
HVL Common Code Base Documentation

Release 0.6.1

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

May 08, 2021

CONTENTS:

1	HVL Common Code Base	1
1.1	Features	1
1.2	Documentation	4
1.3	Credits	4
2	Installation	5
2.1	Stable release	5
2.2	From sources	5
2.3	Additional system libraries	6
3	Usage	7
4	API Documentation	9
4.1	hvl_ccb	9
5	Contributing	169
5.1	Types of Contributions	169
5.2	Get Started!	170
5.3	Merge Request Guidelines	171
5.4	Tips	171
5.5	Deploying	172
6	Credits	173
6.1	Maintainers	173
6.2	Authors	173
6.3	Contributors	173
7	History	175
7.1	0.6.1 (2021-05-08)	175
7.2	0.6.0 (2021-04-23)	175
7.3	0.5.0 (2020-11-11)	176
7.4	0.4.0 (2020-07-16)	176
7.5	0.3.5 (2020-02-18)	177
7.6	0.3.4 (2019-12-20)	177
7.7	0.3.3 (2019-05-08)	178
7.8	0.3.2 (2019-05-08)	178
7.9	0.3.1 (2019-05-02)	178
7.10	0.3 (2019-05-02)	178
7.11	0.2.1 (2019-04-01)	178
7.12	0.2.0 (2019-03-31)	178
7.13	0.1.0 (2019-02-06)	179

8 Indices and tables	181
Python Module Index	183
Index	185

HVL COMMON CODE BASE

Python common code base to control devices high voltage research devices, in particular, as used in Christian Franck's High Voltage Lab (HVL), D-ITET, ETH.

- Free software: GNU General Public License v3
- Copyright (c) 2019-2021 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 devices wrappers in `hvl_ccb` provide a standardised API with configuration dataclasses, various settings and options enumerations, as well as start/stop methods. Currently, wrappers to control the following devices are available:

Function/Type	Devices
Data acquisition and Digital IO	LabJack (T4, T7, T7-PRO)
Experiment control	HVL Supercube with and without Frequency Converter
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)
Laser	CryLaS pulsed laser CryLaS laser attenuator
Oscilloscope	Rhode & Schwarz RTO 1024 TiePie (HS5, HS6, WS5)
Power supply	Elektro-Automatik PS19000 FuG Elektronik Heinzinger PNC Technix capacitor charger
Stepper motor drive	Newport SMC100PP Schneider Electric ILS2T
Waveform generator	TiePie (HS5, WS5)

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 dummy communication protocol concept; this is because these devices build on propriety vendor libraries that communicate with vendor devices, like in case of the TiePie 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 to use the following communication protocols are available:

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

1.2 Documentation

Note: if you're planning to contribute to the `hvl_ccb` project do read beforehand 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 documentation build requirements:

```
$ 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 additional external libraries installations like `tiepie` or `labjack`, specify on installation extra features you want by running e.g. this command in your terminal:

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

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

Please note that for some of the dependencies, like e.g. for `labjack-ljm` or `python-libtiepie`, you may need to separately install additional system libraries.

For [LJM Library](#) please make sure that you have installed version 2019_02_14 or higher.

CHAPTER THREE

USAGE

To use HVL Common Code Base in a project:

```
import hv1_ccb
```


API DOCUMENTATION

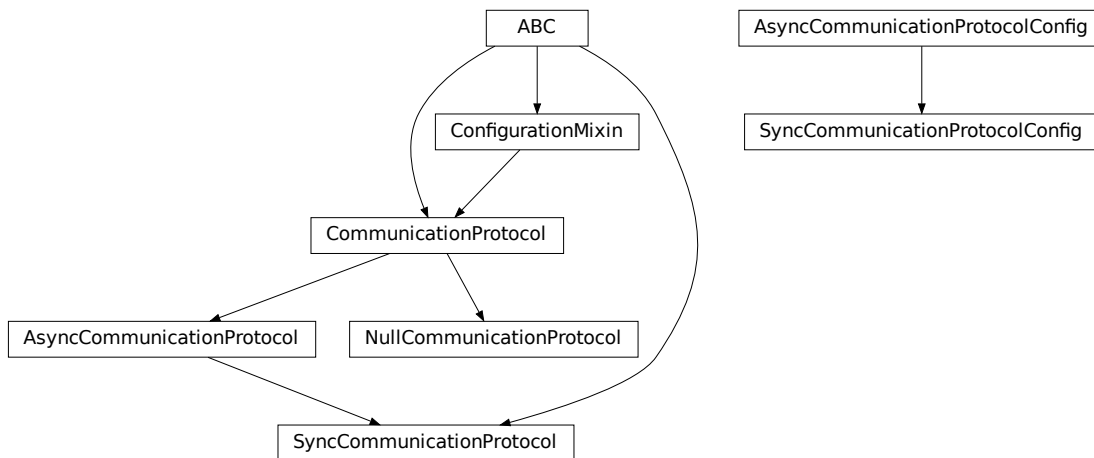
4.1 hvl_ccb

4.1.1 Subpackages

`hvl_ccb.comm`

Submodules

`hvl_ccb.comm.base`



Module with base classes for communication protocols.

```
class AsyncCommunicationProtocol (config)
    Bases: hvl_ccb.comm.base.CommunicationProtocol
```

Abstract base class for asynchronous communication protocols

```
static config_cls () → Type[hvl_ccb.comm.base.AsyncCommunicationProtocolConfig]
    Return the default configdataclass class.
```

Returns a reference to the default configdataclass class

read() → str

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

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

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

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

Parameters

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

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

abstract read_bytes() → bytes

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

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

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

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

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

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

Parameters

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

read_text() → str

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

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

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

Raises *SerialCommunicationIOError* – when communication port is not opened

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

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

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

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

Parameters

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

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

write (*text*: str)

Write text as *str* to the communication.

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

abstract `write_bytes (data: bytes) → int`
Write data as *bytes* to the communication.

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

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

Returns number of bytes written

write_text (text: str)

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

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

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

Raises `SerialCommunicationIOError` – when communication port is not opened

```
class AsyncCommunicationProtocolConfig (terminator: bytes = b'\n', encoding: str =
                                         'utf-8', encoding_error_handling: str = 'replace',
                                         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 = 'replace'

Encoding error handling scheme as defined here: <https://docs.python.org/3/library/codecs.html#error-handlers> By default replacing invalid characters with “*uFFFD*” REPLACEMENT CHARACTER on decoding and with “?” on decoding.

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

class CommunicationProtocol (config)

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

Communication protocol abstract base class.

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

access_lock

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

abstract close ()

Close the communication protocol

abstract open ()

Open communication protocol

class NullCommunicationProtocol (config)

Bases: *hvl_ccb.comm.base.CommunicationProtocol*

Communication protocol that does nothing.

close () → None

Void close function.

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

Empty configuration

Returns EmptyConfig

open () → None

Void open function.

class SyncCommunicationProtocol (config)

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

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

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

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query (command: str) → Optional[str]

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

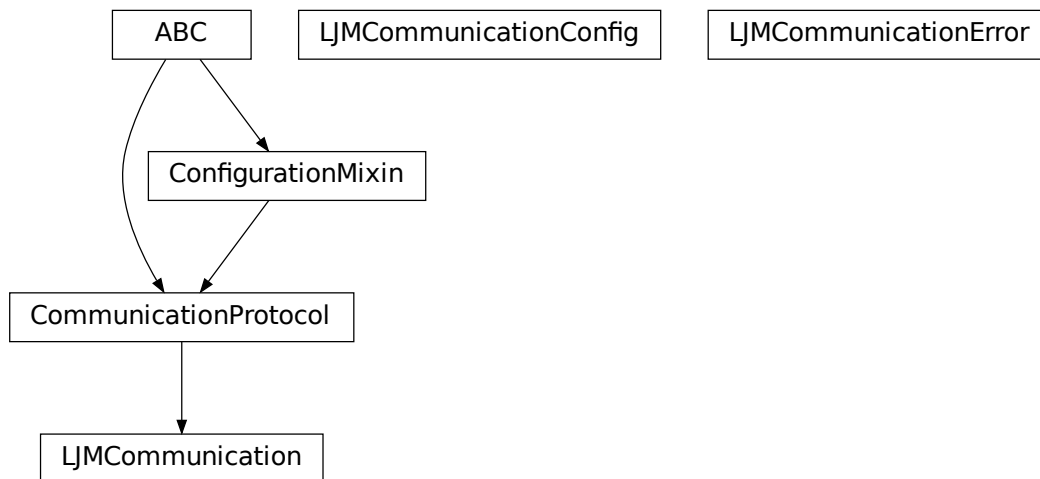
Parameters **command** – Command to send

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


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

Bases: `hvl_ccb.comm.base.AsyncCommunicationProtocolConfig`

`hvl_ccb.comm.labjack_ljm`



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

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

```
class LJMCommunication(configuration)
```

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Communication protocol implementing the LabJack LJM Library Python wrapper.

close() → None

Close the communication port.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

property is_open

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: Type[numbers.Real] = <class 'float'>) → Union[numbers.Real, Sequence[numbers.Real]]
Read one or more input numeric values by name.

Parameters

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

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

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

write_name (name: str, value: numbers.Real) → None
Write one value to a named output.

Parameters

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

write_names (name_value_dict: Dict[str, numbers.Real]) → None
Write more than one value at once to named outputs.

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

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

Bases: object

Configuration dataclass for *LJMCommunication*.

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

Bases: *hvl_ccb_utils.enum.AutoNumberNameEnum*

LabJack connection type.

ANY = 1

ETHERNET = 4

TCP = 3

USB = 2

WIFI = 5

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

Bases: *hvl_ccb_utils.enum.AutoNumberNameEnum*

LabJack device types.

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

ANY = 1

T4 = 2

T7 = 3

T7_PRO = 4

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

Get LabJack device type instance via LabJack product ID.

Note: Product ID is not unambiguous for LabJack devices.

Parameters *p_id* – Product ID of a LabJack device

Returns Instance or list of instances of *LabJackDeviceType*

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

clean_values () → None

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

connection_type: Union[str, *hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType*]

Can be either string or of enum *ConnectionType*.

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

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

force_value (*fieldname, value*)

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

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

Parameters

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

identifier: str = 'ANY'

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

is_configdataclass = True

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

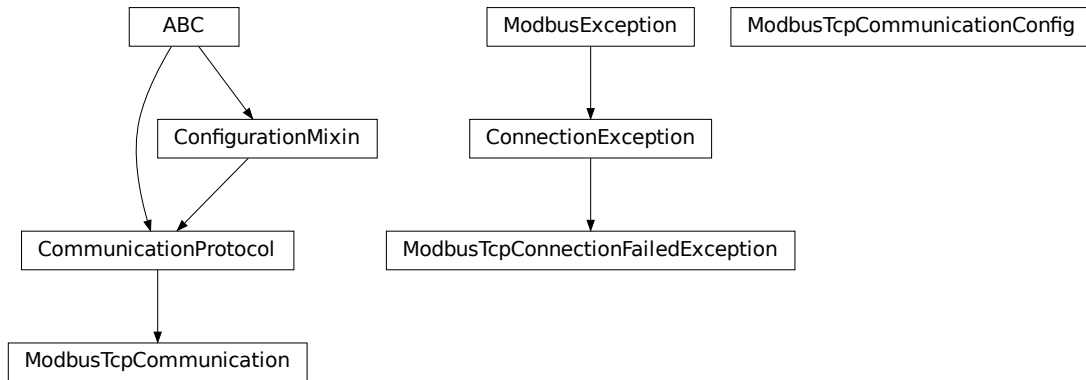
Returns a list of strings containing all required keys.

exception LJMCommunicationError

Bases: *Exception*

Errors coming from LJMCommunication.

hvl_ccb.comm.modbus_tcp



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

class `ModbusTcpCommunication` (*configuration*)

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Implements the Communication Protocol for modbus TCP.

close ()

Close the Modbus TCP connection.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

open () → None

Open the Modbus TCP connection.

Raises `ModbusTcpConnectionFailedException` – if the connection fails.

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

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

Parameters

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

Returns list of *int* values

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

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

Parameters

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

Returns list of *int* values

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

Write values from the specified address forward.

Parameters

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

class ModbusTcpCommunicationConfig (*host: str, 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: str

Host is the IP address of the connected device.

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

TCP port

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.

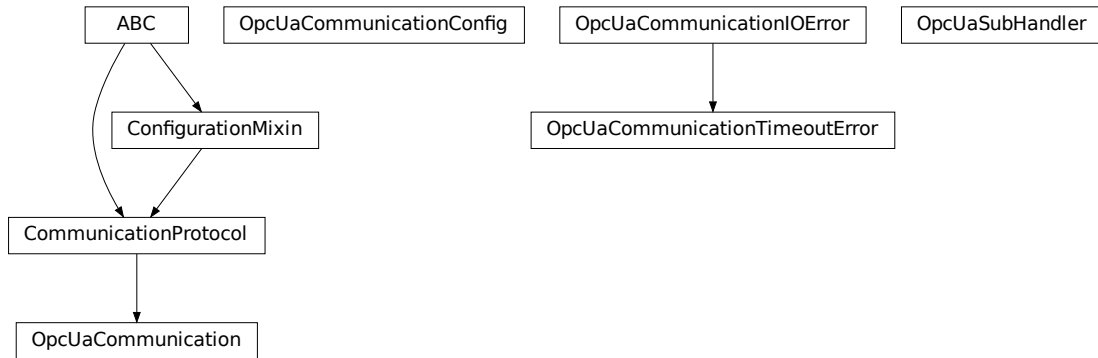
unit: int

Unit number to be used when connecting with Modbus/TCP. Typically this is used when connecting to a relay having Modbus/RTU-connected devices.

exception ModbusTcpConnectionFailedException (*string=""*)

Bases: `pymodbus.exceptions.ConnectionException`

Exception raised when the connection failed.

hvl_ccb.comm.opc

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

class OpcUaCommunication (*config*)

Bases: *hvl_ccb.comm.base.CommunicationProtocol*

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

close () → None

Close the connection to the OPC UA server.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

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

Initialize monitored nodes.

Parameters

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

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

property is_open

Flag indicating if the communication port is open.

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

open () → None

Open the communication to the OPC UA server.

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

read (*node_id*, *ns_index*)

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

Parameters

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

Returns the value of the node object.

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

write (*node_id*, *ns_index*, *value*) → None
Write a value to a node with name *name*.

Parameters

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

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

```
class OpcUaCommunicationConfig (host: str, endpoint_name: str, port: int = 4840,  
                                sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler =  
                                <hvl_ccb.comm.opc.OpcUaSubHandler object>, up-  
                                date_period: int = 500, wait_timeout_retry_sec: 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: str

Hostname or IP-Address of the OPC UA server.

is_configdataclass = True

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

max_timeout_retry_nr: int = 5

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

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

port: `int = 4840`

Port of the OPC UA server to connect to.

classmethod `required_keys()` → Sequence[str]

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

Returns a list of strings containing all required keys.

sub_handler: `hvl_ccb.comm.opc.OpcUaSubHandler` = <hvl_ccb.comm.opc.OpcUaSubHandler obj>
object to use for handling subscriptions.

update_period: `int = 500`

Update period for generating datachange events in OPC UA [milli seconds]

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`

OPC-UA communication I/O error.

exception `OpcUaCommunicationTimeoutError`

Bases: `hvl_ccb.comm.opc.OpcUaCommunicationIOError`

OPC-UA communication timeout error.

class `OpcUaSubHandler`

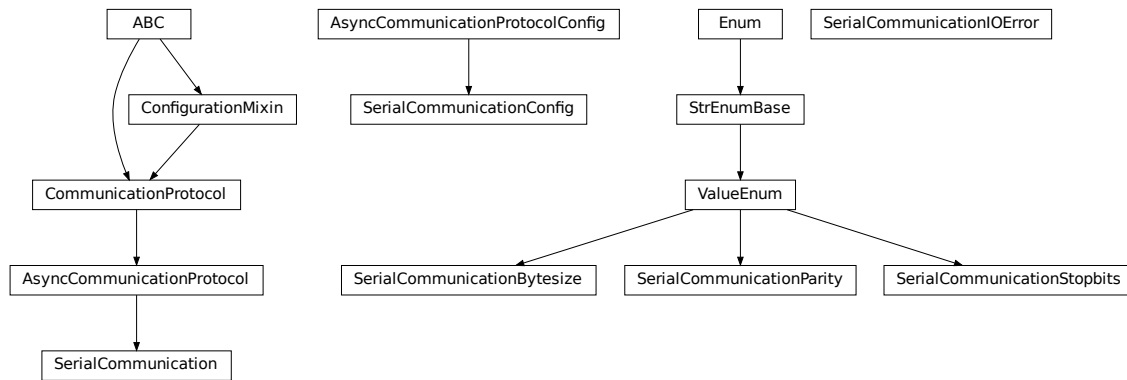
Bases: `object`

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

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

datachange_notification (*node, val, data*)

event_notification (*event*)

hvl_ccb.comm.serial

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

class SerialCommunication (*configuration*)

Bases: `hvl_ccb.comm.base.AsyncCommunicationProtocol`

Implements the Communication Protocol for serial ports.

close()

Close the serial connection.

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

property is_open

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=<object object>, names=None, module=None,  
                                   type=None, start=1, boundary=None)
```

Bases: *hvl_ccb.utils.enum.ValueEnum*

Serial communication bytesize.

EIGHTBITS = 8

FIVEBITS = 5

SEVENBITS = 7

SIXBITS = 6

```
class SerialCommunicationConfig (terminator: bytes = b'\r\n', encoding: str =  
                                'utf-8', encoding_error_handling: str = 'replace',  
                                wait_sec_read_text_nonempty: Union[int, float] = 0.5,  
                                default_n_attempts_read_text_nonempty: int = 10, port:  
                                Optional[str] = None, baudrate: int = 9600, parity: Union[str,  
                                hvl_ccb.comm.serial.SerialCommunicationParity] = <Serial-  
                                CommunicationParity.NONE: 'N'>, stopbits: Union[int, float,  
                                hvl_ccb.comm.serial.SerialCommunicationStopbits] = <Seri-  
                                alCommunicationStopbits.ONE: 1>, bytesize: Union[int,  
                                hvl_ccb.comm.serial.SerialCommunicationBytesize] =  
                                <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:  
                                Union[int, float] = 2)
```

Bases: *hvl_ccb.comm.base.AsyncCommunicationProtocolConfig*

Configuration dataclass for *SerialCommunication*.

Bytesize

alias of *hvl_ccb.comm.serial.SerialCommunicationBytesize*

Parity

alias of *hvl_ccb.comm.serial.SerialCommunicationParity*

Stopbits

alias of *hvl_ccb.comm.serial.SerialCommunicationStopbits*

baudrate: int = 9600

Baudrate of the serial port

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

Size of a byte, 5 to 8

clean_values ()

create_serial_port () → serial.serialposix.Serial

Create a serial port instance according to specification in this configuration

Returns Closed serial port instance

force_value (fieldname, value)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

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

Parity to be used for the connection.

port: Optional[str] = None

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

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

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

Stopbits setting, can be 1, 1.5 or 2.

terminator_str () → str

timeout: Union[int, float] = 2

Timeout in seconds for the serial port

exception SerialCommunicationIOError

Bases: OSError

Serial communication related I/O errors.

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

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

Serial communication parity.

EVEN = 'E'

MARK = 'M'

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

NONE = 'N'

ODD = 'O'

SPACE = 'S'

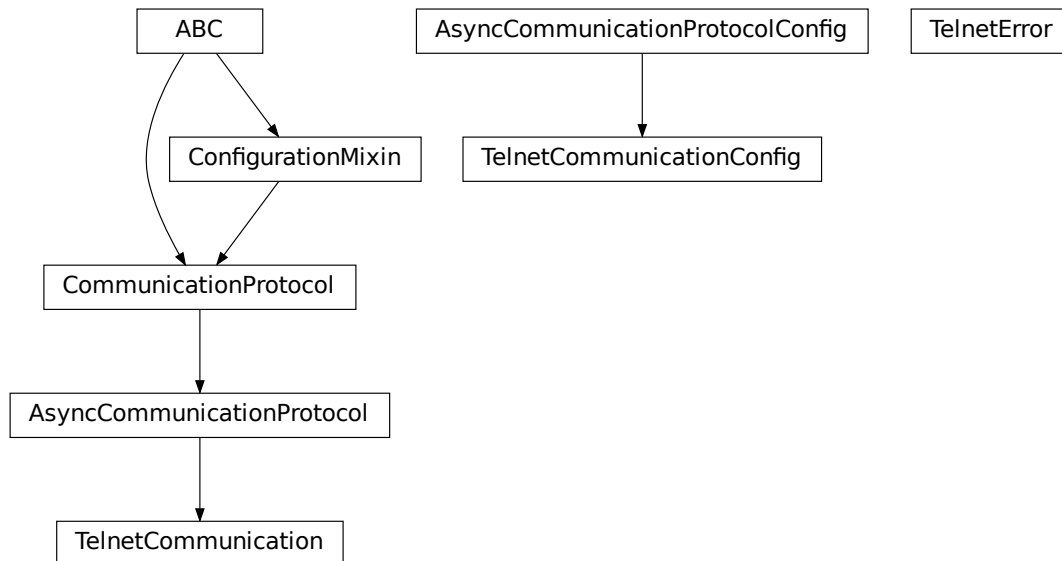
class SerialCommunicationStopbits (value=<object object>, names=None, module=None, type=None, start=1, boundary=None)

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

Serial communication stopbits.

```
ONE = 1
ONE_POINT_FIVE = 1.5
TWO = 2
```

`hvl_ccb.comm.telnet`



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

```
class TelnetCommunication(configuration)
    Bases: hvl_ccb.comm.base.AsyncCommunicationProtocol
```

Implements the Communication Protocol for telnet.

```
close()
    Close the telnet connection unless it is not closed.
```

```
static config_cls()
    Return the default configdataclass class.
```

Returns a reference to the default configdataclass class

```
property is_open
    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 = 'replace',
                                wait_sec_read_text_nonempty: Union[int, float] = 0.5,
                                default_n_attempts_read_text_nonempty: int = 10, host:
                                Optional[str] = None, port: int = 0, timeout: Union[int, float]
                                = 0.2)
```

Bases: *hvl_ccb.comm.base.AsyncCommunicationProtocolConfig*

Configuration dataclass for *TelnetCommunication*.

clean_values ()

create_telnet () → Optional[telnetlib.Telnet]

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

force_value (*fieldname, value*)

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

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

Parameters

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

host: Optional[str] = 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.

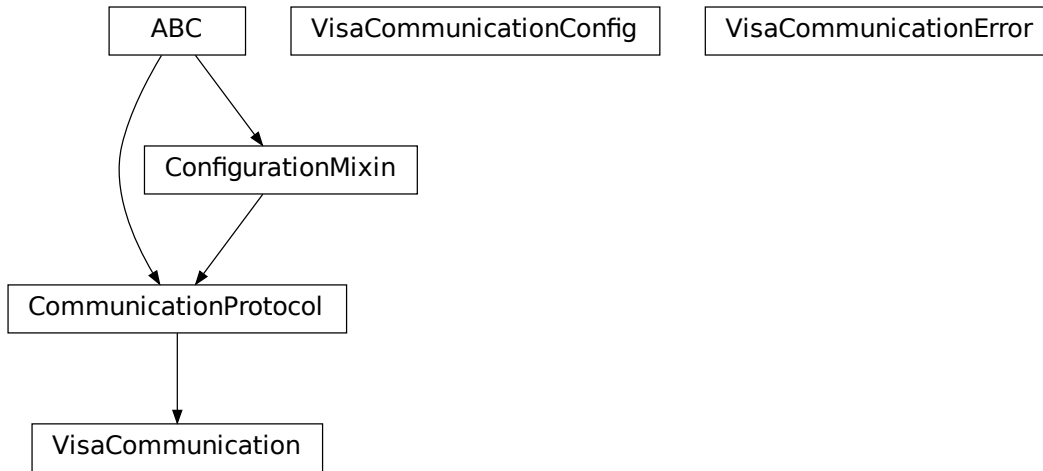
Returns a list of strings containing all required keys.

timeout: Union[int, float] = 0.2

Timeout for reading a line

exception TelnetErrorBases: `Exception`

Telnet communication related errors.

hvl_ccb.comm.visa

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

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

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

class VisaCommunication (*configuration*)

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Implements the Communication Protocol for VISA / SCPI.

MULTI_COMMANDS_MAX = 5

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

MULTI_COMMANDS_SEPARATOR = ';'

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

WAIT_AFTER_WRITE = 0.08

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

close () → None

Close the VISA connection and invalidates the handle.

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

Return the default configdataclass class.

Returns a reference to the default configdataclass class

open () → None

Open the VISA connection and create the resource.

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

A combination of write(message) and read.

Parameters **commands** – list of commands

Returns list of values

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

spoll () → int

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

Returns integer representation of the status byte

Raises *VisaCommunicationError* – when connection was not started

write (*commands: str) → None

Write commands. No answer is read or expected.

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

Raises *VisaCommunicationError* – when connection was not started

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

Bases: object

VisaCommunication configuration dataclass.

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

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

Supported VISA Interface types.

TCPIP_INSTR = 2

VXI-11 protocol

TCPIP_SOCKET = 1

VISA-RAW protocol

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

Address string specific to the VISA interface type.

Parameters

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

Returns address string

property address

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: `str`

IP address of the VISA device. DNS names are currently unsupported.

interface_type: `Union[str, hv1_ccb.comm.visa.VisaCommunicationConfig.InterfaceType]`

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

is_configdataclass = `True`

classmethod keys () → `Sequence[str]`

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

open_timeout: `int = 1000`

Timeout for opening the connection, in milli seconds.

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

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

Returns a list of strings containing all optional keys.

port: `int = 5025`

TCP port, standard is 5025.

read_termination: `str = '\n'`

Read termination character.

classmethod required_keys () → `Sequence[str]`

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

Returns a list of strings containing all required keys.

timeout: `int = 5000`

Timeout for commands in milli seconds.

visa_backend: `str = ''`

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

write_termination: `str = '\n'`

Write termination character.

exception VisaCommunicationError

Bases: `Exception`

Base class for VisaCommunication errors.

Module contents

Communication protocols subpackage.

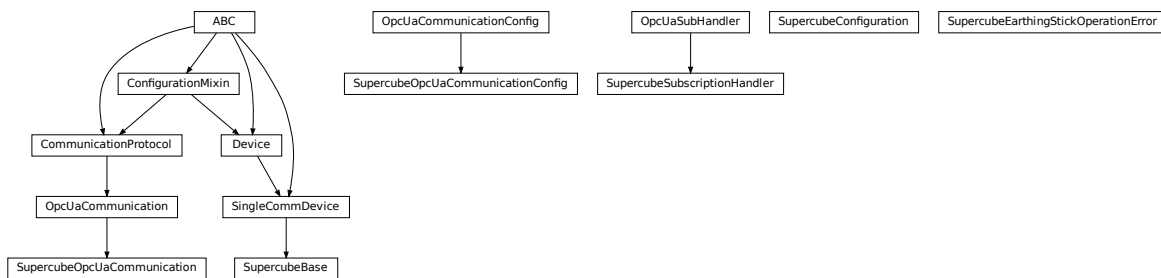
`hvl_ccb.dev`

Subpackages

`hvl_ccb.dev.supercube`

Submodules

`hvl_ccb.dev.supercube.base`



Base classes for the Supercube device.

class `SupercubeBase` (*com*, *dev_config=None*)
 Bases: `hvl_ccb.dev.base.SingleCommDevice`

Base class for Supercube variants.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls ()

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

Returns the type of the standard communication protocol for this device

display_message_board () → None

Display 15 newest messages

display_status_board () → None

Display status board.

get_cee16_socket () → bool

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

Returns the on-state of the CEE16 power socket

get_door_status (*door: int*) → *hvl_ccb.dev.supercube.constants.DoorStatus*

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

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

Returns the door status

get_earthing_rod_status (*earthing_rod: int*) → *hvl_ccb.dev.supercube.constants.EarthingRodStatus*

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

Parameters **earthing_rod** – the earthing rod number (1..3)

Returns the earthing rod status

get_earthing_stick_manual (*number: int*) → *hvl_ccb.dev.supercube.constants.EarthingStickOperation*

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

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

Returns operation of the earthing stick in a manual operating mode (open == 0, close == 1)

Raises **ValueError** – when earthing stick number is not valid

get_earthing_stick_operating_status (*number: int*) → *hvl_ccb.dev.supercube.constants.EarthingStickOperatingStatus*

Get the operating status of an earthing stick.

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

Returns earthing stick operating status (auto == 0, manual == 1)

Raises **ValueError** – when earthing stick number is not valid

get_earthing_stick_status (*number: int*) → *hvl_ccb.dev.supercube.constants.EarthingStickStatus*

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

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

Returns earthing stick status

Raises **ValueError** – when earthing stick number is not valid

get_measurement_ratio (*channel: int*) → float

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

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

Returns the ratio

Raises **ValueError** – when channel is not valid

get_measurement_voltage (*channel: int*) → float

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

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

Returns measured voltage

Raises **ValueError** – when channel is not valid

get_status () → *hvl_ccb.dev.supercube.constants.SafetyStatus*

Get the safety circuit status of the Supercube. :return: the safety status of the supercube's state machine.

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

Get the state of a support socket input.

Parameters

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

Returns digital input read state

Raises **ValueError** – when port or contact number is not valid

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

Get the state of a support socket output.

Parameters

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

Returns digital output read state

Raises **ValueError** – when port or contact number is not valid

get_t13_socket (*port: int*) → bool

Read the state of a SEV T13 power socket.

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

Returns on-state of the power socket

Raises **ValueError** – when port is not valid

operate (*state: bool*) → None

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

Parameters **state** – set operate state

operate_earthing_stick (*number: int, operation: [hvl_ccb.dev.supercube.constants.EarthingStickOperation](#)*)

→ None

Operation of an earthing stick, which is set to manual operation. If an earthing stick is set to manual, it stays closed even if the system is in states RedReady or RedOperate.

Parameters

- **number** – number of the earthing stick (1..6)
- **operation** – earthing stick manual status (close or open)

Raises **SupercubeEarthingStickOperationError** – when operating status of given number's earthing stick is not manual

quit_error () → None

Quits errors that are active on the Supercube.

read (*node_id: str*)

Local wrapper for the OPC UA communication protocol read method.

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

Returns the value of the variable

ready (*state: bool*) → None

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

Parameters **state** – set ready state

set_cee16_socket (*state: bool*) → None

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

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

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

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_remote_control (*state: bool*) → None

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

Parameters **state** – desired remote control state

set_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.

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

Set the state of a support output socket.

Parameters

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

Raises **ValueError** – when port or contact number is not valid

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

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

This function is blocking.

Parameters

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

Raises ValueError – when port or contact number is not valid

set_t13_socket (*port: int, state: bool*) → None
Set the state of a SEV T13 power socket.

Parameters

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

Raises ValueError – when port is not valid or state is not of type bool

start () → None

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

stop () → None

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

write (*node_id, value*) → None

Local wrapper for the OPC UA communication protocol write method.

Parameters

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

class SupercubeConfiguration (*namespace_index: int = 3, polling_delay_sec: Union[int, float] = 5.0, polling_interval_sec: Union[int, float] = 1.0*)

Bases: object

Configuration dataclass for the Supercube 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)

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.

exception **SupercubeEarthingStickOperationError**

Bases: `Exception`

class **SupercubeOpcUaCommunication** (*config*)

Bases: `hvl_ccb.comm.opc.OpcUaCommunication`

Communication protocol specification for Supercube devices.

static **config_cls** ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class SupercubeOpcUaCommunicationConfig (host: str, endpoint_name: str,  
                                         port: int = 4840, sub_handler:  
                                         hvl_ccb.comm.opc.OpcUaSubHandler =  
                                         <hvl_ccb.dev.supercube.base.SupercubeSubscriptionHandler  
                                         object>, update_period: int = 500,  
                                         wait_timeout_retry_sec: Union[int, float] =  
                                         1, max_timeout_retry_nr: int = 5)
```

Bases: `hvl_ccb.comm.opc.OpcUaCommunicationConfig`

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

force_value (*fieldname, value*)

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

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

Parameters

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

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

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

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

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

Returns a list of strings containing all optional keys.

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

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

Returns a list of strings containing all required keys.

sub_handler: `hvl_ccb.comm.opc.OpcUaSubHandler` = `<hvl_ccb.dev.supercube.base.SupercubeSubscriptionHandler for data change events>`

class **SupercubeSubscriptionHandler**

Bases: `hvl_ccb.comm.opc.OpcUaSubHandler`

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

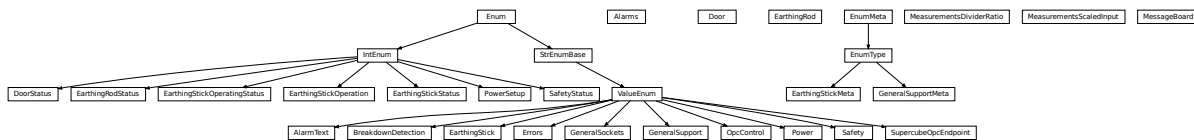
datachange_notification (*node: opcua.common.node.Node, val, data*)

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

Parameters

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

hvl_ccb.dev.supercube.constants



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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

```
Alarm1 = 'STOP Emergency Stop 1'
Alarm10 = 'STOP Earthing stick 2 error while opening'
Alarm11 = 'STOP Earthing stick 3 error while opening'
Alarm12 = 'STOP Earthing stick 4 error while opening'
Alarm13 = 'STOP Earthing stick 5 error while opening'
Alarm14 = 'STOP Earthing stick 6 error while opening'
Alarm15 = 'STOP Earthing stick 1 error while closing'
Alarm16 = 'STOP Earthing stick 2 error while closing'
Alarm17 = 'STOP Earthing stick 3 error while closing'
Alarm18 = 'STOP Earthing stick 4 error while closing'
Alarm19 = 'STOP Earthing stick 5 error while closing'
Alarm2 = 'STOP Emergency Stop 2'
Alarm20 = 'STOP Earthing stick 6 error while closing'
Alarm21 = 'STOP Safety fence 1'
Alarm22 = 'STOP Safety fence 2'
Alarm23 = 'STOP OPC connection error'
Alarm24 = 'STOP Grid power failure'
```

```
Alarm25 = 'STOP UPS failure'
Alarm26 = 'STOP 24V PSU failure'
Alarm3 = 'STOP Emergency Stop 3'
Alarm4 = 'STOP Safety Switch 1 error'
Alarm41 = 'WARNING Door 1: Use earthing rod!'
Alarm42 = 'MESSAGE Door 1: Earthing rod is still in setup.'
Alarm43 = 'WARNING Door 2: Use earthing rod!'
Alarm44 = 'MESSAGE Door 2: Earthing rod is still in setup.'
Alarm45 = 'WARNING Door 3: Use earthing rod!'
Alarm46 = 'MESSAGE Door 3: Earthing rod is still in setup.'
Alarm47 = 'MESSAGE UPS charge < 85%'
Alarm48 = 'MESSAGE UPS running on battery'
Alarm5 = 'STOP Safety Switch 2 error'
Alarm6 = 'STOP Door 1 lock supervision'
Alarm7 = 'STOP Door 2 lock supervision'
Alarm8 = 'STOP Door 3 lock supervision'
Alarm9 = 'STOP Earthing stick 1 error while opening'
```

```
classmethod get (alarm: int)
```

Get the attribute of this enum for an alarm number.

Parameters `alarm` – the alarm number

Returns the enum for the desired alarm number

```
not_defined = 'NO ALARM TEXT DEFINED'
```

```
class Alarms (value=<object object>, names=None, module=None, type=None, start=1, bound-
               ary=None)
```

Bases: `hvl_ccb.dev.supercube.constants._AlarmEnumBase`

Alarms enumeration containing all variable NodeID strings for the alarm array.

```
Alarm1 = '"DB_Alarm_HMI"."Alarm1"'
Alarm10 = '"DB_Alarm_HMI"."Alarm10"'
Alarm100 = '"DB_Alarm_HMI"."Alarm100"'
Alarm101 = '"DB_Alarm_HMI"."Alarm101"'
Alarm102 = '"DB_Alarm_HMI"."Alarm102"'
Alarm103 = '"DB_Alarm_HMI"."Alarm103"'
Alarm104 = '"DB_Alarm_HMI"."Alarm104"'
Alarm105 = '"DB_Alarm_HMI"."Alarm105"'
Alarm106 = '"DB_Alarm_HMI"."Alarm106"'
Alarm107 = '"DB_Alarm_HMI"."Alarm107"'
Alarm108 = '"DB_Alarm_HMI"."Alarm108"'
```



```
Alarm109 = ' "DB_Alarm_HMI"."Alarm109" '
Alarm11 = ' "DB_Alarm_HMI"."Alarm11" '
Alarm110 = ' "DB_Alarm_HMI"."Alarm110" '
Alarm111 = ' "DB_Alarm_HMI"."Alarm111" '
Alarm112 = ' "DB_Alarm_HMI"."Alarm112" '
Alarm113 = ' "DB_Alarm_HMI"."Alarm113" '
Alarm114 = ' "DB_Alarm_HMI"."Alarm114" '
Alarm115 = ' "DB_Alarm_HMI"."Alarm115" '
Alarm116 = ' "DB_Alarm_HMI"."Alarm116" '
Alarm117 = ' "DB_Alarm_HMI"."Alarm117" '
Alarm118 = ' "DB_Alarm_HMI"."Alarm118" '
Alarm119 = ' "DB_Alarm_HMI"."Alarm119" '
Alarm12 = ' "DB_Alarm_HMI"."Alarm12" '
Alarm120 = ' "DB_Alarm_HMI"."Alarm120" '
Alarm121 = ' "DB_Alarm_HMI"."Alarm121" '
Alarm122 = ' "DB_Alarm_HMI"."Alarm122" '
Alarm123 = ' "DB_Alarm_HMI"."Alarm123" '
Alarm124 = ' "DB_Alarm_HMI"."Alarm124" '
Alarm125 = ' "DB_Alarm_HMI"."Alarm125" '
Alarm126 = ' "DB_Alarm_HMI"."Alarm126" '
Alarm127 = ' "DB_Alarm_HMI"."Alarm127" '
Alarm128 = ' "DB_Alarm_HMI"."Alarm128" '
Alarm129 = ' "DB_Alarm_HMI"."Alarm129" '
Alarm13 = ' "DB_Alarm_HMI"."Alarm13" '
Alarm130 = ' "DB_Alarm_HMI"."Alarm130" '
Alarm131 = ' "DB_Alarm_HMI"."Alarm131" '
Alarm132 = ' "DB_Alarm_HMI"."Alarm132" '
Alarm133 = ' "DB_Alarm_HMI"."Alarm133" '
Alarm134 = ' "DB_Alarm_HMI"."Alarm134" '
Alarm135 = ' "DB_Alarm_HMI"."Alarm135" '
Alarm136 = ' "DB_Alarm_HMI"."Alarm136" '
Alarm137 = ' "DB_Alarm_HMI"."Alarm137" '
Alarm138 = ' "DB_Alarm_HMI"."Alarm138" '
Alarm139 = ' "DB_Alarm_HMI"."Alarm139" '
Alarm14 = ' "DB_Alarm_HMI"."Alarm14" '
Alarm140 = ' "DB_Alarm_HMI"."Alarm140" '
```

```
Alarm141 = ' "DB_Alarm_HMI"."Alarm141" '
Alarm142 = ' "DB_Alarm_HMI"."Alarm142" '
Alarm143 = ' "DB_Alarm_HMI"."Alarm143" '
Alarm144 = ' "DB_Alarm_HMI"."Alarm144" '
Alarm145 = ' "DB_Alarm_HMI"."Alarm145" '
Alarm146 = ' "DB_Alarm_HMI"."Alarm146" '
Alarm147 = ' "DB_Alarm_HMI"."Alarm147" '
Alarm148 = ' "DB_Alarm_HMI"."Alarm148" '
Alarm149 = ' "DB_Alarm_HMI"."Alarm149" '
Alarm15 = ' "DB_Alarm_HMI"."Alarm15" '
Alarm150 = ' "DB_Alarm_HMI"."Alarm150" '
Alarm151 = ' "DB_Alarm_HMI"."Alarm151" '
Alarm16 = ' "DB_Alarm_HMI"."Alarm16" '
Alarm17 = ' "DB_Alarm_HMI"."Alarm17" '
Alarm18 = ' "DB_Alarm_HMI"."Alarm18" '
Alarm19 = ' "DB_Alarm_HMI"."Alarm19" '
Alarm2 = ' "DB_Alarm_HMI"."Alarm2" '
Alarm20 = ' "DB_Alarm_HMI"."Alarm20" '
Alarm21 = ' "DB_Alarm_HMI"."Alarm21" '
Alarm22 = ' "DB_Alarm_HMI"."Alarm22" '
Alarm23 = ' "DB_Alarm_HMI"."Alarm23" '
Alarm24 = ' "DB_Alarm_HMI"."Alarm24" '
Alarm25 = ' "DB_Alarm_HMI"."Alarm25" '
Alarm26 = ' "DB_Alarm_HMI"."Alarm26" '
Alarm27 = ' "DB_Alarm_HMI"."Alarm27" '
Alarm28 = ' "DB_Alarm_HMI"."Alarm28" '
Alarm29 = ' "DB_Alarm_HMI"."Alarm29" '
Alarm3 = ' "DB_Alarm_HMI"."Alarm3" '
Alarm30 = ' "DB_Alarm_HMI"."Alarm30" '
Alarm31 = ' "DB_Alarm_HMI"."Alarm31" '
Alarm32 = ' "DB_Alarm_HMI"."Alarm32" '
Alarm33 = ' "DB_Alarm_HMI"."Alarm33" '
Alarm34 = ' "DB_Alarm_HMI"."Alarm34" '
Alarm35 = ' "DB_Alarm_HMI"."Alarm35" '
Alarm36 = ' "DB_Alarm_HMI"."Alarm36" '
Alarm37 = ' "DB_Alarm_HMI"."Alarm37" '
```

```
Alarm38 = ' "DB_Alarm_HMI"."Alarm38" '
Alarm39 = ' "DB_Alarm_HMI"."Alarm39" '
Alarm4 = ' "DB_Alarm_HMI"."Alarm4" '
Alarm40 = ' "DB_Alarm_HMI"."Alarm40" '
Alarm41 = ' "DB_Alarm_HMI"."Alarm41" '
Alarm42 = ' "DB_Alarm_HMI"."Alarm42" '
Alarm43 = ' "DB_Alarm_HMI"."Alarm43" '
Alarm44 = ' "DB_Alarm_HMI"."Alarm44" '
Alarm45 = ' "DB_Alarm_HMI"."Alarm45" '
Alarm46 = ' "DB_Alarm_HMI"."Alarm46" '
Alarm47 = ' "DB_Alarm_HMI"."Alarm47" '
Alarm48 = ' "DB_Alarm_HMI"."Alarm48" '
Alarm49 = ' "DB_Alarm_HMI"."Alarm49" '
Alarm5 = ' "DB_Alarm_HMI"."Alarm5" '
Alarm50 = ' "DB_Alarm_HMI"."Alarm50" '
Alarm51 = ' "DB_Alarm_HMI"."Alarm51" '
Alarm52 = ' "DB_Alarm_HMI"."Alarm52" '
Alarm53 = ' "DB_Alarm_HMI"."Alarm53" '
Alarm54 = ' "DB_Alarm_HMI"."Alarm54" '
Alarm55 = ' "DB_Alarm_HMI"."Alarm55" '
Alarm56 = ' "DB_Alarm_HMI"."Alarm56" '
Alarm57 = ' "DB_Alarm_HMI"."Alarm57" '
Alarm58 = ' "DB_Alarm_HMI"."Alarm58" '
Alarm59 = ' "DB_Alarm_HMI"."Alarm59" '
Alarm6 = ' "DB_Alarm_HMI"."Alarm6" '
Alarm60 = ' "DB_Alarm_HMI"."Alarm60" '
Alarm61 = ' "DB_Alarm_HMI"."Alarm61" '
Alarm62 = ' "DB_Alarm_HMI"."Alarm62" '
Alarm63 = ' "DB_Alarm_HMI"."Alarm63" '
Alarm64 = ' "DB_Alarm_HMI"."Alarm64" '
Alarm65 = ' "DB_Alarm_HMI"."Alarm65" '
Alarm66 = ' "DB_Alarm_HMI"."Alarm66" '
Alarm67 = ' "DB_Alarm_HMI"."Alarm67" '
Alarm68 = ' "DB_Alarm_HMI"."Alarm68" '
Alarm69 = ' "DB_Alarm_HMI"."Alarm69" '
Alarm7 = ' "DB_Alarm_HMI"."Alarm7" '
```

```
Alarm70 = ' "DB_Alarm_HMI"."Alarm70" '
Alarm71 = ' "DB_Alarm_HMI"."Alarm71" '
Alarm72 = ' "DB_Alarm_HMI"."Alarm72" '
Alarm73 = ' "DB_Alarm_HMI"."Alarm73" '
Alarm74 = ' "DB_Alarm_HMI"."Alarm74" '
Alarm75 = ' "DB_Alarm_HMI"."Alarm75" '
Alarm76 = ' "DB_Alarm_HMI"."Alarm76" '
Alarm77 = ' "DB_Alarm_HMI"."Alarm77" '
Alarm78 = ' "DB_Alarm_HMI"."Alarm78" '
Alarm79 = ' "DB_Alarm_HMI"."Alarm79" '
Alarm8 = ' "DB_Alarm_HMI"."Alarm8" '
Alarm80 = ' "DB_Alarm_HMI"."Alarm80" '
Alarm81 = ' "DB_Alarm_HMI"."Alarm81" '
Alarm82 = ' "DB_Alarm_HMI"."Alarm82" '
Alarm83 = ' "DB_Alarm_HMI"."Alarm83" '
Alarm84 = ' "DB_Alarm_HMI"."Alarm84" '
Alarm85 = ' "DB_Alarm_HMI"."Alarm85" '
Alarm86 = ' "DB_Alarm_HMI"."Alarm86" '
Alarm87 = ' "DB_Alarm_HMI"."Alarm87" '
Alarm88 = ' "DB_Alarm_HMI"."Alarm88" '
Alarm89 = ' "DB_Alarm_HMI"."Alarm89" '
Alarm9 = ' "DB_Alarm_HMI"."Alarm9" '
Alarm90 = ' "DB_Alarm_HMI"."Alarm90" '
Alarm91 = ' "DB_Alarm_HMI"."Alarm91" '
Alarm92 = ' "DB_Alarm_HMI"."Alarm92" '
Alarm93 = ' "DB_Alarm_HMI"."Alarm93" '
Alarm94 = ' "DB_Alarm_HMI"."Alarm94" '
Alarm95 = ' "DB_Alarm_HMI"."Alarm95" '
Alarm96 = ' "DB_Alarm_HMI"."Alarm96" '
Alarm97 = ' "DB_Alarm_HMI"."Alarm97" '
Alarm98 = ' "DB_Alarm_HMI"."Alarm98" '
Alarm99 = ' "DB_Alarm_HMI"."Alarm99" '
```

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

Node ID strings for the breakdown detection.

TODO: these variable NodeIDs are not tested and/or correct yet.

activated = `'Ix_Allg_Breakdown_activated'`

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

reset = `'Qx_Allg_Breakdown_reset'`

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

triggered = `'Ix_Allg_Breakdown_triggered'`

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

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

Bases: `hvl_ccb.dev.supercube.constants._DoorEnumBase`

Variable NodeID strings for doors.

status_1 = `'DB_Safety_Circuit"."Door_1"."si_HMI_status'`

status_2 = `'DB_Safety_Circuit"."Door_2"."si_HMI_status'`

status_3 = `'DB_Safety_Circuit"."Door_3"."si_HMI_status'`

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

Bases: `aenum.IntEnum`

Possible status values for doors.

closed = 2

Door is closed, but not locked.

error = 4

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

inactive = 0

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

locked = 3

Door is closed and locked (safe state).

open = 1

Door is open.

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

Bases: `hvl_ccb.dev.supercube.constants._DoorEnumBase`

Variable NodeID strings for earthing rods.

status_1 = `'DB_Safety_Circuit"."Door_1"."Ix_earthingrod'`

status_2 = `'DB_Safety_Circuit"."Door_2"."Ix_earthingrod'`

status_3 = `'DB_Safety_Circuit"."Door_3"."Ix_earthingrod'`

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

Bases: `aenum.IntEnum`

Possible status values for earthing rods.

experiment_blocked = 0

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

experiment_ready = 1

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

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

Bases: `hvl_ccb.utils.enum.ValueEnum`

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

```
classmethod manual(number: int)
```

Get the manual enum instance for an earthing stick number.

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

Returns the manual instance

Raises **ValueError** – when earthing stick number is not valid

```
manual_1 = 'DB_Safety_Circuit"."Earthstick_1"."sx_earthing_manually'
```

```
manual_2 = 'DB_Safety_Circuit"."Earthstick_2"."sx_earthing_manually'
```

```
manual_3 = 'DB_Safety_Circuit"."Earthstick_3"."sx_earthing_manually'
```

```
manual_4 = 'DB_Safety_Circuit"."Earthstick_4"."sx_earthing_manually'
```

```
manual_5 = 'DB_Safety_Circuit"."Earthstick_5"."sx_earthing_manually'
```

```
manual_6 = 'DB_Safety_Circuit"."Earthstick_6"."sx_earthing_manually'
```

```
classmethod manuals() → Tuple[hvl_ccb.dev.supercube.constants.EarthingStick, ...]
```

Get all earthing stick manual instances.

Returns tuple of manual instances

```
property number
```

Get corresponding earthing stick number.

Returns earthing stick number (1..6)

```
classmethod operating_status(number: int)
```

Get the operating status enum instance for an earthing stick number.

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

Returns the operating status instance

Raises **ValueError** – when earthing stick number is not valid

```
operating_status_1 = 'DB_Safety_Circuit"."Earthstick_1"."sx_manual_control_active'
```

```
operating_status_2 = 'DB_Safety_Circuit"."Earthstick_2"."sx_manual_control_active'
```

```
operating_status_3 = 'DB_Safety_Circuit"."Earthstick_3"."sx_manual_control_active'
```

```
operating_status_4 = 'DB_Safety_Circuit"."Earthstick_4"."sx_manual_control_active'
```

```
operating_status_5 = 'DB_Safety_Circuit"."Earthstick_5"."sx_manual_control_active'
```

```
operating_status_6 = 'DB_Safety_Circuit"."Earthstick_6"."sx_manual_control_active'
```

```
classmethod operating_statuses() → Tuple[hvl_ccb.dev.supercube.constants.EarthingStick,  
                                         ...]
```

Get all earthing stick operating status instances.

Returns tuple of operating status instances

```
classmethod range() → Sequence[int]
```

Integer range of all earthing sticks.

Returns sequence of earthing sticks numbers

classmethod status (*number: int*) → *hvl_ccb.dev.supercube.constants.EarthingStick*

Get the status enum instance for an earthing stick number.

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

Returns the status instance

Raises **ValueError** – when earthing stick number is not valid

status_1 = 'DB_Safety_Circuit"."Earthstick_1"."si_HMI_Status"'

status_2 = 'DB_Safety_Circuit"."Earthstick_2"."si_HMI_Status"'

status_3 = 'DB_Safety_Circuit"."Earthstick_3"."si_HMI_Status"'

status_4 = 'DB_Safety_Circuit"."Earthstick_4"."si_HMI_Status"'

status_5 = 'DB_Safety_Circuit"."Earthstick_5"."si_HMI_Status"'

status_6 = 'DB_Safety_Circuit"."Earthstick_6"."si_HMI_Status"'

classmethod statuses () → *Tuple[hvl_ccb.dev.supercube.constants.EarthingStick, ...]*

Get all earthing stick status instances.

Returns tuple of status instances

class EarthingStickMeta (*clsname, bases, clsdict, **kwargs*)

Bases: *aenum.EnumType*

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

Bases: *aenum.IntEnum*

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

auto = 0

manual = 1

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

Bases: *aenum.IntEnum*

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

close = 1

open = 0

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

Bases: *aenum.IntEnum*

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

closed = 1

Earthing is closed (safe).

error = 3

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

inactive = 0

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

open = 2

Earthing is open (not safe).

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

message = **'DB_Message_Buffer"."Info_active"'**

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

quit = **'DB_Message_Buffer"."Reset_button"'**

Writable boolean for the error quit button.

stop = **'DB_Message_Buffer"."Stop_active"'**

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

warning = **'DB_Message_Buffer"."Warning_active"'**

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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

cee16 = **'Qx_Allg_Socket_CEE16"'**

CEE16 socket (writeable boolean).

t13_1 = **'Qx_Allg_Socket_T13_1"'**

SEV T13 socket No. 1 (writable boolean).

t13_2 = **'Qx_Allg_Socket_T13_2"'**

SEV T13 socket No. 2 (writable boolean).

t13_3 = **'Qx_Allg_Socket_T13_3"'**

SEV T13 socket No. 3 (writable boolean).

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

NodeID strings for the support inputs and outputs.

classmethod contact_range () → Sequence[int]

Integer range of all contacts.

Returns sequence of contact numbers

in_1_1 = **'Ix_Allg_Support1_1"'**

in_1_2 = **'Ix_Allg_Support1_2"'**

in_2_1 = **'Ix_Allg_Support2_1"'**

in_2_2 = **'Ix_Allg_Support2_2"'**

in_3_1 = **'Ix_Allg_Support3_1"'**

in_3_2 = **'Ix_Allg_Support3_2"'**

in_4_1 = **'Ix_Allg_Support4_1"'**

in_4_2 = **'Ix_Allg_Support4_2"'**

in_5_1 = **'Ix_Allg_Support5_1"'**


```

in_5_2 = 'Ix_Allg_Support5_2'
in_6_1 = 'Ix_Allg_Support6_1'
in_6_2 = 'Ix_Allg_Support6_2'

classmethod input (port: int, contact: int)
    Get the NodeID string for a support input.

```

Parameters

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

Returns the node id string**Raises** **ValueError** – when port or contact number is not valid

```

out_1_1 = 'Qx_Allg_Support1_1'
out_1_2 = 'Qx_Allg_Support1_2'
out_2_1 = 'Qx_Allg_Support2_1'
out_2_2 = 'Qx_Allg_Support2_2'
out_3_1 = 'Qx_Allg_Support3_1'
out_3_2 = 'Qx_Allg_Support3_2'
out_4_1 = 'Qx_Allg_Support4_1'
out_4_2 = 'Qx_Allg_Support4_2'
out_5_1 = 'Qx_Allg_Support5_1'
out_5_2 = 'Qx_Allg_Support5_2'
out_6_1 = 'Qx_Allg_Support6_1'
out_6_2 = 'Qx_Allg_Support6_2'

classmethod output (port: int, contact: int)
    Get the NodeID string for a support output.

```

Parameters

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

Returns the node id string**Raises** **ValueError** – when port or contact number is not valid

```

classmethod port_range () → Sequence[int]
    Integer range of all ports.

```

Returns sequence of port numbers

```

class GeneralSupportMeta (clsname, bases, clsdict, **kwargs)
    Bases: aenum.EnumType

```

```

class MeasurementsDividerRatio (value=<object object>, names=None, module=None,
                                type=None, start=1, boundary=None)
    Bases: hvl_ccb.dev.supercube.constants._InputEnumBase

```

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

```
input_1 = '"DB_Measurements"."si_Divider_Ratio_1"'
input_2 = '"DB_Measurements"."si_Divider_Ratio_2"'
input_3 = '"DB_Measurements"."si_Divider_Ratio_3"'
input_4 = '"DB_Measurements"."si_Divider_Ratio_4"'
```

```
class MeasurementsScaledInput (value=<object object>, names=None, module=None, type=None,
                                start=1, boundary=None)
Bases: hvl_ccb.dev.supercube.constants._InputEnumBase
```

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

```
input_1 = '"DB_Measurements"."si_scaled_Voltage_Input_1"'
input_2 = '"DB_Measurements"."si_scaled_Voltage_Input_2"'
input_3 = '"DB_Measurements"."si_scaled_Voltage_Input_3"'
input_4 = '"DB_Measurements"."si_scaled_Voltage_Input_4"'
```

```
class MessageBoard (value=<object object>, names=None, module=None, type=None, start=1, bound-
                    ary=None)
Bases: hvl_ccb.dev.supercube.constants._LineEnumBase
```

Variable NodeID strings for message board lines.

```
line_1 = '"DB_OPC_Connection"."Is_status_Line_1"'
line_10 = '"DB_OPC_Connection"."Is_status_Line_10"'
line_11 = '"DB_OPC_Connection"."Is_status_Line_11"'
line_12 = '"DB_OPC_Connection"."Is_status_Line_12"'
line_13 = '"DB_OPC_Connection"."Is_status_Line_13"'
line_14 = '"DB_OPC_Connection"."Is_status_Line_14"'
line_15 = '"DB_OPC_Connection"."Is_status_Line_15"'
line_2 = '"DB_OPC_Connection"."Is_status_Line_2"'
line_3 = '"DB_OPC_Connection"."Is_status_Line_3"'
line_4 = '"DB_OPC_Connection"."Is_status_Line_4"'
line_5 = '"DB_OPC_Connection"."Is_status_Line_5"'
line_6 = '"DB_OPC_Connection"."Is_status_Line_6"'
line_7 = '"DB_OPC_Connection"."Is_status_Line_7"'
line_8 = '"DB_OPC_Connection"."Is_status_Line_8"'
line_9 = '"DB_OPC_Connection"."Is_status_Line_9"'
```

```
class OpcControl (value=<object object>, names=None, module=None, type=None, start=1, bound-
                  ary=None)
Bases: hvl_ccb.utils.enum.ValueEnum
```

Variable NodeID strings for supervision of the OPC connection from the controlling workstation to the Supercube.

```
active = '"DB_OPC_Connection"."sx OPC_active"'
```

writable boolean to enable OPC remote control and display a message window on the Supercube HMI.

```
live = '"DB_OPC_Connection"."sx_OPC_lifebit"'
```

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

Variable NodeID strings concerning power data.

TODO: these variable NodeIDs are not tested and/or correct yet, they don't exist yet on Supercube side.

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

frequency = 'Ir_Power_FU_Frequency'
 Frequency converter output frequency. (read-only)

setup = 'Qi_Power_Setup'
 Power setup that is configured using the Supercube HMI. The value corresponds to the ones in *PowerSetup*. (read-only)

voltage_max = 'Iw_Power_max_Voltage'
 Maximum voltage allowed by the current experimental setup. (read-only)

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

voltage_slope = 'Ir_Power_dUdt'
 Voltage slope in V/s.

voltage_target = 'Ir_Power_Target_Voltage'
 Target voltage setpoint in V.

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

Possible power setups corresponding to the value of variable *Power.setup*.

AC_DoubleStage_150kV = 4
 AC voltage with two MWB transformers, one at 100kV and the other at 50kV, resulting in a total maximum voltage of 150kV.

AC_DoubleStage_200kV = 5
 AC voltage with two MWB transformers both at 100kV, resulting in a total maximum voltage of 200kV

AC_SingleStage_100kV = 3
 AC voltage with MWB transformer set to 100kV maximum voltage.

AC_SingleStage_50kV = 2
 AC voltage with MWB transformer set to 50kV maximum voltage.

DC_DoubleStage_280kV = 8
 DC voltage with two AC transformers set to 100kV AC each, resulting in 280kV DC in total (or a single stage transformer with Greinacher voltage doubling rectifier)

DC_SingleStage_140kV = 7
 DC voltage with one AC transformer set to 100kV AC, resulting in 140kV DC

External = 1
 External power supply fed through blue CEE32 input using isolation transformer and safety switches of the Supercube, or using an external safety switch attached to the Supercube Type B.

Internal = 6

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

NoPower = 0

No safety switches, use only safety components (doors, fence, earthing...) without any power.

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

status = "DB_Safety_Circuit"."si_safe_status"

Status is a read-only integer containing the state number of the supercube-internal state machine. The values correspond to numbers in *SafetyStatus*.

switch_to_operate = "DB_Safety_Circuit"."sx_safe_switch_to_operate"

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

switch_to_ready = "DB_Safety_Circuit"."sx_safe_switch_to_ready"

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

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

Bases: *aenum.IntEnum*

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

Error = 6

System is in error mode.

GreenNotReady = 1

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

GreenReady = 2

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

Initializing = 0

System is initializing or booting.

QuickStop = 5

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

RedOperate = 4

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

RedReady = 3

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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

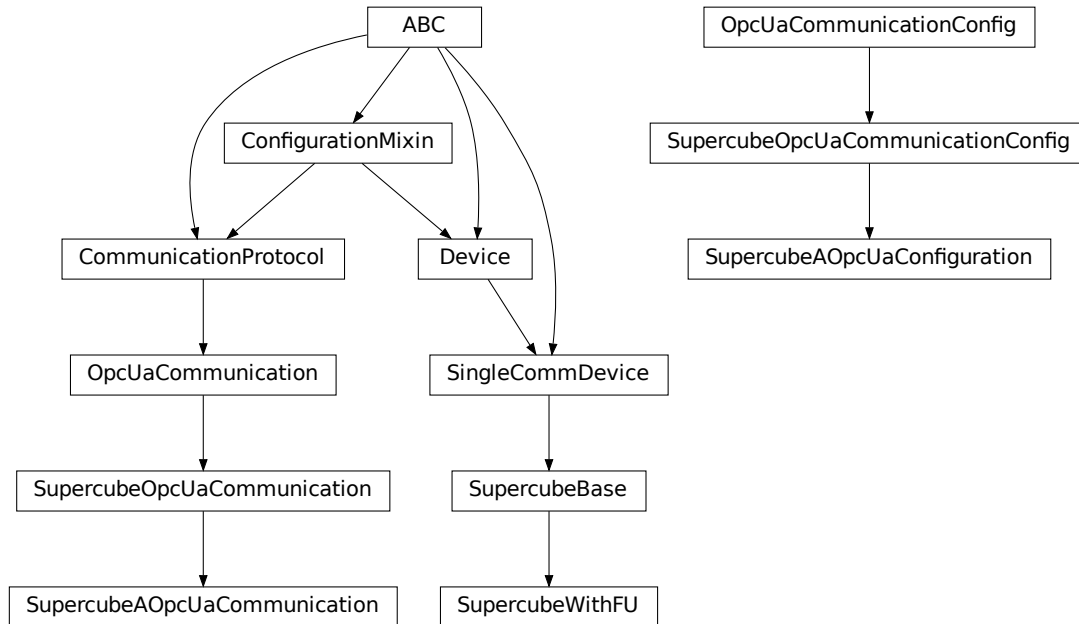
OPC Server Endpoint strings for the supercube variants.

A = 'Supercube Typ A'

B = 'Supercube Typ B'

T13_SOCKET_PORTS = (1, 2, 3)
 Port numbers of SEV T13 power socket

hvl_ccb.dev.supercube.typ_a



Supercube Typ A module.

```

class SupercubeAOpcUaCommunication(config)
    Bases: hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunication

    static config_cls()
        Return the default configdataclass class.

        Returns a reference to the default configdataclass class

class SupercubeAOpcUaConfiguration(host: str, endpoint_name: str = 'Supercube Typ A', port: int = 4840, sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler = <hvl_ccb.dev.supercube.base.SupercubeSubscriptionHandler object at 0x7f37219f9590>, update_period: int = 500, wait_timeout_retry_sec: Union[int, float] = 1, max_timeout_retry_nr: int = 5)
    Bases: hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunicationConfig

    endpoint_name: str = 'Supercube Typ A'
        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.

class SupercubeWithFU (com, dev_config=None)

Bases: *hvl_ccb.dev.supercube.base.SupercubeBase*

Variant A of the Supercube with frequency converter.

static default_com_cls ()

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

Returns the type of the standard communication protocol for this device

fso_reset () → None

TODO: test fso_reset with device

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

get_frequency () → float

TODO: test get_frequency with device

Read the electrical frequency of the current Supercube setup.

Returns the frequency in Hz

get_fso_active () → bool

TODO: test get_fso_active with device

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

Returns state of the FSO functionality

get_max_voltage () → float

TODO: test get_max_voltage with device

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

Returns the maximum voltage of the setup in V.

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

TODO: test get_power_setup with device

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

Returns the power setup

get_primary_current () → float

TODO: get_primary_current with device

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

Returns primary current in A

get_primary_voltage () → float

TODO: test get_primary_voltage with device

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

Returns primary voltage in V

get_target_voltage () → float

TODO: test get_target_voltage with device

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

Returns the setpoint voltage in V.

set_slope (*slope: float*) → None

TODO: test set_slope with device

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

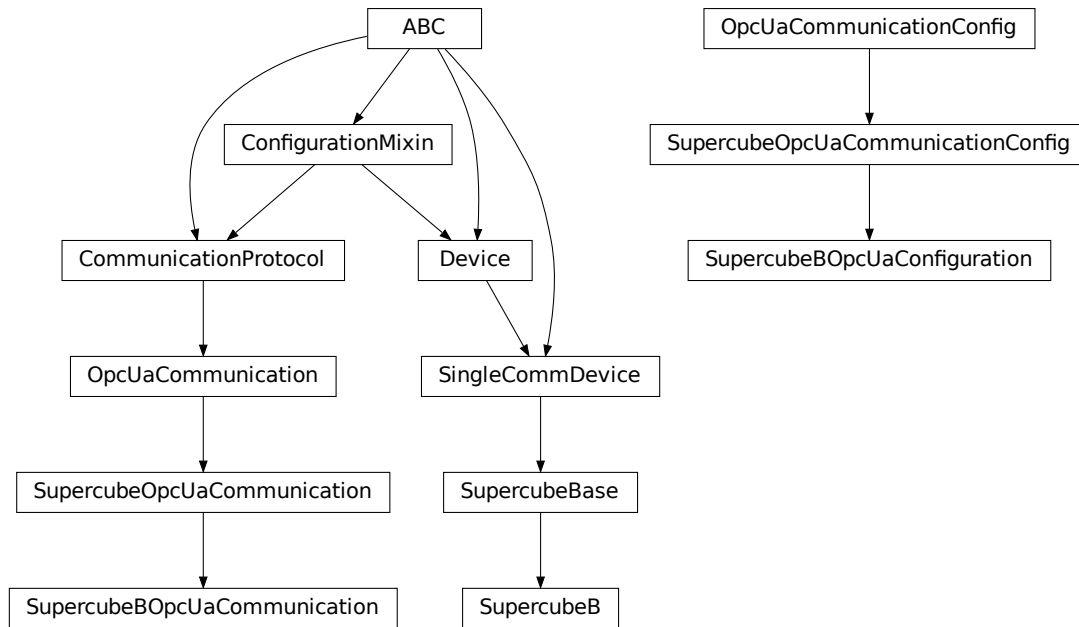
Parameters **slope** – voltage slope in V/s (0..15)

set_target_voltage (*volt_v: float*) → None

TODO: test set_target_voltage with device

Set the output voltage to a defined value in V.

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

hvl_ccb.dev.supercube.typ_b

Supercube Typ B module.

class SupercubeB (*com, dev_config=None*)

Bases: *hvl_ccb.dev.supercube.base.SupercubeBase*

Variant B of the Supercube without frequency converter but external safety switches.

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

class SupercubeBOpcUaCommunication (*config*)

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

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class SupercubeBOpcUaConfiguration (*host: str, endpoint_name: str = 'Supercube Typ B', port: int = 4840, sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler = <hvl_ccb.dev.supercube.base.SupercubeSubscriptionHandler object at 0x7f37219f9590>, update_period: int = 500, wait_timeout_retry_sec: Union[int, float] = 1, max_timeout_retry_nr: int = 5*)

Bases: *hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunicationConfig*

endpoint_name: str = 'Supercube Typ B'

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

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

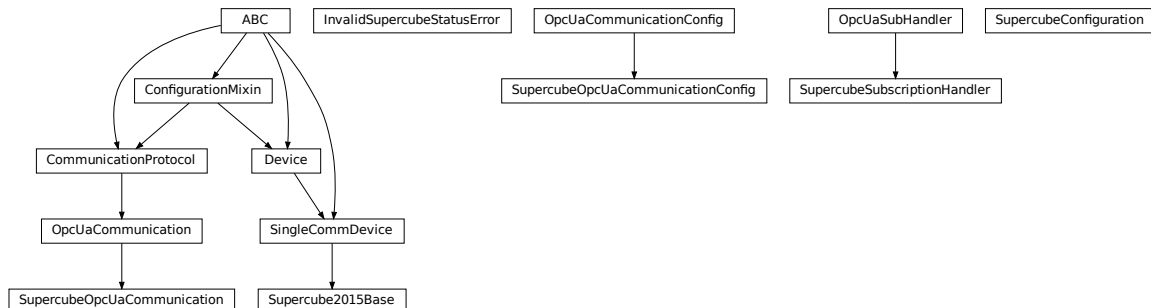
Module contents

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

hvl_ccb.dev.supercube2015

Submodules

hvl_ccb.dev.supercube2015.base



Base classes for the Supercube device.

exception InvalidSupercubeStatusError

Bases: `Exception`

Exception raised when supercube has invalid status.

class Supercube2015Base (*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Base class for Supercube variants.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls ()

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

Returns the type of the standard communication protocol for this device

get_cee16_socket () → bool

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

Returns the on-state of the CEE16 power socket

get_door_status (*door: int*) → `hvl_ccb.dev.supercube2015.constants.DoorStatus`

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

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

Returns the door status

get_earthing_manual (*number: int*) → bool

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

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

Returns earthing stick manual status

get_earthing_status (*number: int*) → int

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

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

Returns earthing stick status; see `constants.EarthingStickStatus`

get_measurement_ratio (*channel: int*) → float

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

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

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

Returns the ratio

get_measurement_voltage (*channel: int*) → float

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

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

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

Returns measured voltage

get_status () → int

Get the safety circuit status of the Supercube.

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

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

Get the state of a support socket input.

Parameters

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

Returns digital input read state

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

Get the state of a support socket output.

Parameters

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

Returns digital output read state

get_t13_socket (port: int) → bool

Read the state of a SEV T13 power socket.

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

Returns on-state of the power socket

horn (state: bool) → None

Turns acoustic horn on or off.

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

operate (state: bool) → None

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

Parameters **state** – set operate state

quit_error () → None

Quits errors that are active on the Supercube.

read (node_id: str)

Local wrapper for the OPC UA communication protocol read method.

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

Returns the value of the variable

ready (state: bool) → None

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

Parameters **state** – set ready state

set_cee16_socket (state: bool) → None

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

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

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

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

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

Parameters

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

set_remote_control (*state: bool*) → None

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

Parameters **state** – desired remote control state

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

Set the state of a support output socket.

Parameters

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

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

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

This function is blocking.

Parameters

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

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

Set the state of a SEV T13 power socket.

Parameters

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

start () → None

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

stop () → None

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

write (*node_id, value*) → None

Local wrapper for the OPC UA communication protocol write method.

Parameters

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

```
class SupercubeConfiguration (namespace_index: int = 7)
```

Bases: `object`

Configuration dataclass for the Supercube devices.

```
clean_values ()
```

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

```
force_value (fieldname, value)
```

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

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

Parameters

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

```
is_configdataclass = True
```

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

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

```
namespace_index: int = 7
```

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

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

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

Returns a list of strings containing all optional keys.

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

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

Returns a list of strings containing all required keys.

```
class SupercubeOpcUaCommunication (config)
```

Bases: `hvl_ccb.comm.opc.OpcUaCommunication`

Communication protocol specification for Supercube devices.

```
static config_cls ()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class SupercubeOpcUaCommunicationConfig (host: str, endpoint_name: str,  
                                           port: int = 4845, sub_handler:  
                                           hvl_ccb.comm.opc.OpcUaSubHandler =  
                                           <hvl_ccb.dev.supercube2015.base.SupercubeSubscriptionHandler  
                                           object>, update_period: int = 500,  
                                           wait_timeout_retry_sec: Union[int, float] =  
                                           1, max_timeout_retry_nr: int = 5)
```

Bases: `hvl_ccb.comm.opc.OpcUaCommunicationConfig`

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

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

port: **int** = **4845**

Port of the OPC UA server to connect to.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

sub_handler: *hvl_ccb.comm.opc.OpcUaSubHandler* = *<hvl_ccb.dev.supercube2015.base.Super*

Subscription handler for data change events

class SupercubeSubscriptionHandler

Bases: *hvl_ccb.comm.opc.OpcUaSubHandler*

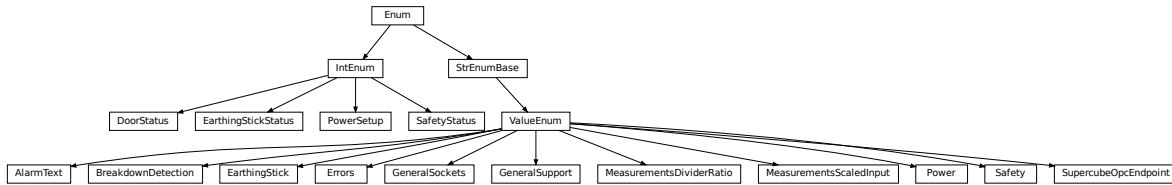
OPC Subscription handler for datachange events and normal events specifically implemented for the Supercube devices.

datachange_notification (*node: opcua.common.node.Node*, *val*, *data*)

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

Parameters

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

hvl_ccb.dev.supercube2015.constants

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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

Alarm0 = 'No Alarm.'

Alarm1 = 'STOP Safety switch 1 error'

Alarm10 = 'STOP Earthing stick 2 error'

Alarm11 = 'STOP Earthing stick 3 error'

Alarm12 = 'STOP Earthing stick 4 error'

Alarm13 = 'STOP Earthing stick 5 error'

Alarm14 = 'STOP Earthing stick 6 error'

Alarm17 = 'STOP Source switch error'

Alarm19 = 'STOP Fence 1 error'

Alarm2 = 'STOP Safety switch 2 error'

Alarm20 = 'STOP Fence 2 error'

Alarm21 = 'STOP Control error'

Alarm22 = 'STOP Power outage'

Alarm3 = 'STOP Emergency Stop 1'

Alarm4 = 'STOP Emergency Stop 2'

Alarm5 = 'STOP Emergency Stop 3'

Alarm6 = 'STOP Door 1 lock supervision'

Alarm7 = 'STOP Door 2 lock supervision'

Alarm8 = 'STOP Door 3 lock supervision'

Alarm9 = 'STOP Earthing stick 1 error'

classmethod get (*alarm: int*)

Get the attribute of this enum for an alarm number.

Parameters **alarm** – the alarm number

Returns the enum for the desired alarm number

```
not_defined = 'NO ALARM TEXT DEFINED'
```

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

Bases: `hvl_ccb.utils.enum.ValueEnum`

Node ID strings for the breakdown detection.

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

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

```
reset = 'hvl-ipc.WINAC.Support6OutA'
```

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

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

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

```
class DoorStatus(value=<object object>, names=None, module=None, type=None, start=1, bound-  
                ary=None)
```

Bases: `aenum.IntEnum`

Possible status values for doors.

```
closed = 2
```

Door is closed, but not locked.

```
error = 4
```

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

```
inactive = 0
```

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

```
locked = 3
```

Door is closed and locked (safe state).

```
open = 1
```

Door is open.

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

Bases: `hvl_ccb.utils.enum.ValueEnum`

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

```
classmethod manual(number: int)
```

Get the manual enum attribute for an earthing stick number.

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

Returns the manual enum

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

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

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

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

```
manual_5 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_5.MANUAL'
```

```
manual_6 = 'hvl-ipc.WINAC.SYSTEM_COMPONENTS.STICK_6.MANUAL'
```



```

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

```

classmethod status_closed (*number: int*)

Get the status enum attribute for an earthing stick number.

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

Returns the status enum

classmethod status_connected (*number: int*)

Get the status enum attribute for an earthing stick number.

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

Returns the status enum

classmethod status_open (*number: int*)

Get the status enum attribute for an earthing stick number.

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

Returns the status enum

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

Bases: `aenum.IntEnum`

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

closed = 1

Earthing is closed (safe).

error = 3

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

inactive = 0

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

open = 2

Earthing is open (not safe).

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

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

Writable boolean for the error quit button.

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

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

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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

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

CEE16 socket (writeable boolean).

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

SEV T13 socket No. 1 (writable boolean).

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

SEV T13 socket No. 2 (writable boolean).

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

SEV T13 socket No. 3 (writable boolean).

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

NodeID strings for the support inputs and outputs.

in_1_1 = 'hvl-ipc.WINAC.Support1InA'

in_1_2 = 'hvl-ipc.WINAC.Support1InB'

in_2_1 = 'hvl-ipc.WINAC.Support2InA'

in_2_2 = 'hvl-ipc.WINAC.Support2InB'

in_3_1 = 'hvl-ipc.WINAC.Support3InA'

in_3_2 = 'hvl-ipc.WINAC.Support3InB'

in_4_1 = 'hvl-ipc.WINAC.Support4InA'

in_4_2 = 'hvl-ipc.WINAC.Support4InB'

in_5_1 = 'hvl-ipc.WINAC.Support5InA'

in_5_2 = 'hvl-ipc.WINAC.Support5InB'

in_6_1 = 'hvl-ipc.WINAC.Support6InA'

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

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

Get the NodeID string for a support input.

Parameters

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

Returns the node id string

```
out_1_1 = 'hvl-ipc.WINAC.Support1OutA'
```

```
out_1_2 = 'hvl-ipc.WINAC.Support1OutB'
```

```
out_2_1 = 'hvl-ipc.WINAC.Support2OutA'
```

```
out_2_2 = 'hvl-ipc.WINAC.Support2OutB'
```

```
out_3_1 = 'hvl-ipc.WINAC.Support3OutA'
```

```
out_3_2 = 'hvl-ipc.WINAC.Support3OutB'
```

```
out_4_1 = 'hvl-ipc.WINAC.Support4OutA'
```

```
out_4_2 = 'hvl-ipc.WINAC.Support4OutB'
```

```
out_5_1 = 'hvl-ipc.WINAC.Support5OutA'
```

```
out_5_2 = 'hvl-ipc.WINAC.Support5OutB'
```

```
out_6_1 = 'hvl-ipc.WINAC.Support6OutA'
```

```
out_6_2 = 'hvl-ipc.WINAC.Support6OutB'
```

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

Get the NodeID string for a support output.

Parameters

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

Returns the node id string

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

Bases: `hvl_ccb.utils.enum.ValueEnum`

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

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

Get the attribute for an input number.

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

Returns the enum for the desired channel.

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

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

Bases: `hvl_ccb.utils.enum.ValueEnum`

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

classmethod **get** (*channel: int*)

Get the attribute for an input number.

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

Returns the enum for the desired channel.

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

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

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

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

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

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

Variable NodeID strings concerning power data.

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

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

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

Frequency converter output frequency. (read-only)

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

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

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

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

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

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

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

Voltage slope in V/s.

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

Target voltage setpoint in V.

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

Bases: [aenum.IntEnum](#)

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

AC_DoubleStage_150kV = 3

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

AC_DoubleStage_200kV = 4

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

AC_SingleStage_100kV = 2

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

AC_SingleStage_50kV = 1

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

DC_DoubleStage_280kV = 7

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

DC_SingleStage_140kV = 6

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

External = 0

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

Internal = 5

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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

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

Writeable boolean to manually turn on or off the horn

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

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

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

Status is a read-only integer containing the state number of the supercube-internal state machine. The values correspond to numbers in *SafetyStatus*.

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

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

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

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

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

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

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

Bases: *aenum.IntEnum*

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

Error = 6

System is in error mode.

GreenNotReady = 1

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

GreenReady = 2

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

Initializing = 0

System is initializing or booting.

QuickStop = 5

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

RedOperate = 4

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

RedReady = 3

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

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

Bases: *hvl_ccb.utils.enum.ValueEnum*

OPC Server Endpoint strings for the supercube variants.

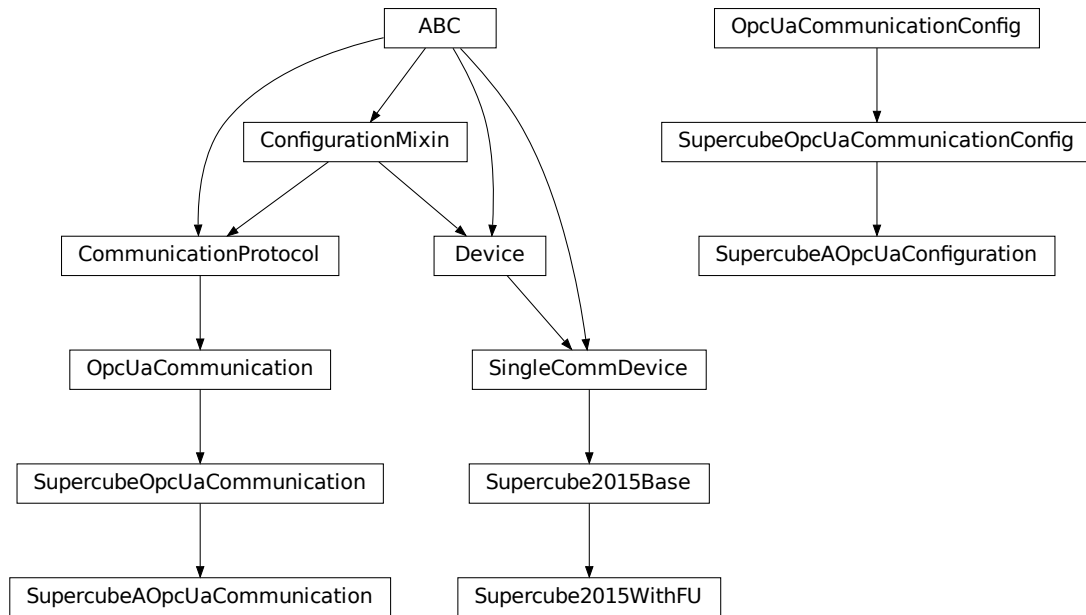
A = 'OPC.SimaticNET.S7'

B = 'OPC.SimaticNET.S7'

T13_SOCKET_PORTS = (1, 2, 3)

Port numbers of SEV T13 power socket

hvl_ccb.dev.supercube2015.typ_a



Supercube Typ A module.

class Supercube2015WithFU (*com, dev_config=None*)

Bases: *hvl_ccb.dev.supercube2015.base.Supercube2015Base*

Variant A of the Supercube with frequency converter.

static default_com_cls()

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

Returns the type of the standard communication protocol for this device

fso_reset () → None

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

get_frequency () → float

Read the electrical frequency of the current Supercube setup.

Returns the frequency in Hz

get_fso_active () → bool

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

Returns state of the FSO functionality

get_max_voltage () → float

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

Returns the maximum voltage of the setup in V.

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

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

Returns the power setup

get_primary_current () → float

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

Returns primary current in A

get_primary_voltage () → float

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

Returns primary voltage in V

get_target_voltage () → float

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

Returns the setpoint voltage in V.

set_slope (slope: float) → None

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

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

set_target_voltage (volt_v: float) → None

Set the output voltage to a defined value in V.

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

class SupercubeAOpcUaCommunication (config)

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

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class SupercubeAOpcUaConfiguration (host: str, endpoint_name: str =
    'OPC.SimaticNET.S7', port: int = 4845,
    sub_handler: hvl_ccb.comm.opc.OpcUaSubHandler =
    <hvl_ccb.dev.supercube2015.base.SupercubeSubscriptionHandler
    object at 0x7f3720d9e4d0>, update_period: int =
    500, wait_timeout_retry_sec: Union[int, float] = 1,
    max_timeout_retry_nr: int = 5)
```

Bases: `hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunicationConfig`

endpoint_name: `str = 'OPC.SimaticNET.S7'`

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

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

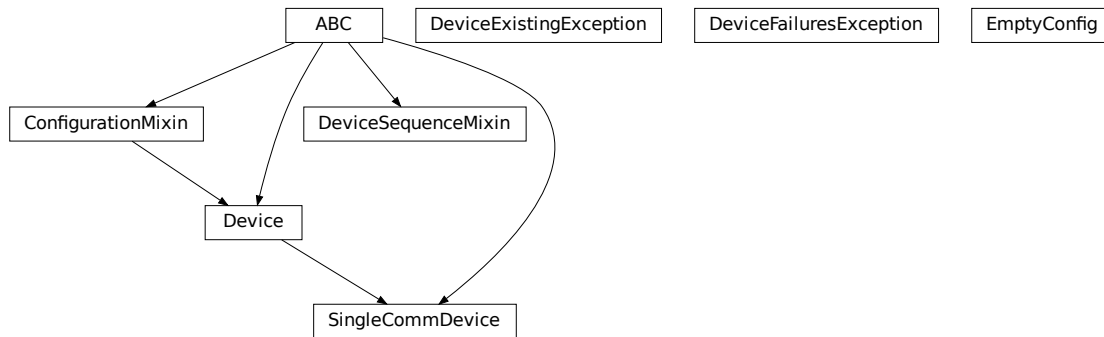
Returns a list of strings containing all required keys.

Module contents

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

Submodules

hvl_ccb.dev.base



Module with base classes for devices.

class Device (*dev_config=None*)

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

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

Specifies the methods to implement for a device.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

abstract start () → None

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

abstract stop () → None

Stop this Device. To be implemented in the subclass.

exception DeviceExistingException

Bases: `Exception`

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

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

Bases: `Exception`

Exception to indicate that one or several devices failed.

failures: `Dict[str, Exception]`

A dictionary of named devices failures (exceptions).

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

Bases: `abc.ABC`

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

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

Add a new device to the device sequence.

Parameters

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

Raises *DeviceExistingException* –

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

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

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

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

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

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

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

Get a device by name.

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

Returns the device object from this sequence.

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

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

Returns A list of tuples with name and device each.

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

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

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

Returns device object or *None* if such device was not in the sequence.

Raises **ValueError** – when device with given name was not found

start () → None

Start all devices in this sequence in their added order.

Raises *DeviceFailuresException* – if one or several devices failed to start

stop () → None

Stop all devices in this sequence in their reverse order.

Raises *DeviceFailuresException* – if one or several devices failed to stop

class **EmptyConfig**

Bases: object

Empty configuration dataclass that is the default configuration for a Device.

clean_values ()

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

force_value (*fieldname, value*)

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

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

Parameters

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

is_configdataclass = **True**

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

class SingleCommDevice (*com, dev_config=None*)

Bases: *hvl_ccb.dev.base.Device*, *abc.ABC*

Base class for devices with a single communication protocol.

property com

Get the communication protocol of this device.

Returns an instance of CommunicationProtocol subtype

abstract static default_com_cls () → Type[*hvl_ccb.comm.base.CommunicationProtocol*]

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

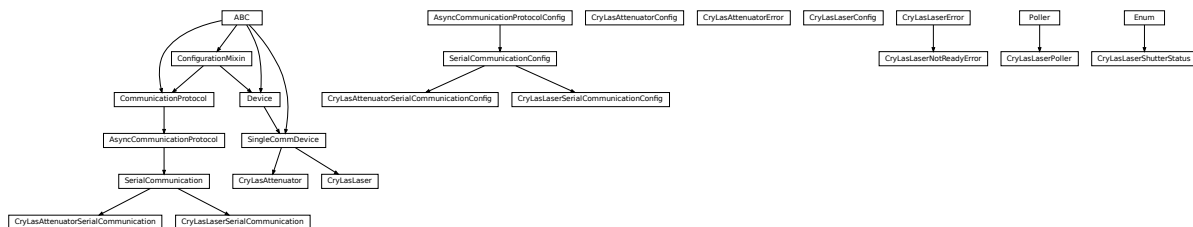
Returns the type of the standard communication protocol for this device

start () → None

Open the associated communication protocol.

stop () → None

Close the associated communication protocol.

hvl_ccb.dev.crylas

Device classes for a CryLas pulsed laser controller and a CryLas laser attenuator, using serial communication.

There are three modes of operation for the laser 1. Laser-internal hardware trigger (default): fixed to 20 Hz and max energy per pulse. 2. Laser-internal software trigger (for diagnosis only). 3. External trigger: required for arbitrary pulse energy or repetition rate. Switch to “external” on the front panel of laser controller for using option 3.

After switching on the laser with `laser_on()`, the system must stabilize for some minutes. Do not apply abrupt changes of pulse energy or repetition rate.

Manufacturer homepage: https://www.crylas.de/products/pulsed_laser.html

class CryLasAttenuator (*com, dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Device class for the CryLas laser attenuator.

property attenuation

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

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: Exception

General error with the CryLas Attenuator.

```
class CryLasAttenuatorSerialCommunication (configuration)
```

Bases: *hvl_ccb.comm.serial.SerialCommunication*

Specific communication protocol implementation for the CryLas attenuator. Already predefines device-specific protocol parameters in config.

```
static config_cls ()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class CryLasAttenuatorSerialCommunicationConfig (terminator: bytes = b'', encoding: str = 'utf-8', encoding_error_handling: str = '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 = 9600, parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout: Union[int, float] = 3)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

```
baudrate: int = 9600
```

Baudrate for CryLas attenuator is 9600 baud

bytesize: Union[int, *hvl_ccb.comm.serial.SerialCommunicationBytesize*] = 8
One byte is eight bits long

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: Union[str, *hvl_ccb.comm.serial.SerialCommunicationParity*] = 'N'

CryLas attenuator does not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: Union[int, *hvl_ccb.comm.serial.SerialCommunicationStopbits*] = 1

CryLas attenuator uses one stop bit

terminator: bytes = b''

No terminator

timeout: Union[int, float] = 3

use 3 seconds timeout as default

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

Bases: *hvl_ccb.dev.base.SingleCommDevice*

CryLas laser controller device class.

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

Bases: *aenum.Enum*

Standard answers of the CryLas laser controller to 'Shutter' command passed via *com*.

CLOSED = 'Shutter inaktiv'

OPENED = 'Shutter aktiv'

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

Bases: *aenum.Enum*

Standard answers of the CryLas laser controller to 'STATUS' command passed via *com*.

ACTIVE = 'STATUS: Laser active'

```

HEAD = 'STATUS: Head ok'
INACTIVE = 'STATUS: Laser inactive'
READY = 'STATUS: System ready'
TEC1 = 'STATUS: TEC1 Regulation ok'
TEC2 = 'STATUS: TEC2 Regulation ok'

class LaserStatus (value=<object object>, names=None, module=None, type=None, start=1,
                    boundary=None)
    Bases: aenum.Enum
    Status of the CryLas laser
    READY_ACTIVE = 2
    READY_INACTIVE = 1
    UNREADY_INACTIVE = 0
    property is_inactive
    property is_ready

class RepetitionRates (value=<object object>, names=None, module=None, type=None,
                        start=1, boundary=None)
    Bases: aenum.IntEnum
    Repetition rates for the internal software trigger in Hz
    HARDWARE = 0
    SOFTWARE_INTERNAL_SIXTY = 60
    SOFTWARE_INTERNAL_TEN = 10
    SOFTWARE_INTERNAL_TWENTY = 20

ShutterStatus
    alias of hvl_ccb.dev.crylas.CryLasLaserShutterStatus

close_shutter () → None
    Close the laser shutter.

    Raises
        • SerialCommunicationIOError – when communication port is not opened
        • CryLasLaserError – if success is not confirmed by the device

static config_cls ()
    Return the default configdataclass class.

    Returns a reference to the default configdataclass class

static default_com_cls ()
    Get the class for the default communication protocol used with this device.

    Returns the type of the standard communication protocol for this device

get_pulse_energy_and_rate () → Tuple[int, int]
    Use the debug mode, return the measured pulse energy and rate.

    Returns (energy in micro joule, rate in Hz)

    Raises

```

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if the device does not answer the query

laser_off () → None

Turn the laser off.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

laser_on () → None

Turn the laser on.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserNotReadyError* – if the laser is not ready to be turned on
- *CryLasLaserError* – if success is not confirmed by the device

open_shutter () → None

Open the laser shutter.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

set_init_shutter_status () → None

Open or close the shutter, to match the configured shutter_status.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

set_pulse_energy (energy: int) → None

Sets the energy of pulses (works only with external hardware trigger). Proceed with small energy steps, or the regulation may fail.

Parameters **energy** – energy in micro joule

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if the device does not confirm success

set_repetition_rate (rate: Union[int, hvl_ccb.dev.crylas.CryLasLaser.RepetitionRates]) → None

Sets the repetition rate of the internal software trigger.

Parameters **rate** – frequency (Hz) as an integer

Raises

- **ValueError** – if rate is not an accepted value in RepetitionRates Enum
- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

start () → None

Opens the communication protocol and configures the device.

Raises *SerialCommunicationIOError* – when communication port cannot be opened

stop () → None

Stops the device and closes the communication protocol.

Raises

- *SerialCommunicationIOError* – if com port is closed unexpectedly
- *CryLasLaserError* – if laser_off() or close_shutter() fail

property target_pulse_energy

update_laser_status () → None

Update the laser status to *LaserStatus.NOT_READY* or *LaserStatus.INACTIVE* or *LaserStatus.ACTIVE*.

Note: laser never explicitly says that it is not ready (*LaserStatus.NOT_READY*) in response to ‘STATUS’ command. It only says that it is ready (heated-up and implicitly inactive/off) or active (on). If it’s not either of these then the answer is *Answers.HEAD*. Moreover, the only time the laser explicitly says that its status is inactive (*Answers.INACTIVE*) is after issuing a ‘LASER OFF’ command.

Raises *SerialCommunicationIOError* – when communication port is not opened

update_repetition_rate () → None

Query the laser repetition rate.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

update_shutter_status () → None

Update the shutter status (OPENED or CLOSED)

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

update_target_pulse_energy () → None

Query the laser pulse energy.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *CryLasLaserError* – if success is not confirmed by the device

wait_until_ready () → None

Block execution until the laser is ready

Raises *CryLasLaserError* – if the polling thread stops before the laser is ready

```
class CryLasLaserConfig (calibration_factor: Union[int, float] = 4.35, polling_period:
                        Union[int, float] = 12, polling_timeout: Union[int, float] =
                        300, auto_laser_on: bool = True, init_shutter_status: Union[int,
                        hvl_ccb.dev.crylas.CryLasLaserShutterStatus] = <CryLasLaserShutterSta-
                        tus.CLOSED: 0>)
```

Bases: object

Device configuration dataclass for the CryLas laser controller.

ShutterStatus

alias of `hvl_ccb.dev.crylas.CryLasLaserShutterStatus`

auto_laser_on: `bool = True`

calibration_factor: `Union[int, float] = 4.35`

clean_values()

force_value (*fieldname, value*)

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

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

Parameters

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

init_shutter_status: `Union[int, hvl_ccb.dev.crylas.CryLasLaserShutterStatus] = 0`

is_configdataclass = `True`

classmethod keys() → `Sequence[str]`

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

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

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

Returns a list of strings containing all optional keys.

polling_period: `Union[int, float] = 12`

polling_timeout: `Union[int, float] = 300`

classmethod required_keys() → `Sequence[str]`

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

Returns a list of strings containing all required keys.

exception CryLasLaserError

Bases: `Exception`

General error with the CryLas Laser.

exception CryLasLaserNotReadyError

Bases: `hvl_ccb.dev.crylas.CryLasLaserError`

Error when trying to turn on the CryLas Laser before it is ready.

class CryLasLaserPoller (*spoll_handler: Callable, check_handler: Callable, check_laser_status_handler: Callable, polling_delay_sec: Union[int, float] = 0, polling_interval_sec: Union[int, float] = 1, polling_timeout_sec: Optional[Union[int, float]] = None*)

Bases: `hvl_ccb.dev.utils.Poller`

Poller class for polling the laser status until the laser is ready.

Raises

- `CryLasLaserError` – if the timeout is reached before the laser is ready

- **`SerialCommunicationIOError`** – when communication port is closed.

class `CryLasLaserSerialCommunication` (*configuration*)

Bases: `hvl_ccb.comm.serial.SerialCommunication`

Specific communication protocol implementation for the CryLas laser controller. Already predefines device-specific protocol parameters in config.

READ_TEXT_SKIP_PREFIXES = ('>', 'MODE:')

Prefixes of lines that are skipped when read from the serial port.

static `config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query (*cmd: str, prefix: str, post_cmd: Optional[str] = None*) → str

Send a command, then read the com until a line starting with prefix, or an empty line, is found. Returns the line in question.

Parameters

- **cmd** – query message to send to the device
- **prefix** – start of the line to look for in the device answer
- **post_cmd** – optional additional command to send after the query

Returns line in question as a string

Raises **`SerialCommunicationIOError`** – when communication port is not opened

query_all (*cmd: str, prefix: str*)

Send a command, then read the com until a line starting with prefix, or an empty line, is found. Returns a list of successive lines starting with prefix.

Parameters

- **cmd** – query message to send to the device
- **prefix** – start of the line to look for in the device answer

Returns line in question as a string

Raises **`SerialCommunicationIOError`** – when communication port is not opened

read () → str

Read first line of text from the serial port that does not start with any of `self.READ_TEXT_SKIP_PREFIXES`.

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

Raises **`SerialCommunicationIOError`** – when communication port is not opened

```
class CryLasLaserSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str = 'utf-8', encoding_error_handling: str = '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 = 19200, parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout: Union[int, float] = 10)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: `int = 19200`

Baudrate for CryLas laser is 19200 baud

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

One byte is eight bits long

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

CryLas laser does not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

CryLas laser uses one stop bit

terminator: `bytes = b'\n'`

The terminator is LF

```
timeout: Union[int, float] = 10
    use 10 seconds timeout as default (a long timeout is needed!)
```

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

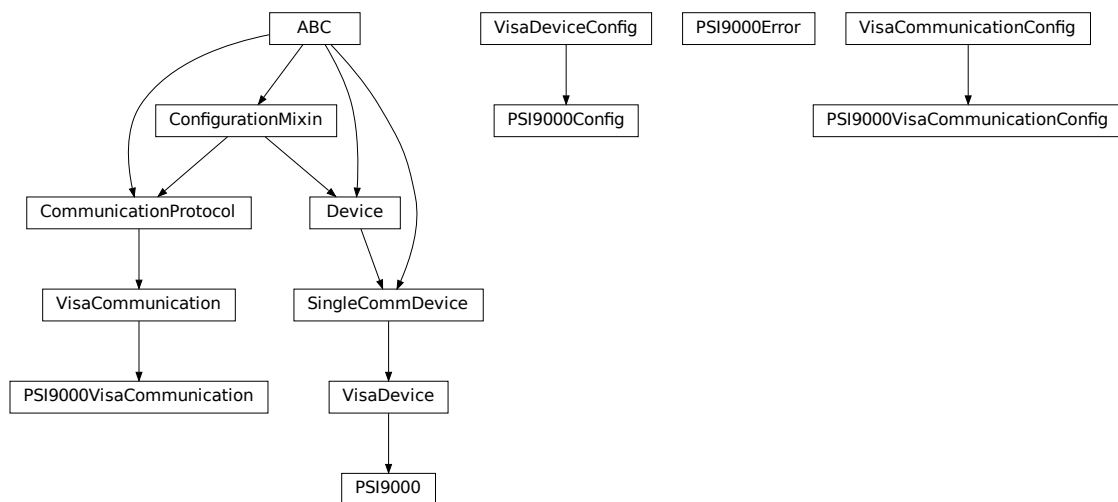
Bases: `aenum.Enum`

Status of the CryLas laser shutter

CLOSED = 0

OPENED = 1

hvl_ccb.dev.ea_psi9000



Device class for controlling a Elektro Automatik PSI 9000 power supply over VISA.

It is necessary that a backend for pyvisa is installed. This can be NI-Visa oder pyvisa-py (up to know, all the testing was done with NI-Visa)

```
class PSI9000 (com: Union[hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunication,
                        hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig, dict], dev_config: Op-
                        tional[Union[hvl_ccb.dev.ea_psi9000.PSI9000Config, dict]] = None)
```

Bases: `hvl_ccb.dev.visa.VisaDevice`

Elektro Automatik PSI 9000 power supply.

MS_NOMINAL_CURRENT = 2040

MS_NOMINAL_VOLTAGE = 80

SHUTDOWN_CURRENT_LIMIT = 0.1

SHUTDOWN_VOLTAGE_LIMIT = 0.1

check_master_slave_config() → None

Checks if the master / slave configuration and initializes if successful

Raises **PSI9000Error** – if master-slave configuration failed

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls()

Return the default communication protocol for this device type, which is VisaCommunication.

Returns the VisaCommunication class

get_output() → bool

Reads the current state of the DC output of the source. Returns True, if it is enabled, false otherwise.

Returns the state of the DC output

get_system_lock() → bool

Get the current lock state of the system. The lock state is true, if the remote control is active and false, if not.

Returns the current lock state of the device

get_ui_lower_limits() → Tuple[float, float]

Get the lower voltage and current limits. A lower power limit does not exist.

Returns Umin in V, Imin in A

get_uip_upper_limits() → Tuple[float, float, float]

Get the upper voltage, current and power limits.

Returns Umax in V, Imax in A, Pmax in W

get_voltage_current_setpoint() → Tuple[float, float]

Get the voltage and current setpoint of the current source.

Returns Uset in V, Iset in A

measure_voltage_current() → Tuple[float, float]

Measure the DC output voltage and current

Returns Umeas in V, Imeas in A

set_lower_limits (*voltage_limit: Optional[float] = None, current_limit: Optional[float] = None*)
→ None

Set the lower limits for voltage and current. After writing the values a check is performed if the values are set correctly.

Parameters

- **voltage_limit** – is the lower voltage limit in V
- **current_limit** – is the lower current limit in A

Raises *PSI9000Error* – if the limits are out of range

set_output (*target_onstate: bool*) → None

Enables / disables the DC output.

Parameters **target_onstate** – enable or disable the output power

Raises *PSI9000Error* – if operation was not successful

set_system_lock (*lock: bool*) → None

Lock / unlock the device, after locking the control is limited to this class unlocking only possible when voltage and current are below the defined limits

Parameters **lock** – True: locking, False: unlocking

set_upper_limits (*voltage_limit: Optional[float] = None, current_limit: Optional[float] = None, power_limit: Optional[float] = None*) → None

Set the upper limits for voltage, current and power. After writing the values a check is performed if the values are set. If a parameter is left blank, the maximum configurable limit is set.

Parameters

- **voltage_limit** – is the voltage limit in V
- **current_limit** – is the current limit in A
- **power_limit** – is the power limit in W

Raises **PSI9000Error** – if limits are out of range

set_voltage_current (*volt: float, current: float*) → None

Set voltage and current setpoints.

After setting voltage and current, a check is performed if writing was successful.

Parameters

- **volt** – is the setpoint voltage: 0..81.6 V (1.02 * 0-80 V) (absolute max, can be smaller if limits are set)
- **current** – is the setpoint current: 0..2080.8 A (1.02 * 0 - 2040 A) (absolute max, can be smaller if limits are set)

Raises **PSI9000Error** – if the desired setpoint is out of limits

start () → None

Start this device.

stop () → None

Stop this device. Turns off output and lock, if enabled.

class PSI9000Config (*spoll_interval: Union[int, float] = 0.5, spoll_start_delay: Union[int, float] = 2, power_limit: Union[int, float] = 43500, voltage_lower_limit: Union[int, float] = 0.0, voltage_upper_limit: Union[int, float] = 10.0, current_lower_limit: Union[int, float] = 0.0, current_upper_limit: Union[int, float] = 2040.0, wait_sec_system_lock: Union[int, float] = 0.5, wait_sec_settings_effect: Union[int, float] = 1, wait_sec_initialisation: Union[int, float] = 2*)

Bases: `hvl_ccb.dev.visa.VisaDeviceConfig`

Elektro Automatik PSI 9000 power supply device class. The device is communicating over a VISA TCP socket.

Using this power supply, DC voltage and current can be supplied to a load with up to 2040 A and 80 V (using all four available units in parallel). The maximum power is limited by the grid, being at 43.5 kW available through the CEE63 power socket.

clean_values () → None

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

current_lower_limit: Union[int, float] = 0.0

Lower current limit in A, depending on the experimental setup.

current_upper_limit: Union[int, float] = 2040.0

Upper current limit in A, depending on the experimental setup.

force_value (*fieldname, value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

power_limit: Union[int, float] = 43500

Power limit in W depending on the experimental setup. With 3x63A, this is 43.5kW. Do not change this value, if you do not know what you are doing. There is no lower power limit.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

voltage_lower_limit: Union[int, float] = 0.0

Lower voltage limit in V, depending on the experimental setup.

voltage_upper_limit: Union[int, float] = 10.0

Upper voltage limit in V, depending on the experimental setup.

wait_sec_initialisation: Union[int, float] = 2

wait_sec_settings_effect: Union[int, float] = 1

wait_sec_system_lock: Union[int, float] = 0.5

exception PSI9000Error

Bases: Exception

Base error class regarding problems with the PSI 9000 supply.

class PSI9000VisaCommunication (*configuration*)

Bases: *hvl_ccb.comm.visa.VisaCommunication*

Communication protocol used with the PSI 9000 power supply.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class PSI9000VisaCommunicationConfig (*host: str, interface_type: Union[str, hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType]*
= <InterfaceType.TCPIP_SOCKET: 1>, *board: int = 0,*
port: int = 5025, timeout: int = 5000, chunk_size: int =
204800, open_timeout: int = 1000, write_termination:
str = '\n', read_termination: str = '\n', visa_backend:
str = '')

Bases: *hvl_ccb.comm.visa.VisaCommunicationConfig*

Visa communication protocol config dataclass with specification for the PSI 9000 power supply.

force_value (*fieldname*, *value*)

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

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

Parameters

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

interface_type: Union[str, [hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType](#)] =
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.

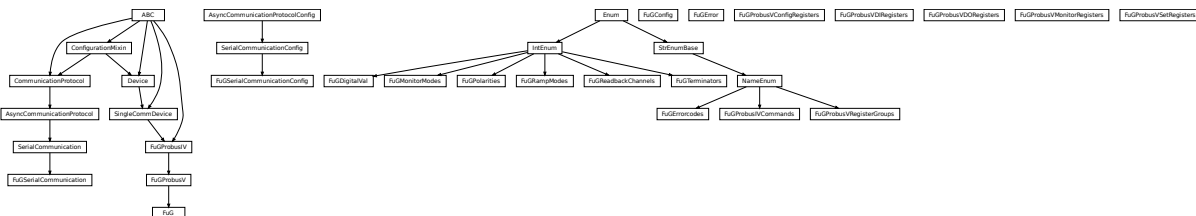
classmethod optional_defaults () → Dict[str, object]
Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified on instantiation.

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.fug



Device classes for “Probus V - ADDAT30” Interfaces which are used to control power supplies from FuG Elektronik GmbH

This interface is used for many FuG power units. Manufacturer homepage: <https://www.fug-elektronik.de>

The Professional Series of Power Supplies from FuG is a series of low, medium and high voltage direct current power supplies as well as capacitor chargers. The class FuG is tested with a HCK 800-20 000 in Standard Mode. The addressable mode is not implemented. Check the code carefully before using it with other devices. Manufacturer homepage: <https://www.fug-elektronik.de/netzgeraete/professional-series/>

The documentation of the interface from the manufacturer can be found here: https://www.fug-elektronik.de/wp-content/uploads/download/de/SOFTWARE/Probus_V.zip

The provided classes support the basic and some advanced commands. The commands for calibrating the power supplies are not implemented, as they are only for very special porpoises and should not be used by “normal” customers.

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

Bases: *hvl_ccb.dev.fug.FuGProbusV*

FuG power supply device class.

The power supply is controlled over a FuG ADDA Interface with the PROBUS V protocol

property **config_status**

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

Returns *FuGProbusVConfigRegisters*

property **current**

Returns the registers for the current output

Returns

property **current_monitor**

Returns the registers for the current monitor.

A typically usage will be “self.current_monitor.value” to measure the output current

Returns

property **di**

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**

Returns the maximal current which could provided within the test setup

Returns

property **max_current_hardware**

Returns the maximal current which could provided with the power supply

Returns

property **max_voltage**

Returns the maximal voltage which could provided within the test setup

Returns

property **max_voltage_hardware**

Returns the maximal voltage which could provided with the power supply

Returns

property **on**

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

Returns *FuGProbusVDORegisters*

property **outX0**

Returns the registers for the digital output X0

Returns *FuGProbusVDORegisters*

property **outX1**

Returns the registers for the digital output X1

Returns FuGProbusVDORegisters

property outX2

Returns the registers for the digital output X2

Returns FuGProbusVDORegisters

property outXCMD

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

Returns the registers for the voltage output

Returns

property voltage_monitor

Returns the registers for the voltage monitor.

A typically usage will be “self.voltage_monitor.value” to measure the output voltage

Returns

class FuGConfig (*wait_sec_stop_commands: Union[int, float] = 0.5*)

Bases: object

Device configuration dataclass for FuG power supplies.

clean_values ()

force_value (*fieldname, value*)

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

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

Parameters

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

is_configdataclass = True

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

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

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

Returns a list of strings containing all required keys.

wait_sec_stop_commands: Union[int, float] = 0.5

Time to wait after subsequent commands during stop (in seconds)

class **FuGDigitalVal** (*value*)

Bases: enum.IntEnum

An enumeration.

NO = 0

OFF = 0

ON = 1

YES = 1

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

Bases: Exception

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=<object object>, names=None, module=None, type=None, start=1, boundary=None*)

Bases: *hvl_ccb.utils.enum.NameEnum*

The power supply can return an errorcode. These errorcodes are handled by this class. The original errorcodes from the source are with one or two digits, see documentation of Probus V chapter 5. All three-digit errorcodes are from this python module.

E0 = ('no error', 'standard response on each command')

E1 = ('no data available', 'Customer tried to read from GPIB but there were no data pr

E10 = ('unknown SCPI command', 'This SCPI command is not implemented')

E100 = ('Command is not implemented', 'You tried to execute a command, which is not imp

E106 = ('The rampstate is a read-only register', 'You tried to write data to the regis

E11 = ('not allowed Trigger-on-Talk', 'Not allowed attempt to Trigger-on-Talk (~T1) wh

E115 = ('The given index to select a digital value is out of range', 'Only integer val

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 1

E13 = ('invalid N-value', 'Register > K8 contained an invalid value. Error code is outp

E135 = ('The given index to select the readback channel is out of range', 'Only intege

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-c

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 o

```
E16 = ('wrong checksum', 'checksum over command string was not correct, refer also to  
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  
E306 = ('The monitor register is read-only', 'You tried to write data to a monitor, wh  
E4 = ('invalid argument', 'The argument of the command was rejected .i.e. malformed num  
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 overbu  
E6 = ('register is read only', 'Some registers can only be read but not written to. (i  
E666 = ('You cannot overwrite the most recent error in the interface of the power supp  
E7 = ('Receive Overflow', 'Command string was longer than 50 characters.')  
E8 = ('EEPROM is write protected', 'Write attempt to calibration data while the write p  
E9 = ('address error', 'A non addressed command was sent to ADDA while it was in addre  
raise_()
```

```
class FuGMonitorModes(value)
```

```
    Bases: enum.IntEnum
```

```
    An enumeration.
```

```
    T1MS = 1
```

```
        15 bit + sign, 1 ms integration time
```

```
    T200MS = 6
```

```
        typ. 19 bit + sign, 200 ms integration time
```

```
    T20MS = 3
```

```
        17 bit + sign, 20 ms integration time
```

```
    T256US = 0
```

```
        14 bit + sign, 256 us integration time
```

```
    T40MS = 4
```

```
        17 bit + sign, 40 ms integration time
```

```
    T4MS = 2
```

```
        15 bit + sign, 4 ms integration time
```

```
    T800MS = 7
```

```
        typ. 20 bit + sign, 800 ms integration time
```

```
    T80MS = 5
```

```
        typ. 18 bit + sign, 80 ms integration time
```

```
class FuGPolarities(value)
```

```
    Bases: enum.IntEnum
```

```
    An enumeration.
```

```
    NEGATIVE = 1
```

```
    POSITIVE = 0
```

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

Bases: `hvl_ccb.dev.base.SingleCommDevice`, `abc.ABC`

FuG Probus IV device class

Sends basic SCPI commands and reads the answer. Only the special commands and PROBUS IV instruction set is implemented.

```
command (command: hvl_ccb.dev.fug.FuGProbusIVCommands, value=None) → str
```

Parameters

- **command** – one of the commands given within `FuGProbusIVCommands`
- **value** – an optional value, depending on the command

Returns a String if a query was performed

```
static config_cls ()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
static default_com_cls ()
```

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

Returns the type of the standard communication protocol for this device

```
output_off () → None
```

Switch DC voltage output off.

```
reset () → None
```

Reset of the interface: All setvalues are set to zero

```
abstract start ()
```

Open the associated communication protocol.

```
stop () → None
```

Close the associated communication protocol.

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

Bases: `hvl_ccb.utils.enum.NameEnum`

An enumeration.

```
ADMODE = ('S', (<enum 'FuGMonitorModes'>, <class 'int'>))
```

```
CURRENT = ('I', (<class 'int'>, <class 'float'>))
```

```
EXECUTE = ('X', None)
```

```
EXECUTEONX = ('G', (<enum 'FuGDigitalVal'>, <class 'int'>))
```

Wait for “X” to execute pending commands

```
ID = ('*IDN?', None)
```

```
OUTPUT = ('F', (<enum 'FuGDigitalVal'>, <class 'int'>))
```

```
POLARITY = ('P', (<enum 'FuGPolarities'>, <class 'int'>))
```

```
QUERY = ('?', None)
```

```
READBACKCHANNEL = ('N', (<enum 'FuGReadbackChannels'>, <class 'int'>))
```

```
RESET = ('=', None)
```

```
TERMINATOR = ('Y', (<enum 'FuGTerminators'>, <class 'int'>))
```

```
VOLTAGE = ('U', (<class 'int'>, <class 'float'>))
```

```
XOUTPUTS = ('R', <class 'int'>)
```

TODO: the possible values are limited to 0..13

```
class FuGProbusV(com, dev_config=None)
```

Bases: `hvl_ccb.dev.fug.FuGProbusIV`

FuG Probus V class which uses register based commands to control the power supplies

```
get_register(register: str) → str
```

get the value from a register

Parameters `register` – the register from which the value is requested

Returns the value of the register as a String

```
set_register(register: str, value: Union[int, float, str]) → None
```

generic method to set value to register

Parameters

- **register** – the name of the register to set the value
- **value** – which should be written to the register

```
class FuGProbusVConfigRegisters(fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups)
```

Bases: `object`

Configuration and Status values, acc. 4.2.5

```
property execute_on_x
```

status of Execute-on-X

Returns `FuGDigitalVal` of the status

```
property most_recent_error
```

Reads the Error-Code of the most recent command

Return `FuGError`

Raises `FuGError` – if code is not “E0”

```
property readback_data
```

Preselection of readout data for Trigger-on-Talk

Returns index for the readback channel

```
property srq_mask
```

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
```

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
```

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

Terminator character for answer strings from ADDA

Returns FuGTerminators

class FuGProbusVDIRegisters (*fug, super_register: [hvl_ccb.dev.fug.FuGProbusVRegisterGroups](#)*)

Bases: object

Digital Inputs acc. 4.2.4

property analog_control

Returns shows 1 if power supply is controlled by the analog interface

property calibration_mode

Returns shows 1 if power supply is in calibration mode

property cc_mode

Returns shows 1 if power supply is in CC mode

property cv_mode

Returns shows 1 if power supply is in CV mode

property digital_control

Returns shows 1 if power supply is digitally controlled

property on

Returns shows 1 if power supply ON

property reg_3

For special applications.

Returns input from bit 3-REG

property x_stat

Returns polarity of HVPS with polarity reversal

class FuGProbusVDORegisters (*fug, super_register: [hvl_ccb.dev.fug.FuGProbusVRegisterGroups](#)*)

Bases: object

Digital outputs acc. 4.2.2

property out

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

Returns the actual value of output. This can differ from the set value if pulse function is used.

Returns FuGDigitalVal

class FuGProbusVMonitorRegisters (*fug, super_register: [hvl_ccb.dev.fug.FuGProbusVRegisterGroups](#)*)

Bases: object

Analog monitors acc. 4.2.3

property adc_mode

The programmed resolution and integration time of the AD converter

Returns FuGMonitorModes

property value

Value from the monitor.

Returns a float value in V or A

property value_raw

uncalibrated raw value from AD converter

Returns float value from ADC

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

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

CONFIG = 'K'

INPUT = 'D'

MONITOR_I = 'M1'

MONITOR_V = 'M0'

OUTPUTONCMD = 'BON'

OUTPUTX0 = 'B0'

OUTPUTX1 = 'B1'

OUTPUTX2 = 'B2'

OUTPUTXCMD = 'BX'

SETCURRENT = 'S1'

SETVOLTAGE = 'S0'

```
class FuGProbusVSetRegisters (fug, super_register: hvl_ccb.dev.fug.FuGProbusVRegisterGroups)
```

Bases: *object*

Setvalue control acc. 4.2.1 for the voltage and the current output

property actualsetvalue

The actual valid set value, which depends on the ramp function.

Returns actual valid set value

property high_resolution

Status of the high resolution mode of the output.

Return 0 normal operation

Return 1 High Res. Mode

property rampmode

The set ramp mode to control the setvalue.

Returns the mode of the ramp as instance of FuGRampModes

property ramprate

The set ramp rate in V/s.

Returns ramp rate in V/s

property rampstate

Status of ramp function.

Return 0 if final setvalue is reached

Return 1 if still ramping up

property setvalue

For the voltage or current output this setvalue was programmed.

Returns the programmed setvalue

class FuGRampModes (*value*)

Bases: `enum.IntEnum`

An enumeration.

FOLLOWRAMP = 1

Follow the ramp up- and downwards

IMMEDIATELY = 0

Standard mode: no ramp

ONLYUPWARDSOFFTOZERO = 4

Follow the ramp up- and downwards, if output is OFF set value is zero

RAMPUPWARDS = 2

Follow the ramp only upwards, downwards immediately

SPECIALRAMPUPWARDS = 3

Follow a special ramp function only upwards

class FuGReadbackChannels (*value*)

Bases: `enum.IntEnum`

An enumeration.

CURRENT = 1

FIRMWARE = 5

RATEDCURRENT = 4

RATEDVOLTAGE = 3

SN = 6

STATUSBYTE = 2

VOLTAGE = 0

class FuGSerialCommunication (*configuration*)

Bases: `hvl_ccb.comm.serial.SerialCommunication`

Specific communication protocol implementation for FuG power supplies. Already predefines device-specific protocol parameters in config.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query (*command: str*) → str

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

Parameters **command** – Command to send

Raises *FuGError* – if the connection is broken or the error from the power source itself

Returns Answer from the interface or empty string

```
class FuGSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str = 'utf-
8', encoding_error_handling: str = '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 = 9600, parity: Union[str,
hvl_ccb.comm.serial.SerialCommunicationParity]
= <SerialCommunicationPar-
ity.NONE: 'N'>, stopbits: Union[int,
hvl_ccb.comm.serial.SerialCommunicationStopbits]
= <SerialCommunicationStop-
bits.ONE: 1>, bytesize: Union[int,
hvl_ccb.comm.serial.SerialCommunicationBytesize] =
<SerialCommunicationBytesize.EIGHTBITS: 8>, timeout:
Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for FuG power supplies is 9600 baud

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

One byte is eight bits long

default_n_attempts_read_text_nonempty: `int = 10`

default number of attempts to read a non-empty text

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

FuG does not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

FuG uses one stop bit

terminator: `bytes = b'\n'`

The terminator is LF

```
timeout: Union[int, float] = 3
    use 3 seconds timeout as default
```

```
wait_sec_read_text_nonempty: Union[int, float] = 0.5
    default time to wait between attempts of reading a non-empty text
```

```
class FuGTerminators(value)
```

```
    Bases: enum.IntEnum
```

```
    An enumeration.
```

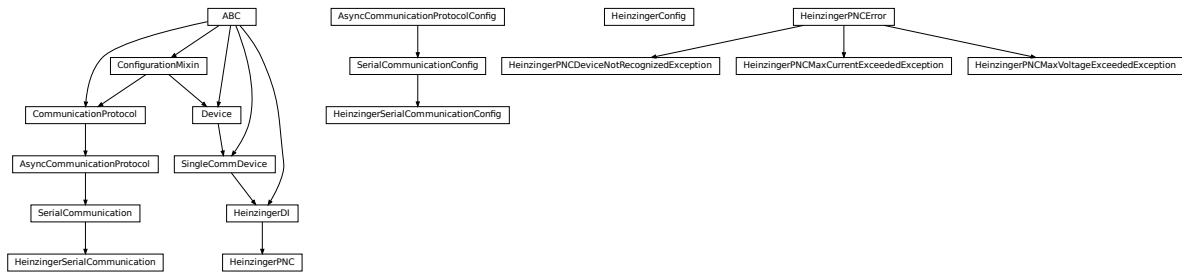
```
    CR = 3
```

```
    CRLF = 0
```

```
    LF = 2
```

```
    LFCR = 1
```

hvl_ccb.dev.heinzinger



Device classes for Heinzinger Digital Interface I/II and Heinzinger PNC power supply.

The Heinzinger Digital Interface I/II is used for many Heinzinger power units. Manufacturer homepage: <https://www.heinzinger.com/products/accessories-and-more/digital-interfaces/>

The Heinzinger PNC series is a series of high voltage direct current power supplies. The class `HeinzingerPNC` is tested with two PNChp 60000-1neg and a PNChp 1500-1neg. Check the code carefully before using it with other PNC devices, especially PNC3p or PNCcap. Manufacturer homepage: <https://www.heinzinger.com/products/high-voltage/universal-high-voltage-power-supplies/>

```
class HeinzingerConfig(default_number_of_recordings: Union[int,
    hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum] = 1, num-
    ber_of_decimals: int = 6, wait_sec_stop_commands: Union[int, float] =
    0.5)
```

```
    Bases: object
```

```
    Device configuration dataclass for Heinzinger power supplies.
```

```
class RecordingsEnum(value)
```

```
    Bases: enum.IntEnum
```

```
    An enumeration.
```

```
    EIGHT = 8
```

```
    FOUR = 4
```

```
    ONE = 1
```

```
    SIXTEEN = 16
```

```

    TWO = 2

    clean_values ()

    default_number_of_recordings: Union[int, hvl_ccb.dev.heinzinger.HeinzingerConfig.Reco

    force_value (fieldname, value)
        Forces a value to a dataclass field despite the class being frozen.

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

        Parameters
            • fieldname – name of the field
            • value – value to assign

    is_configdataclass = True

    classmethod keys () → Sequence[str]
        Returns a list of all configdataclass fields key-names.

        Returns a list of strings containing all keys.

    number_of_decimals: int = 6

    classmethod optional_defaults () → Dict[str, object]
        Returns a list of all configdataclass fields, that have a default value assigned and may be optionally specified
        on instantiation.

        Returns a list of strings containing all optional keys.

    classmethod required_keys () → Sequence[str]
        Returns a list of all configdataclass fields, that have no default value assigned and need to be specified on
        instantiation.

        Returns a list of strings containing all required keys.

    wait_sec_stop_commands: Union[int, float] = 0.5
        Time to wait after subsequent commands during stop (in seconds)

class HeinzingerDI (com, dev_config=None)
    Bases: hvl_ccb.dev.base.SingleCommDevice, abc.ABC
    Heinzinger Digital Interface I/II device class

    Sends basic SCPI commands and reads the answer. Only the standard instruction set from the manual is imple-
    mented.

    class OutputStatus (value)
        Bases: enum.IntEnum
        Status of the voltage output

        OFF = 0
        ON = 1
        UNKNOWN = -1

    static config_cls ()
        Return the default configdataclass class.

        Returns a reference to the default configdataclass class

```

static default_com_cls()

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

Returns the type of the standard communication protocol for this device

get_current() → float

Queries the set current of the Heinzinger PNC (not the measured current!).

Raises *SerialCommunicationIOError* – when communication port is not opened

get_interface_version() → str

Queries the version number of the digital interface.

Raises *SerialCommunicationIOError* – when communication port is not opened

get_number_of_recordings() → int

Queries the number of recordings the device is using for average value calculation.

Returns int number of recordings

Raises *SerialCommunicationIOError* – when communication port is not opened

get_serial_number() → str

Ask the device for its serial number and returns the answer as a string.

Returns string containing the device serial number

Raises *SerialCommunicationIOError* – when communication port is not opened

get_voltage() → float

Queries the set voltage of the Heinzinger PNC (not the measured voltage!).

Raises *SerialCommunicationIOError* – when communication port is not opened

measure_current() → float

Ask the Device to measure its output current and return the measurement result.

Returns measured current as float

Raises *SerialCommunicationIOError* – when communication port is not opened

measure_voltage() → float

Ask the Device to measure its output voltage and return the measurement result.

Returns measured voltage as float

Raises *SerialCommunicationIOError* – when communication port is not opened

output_off() → None

Switch DC voltage output off and updates the output status.

Raises *SerialCommunicationIOError* – when communication port is not opened

output_on() → None

Switch DC voltage output on and updates the output status.

Raises *SerialCommunicationIOError* – when communication port is not opened

property output_status

reset_interface() → None

Reset of the digital interface; only Digital Interface I: Power supply is switched to the Local-Mode (Manual operation)

Raises *SerialCommunicationIOError* – when communication port is not opened

set_current (*value: Union[int, float]*) → None

Sets the output current of the Heinzinger PNC to the given value.

Parameters *value* – current expressed in *self.unit_current*

Raises *SerialCommunicationIOError* – when communication port is not opened

set_number_of_recordings (*value: Union[int, hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum]*) → None

Sets the number of recordings the device is using for average value calculation. The possible values are 1, 2, 4, 8 and 16.

Raises *SerialCommunicationIOError* – when communication port is not opened

set_voltage (*value: Union[int, float]*) → None

Sets the output voltage of the Heinzinger PNC to the given value.

Parameters *value* – voltage expressed in *self.unit_voltage*

Raises *SerialCommunicationIOError* – when communication port is not opened

abstract start ()

Opens the communication protocol.

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

stop () → None

Stop the device. Closes also the communication protocol.

class HeinzingerPNC (*com, dev_config=None*)

Bases: *hvl_ccb.dev.heinzinger.HeinzingerDI*

Heinzinger PNC power supply device class.

The power supply is controlled over a Heinzinger Digital Interface I/II

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

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

An enumeration.

A = 3

UNKNOWN = 1

mA = 2

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

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

An enumeration.

UNKNOWN = 1

V = 2

kV = 3

identify_device () → None

Identify the device nominal voltage and current based on its serial number.

Raises *SerialCommunicationIOError* – when communication port is not opened

property max_current

property max_current_hardware

property max_voltage

property max_voltage_hardware

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

property unit_voltage

exception HeinzingerPNCDeviceNotRecognizedException

Bases: *hvl_ccb.dev.heinzinger.HeinzingerPNCError*

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

exception HeinzingerPNCError

Bases: *Exception*

General error with the Heinzinger PNC voltage source.

exception HeinzingerPNCMaxCurrentExceededException

Bases: *hvl_ccb.dev.heinzinger.HeinzingerPNCError*

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

exception HeinzingerPNCMaxVoltageExceededException

Bases: *hvl_ccb.dev.heinzinger.HeinzingerPNCError*

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

class HeinzingerSerialCommunication (*configuration*)

Bases: *hvl_ccb.comm.serial.SerialCommunication*

Specific communication protocol implementation for Heinzinger power supplies. Already predefines device-specific protocol parameters in config.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class


```
class HeinzingerSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str
                                         = 'utf-8', encoding_error_handling: str =
                                         'replace', wait_sec_read_text_nonempty:
                                         Union[int, float] = 0.5, default_n_attempts_read_text_nonempty: int
                                         = 40, port: Union[str, NoneType] = None,
                                         baudrate: int = 9600, parity: Union[str,
                                         hvl_ccb.comm.serial.SerialCommunicationParity]
                                         = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int,
                                         hvl_ccb.comm.serial.SerialCommunicationStopbits]
                                         = <SerialCommunicationStopbits.ONE: 1>,
                                         bytesize: Union[int,
                                         hvl_ccb.comm.serial.SerialCommunicationBytesize]
                                         = <SerialCommunicationBytesize.EIGHTBITS:
                                         8>, timeout: Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for Heinzinger power supplies is 9600 baud

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

One byte is eight bits long

default_n_attempts_read_text_nonempty: `int = 40`

increased to 40 default number of attempts to read a non-empty text

force_value (*fieldname, value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

Heinzinger does not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

Heinzinger uses one stop bit

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

The terminator is LF

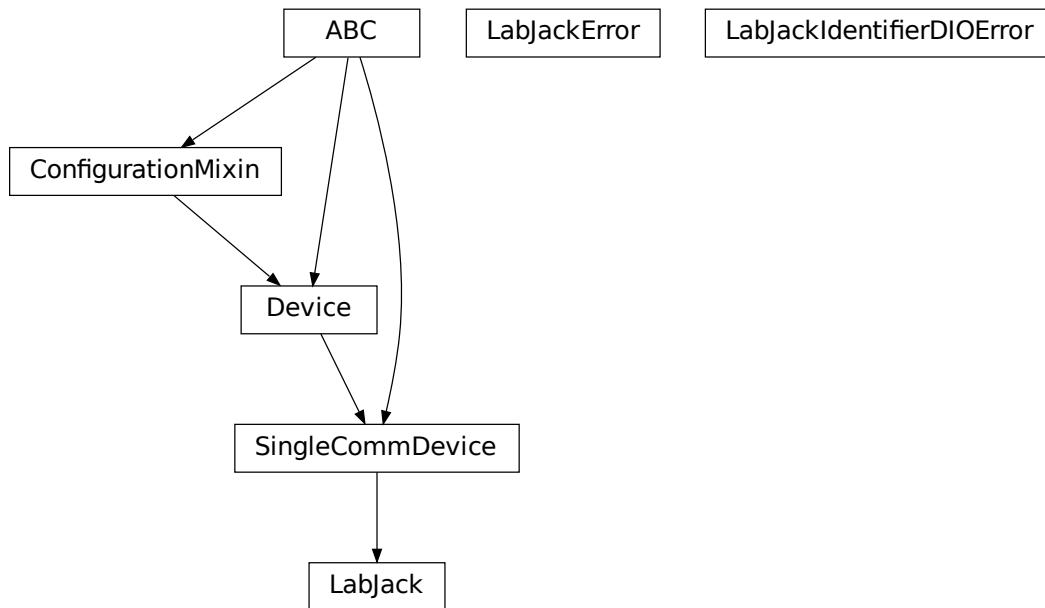
```
timeout: Union[int, float] = 3
```

use 3 seconds timeout as default

```
wait_sec_read_text_nonempty: Union[int, float] = 0.5
```

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

hvl_ccb.dev.labjack



Labjack Device for hvl_ccb. 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 LabJack (com, dev_config=None)
```

Bases: `hvl_ccb.dev.base.SingleCommDevice`

LabJack Device.

This class is tested with a LabJack T7-Pro and should also work with T4 and T7 devices communicating through the LJM Library. Other or older hardware versions and variants of LabJack devices are not supported.

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

Bases: `hvl_ccb.utils.enum.StrEnumBase`

An enumeration.

```
ONE = 1.0
```

```

ONE_HUNDREDTH = 0.01
ONE_TENTH = 0.1
TEN = 10.0
property value
class CalMicroAmpere (value=<object object>, names=None, module=None, type=None,
                      start=1, boundary=None)
    Bases: aenum.Enum
    Pre-defined microampere (uA) values for calibration current source query.
    TEN = '10uA'
    TWO_HUNDRED = '200uA'
class CjcType (value=<object object>, names=None, module=None, type=None, start=1, bound-
               ary=None)
    Bases: hvl_ccb.utils.enum.NameEnum
    CJC slope and offset
    internal = (1, 0)
    lm34 = (55.56, 255.37)
DIOChannel
    alias of hvl_ccb._dev.labjack.TSeriesDIOChannel
class DIOStatus (value=<object object>, names=None, module=None, type=None, start=1,
                 boundary=None)
    Bases: aenum.IntEnum
    State of a digital I/O channel.
    HIGH = 1
    LOW = 0
class DeviceType (value=<object object>, names=None, module=None, type=None, start=1,
                  boundary=None)
    Bases: hvl_ccb.utils.enum.AutoNumberNameEnum
    LabJack device types.
    Can be also looked up by ambiguous Product ID (p_id) or by instance name: `python
    LabJackDeviceType(4) is LabJackDeviceType('T4')`
    ANY = 1
    T4 = 2
    T7 = 3
    T7_PRO = 4
classmethod get_by_p_id (p_id: int) → Union[hvl_ccb._dev.labjack.DeviceType,
                                             List[hvl_ccb._dev.labjack.DeviceType]]
    Get LabJack device type instance via LabJack product ID.
    Note: Product ID is not unambiguous for LabJack devices.
    Parameters p_id – Product ID of a LabJack device
    Returns Instance or list of instances of LabJackDeviceType
    Raises ValueError – when Product ID is unknown

```

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

Bases: `hvl_ccb.utils.enum.NameEnum`

Temperature unit (to be returned)

C = 1

F = 2

K = 0

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

Bases: `hvl_ccb.utils.enum.NameEnum`

Thermocouple type; NONE means disable thermocouple mode.

C = 30

E = 20

J = 21

K = 22

NONE = 0

PT100 = 40

PT1000 = 42

PT500 = 41

R = 23

S = 25

T = 24

```
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_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_digital_input (address: Union[str, hvl_ccb.dev.labjack.TSeriesDIOChannel]) →  
                    hvl_ccb.dev.labjack.LabJack.DIOStatus
```

Get the value of a digital input.

allowed names for T7 (Pro): FIO0 - FIO7, EIO0 - EIO 7, CIO0- CIO3, MIO0 - MIO2 :param address: name of the output -> 'FIO0' :return: HIGH when *address* DIO is high, and LOW when *address* DIO is low

get_product_id () → int

This function returns the product ID reported by the connected device.

Attention: returns 7 for both T7 and T7-Pro devices!

Returns integer product ID of the device

get_product_name (*force_query_id=False*) → str

This function will return the product name based on product ID reported by the device.

Attention: returns "T7" for both T7 and T7-Pro devices!

Parameters **force_query_id** – boolean flag to force *get_product_id* query to device instead of using cached device type from previous queries.

Returns device name string, compatible with *LabJack.DeviceType*

get_product_type (*force_query_id: bool = False*) → *hvl_ccb._dev.labjack.DeviceType*

This function will return the device type based on reported device type and in case of unambiguity based on configuration of device's communication protocol (e.g. for "T7" and "T7_PRO" devices), or, if not available first matching.

Parameters **force_query_id** – boolean flag to force *get_product_id* query to device instead of using cached device type from previous queries.

Returns *DeviceType* instance

Raises *LabJackIdentifierDIOError* – when read Product ID is unknown

get_sbus_rh (*number: int*) → float

Read the relative humidity value from a serial SBUS sensor.

Parameters **number** – port number (0..22)

Returns relative humidity in %RH

get_sbus_temp (*number: int*) → float

Read the temperature value from a serial SBUS sensor.

Parameters **number** – port number (0..22)

Returns temperature in Kelvin

get_serial_number () → int

Returns the serial number of the connected LabJack.

Returns Serial number.

read_resistance (*channel: int*) → float

Read resistance from specified channel.

Parameters **channel** – channel with resistor

Returns resistance value with 2 decimal places

read_thermocouple (*pos_channel: int*) → float

Read the temperature of a connected thermocouple.

Parameters **pos_channel** – is the AIN number of the positive pin

Returns temperature in specified unit

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: (1, 0)>, vrange: Union[Real, AInRange] = <AInRange.ONE_HUNDREDTH: '0.01'>, resolution: int = 10, unit: Union[str, TemperatureUnit] = <TemperatureUnit.K: 0>*) → 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_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: Exception

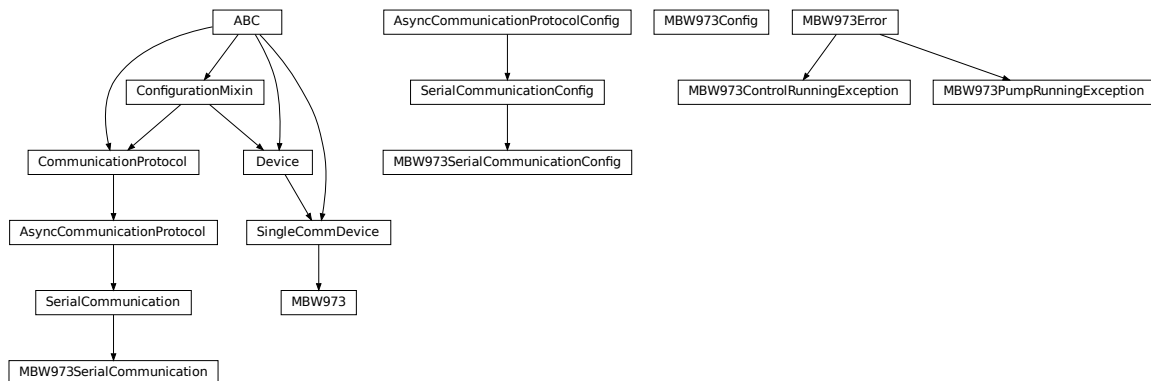
Errors of the LabJack device.

exception LabJackIdentifierDIOError

Bases: Exception

Error indicating a wrong DIO identifier

hvl_ccb.dev.mbw973



Device class for controlling a MBW 973 SF6 Analyzer over a serial connection.

The MBW 973 is a gas analyzer designed for gas insulated switchgear and measures humidity, SF6 purity and SO2 contamination in one go. Manufacturer homepage: <https://www.mbw.ch/products/sf6-gas-analysis/973-sf6-analyzer/>

class MBW973 (*com, dev_config=None*)

Bases: *hvl_ccb.dev.base.SingleCommDevice*

MBW 973 dew point mirror device class.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls ()

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

Returns the type of the standard communication protocol for this device

is_done () → bool

Poll status of the dew point mirror and return True, if all measurements are done.

Returns True, if all measurements are done; False otherwise.

Raises *SerialCommunicationIOError* – when communication port is not opened

read (*cast_type*: Type = <class 'str'>)

Read value from *self.com* and cast to *cast_type*. Raises *ValueError* if read text (*str*) is not convertible to *cast_type*, e.g. to *float* or to *int*.

Returns Read value of *cast_type* type.

read_float () → float

Convenience wrapper for *self.read()*, with typing hint for return value.

Returns Read *float* value.

read_int () → int

Convenience wrapper for *self.read()*, with typing hint for return value.

Returns Read *int* value.

read_measurements () → Dict[str, float]

Read out measurement values and return them as a dictionary.

Returns Dictionary with values.

Raises *SerialCommunicationIOError* – when communication port is not opened

set_measuring_options (*humidity*: bool = True, *sf6_purity*: bool = False) → None

Send measuring options to the dew point mirror.

Parameters

- **humidity** – Perform humidity test or not?
- **sf6_purity** – Perform SF6 purity test or not?

Raises *SerialCommunicationIOError* – when communication port is not opened

start () → None

Start this device. Opens the communication protocol and retrieves the set measurement options from the device.

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

start_control () → None

Start dew point control to acquire a new value set.

Raises *SerialCommunicationIOError* – when communication port is not opened

stop () → None

Stop the device. Closes also the communication protocol.

write (*value*) → None

Send *value* to *self.com*.

Parameters **value** – Value to send, converted to *str*.

Raises *SerialCommunicationIOError* – when communication port is not opened

class MBW973Config (*polling_interval*: Union[int, float] = 2)

Bases: object

Device configuration dataclass for MBW973.

clean_values ()

force_value (*fieldname*, *value*)

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

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

Parameters

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

is_configdataclass = **True**

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

polling_interval: **Union[int, float] = 2**

Polling period for *is_done* status queries [in seconds].

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

exception MBW973ControlRunningException

Bases: *hvl_ccb.dev.mbw973.MBW973Error*

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

exception MBW973Error

Bases: *Exception*

General error with the MBW973 dew point mirror device.

exception MBW973PumpRunningException

Bases: *hvl_ccb.dev.mbw973.MBW973Error*

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

class MBW973SerialCommunication (*configuration*)

Bases: *hvl_ccb.comm.serial.SerialCommunication*

Specific communication protocol implementation for the MBW973 dew point mirror. Already predefines device-specific protocol parameters in config.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class MBW973SerialCommunicationConfig(terminator: bytes = b'\r', encoding: str = 'utf-8', encoding_error_handling: str = '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 = 9600, parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout: Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: int = 9600

Baudrate for MBW973 is 9600 baud

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

One byte is eight bits long

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: Union[str, `hvl_ccb.comm.serial.SerialCommunicationParity`] = 'N'

MBW973 does not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: Union[int, `hvl_ccb.comm.serial.SerialCommunicationStopbits`] = 1

MBW973 does use one stop bit

terminator: bytes = b'\r'

The terminator is only CR

timeout: Union[int, float] = 3

use 3 seconds timeout as default

exception NewportMotorPowerSupplyWasCutError

Bases: `Exception`

Error with the Newport motor after the power supply was cut and then restored, without interrupting the communication with the controller.

class NewportSMC100PP (*com, dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Device class of the Newport motor controller SMC100PP

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

Bases: `aenum.Enum`

Possible motor errors reported by the motor during `get_state()`.

`DC_VOLTAGE_TOO_LOW = 3`

`FOLLOWING_ERROR = 6`

`HOMING_TIMEOUT = 5`

`NED_END_OF_TURN = 11`

`OUTPUT_POWER_EXCEEDED = 2`

`PEAK_CURRENT_LIMIT = 9`

`POS_END_OF_TURN = 10`

`RMS_CURRENT_LIMIT = 8`

`SHORT_CIRCUIT = 7`

`WRONG_ESP_STAGE = 4`

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

Bases: `aenum.Enum`

Possible messages returned by the controller on `get_state()` query.

`CONFIG = '14'`

`DISABLE_FROM_JOGGING = '3E'`

`DISABLE_FROM_MOVING = '3D'`

`DISABLE_FROM_READY = '3C'`

`HOMING_FROM_RS232 = '1E'`

`HOMING_FROM_SMC = '1F'`

`JOGGING_FROM_DISABLE = '47'`

`JOGGING_FROM_READY = '46'`

`MOVING = '28'`

`NO_REF_ESP_STAGE_ERROR = '10'`

`NO_REF_FROM_CONFIG = '0C'`

`NO_REF_FROM_DISABLED = '0D'`

`NO_REF_FROM_HOMING = '0B'`

`NO_REF_FROM_JOGGING = '11'`

```

NO_REF_FROM_MOVING = '0F'
NO_REF_FROM_READY = '0E'
NO_REF_FROM_RESET = '0A'
READY_FROM_DISABLE = '34'
READY_FROM_HOMING = '32'
READY_FROM_JOGGING = '35'
READY_FROM_MOVING = '33'

```

States

alias of `hvl_ccb.dev.newport.NewportStates`

`static config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

`static default_com_cls()`

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

Returns the type of the standard communication protocol for this device

`exit_configuration(add: Optional[int] = None) → None`

Exit the CONFIGURATION state and go back to the NOT REFERENCED state. All configuration parameters are saved to the device's memory.

Parameters `add` – controller address (1 to 31)

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

`get_acceleration(add: Optional[int] = None) → Union[int, float]`

Leave the configuration state. The configuration parameters are saved to the device's memory.

Parameters `add` – controller address (1 to 31)

Returns acceleration (preset units/s²), value between 1e-6 and 1e12

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained
- `NewportControllerError` – if the controller reports an error

`get_controller_information(add: Optional[int] = None) → str`

Get information on the controller name and driver version

Parameters `add` – controller address (1 to 31)

Returns controller information

Raises

- `SerialCommunicationIOError` – if the com is closed
- `NewportSerialCommunicationError` – if an unexpected answer is obtained

- **NewportControllerError** – if the controller reports an error

get_motor_configuration (*add: Optional[int] = None*) → Dict[str, float]

Query the motor configuration and returns it in a dictionary.

Parameters **add** – controller address (1 to 31)

Returns dictionary containing the motor's configuration

Raises

- **SerialCommunicationIOError** – if the com is closed
- **NewportSerialCommunicationError** – if an unexpected answer is obtained
- **NewportControllerError** – if the controller reports an error

get_move_duration (*dist: Union[int, float], add: Optional[int] = None*) → float

Estimate the time necessary to move the motor of the specified distance.

Parameters

- **dist** – distance to travel
- **add** – controller address (1 to 31), defaults to self.address

Raises

- **SerialCommunicationIOError** – if the com is closed
- **NewportSerialCommunicationError** – if an unexpected answer is obtained
- **NewportControllerError** – if the controller reports an error

get_negative_software_limit (*add: Optional[int] = None*) → Union[int, float]

Get the negative software limit (the maximum position that the motor is allowed to travel to towards the left).

Parameters **add** – controller address (1 to 31)

Returns negative software limit (preset units), value between -1e12 and 0

Raises

- **SerialCommunicationIOError** – if the com is closed
- **NewportSerialCommunicationError** – if an unexpected answer is obtained
- **NewportControllerError** – if the controller reports an error

get_position (*add: Optional[int] = None*) → float

Returns the value of the current position.

Parameters **add** – controller address (1 to 31)

Raises

- **SerialCommunicationIOError** – if the com is closed
- **NewportSerialCommunicationError** – if an unexpected answer is obtained
- **NewportControllerError** – if the controller reports an error
- **NewportUncertainPositionError** – if the position is ambiguous

get_positive_software_limit (*add: Optional[int] = None*) → Union[int, float]

Get the positive software limit (the maximum position that the motor is allowed to travel to towards the right).

Parameters `add` – controller address (1 to 31)

Returns positive software limit (preset units), value between 0 and 1e12

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

get_state (`add: int = None`) → *StateMessages*

Check on the motor errors and the controller state

Parameters `add` – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error
- *NewportMotorError* – if the motor reports an error

Returns state message from the device (member of *StateMessages*)

go_home (`add: Optional[int] = None`) → None

Move the motor to its home position.

Parameters `add` – controller address (1 to 31), defaults to `self.address`

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

go_to_configuration (`add: Optional[int] = None`) → None

This method is executed during `start()`. It can also be executed after a `reset()`. The controller is put in CONFIG state, where configuration parameters can be changed.

Parameters `add` – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

initialize (`add: Optional[int] = None`) → None

Puts the controller from the NOT_REF state to the READY state. Sends the motor to its “home” position.

Parameters `add` – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

move_to_absolute_position (*pos*: Union[int, float], *add*: Optional[int] = None) → None
Move the motor to the specified position.

Parameters

- **pos** – target absolute position (affected by the configured offset)
- **add** – controller address (1 to 31), defaults to self.address

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

move_to_relative_position (*pos*: Union[int, float], *add*: Optional[int] = None) → None
Move the motor of the specified distance.

Parameters

- **pos** – distance to travel (the sign gives the direction)
- **add** – controller address (1 to 31), defaults to self.address

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

reset (*add*: Optional[int] = None) → None
Resets the controller, equivalent to a power-up. This puts the controller back to NOT REFERENCED state, which is necessary for configuring the controller.

Parameters **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_acceleration (*acc*: Union[int, float], *add*: Optional[int] = None) → None
Leave the configuration state. The configuration parameters are saved to the device's memory.

Parameters

- **acc** – acceleration (preset units/s²), value between 1e-6 and 1e12
- **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_motor_configuration (*add*: Optional[int] = None, *config*: Optional[dict] = None) → None
Set the motor configuration. The motor must be in CONFIG state.

Parameters

- **add** – controller address (1 to 31)
- **config** – dictionary containing the motor’s configuration

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_negative_software_limit (*lim: Union[int, float], add: Optional[int] = None*) → None

Set the negative software limit (the maximum position that the motor is allowed to travel to towards the left).

Parameters

- **lim** – negative software limit (preset units), value between -1e12 and 0
- **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

set_positive_software_limit (*lim: Union[int, float], add: Optional[int] = None*) → None

Set the positive software limit (the maximum position that the motor is allowed to travel to towards the right).

Parameters

- **lim** – positive software limit (preset units), value between 0 and 1e12
- **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

start ()

Opens the communication protocol and applies the config.

Raises *SerialCommunicationIOError* – when communication port cannot be opened

stop () → None

Stop the device. Close the communication protocol.

stop_motion (*add: Optional[int] = None*) → None

Stop a move in progress by decelerating the positioner immediately with the configured acceleration until it stops. If a controller address is provided, stops a move in progress on this controller, else stops the moves on all controllers.

Parameters **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained

- **NewportControllerError** – if the controller reports an error

wait_until_motor_initialized (*add: Optional[int] = None*) → None

Wait until the motor leaves the HOMING state (at which point it should have arrived to the home position).

Parameters **add** – controller address (1 to 31)

Raises

- **SerialCommunicationIOError** – if the com is closed
- **NewportSerialCommunicationError** – if an unexpected answer is obtained
- **NewportControllerError** – if the controller reports an error

```
class NewportSMC100PPConfig (address: int = 1, user_position_offset: Union[int, float] = 23.987, screw_scaling: Union[int, float] = 1, exit_configuration_wait_sec: Union[int, float] = 5, move_wait_sec: Union[int, float] = 1, acceleration: Union[int, float] = 10, backlash_compensation: Union[int, float] = 0, hysteresis_compensation: Union[int, float] = 0.015, micro_step_per_full_step_factor: int = 100, motion_distance_per_full_step: Union[int, float] = 0.01, home_search_type: Union[int, hvl_ccb.dev.newport.NewportSMC100PPConfig.HomeSearch] = <HomeSearch.HomeSwitch: 2>, jerk_time: Union[int, float] = 0.04, home_search_velocity: Union[int, float] = 4, home_search_timeout: Union[int, float] = 27.5, home_search_polling_interval: Union[int, float] = 1, peak_output_current_limit: Union[int, float] = 0.4, rs485_address: int = 2, negative_software_limit: Union[int, float] = -23.5, positive_software_limit: Union[int, float] = 25, velocity: Union[int, float] = 4, base_velocity: Union[int, float] = 0, stage_configuration: Union[int, hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageConfig] = <EspStageConfig.EnableEspStageCheck: 3>)
```

Bases: object

Configuration dataclass for the Newport motor controller SMC100PP.

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

Bases: aenum.IntEnum

Different configurations to check or not the motor configuration upon power-up.

DisableEspStageCheck = 1

EnableEspStageCheck = 3

UpdateEspStageInfo = 2

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

Bases: aenum.IntEnum

Different methods for the motor to search its home position during initialization.

CurrentPosition = 1

EndOfRunSwitch = 4

EndOfRunSwitch_and_Index = 3

HomeSwitch = 2

```

    HomeSwitch_and_Index = 0
    acceleration: Union[int, float] = 10
    address: int = 1
    backlash_compensation: Union[int, float] = 0
    base_velocity: Union[int, float] = 0
    clean_values()
    exit_configuration_wait_sec: Union[int, float] = 5
    force_value(fieldname, value)

```

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

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

Parameters

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

```

home_search_polling_interval: Union[int, float] = 1
home_search_timeout: Union[int, float] = 27.5
home_search_type: Union[int, hvl\_ccb.dev.newport.NewportSMC100PPConfig.HomeSearch] = 1
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
    Gather the configuration parameters of the motor into a dictionary.

```

Returns dict containing the configuration parameters of the motor

```

move_wait_sec: Union[int, float] = 1
negative_software_limit: Union[int, float] = -23.5
classmethod optional_defaults() → Dict[str, object]

```

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

Returns a list of strings containing all optional keys.

```

peak_output_current_limit: Union[int, float] = 0.4
positive_software_limit: Union[int, float] = 25
post_force_value(fieldname, value)

```

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

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

Returns a list of strings containing all required keys.

rs485_address: int = 2

screw_scaling: Union[int, float] = 1

stage_configuration: Union[int, *hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageCon*

user_position_offset: Union[int, float] = 23.987

velocity: Union[int, float] = 4

class **NewportSMC100PPSerialCommunication** (*configuration*)

Bases: *hvl_ccb.comm.serial.SerialCommunication*

Specific communication protocol implementation Heinzinger power supplies. Already predefines device-specific protocol parameters in config.

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

Bases: *aenum.Enum*

Possible controller errors with values as returned by the device in response to sent commands.

ADDR_INCORRECT = 'B'

CMD_EXEC_ERROR = 'V'

CMD_NOT_ALLOWED = 'D'

CMD_NOT_ALLOWED_CC = 'X'

CMD_NOT_ALLOWED_CONFIGURATION = 'I'

CMD_NOT_ALLOWED_DISABLE = 'J'

CMD_NOT_ALLOWED_HOMING = 'L'

CMD_NOT_ALLOWED_MOVING = 'M'

CMD_NOT_ALLOWED_NOT_REFERENCED = 'H'

CMD_NOT_ALLOWED_PP = 'W'

CMD_NOT_ALLOWED_READY = 'K'

CODE_OR_ADDR_INVALID = 'A'

COM_TIMEOUT = 'S'

DISPLACEMENT_OUT_OF_LIMIT = 'G'

EEPROM_ACCESS_ERROR = 'U'

ESP_STAGE_NAME_INVALID = 'F'

HOME_STARTED = 'E'

NO_ERROR = '@'

PARAM_MISSING_OR_INVALID = 'C'

POSITION_OUT_OF_LIMIT = 'N'

check_for_error (*add: int*) → None

Ask the Newport controller for the last error it recorded.

This method is called after every command or query.

Parameters **add** – controller address (1 to 31)

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query (*add: int, cmd: str, param: Optional[Union[int, float, str]] = None*) → str

Send a query to the controller, read the answer, and check for errors. The prefix add+cmd is removed from the answer.

Parameters

- **add** – the controller address (1 to 31)
- **cmd** – the command to be sent
- **param** – optional parameter (int/float/str) appended to the command

Returns the answer from the device without the prefix

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

query_multiple (*add: int, cmd: str, prefixes: List[str]*) → List[str]

Send a query to the controller, read the answers, and check for errors. The prefixes are removed from the answers.

Parameters

- **add** – the controller address (1 to 31)
- **cmd** – the command to be sent
- **prefixes** – prefixes of each line expected in the answer

Returns list of answers from the device without prefix

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

read_text () → str

Read one line of text from the serial port, and check for presence of a null char which indicates that the motor power supply was cut and then restored. The input buffer may hold additional data afterwards, since only one line is read.

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

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

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *NewportMotorPowerSupplyWasCutError* – if a null char is read

send_command (*add*: int, *cmd*: str, *param*: Optional[Union[int, float, str]] = None) → None
Send a command to the controller, and check for errors.

Parameters

- **add** – the controller address (1 to 31)
- **cmd** – the command to be sent
- **param** – optional parameter (int/float/str) appended to the command

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained
- *NewportControllerError* – if the controller reports an error

send_stop (*add*: int) → None
Send the general stop ST command to the controller, and check for errors.

Parameters **add** – the controller address (1 to 31)

Returns ControllerErrors reported by Newport Controller

Raises

- *SerialCommunicationIOError* – if the com is closed
- *NewportSerialCommunicationError* – if an unexpected answer is obtained

```
class NewportSMC100PPSerialCommunicationConfig (terminator: bytes = b'\r\n', en-
coding: str = 'utf-8', encod-
ing_error_handling: str = 'replace',
wait_sec_read_text_nonempty:
Union[int, float] = 0.5, de-
fault_n_attempts_read_text_nonempty:
int = 10, port: Union[str, None-
Type] = None, baudrate: int
= 57600, parity: Union[str,
hvl_ccb.comm.serial.SerialCommunicationParity]
=
<SerialCommunicationPar-
ity.NONE: 'N'>, stopbits: Union[int,
hvl_ccb.comm.serial.SerialCommunicationStopbits]
=
<SerialCommunicationStop-
bits.ONE: 1>, bytesize: Union[int,
hvl_ccb.comm.serial.SerialCommunicationBytesize]
=
<SerialCommunicationByte-
size.EIGHTBITS: 8>, timeout:
Union[int, float] = 10)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate: int = 57600

Baudrate for Heinzinger power supplies is 9600 baud

bytesize: `Union[int, hv1_ccb.comm.serial.SerialCommunicationBytesize] = 8`
 One byte is eight bits long

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: `Union[str, hv1_ccb.comm.serial.SerialCommunicationParity] = 'N'`

Heinzinger does not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: `Union[int, hv1_ccb.comm.serial.SerialCommunicationStopbits] = 1`

Heinzinger 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: `Exception`

Communication error with the Newport controller.

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

Bases: `hv1_ccb.utils.enum.AutoNumberNameEnum`

States of the Newport controller. Certain commands are allowed only in certain states.

CONFIG = 3

DISABLE = 6

HOMING = 2

JOGGING = 7

MOVING = 5

NO_REF = 1

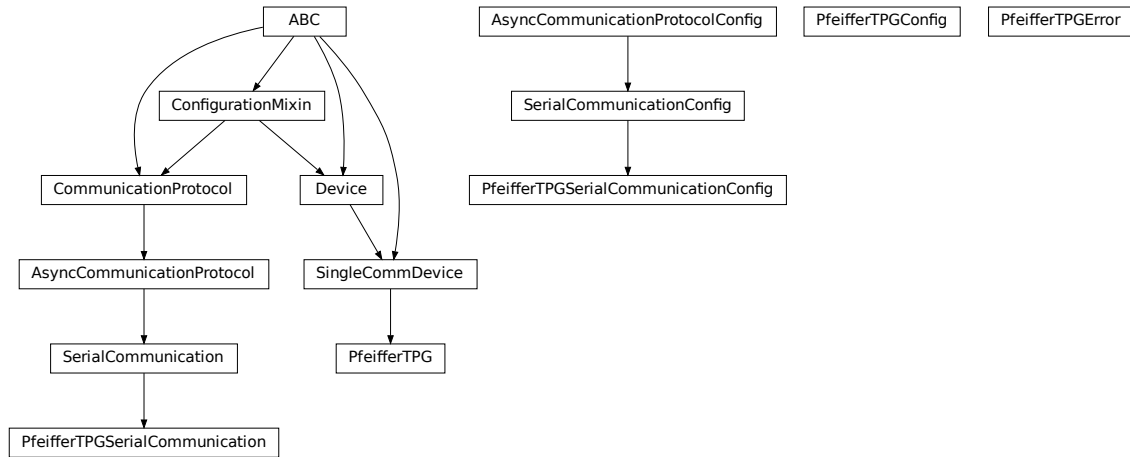
READY = 4

exception NewportUncertainPositionError

Bases: `Exception`

Error with the position of the Newport motor.

`hvl_ccb.dev.pfeiffer_tpg`



Device class for Pfeiffer TPG controllers.

The Pfeiffer TPG control units are used to control Pfeiffer Compact Gauges. Models: TPG 251 A, TPG 252 A, TPG 256A, TPG 261, TPG 262, TPG 361, TPG 362 and TPG 366.

Manufacturer homepage: <https://www.pfeiffer-vacuum.com/en/products/measurement-analysis/measurement/activeline/controllers/>

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

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Pfeiffer TPG control unit device class

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

Bases: `hvl_ccb.utils.enum.NameEnum`

Enum of available pressure units for the digital display. “0” corresponds either to bar or to mbar depending on the TPG model. In case of doubt, the unit is visible on the digital display.

Micron = 3

Pascal = 2

Torr = 1

Volt = 5

bar = 0

hPascal = 4

mbar = 0


```
class SensorStatus (value)
```

Bases: `enum.IntEnum`

An enumeration.

Identification_error = 6

No_sensor = 5

Ok = 0

Overrange = 2

Sensor_error = 3

Sensor_off = 4

Underrange = 1

```
class SensorTypes (value)
```

Bases: `enum.Enum`

An enumeration.

CMR = 4

IKR = 2

IKR11 = 2

IKR9 = 2

IMR = 5

None = 7

PBR = 6

PKR = 3

TPR = 1

noSENSOR = 7

noSen = 7

```
static config_cls ()
```

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
static default_com_cls ()
```

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

Returns the type of the standard communication protocol for this device

```
get_full_scale_mbar () → List[Union[int, float]]
```

Get the full scale range of the attached sensors

Returns full scale range values in mbar, like `[0.01, 1, 0.1, 1000, 50000, 10]`

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

get_full_scale_unitless () → List[int]

Get the full scale range of the attached sensors. See lookup table between command and corresponding pressure in the device user manual.

Returns list of full scale range values, like *[0, 1, 3, 3, 2, 0]*

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

identify_sensors () → None

Send identification request TID to sensors on all channels.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

measure (*channel: int*) → Tuple[str, float]

Get the status and measurement of one sensor

Parameters **channel** – int channel on which the sensor is connected, with $1 \leq \text{channel} \leq \text{number_of_sensors}$

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_display_unit (*unit: Union[str, hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits]*) → None

Set the unit in which the measurements are shown on the display.

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

set_full_scale_mbar (*fsr: List[Union[int, float]]*) → None

Set the full scale range of the attached sensors (in unit mbar)

Parameters **fsr** – full scale range values in mbar, for example *[0.01, 1000]*

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

set_full_scale_unitless (*fsr: List[int]*) → None

Set the full scale range of the attached sensors. See lookup table between command and corresponding pressure in the device user manual.

Parameters *fsr* – list of full scale range values, like *[0, 1, 3, 3, 2, 0]*

Raises

- *SerialCommunicationIOError* – when communication port is not opened
- *PfeifferTPGError* – if command fails

start () → None

Start this device. Opens the communication protocol, and identify the sensors.

Raises *SerialCommunicationIOError* – when communication port cannot be opened

stop () → None

Stop the device. Closes also the communication protocol.

property unit

The pressure unit of readings is always mbar, regardless of the display unit.

```
class PfeifferTPGConfig (model: Union[str, hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model] =
    <Model.TPG25xA: {1: 0, 10: 1, 100: 2, 1000: 3, 2000: 4, 5000: 5, 10000:
    6, 50000: 7, 0.1: 8}>)
```

Bases: object

Device configuration dataclass for Pfeiffer TPG controllers.

```
class Model (value=<object object>, names=None, module=None, type=None, start=1, bound-
    ary=None)
```

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

```
TPG25xA = {0.1: 8, 1: 0, 10: 1, 100: 2, 1000: 3, 2000: 4, 5000: 5, 10000: 6}
```

```
TPGx6x = {0.01: 0, 0.1: 1, 1: 2, 10: 3, 100: 4, 1000: 5, 2000: 6, 5000: 7, 10000: 8}
```

```
is_valid_scale_range_reversed_str (v: str) → bool
```

Check if given string represents a valid reversed scale range of a model.

Parameters *v* – Reversed scale range string.

Returns *True* if valid, *False* otherwise.

```
clean_values ()
```

```
force_value (fieldname, value)
```

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

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

Parameters

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

```
is_configdataclass = True
```

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

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

```
model: Union[str, hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model] = {0.1: 8, 1: 0, 10: 1, 100: 2, 1000: 3, 2000: 4, 5000: 5, 10000: 6, 50000: 7, 0.01: 0}
```

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: `Exception`

Error with the Pfeiffer TPG Controller.

class `PfeifferTPGSerialCommunication` (*configuration*)

Bases: `hvl_ccb.comm.serial.SerialCommunication`

Specific communication protocol implementation for Pfeiffer TPG controllers. Already predefines device-specific protocol parameters in config.

static `config_cls()`

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query (*cmd: str*) → str

Send a query, then read and returns the first line from the com port.

Parameters `cmd` – query message to send to the device

Returns first line read on the com

Raises

- `SerialCommunicationIOError` – when communication port is not opened
- `PfeifferTPGError` – if the device does not acknowledge the command or if the answer from the device is empty

send_command (*cmd: str*) → None

Send a command to the device and check for acknowledgement.

Parameters `cmd` – command to send to the device

Raises

- `SerialCommunicationIOError` – when communication port is not opened
- `PfeifferTPGError` – if the answer from the device differs from the expected acknowledgement character `'chr(6)'`.

```
class PfeifferTPGSerialCommunicationConfig(terminator: bytes = b'\n', encoding: str
    = 'utf-8', 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 = 9600, parity: Union[str,
    hvl_ccb.comm.serial.SerialCommunicationParity]
    = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int,
    hvl_ccb.comm.serial.SerialCommunicationStopbits]
    = <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int,
    hvl_ccb.comm.serial.SerialCommunicationBytesize]
    = <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout: Union[int,
    float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: `int = 9600`

Baudrate for Pfeiffer TPG controllers is 9600 baud

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

One byte is eight bits long

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity: `Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = 'N'`

Pfeiffer TPG controllers do not use parity

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

stopbits: `Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits] = 1`

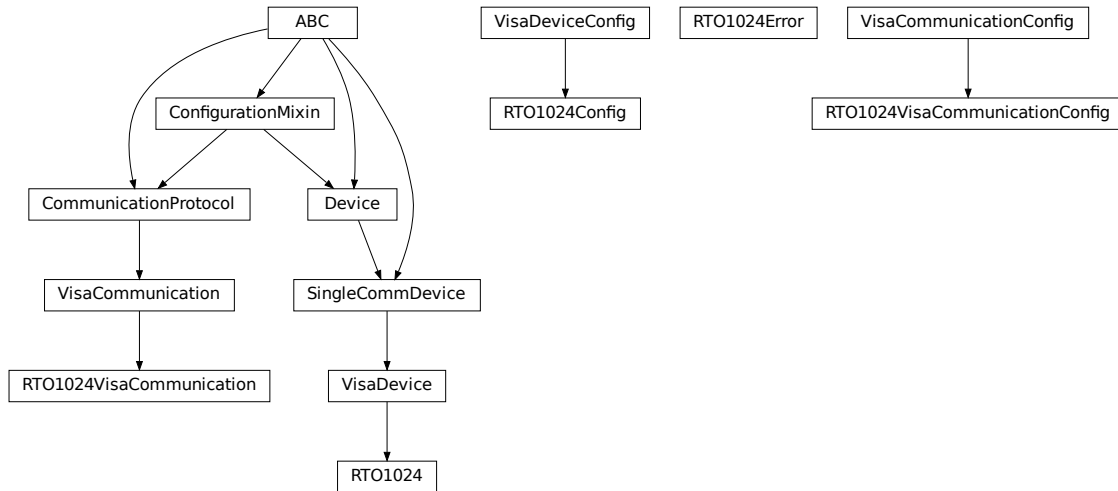
Pfeiffer TPG controllers use one stop bit

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

The terminator is <CR><LF>

```
timeout: Union[int, float] = 3
    use 3 seconds timeout as default
```

hvl_ccb.dev.rs_rto1024



Python module for the Rhode & Schwarz RTO 1024 oscilloscope. The communication to the device is through VISA, type TCPIP / INSTR.

```
class RTO1024 (com: Union[hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunication,
    hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunicationConfig, dict], dev_config:
    Union[hvl_ccb.dev.rs_rto1024.RTO1024Config, dict])
```

Bases: `hvl_ccb.dev.visa.VisaDevice`

Device class for the Rhode & Schwarz RTO 1024 oscilloscope.

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

Bases: `hvl_ccb.utils.enum.AutoNumberNameEnum`

Enumeration for the three available trigger modes.

AUTO = 1

FREERUN = 3

NORMAL = 2

```
classmethod names()
```

Returns a list of the available trigger modes. :return: list of strings

```
activate_measurements (meas_n: int, source: str, measurements: List[str], category: str = 'AMP-
    Time')
```

Activate the list of 'measurements' of the waveform 'source' in the measurement box number 'meas_n'. The list 'measurements' starts with the main measurement and continues with additional measurements of the same 'category'.

Parameters

- **meas_n** – measurement number 1..8

- **source** – measurement source, for example C1W1
- **measurements** – list of measurements, the first one will be the main measurement.
- **category** – the category of measurements, by default AMPTIME

backup_waveform (*filename: str*) → None

Backup a waveform file from the standard directory specified in the device configuration to the standard backup destination specified in the device configuration. The filename has to be specified without .bin or path.

Parameters filename – The waveform filename without extension and path

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls ()

Return the default communication protocol for this device type, which is VisaCommunication.

Returns the VisaCommunication class

file_copy (*source: str, destination: str*) → None

Copy a file from one destination to another on the oscilloscope drive. If the destination file already exists, it is overwritten without notice.

Parameters

- **source** – absolute path to the source file on the DSO filesystem
- **destination** – absolute path to the destination file on the DSO filesystem

Raises *RT01024Error* – if the operation did not complete

get_acquire_length () → float

Gets the time of one acquisition, that is the time across the 10 divisions of the diagram.

- Range: 250E-12 ... 500 [s]
- Increment: 1E-12 [s]

Returns the time for one acquisition. Range: 250e-12 ... 500 [s]

get_channel_offset (*channel: int*) → float

Gets the voltage offset of the indicated channel.

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

Returns channel offset voltage in V (value between -1 and 1)

get_channel_position (*channel: int*) → float

Gets the vertical position of the indicated channel.

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

Returns channel position in div (value between -5 and 5)

get_channel_range (*channel: int*) → float

Queries the channel range in V.

Parameters channel – is the input channel (1..4)

Returns channel range in V

get_channel_scale (*channel: int*) → float

Queries the channel scale in V/div.

Parameters **channel** – is the input channel (1..4)

Returns channel scale in V/div

get_channel_state (*channel: int*) → bool

Queries if the channel is active or not.

Parameters **channel** – is the input channel (1..4)

Returns True if active, else False

get_reference_point () → int

Gets the reference point of the time scale in % of the display. If the “Trigger offset” is zero, the trigger point matches the reference point. ReferencePoint = zero pint of the time scale

- Range: 0 ... 100 [%]
- Increment: 1 [%]

Returns the reference in %

get_repetitions () → int

Get the number of acquired waveforms with RUN Nx SINGLE. Also defines the number of waveforms used to calculate the average waveform.

- Range: 1 ... 16777215
- Increment: 10
- *RST = 1

Returns the number of waveforms to acquire

get_timestamps () → List[float]

Gets the timestamps of all recorded frames in the history and returns them as a list of floats.

Returns list of timestamps in [s]

Raises *RT01024Error* – if the timestamps are invalid

list_directory (*path: str*) → List[Tuple[str, str, int]]

List the contents of a given directory on the oscilloscope filesystem.

Parameters **path** – is the path to a folder

Returns a list of filenames in the given folder

load_configuration (*filename: str*) → None

Load current settings from a configuration file. The filename has to be specified without base directory and ‘.dff’ extension.

Information from the manual *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 ‘.dfl’ extension, the file will be saved to the configured settings directory.

Information from the manual *SAVe* stores the current instrument settings under the specified number in an intermediate memory. The settings can be recalled using the command **RCL* with the associated number. To transfer the stored instrument settings to a file, use *MMEMory:STORe:STATe* .

Parameters **filename** – is the name of the settings file without path and extension

save_waveform_history (*filename: str, channel: int, waveform: int = 1*) → None

Save the history of one channel and one waveform to a .bin file. This function is used after an acquisition using sequence trigger mode (with or without ultra segmentation) was performed.

Parameters

- **filename** – is the name (without extension) of the file
- **channel** – is the channel number
- **waveform** – is the waveform number (typically 1)

Raises *RT01024Error* – if storing waveform times out

set_acquire_length (*timerange: float*) → None

Defines the time of one acquisition, that is the time across the 10 divisions of the diagram.

- Range: 250E-12 ... 500 [s]
- Increment: 1E-12 [s]
- *RST = 0.5 [s]

Parameters **timerange** – is the time for one acquisition. Range: 250e-12 ... 500 [s]

set_channel_offset (*channel: int, offset: float*) → None

Sets the voltage offset of the indicated channel.

- Range: Dependent on the channel scale and coupling [V]
- Increment: Minimum 0.001 [V], may be higher depending on the channel scale and coupling
- *RST = 0

Parameters

- **channel** – is the channel number (1..4)
- **offset** – Offset voltage. Positive values move the waveform down, negative values move it up.

set_channel_position (*channel: int, position: float*) → None

Sets the vertical position of the indicated channel as a graphical value.

- Range: -5.0 ... 5.0 [div]
- Increment: 0.02
- *RST = 0

Parameters

- **channel** – is the channel number (1..4)
- **position** – is the position. Positive values move the waveform up, negative values move it down.

set_channel_range (*channel: int, v_range: float*) → None

Sets the voltage range across the 10 vertical divisions of the diagram. Use the command alternatively instead of set_channel_scale.

- Range for range: Depends on attenuation factors and coupling. With 1:1 probe and external attenuations and 50 input coupling, the range is 10 mV to 10 V. For 1 M input coupling, it is 10 mV to 100 V. If the probe and/or external attenuation is changed, multiply the range values by the attenuation factors.
- Increment: 0.01
- *RST = 0.5

Parameters

- **channel** – is the channel number (1..4)
- **v_range** – is the vertical range [V]

set_channel_scale (*channel: int, scale: float*) → None

Sets the vertical scale for the indicated channel. The scale value is given in volts per division.

- Range for scale: depends on attenuation factor and coupling. With 1:1 probe and external attenuations and 50 input coupling, the vertical scale (input sensitivity) is 1 mV/div to 1 V/div. For 1 M input coupling, it is 1 mV/div to 10 V/div. If the probe and/or external attenuation is changed, multiply the values by the attenuation factors to get the actual scale range.
- Increment: 1e-3
- *RST = 0.05

See also: set_channel_range

Parameters

- **channel** – is the channel number (1..4)
- **scale** – is the vertical scaling [V/div]

set_channel_state (*channel: int, state: bool*) → None

Switches the channel signal on or off.

Parameters

- **channel** – is the input channel (1..4)
- **state** – is True for on, False for off

set_reference_point (*percentage: int*) → None

Sets the reference point of the time scale in % of the display. If the “Trigger offset” is zero, the trigger point matches the reference point. ReferencePoint = zero pint of the time scale

- Range: 0 ... 100 [%]
- Increment: 1 [%]
- *RST = 50 [%]

Parameters percentage – is the reference in %

set_repetitions (*number: int*) → None

Set the number of acquired waveforms with RUN Nx SINGLE. Also defines the number of waveforms used to calculate the average waveform.

- Range: 1 ... 16777215
- Increment: 10
- *RST = 1

Parameters number – is the number of waveforms to acquire

set_trigger_level (*channel: int, level: float, event_type: int = 1*) → None

Sets the trigger level for the specified event and source.

- Range: -10 to 10 V
- Increment: 1e-3 V
- *RST = 0 V

Parameters

- **channel** – indicates the trigger source.
 - 1..4 = channel 1 to 4, available for all event types 1..3
 - 5 = external trigger input on the rear panel for analog signals, available for A-event type = 1
 - 6..9 = not available
- **level** – is the voltage for the trigger level in [V].
- **event_type** – is the event type. 1: A-Event, 2: B-Event, 3: R-Event

set_trigger_mode (*mode: Union[str, hvl_ccb.dev.rs_rto1024.RTO1024.TriggerModes]*) → None

Sets the trigger mode which determines the behavior of the instrument if no trigger occurs.

Parameters mode – is either auto, normal, or freerun.

Raises [RTO1024Error](#) – if an invalid triggermode is selected

set_trigger_source (*channel: int, event_type: int = 1*) → None

Set the trigger (Event A) source channel.

Parameters

- **channel** – is the channel number (1..4)
- **event_type** – is the event type. 1: A-Event, 2: B-Event, 3: R-Event

start () → None

Start the RTO1024 oscilloscope and bring it into a defined state and remote mode.

stop () → None

Stop the RTO1024 oscilloscope, reset events and close communication. Brings back the device to a state where local operation is possible.

stop_acquisition () → None

Stop any acquisition.

```
class RTO1024Config (waveforms_path: str, settings_path: str, backup_path: str, spoll_interval:
    Union[int, float] = 0.5, spoll_start_delay: Union[int, float] = 2, com-
    mand_timeout_seconds: Union[int, float] = 60, wait_sec_short_pause:
    Union[int, float] = 0.1, wait_sec_enable_history: Union[int, float] = 1,
    wait_sec_post_acquisition_start: Union[int, float] = 2)
```

Bases: `hvl_ccb.dev.visa.VisaDeviceConfig`, `hvl_ccb.dev.rs_rto1024._RTO1024ConfigDefaultsBase`, `hvl_ccb.dev.rs_rto1024._RTO1024ConfigBase`

Configdataclass for the RTO1024 device.

force_value (fieldname, value)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

exception RTO1024Error

Bases: `Exception`

class RTO1024VisaCommunication (configuration)

Bases: `hvl_ccb.comm.visa.VisaCommunication`

Specialization of VisaCommunication for the RTO1024 oscilloscope

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class RTO1024VisaCommunicationConfig (host: str, interface_type: Union[str,
    hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType]
    = <InterfaceType.TCPIP_INSTR: 2>, board: int = 0,
    port: int = 5025, timeout: int = 5000, chunk_size: int =
    204800, open_timeout: int = 1000, write_termination:
    str = '\n', read_termination: str = '\n', visa_backend:
    str = ")
Bases: hvl_ccb.comm.visa.VisaCommunicationConfig
```

Configuration dataclass for VisaCommunication with specifications for the RTO1024 device class.

force_value (*fieldname*, *value*)

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

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

Parameters

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

interface_type: Union[str, [hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType](#)] =
Interface type of the VISA connection, being one of InterfaceType.

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

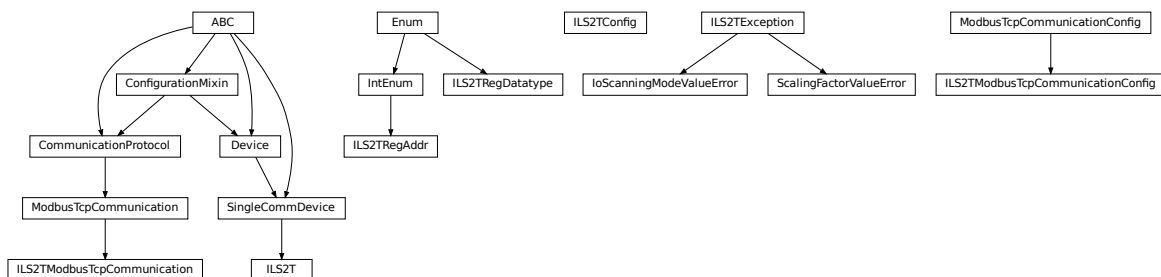
Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

[hvl_ccb.dev.se_ils2t](#)



Device class for controlling a Schneider Electric ILS2T stepper drive over modbus TCP.

```
class ILS2T (com, dev_config=None)
```

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

Schneider Electric ILS2T stepper drive class.

ACTION_JOG_VALUE = 0

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

class ActionsPtp (*value*)

Bases: `enum.IntEnum`

Allowed actions in the point to point mode (*ILS2T.Mode.PTP*).

ABSOLUTE_POSITION = 0

RELATIVE_POSITION_MOTOR = 2

RELATIVE_POSITION_TARGET = 1

DEFAULT_IO_SCANNING_CONTROL_VALUES = {'action': 2, 'continue_after_stop_cu': 0, 'dis

Default IO Scanning control mode values

class Mode (*value*)

Bases: `enum.IntEnum`

ILS2T device modes

JOG = 1

PTP = 3

class Ref16Jog (*value*)

Bases: `enum.Flag`

Allowed values for ILS2T ref_16 register (the shown values are the integer representation of the bits), all in Jog mode = 1

FAST = 4

NEG = 2

NEG_FAST = 6

NONE = 0

POS = 1

POS_FAST = 5

RegAddr

Modbus Register Adresses

alias of `hvl_ccb.dev.se_ils2t.ILS2TRegAddr`

RegDatatype

Modbus Register Datatypes

alias of `hvl_ccb.dev.se_ils2t.ILS2TRegDatatype`

class State (*value*)

Bases: `enum.IntEnum`

State machine status values

ON = 6

QUICKSTOP = 7

READY = 4

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

static default_com_cls ()

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

Returns the type of the standard communication protocol for this device

disable (*log_warn: bool = True, wait_sec_max: Optional[int] = None*) → bool

Disable the driver of the stepper motor and enable the brake.

Note: the driver cannot be disabled if the motor is still running.

Parameters

- **log_warn** – if log a warning in case the motor cannot be disabled.
- **wait_sec_max** – maximal wait time for the motor to stop running and to disable it; by default, with *None*, use a config value

Returns *True* if disable request could and was sent, *False* otherwise.

do_ioscanning_write (***kwargs: int*) → None

Perform a write operation using IO Scanning mode.

Parameters **kwargs** – Keyword-argument list with options to send, remaining are taken from the defaults.

enable () → None

Enable the driver of the stepper motor and disable the brake.

execute_absolute_position (*position: int*) → bool

Execute a absolute position change, i.e. enable motor, perform absolute position change, wait until done and disable motor afterwards.

Check position at the end if wrong do not raise error; instead just log and return check result.

Parameters **position** – absolute position of motor in user defined steps.

Returns *True* if actual position is as expected, *False* otherwise.

execute_relative_step (*steps: int*) → bool

Execute a relative step, i.e. enable motor, perform relative steps, wait until done and disable motor afterwards.

Check position at the end if wrong do not raise error; instead just log and return check result.

Parameters **steps** – Number of steps.

Returns *True* if actual position is as expected, *False* otherwise.

get_dc_volt () → float

Read the DC supply voltage of the motor.

Returns DC input voltage.

get_error_code () → Dict[int, Dict[str, Any]]

Read all messages in fault memory. Will read the full error message and return the decoded values. At the end the fault memory of the motor will be deleted. In addition, `reset_error` is called to re-enable the motor for operation.

Returns Dictionary with all information

get_position() → int

Read the position of the drive and store into status.

Returns Position step value

get_status() → Dict[str, int]

Perform an IO Scanning read and return the status of the motor.

Returns dict with status information.

get_temperature() → int

Read the temperature of the motor.

Returns Temperature in degrees Celsius.

jog_run(direction: bool = True, fast: bool = False) → None

Slowly turn the motor in positive direction.

jog_stop() → None

Stop turning the motor in Jog mode.

quickstop() → None

Stops the motor with high deceleration rate and falls into error state. Reset with *reset_error* to recover into normal state.

reset_error() → None

Resets the motor into normal state after quick stop or another error occurred.

set_jog_speed(slow: int = 60, fast: int = 180) → None

Set the speed for jog mode. Default values correspond to startup values of the motor.

Parameters

- **slow** – RPM for slow jog mode.
- **fast** – RPM for fast jog mode.

set_max_acceleration(rpm_minute: int) → None

Set the maximum acceleration of the motor.

Parameters rpm_minute – revolution per minute per minute

set_max_deceleration(rpm_minute: int) → None

Set the maximum deceleration of the motor.

Parameters rpm_minute – revolution per minute per minute

set_max_rpm(rpm: int) → None

Set the maximum RPM.

Parameters rpm – revolution per minute (0 < rpm <= RPM_MAX)

Raises *ILS2TException* – if RPM is out of range

set_ramp_type(ramp_type: int = -1) → None

Set the ramp type. There are two options available: 0: linear ramp -1: motor optimized ramp

Parameters ramp_type – 0: linear ramp | -1: motor optimized ramp

start() → None

Start this device.

stop () → None

Stop this device. Disables the motor (applies brake), disables access and closes the communication protocol.

user_steps (*steps: int = 16384, revolutions: int = 1*) → None

Define steps per revolution. Default is 16384 steps per revolution. Maximum precision is 32768 steps per revolution.

Parameters

- **steps** – number of steps in *revolutions*.
- **revolutions** – number of revolutions corresponding to *steps*.

write_absolute_position (*position: int*) → None

Write instruction to turn the motor until it reaches the absolute position. This function does not enable or disable the motor automatically.

Parameters position – absolute position of motor in user defined steps.

write_relative_step (*steps: int*) → None

Write instruction to turn the motor the relative amount of steps. This function does not enable or disable the motor automatically.

Parameters steps – Number of steps to turn the motor.

```
class ILS2TConfig (rpm_max_init: numbers.Integral = 1500, wait_sec_post_enable: Union[int, float] =
    1, wait_sec_max_disable: Union[int, float] = 10, wait_sec_post_cannot_disable:
    Union[int, float] = 1, wait_sec_post_relative_step: Union[int, float] = 2,
    wait_sec_post_absolute_position: Union[int, float] = 2)
```

Bases: object

Configuration for the ILS2T stepper motor device.

clean_values ()

force_value (*fieldname, value*)

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

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

Parameters

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

is_configdataclass = True

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

rpm_max_init: `numbers.Integral = 1500`

initial maximum RPM for the motor, can be set up to 3000 RPM. The user is allowed to set a new max RPM at runtime using `ILS2T.set_max_rpm()`, but the value must never exceed this configuration setting.

wait_sec_max_disable: `Union[int, float] = 10`

wait_sec_post_absolute_position: `Union[int, float] = 2`

wait_sec_post_cannot_disable: `Union[int, float] = 1`

wait_sec_post_enable: `Union[int, float] = 1`

wait_sec_post_relative_step: `Union[int, float] = 2`

exception ILS2TException

Bases: `Exception`

Exception to indicate problems with the SE ILS2T stepper motor.

class ILS2TModbusTcpCommunication (*configuration*)

Bases: `hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication`

Specific implementation of Modbus/TCP for the Schneider Electric ILS2T stepper motor.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class ILS2TModbusTcpCommunicationConfig (*host: str, unit: int = 255, port: int = 502*)

Bases: `hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig`

Configuration dataclass for Modbus/TCP communication specific for the Schneider Electric ILS2T stepper motor.

force_value (*fieldname, value*)

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

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

Parameters

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

classmethod keys () → `Sequence[str]`

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

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

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → `Sequence[str]`

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

Returns a list of strings containing all required keys.

```
unit: int = 255
```

The unit has to be 255 such that IO scanning mode works.

```
class ILS2TRegAddr(value)
```

Bases: `enum.IntEnum`

Modbus Register Adresses for for Schneider Electric ILS2T stepper drive.

```
ACCESS_ENABLE = 282
```

```
FLT_INFO = 15362
```

```
FLT_MEM_DEL = 15112
```

```
FLT_MEM_RESET = 15114
```

```
IO_SCANNING = 6922
```

```
JOGN_FAST = 10506
```

```
JOGN_SLOW = 10504
```

```
POSITION = 7706
```

```
RAMP_ACC = 1556
```

```
RAMP_DECEL = 1558
```

```
RAMP_N_MAX = 1554
```

```
RAMP_TYPE = 1574
```

```
SCALE = 1550
```

```
TEMP = 7200
```

```
VOLT = 7198
```

```
class ILS2TRegDatatype(value=<object object>, names=None, module=None, type=None, start=1,
                        boundary=None)
```

Bases: `aenum.Enum`

Modbus Register Datatypes for Schneider Electric ILS2T stepper drive.

From the manual of the drive:

datatype	byte	min	max
INT8	1 Byte	-128	127
UINT8	1 Byte	0	255
INT16	2 Byte	-32_768	32_767
UINT16	2 Byte	0	65_535
INT32	4 Byte	-2_147_483_648	2_147_483_647
UINT32	4 Byte	0	4_294_967_295
BITS	just 32bits	N/A	N/A

```
INT32 = (-2147483648, 2147483647)
```

```
is_in_range(value: int) → bool
```

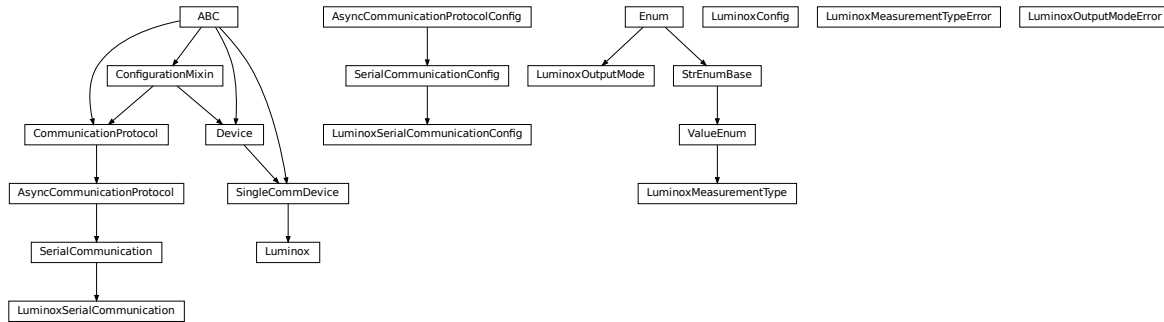
```
exception IoScanningModeValueError
```

Bases: `hvl_ccb.dev.se_ils2t.ILS2TException`

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

exception ScalingFactorValueErrorBases: `hvl_ccb.dev.se_ils2t.ILS2TException`

Exception to indicate that a scaling factor value is invalid.

hvl_ccb.dev.sst_luminox

Device class for a SST Luminox Oxygen sensor. This device can measure the oxygen concentration between 0 % and 25 %.

Furthermore, it measures the barometric pressure and internal temperature. The device supports two operating modes: in streaming mode the device measures all parameters every second, in polling mode the device measures only after a query.

Technical specification and documentation for the device can be found at the manufacturer's page: <https://www.sstsensing.com/product/luminox-optical-oxygen-sensors-2/>

class Luminox (*com, dev_config=None*)Bases: `hvl_ccb.dev.base.SingleCommDevice`

Luminox oxygen sensor device class.

activate_output (*mode: hvl_ccb.dev.sst_luminox.LuminoxOutputMode*) → None

activate the selected output mode of the Luminox Sensor. :param mode: polling or streaming

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class**static default_com_cls** ()

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

Returns the type of the standard communication protocol for this device**query_polling** (*measurement: Union[str, hvl_ccb.dev.sst_luminox.LuminoxMeasurementType]*)→ Union[Dict[Union[str, `hvl_ccb.dev.sst_luminox.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, *hvl_ccb.dev.sst_luminox.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
```

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

Bases: `hvl_ccb.utils.enum.ValueEnum`

Measurement types for *LuminosOutputMode.polling*.

The *all_measurements* type will read values for the actual measurement types as given in *LuminosOutputMode.all_measurements_types()*; it parses multiple single values using regexp's for other measurement types, therefore, no regexp is defined for this measurement type.

```
all_measurements = 'A'
```

```
classmethod all_measurements_types () → Tuple[hvl_ccb.dev.sst_luminos.LuminosMeasurementType,  
                                                ...]
```

A tuple of *LuminosMeasurementType* enum instances which are actual measurements, i.e. not date of manufacture or software revision.

```
barometric_pressure = 'P'
```

property command

```
date_of_manufacture = '# 0'
```

```
parse_read_measurement_value (read_txt: str) → Union[Dict[Union[str,  
hvl_ccb.dev.sst_luminos.LuminosMeasurementType],  
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'
```

LuminosMeasurementTypeDict

A typing hint for a dictionary holding *LuminosMeasurementType* values. Keys are allowed as strings because *LuminosMeasurementType* is of a *StrEnumBase* type.

alias of `Dict[Union[str, LuminosMeasurementType], Union[float, int, str]]`

exception LuminosMeasurementTypeError

Bases: `Exception`

Wrong measurement type for requested data

LuminosMeasurementTypeValue

A typing hint for all possible *LuminosMeasurementType* values as read in either streaming mode or in a polling mode with *LuminosMeasurementType.all_measurements*.

Beware: has to be manually kept in sync with *LuminosMeasurementType* instances *cast_type* attribute values.

alias of `Union[float, int, str]`

```
class LuminosOutputMode (value)
```

Bases: `enum.Enum`

output mode.

```
polling = 1
```

```
streaming = 0
```

exception LuminosOutputModeError

Bases: `Exception`

Wrong output mode for requested data

class LuminosSerialCommunication (*configuration*)

Bases: `hvl_ccb.comm.serial.SerialCommunication`

Specific communication protocol implementation for the SST Luminos 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 LuminosSerialCommunicationConfig (terminator: bytes = b'\n', encoding: str =
    'utf-8', 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 = 9600, parity: Union[str,
    hvl_ccb.comm.serial.SerialCommunicationParity]
    = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int,
    hvl_ccb.comm.serial.SerialCommunicationStopbits]
    = <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int,
    hvl_ccb.comm.serial.SerialCommunicationBytesize]
    = <SerialCommunicationBytesize.EIGHTBITS: 8>,
    timeout: Union[int, float] = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate: int = 9600

Baudrate for SST Luminos is 9600 baud

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

One byte is eight bits long

force_value (*fieldname*, *value*)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

property current

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

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

Returns the type of the standard communication protocol for this device

get_status_byte () → hvl_ccb.dev.technix.TechnixStatusByte

property hv

property inhibit

property max_current

property max_voltage

property remote

start ()

Open the associated communication protocol.

stop ()

Close the associated communication protocol.

property voltage

property voltage_regulation

class TechnixCommunication (config)

Bases: hvl_ccb.comm.base.SyncCommunicationProtocol, abc.ABC

Generic communication class for Technix, which can be implemented via *TechnixSerialCommunication* or *TechnixTelnetCommunication*

query (command: str) → str

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

Parameters **command** – Command to send

Raises *TechnixError* – if the connection is broken

Returns Answer from the interface

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

Bases: hvl_ccb.comm.base.SyncCommunicationProtocolConfig

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: bytes = b'\r'

The terminator is CR

class TechnixConfig (*communication_channel: Union[Type[hvl_ccb.dev.technix.TechnixSerialCommunication], Type[hvl_ccb.dev.technix.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)*

Bases: object

clean_values ()

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

communication_channel: Union[Type[hvl_ccb.dev.technix.TechnixSerialCommunication], Type[hvl_ccb.dev.technix.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

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.

exception TechnixError

Bases: Exception

Technix related errors.

class TechnixSerialCommunication (configuration)

Bases: `hvl_ccb.dev.technix.TechnixCommunication`, `hvl_ccb.comm.serial.SerialCommunication`

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 = '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 = 9600, parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity] = <SerialCommunicationParity.NONE: 'N'>, stopbits: Union[int, float, hvl_ccb.comm.serial.SerialCommunicationStopbits] = <SerialCommunicationStopbits.ONE: 1>, bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize] = <SerialCommunicationBytesize.EIGHTBITS: 8>, timeout: Union[int, float] = 2)

Bases: `hvl_ccb.dev.technix.TechnixCommunicationConfig`, `hvl_ccb.comm.serial.SerialCommunicationConfig`

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 **TechnixStatusByte** (*value: int*)

Bases: object

msb_first (*idx: int*) → Optional[bool]

Give the Bit at position idx with MSB first

Parameters **idx** – Position of Bit as 1...8

Returns

class **TechnixTelnetCommunication** (*configuration*)

Bases: `hvl_ccb.comm.telnet.TelnetCommunication`, `hvl_ccb.dev.technix.TelnetCommunication`

static **config_cls** ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

class **TechnixTelnetCommunicationConfig** (*terminator: bytes = b'\r\n', encoding: str = 'utf-8', encoding_error_handling: str = 'replace', wait_sec_read_text_nonempty: Union[int, float] = 0.5, default_n_attempts_read_text_nonempty: int = 10, host: Union[str, NoneType] = None, port: int = 4660, timeout: Union[int, float] = 0.2*)

Bases: `hvl_ccb.comm.telnet.TelnetCommunicationConfig`, `hvl_ccb.dev.technix.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** = 4660

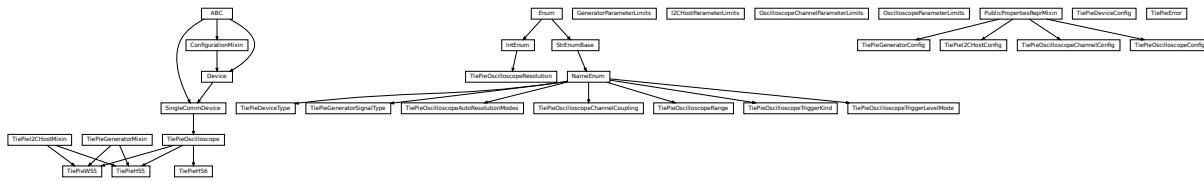
Port at which Technix is listening

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.tiepie



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

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

To install libtiepie on Windows: The installation of the Python bindings “python-libtiepie” is done automatically with the dependencies of the hvl_ccb. The additional DLL for Windows is included in that package.

On a Linux-system additional libraries have to be installed; see <https://www.tiepie.com/en/libtiepie-sdk/linux>.

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

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

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

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

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

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

Bases: object

Default limits for generator parameters.

class I2CHostParameterLimits (*dev_i2c: libtiepie.i2chost.I2CHost*)

Bases: object

Default limits for I2C host parameters.

class OscilloscopeChannelParameterLimits (*osc_channel: libtiepie.oscilloscopechannel.OscilloscopeChannel*)

Bases: object

Default limits for oscilloscope channel parameters.

class OscilloscopeParameterLimits (*dev_osc: libtiepie.oscilloscope.Oscilloscope*)

Bases: object

Default limits for oscilloscope parameters.

class PublicPropertiesReprMixin

Bases: object

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

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

Bases: object

Configuration dataclass for TiePie

clean_values ()

force_value (fieldname, value)

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

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

Parameters

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

is_configdataclass = True

is_data_ready_polling_interval_sec: Union[int, float] = 0.01

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

n_max_try_get_device: int = 10

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

require_block_measurement_support: bool = True

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

serial_number: int

wait_sec_retry_get_device: Union[int, float] = 1.0

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

Bases: *hvl_ccb.utils.enum.NameEnum*

TiePie device type.

GENERATOR = 2

I2C = 4

OSCILLOSCOPE = 1

```
exception TiePieError
```

Bases: Exception

Error of the class TiePie

```

class TiePieGeneratorConfig (dev_gen: libtiepie.generator.Generator)
    Bases: hvl_ccb.dev.tiepie.PublicPropertiesReprMixin
    Generator's configuration with cleaning of values in properties setters.

    property amplitude
    clean_amplitude (amplitude: float) → float
    static clean_enabled (enabled: bool) → bool
    clean_frequency (frequency: float) → float
    clean_offset (offset: float) → float
    static clean_signal_type (signal_type: Union[int, hvl_ccb.dev.tiepie.TiePieGeneratorSignalType])
        → hvl_ccb.dev.tiepie.TiePieGeneratorSignalType

    property enabled
    property frequency
    property offset
    property signal_type

class TiePieGeneratorMixin (com, dev_config)
    Bases: object
    TiePie Generator sub-device.

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

    config_gen: Optional[hvl_ccb.dev.tiepie.TiePieGeneratorConfig]
        Generator's dynamical configuration.

    generator_start ()
        Start signal generation.

    generator_stop ()
        Stop signal generation.

    start () → None
        Start the Generator.

    stop () → None
        Stop the generator.

class TiePieGeneratorSignalType (value=<object object>, names=None, module=None,
                                type=None, start=1, boundary=None)
    Bases: hvl_ccb.utils.enum.NameEnum
    An enumeration.

    ARBITRARY = 32
    DC = 8
    NOISE = 16
    PULSE = 64
    SINE = 1
    SQUARE = 4
    TRIANGLE = 2
    UNKNOWN = 0

```

class TiePieHS5 (*com, dev_config*)

Bases: *hvl_ccb.dev.tiepie.TiePieI2CHostMixin, hvl_ccb.dev.tiepie.TiePieGeneratorMixin, hvl_ccb.dev.tiepie.TiePieOscilloscope*

TiePie HS5 device.

config_gen: **Optional**[*hvl_ccb.dev.tiepie.TiePieGeneratorConfig*]

Generator's dynamical configuration.

class TiePieHS6 (*com, dev_config*)

Bases: *hvl_ccb.dev.tiepie.TiePieOscilloscope*

TiePie HS6 DIFF device.

class TiePieI2CHostConfig (*dev_i2c: libtiepie.i2chost.I2CHost*)

Bases: *hvl_ccb.dev.tiepie.PublicPropertiesReprMixin*

I2C Host's configuration with cleaning of values in properties setters.

class TiePieI2CHostMixin (*com, dev_config*)

Bases: *object*

TiePie I2CHost sub-device.

A wrapper for the *libtiepie.i2chost.I2CHost* class. To be mixed in with *TiePieOscilloscope* base class.

config_i2c: **Optional**[*hvl_ccb.dev.tiepie.TiePieI2CHostConfig*]

I2C host's dynamical configuration.

start () → *None*

Start the I2C Host.

stop () → *None*

Stop the I2C host.

class TiePieOscilloscope (*com, dev_config*)

Bases: *hvl_ccb.dev.base.SingleCommDevice*

TiePie oscilloscope.

A wrapper for TiePie oscilloscopes, based on the class *libtiepie.oscilloscope.Oscilloscope* with simplifications for starting of the device (using serial number) and managing mutable configuration of both the device and its channels, including extra validation and typing hints support for configurations.

Note that, in contrast to *libtiepie* library, since all physical TiePie devices include an oscilloscope, this is the base class for all physical TiePie devices. The additional TiePie sub-devices: “Generator” and “I2CHost”, are mixed-in to this base class in subclasses.

The channels use *1..N* numbering (not *0..N-1*), as in, e.g., the Multi Channel software.

property channels_enabled

Yield numbers of enabled channels.

Returns Numbers of enabled channels

collect_measurement_data (*timeout: Optional[Union[int, float]] = 0*) → *Optional[numpy.ndarray]*

Try to collect the data from TiePie; return *None* if data is not ready.

Parameters *timeout* – The timeout to wait until data is available. This option makes this function blocking the code. *timeout = None* blocks the code infinitely till data will be available. Per default, the *timeout* is set to 0: The function will not block.

Returns Measurement data of only enabled channels and time vector in a 2D-*numpy.ndarray* with float sample data; or *None* if there is no data available.


```

static config_cls () → Type[hvl_ccb.dev.tiepie.TiePieDeviceConfig]
    Return the default configdataclass class.

    Returns a reference to the default configdataclass class

config_osc: Optional[hvl_ccb.dev.tiepie.TiePieOscilloscopeConfig]
    Oscilloscope's dynamical configuration.

config_osc_channel_dict: Dict[int, hvl_ccb.dev.tiepie.TiePieOscilloscopeChannelConfig]
    Channel configuration. A dict mapping actual channel number, numbered 1..N, to channel configuration.
    The channel info is dynamically read from the device only on the first start(); beforehand the dict is empty.

static default_com_cls () → Type[hvl_ccb.comm.base.NullCommunicationProtocol]
    Get the class for the default communication protocol used with this device.

    Returns the type of the standard communication protocol for this device

force_trigger () → None
    Forces the TiePie to trigger with a software sided trigger event.

    Return None

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

is_measurement_data_ready () → bool
    Reports if TiePie has data which is ready to collect

    Returns if the data is ready to collect.

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

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

    Returns if a trigger event occurred

static list_devices () → libtiepie.devicelist.DeviceList
    List available TiePie devices.

    Returns libtiepie up to date list of devices

property n_channels
    Number of channels in the oscilloscope.

    Returns Number of channels.

start () → None
    Start the oscilloscope.

start_measurement () → None
    Start a measurement using set configuration.

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

stop () → None
    Stop the oscilloscope.

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

    Bases: hvl_ccb.utils.enum.NameEnum

    An enumeration.

    ALL = 4

```

```
DISABLED = 1
NATIVEONLY = 2
UNKNOWN = 0

class TiePieOscilloscopeChannelConfig (ch_number: int, channel:
    libtiepie.oscilloscopechannel.OscilloscopeChannel)
    Bases: hvl_ccb.dev.tiepie.PublicPropertiesReprMixin

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

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

    static clean_enabled (enabled: bool) → bool

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

    clean_probe_offset (probe_offset: float) → float

    static clean_safe_ground_enabled (safe_ground_enabled: bool) → bool

    static clean_trigger_enabled (trigger_enabled)

    clean_trigger_hysteresis (trigger_hysteresis: float) → float

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

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

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

    property coupling
    property enabled
    property has_safe_ground
        Check whether bound oscilloscope device has "safe ground" option

        Returns bool: 1=safe ground available

    property input_range
    property probe_offset
    property safe_ground_enabled
    property trigger_enabled
    property trigger_hysteresis
    property trigger_kind
    property trigger_level
    property trigger_level_mode

class TiePieOscilloscopeChannelCoupling (value=<object object>, names=None, mod-
    ule=None, type=None, start=1, boundary=None)
    Bases: hvl_ccb.utils.enum.NameEnum

    An enumeration.

    ACA = 8
```

ACV = 2

DCA = 4

DCV = 1

class TiePieOscilloscopeConfig (*dev_osc: libtiepie.oscilloscope.Oscilloscope*)

Bases: *hvl_ccb.dev.tiepie.PublicPropertiesReprMixin*

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

property auto_resolution_mode

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

clean_pre_sample_ratio (*pre_sample_ratio: float*) → float

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

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

clean_sample_frequency (*sample_frequency: float*) → float

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

property pre_sample_ratio

property record_length

property resolution

property sample_frequency

property trigger_timeout

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

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

EIGHTY_VOLT = 80

EIGHT_HUNDRED_MILLI_VOLT = 0.8

EIGHT_VOLT = 8

FORTY_VOLT = 40

FOUR_HUNDRED_MILLI_VOLT = 0.4

FOUR_VOLT = 4

TWENTY_VOLT = 20

TWO_HUNDRED_MILLI_VOLT = 0.2

TWO_VOLT = 2

static suitable_range (*value*)

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

Bases: *aenum.IntEnum*

An enumeration.

EIGHT_BIT = 8

FOURTEEN_BIT = 14

SIXTEEN_BIT = 16

TWELVE_BIT = 12

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

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

ANY = 16

FALLING = 2

RISING = 1

RISING_OR_FALLING = 16

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

Bases: *hvl_ccb.utils.enum.NameEnum*

An enumeration.

ABSOLUTE = 2

RELATIVE = 1

UNKNOWN = 0

class TiePieWS5 (*com, dev_config*)

Bases: *hvl_ccb.dev.tiepie.TiePieI2CHostMixin, hvl_ccb.dev.tiepie.TiePieGeneratorMixin, hvl_ccb.dev.tiepie.TiePieOscilloscope*

TiePie WS5 device.

config_i2c: **Optional**[*hvl_ccb.dev.tiepie.TiePieI2CHostConfig*]

I2C host's dynamical configuration.

get_device_by_serial_number (*serial_number: int, device_type: Union[str, Tuple[int, _LtpDeviceReturnType]], n_max_try_get_device: int = 10, wait_sec_retry_get_device: float = 1.0*) → *_LtpDeviceReturnType*

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

Parameters

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

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

Raises

- *TiePieError* – when there is no device with given serial number

- **ValueError** – when *device_type* is not an instance of *TiePieDeviceType*

log_set (*prop_name*: str, *prop_value*: object, *value_suffix*: str = "") → None

wrap_libtiepie_exception (*func*: Callable) → Callable

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

Parameters *func* – Function or method to be wrapped

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

Returns whatever *func* returns

hvl_ccb.dev.utils

Poller

class Poller (*spoll_handler*: Callable, *polling_delay_sec*: Union[int, float] = 0, *polling_interval_sec*: Union[int, float] = 1, *polling_timeout_sec*: Optional[Union[int, float]] = None)

Bases: object

Poller class wrapping *concurrent.futures.ThreadPoolExecutor* which enables passing of results and errors out of the polling thread.

is_polling () → bool

Check if device status is being polled.

Returns *True* when polling thread is set and alive

start_polling () → bool

Start polling.

Returns *True* if was not polling before, *False* otherwise

stop_polling () → bool

Stop polling.

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

Returns *True* if was polling before, *False* otherwise, and last result of the polling function call.

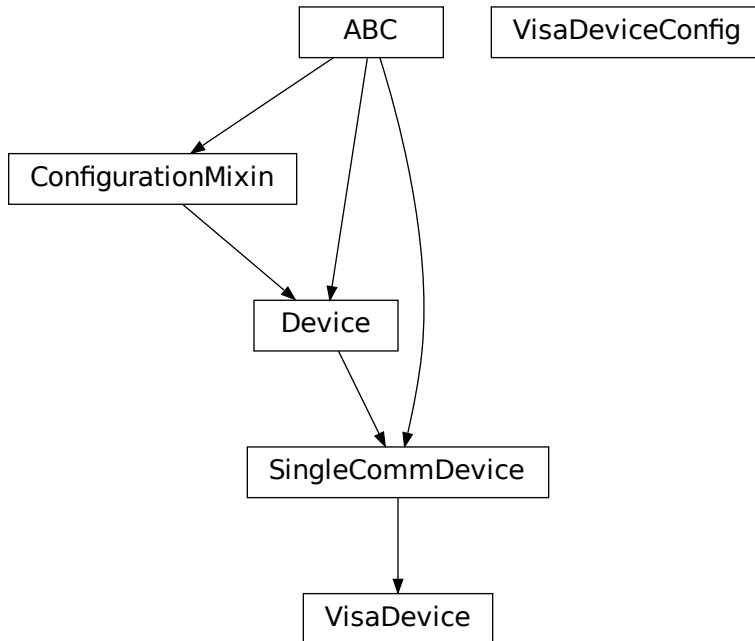
Raises polling function exceptions

wait_for_polling_result ()

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

Returns polling function result

Raises polling function errors

`hvl_ccb.dev.visa`

```

class VisaDevice (com: Union[hvl_ccb.comm.visa.VisaCommunication,
                        hvl_ccb.comm.visa.VisaCommunicationConfig, dict], dev_config: Op-
                        tional[Union[hvl_ccb.dev.visa.VisaDeviceConfig, dict]] = None)
Bases: hvl_ccb.dev.base.SingleCommDevice

Device communicating over the VISA protocol using VisaCommunication.

static config_cls ()
    Return the default configdataclass class.

    Returns a reference to the default configdataclass class

static default_com_cls () → Type[hvl_ccb.comm.visa.VisaCommunication]
    Return the default communication protocol for this device type, which is VisaCommunication.

    Returns the VisaCommunication class

get_error_queue () → str
    Read out error queue and logs the error.

    Returns Error string

get_identification () → str
    Queries “*IDN?” and returns the identification string of the connected device.

    Returns the identification string of the connected device

reset () → None
    Send “*RST” and “*CLS” to the device. Typically sets a defined state.

```

spoll_handler()

Reads the status byte and decodes it. The status byte STB is defined in IEEE 488.2. It provides a rough overview of the instrument status.

Returns

start() → None

Start the VisaDevice. Sets up the status poller and starts it.

Returns

stop() → None

Stop the VisaDevice. Stops the polling thread and closes the communication protocol.

Returns

wait_operation_complete(*timeout: Optional[float] = None*) → bool

Waits for a operation complete event. Returns after timeout [s] has expired or the operation complete event has been caught.

Parameters **timeout** – Time in seconds to wait for the event; *None* for no timeout.

Returns True, if OPC event is caught, False if timeout expired

class VisaDeviceConfig(*spoll_interval: Union[int, float] = 0.5, spoll_start_delay: Union[int, float] = 2*)

Bases: `hvl_ccb.dev.visa._VisaDeviceConfigDefaultsBase`, `hvl_ccb.dev.visa._VisaDeviceConfigBase`

Configdataclass for a VISA device.

force_value(*fieldname, value*)

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

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

Parameters

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

classmethod keys() → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults() → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys() → Sequence[str]

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

Returns a list of strings containing all required keys.

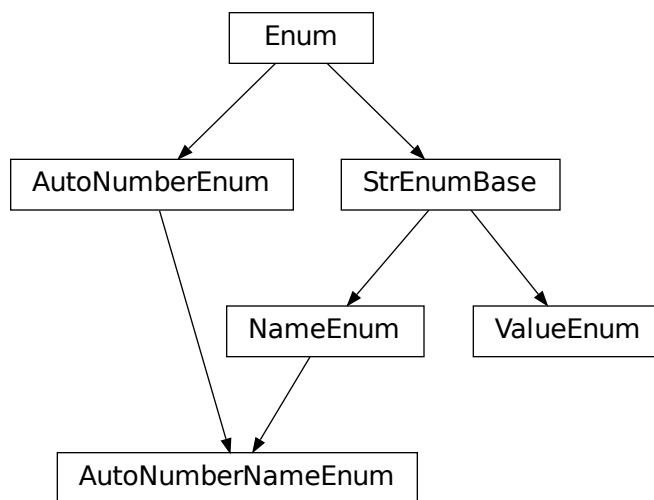
Module contents

Devices subpackage.

hvl_ccb.utils

Submodules

hvl_ccb.utils.enum



```
class AutoNumberNameEnum (value=<object object>, names=None, module=None, type=None,
                           start=1, boundary=None)
    Bases: hvl_ccb.utils.enum.NameEnum, aenum.AutoNumberEnum
```

Auto-numbered enum with names used as string representation, and with lookup and equality based on this representation.

```
class NameEnum (value=<object object>, names=None, module=None, type=None, start=1, bound-
                 ary=None)
    Bases: hvl_ccb.utils.enum.StrEnumBase
```

Enum with names used as string representation, and with lookup and equality based on this representation.

```
class StrEnumBase (value=<object object>, names=None, module=None, type=None, start=1, bound-
                   ary=None)
    Bases: aenum.Enum
```

String representation-based equality and lookup.

```
class ValueEnum (value=<object object>, names=None, module=None, type=None, start=1, bound-
                  ary=None)
    Bases: hvl_ccb.utils.enum.StrEnumBase
```


Enum with string representation of values used as string representation, and with lookup and equality based on this representation.

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

hvl_ccb.utils.typing

Additional Python typing module utilities

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_*)

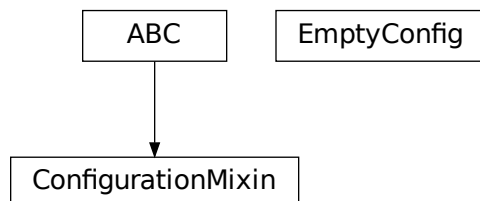
Check if class is a generic type, for example *Union[int, float]*, *List[int]*, or *List*.

Parameters **type** – type to check

Module contents

4.1.2 Submodules

hvl_ccb.configuration



Facilities providing classes for handling configura-

tion for communication protocols and devices.

class ConfigurationMixin (*configuration*)

Bases: `abc.ABC`

Mixin providing configuration to a class.

property config

ConfigDataclass property.

Returns the configuration

abstract static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

configuration_save_json (*path: str*) → None

Save current configuration as JSON file.

Parameters **path** – path to the JSON file.

classmethod from_json (*filename: str*)

Instantiate communication protocol using configuration from a JSON file.

Parameters **filename** – Path and filename to the JSON configuration

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.

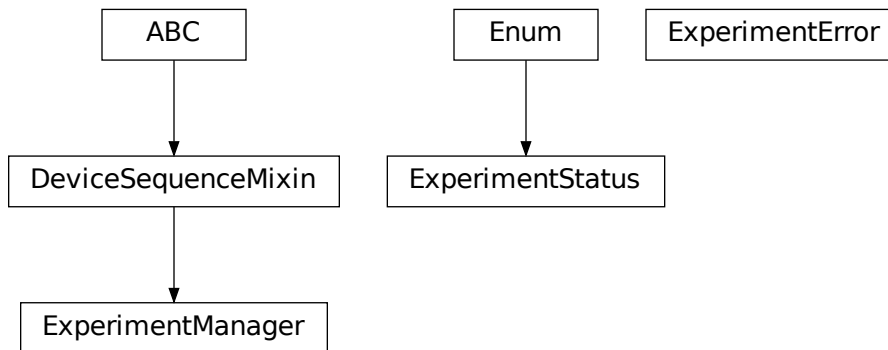
configdataclass (*direct_decoration=None, frozen=True*)

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.experiment_manager



Main module containing the top level ExperimentManager class. Inherit from this class to implement your own experiment functionality in another project and it will help you start, stop and manage your devices.

exception ExperimentError

Bases: `Exception`

Exception to indicate that the current status of the experiment manager is on ERROR and thus no operations can be made until reset.

class ExperimentManager(*args, **kwargs)

Bases: `hvl_ccb.dev.base.DeviceSequenceMixin`

Experiment Manager can start and stop communication protocols and devices. It provides methods to queue commands to devices and collect results.

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

Add a new device to the manager. If the experiment is running, automatically start the device. If a device with this name already exists, raise an exception.

Parameters

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

Raises `DeviceExistingException` –

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

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

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

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

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

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

finish() → None

Stop experimental setup, stop all devices.

is_error() → bool

Returns true, if the status of the experiment manager is *error*.

Returns True if on error, false otherwise

is_finished() → bool

Returns true, if the status of the experiment manager is *finished*.

Returns True if finished, false otherwise

is_running() → bool

Returns true, if the status of the experiment manager is *running*.

Returns True if running, false otherwise

run() → None

Start experimental setup, start all devices.

start() → None

Alias for ExperimentManager.run()

property status

Get experiment status.

Returns experiment status enum code.

stop() → None

Alias for ExperimentManager.finish()

class ExperimentStatus (*value*)

Bases: `enum.Enum`

Enumeration for the experiment status

ERROR = 5

FINISHED = 4

FINISHING = 3

INITIALIZED = 0

INITIALIZING = -1

RUNNING = 2

STARTING = 1

4.1.3 Module contents

Top-level package for HVL Common Code Base.

CONTRIBUTING

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given. You can contribute in many ways:

5.1 Types of Contributions

5.1.1 Report Bugs

Report bugs at https://gitlab.com/ethz_hvl/hvl_ccb/issues.

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

5.1.2 Fix Bugs

Look through the GitLab issues for bugs. Anything tagged with “bug” and “help wanted” is open to whoever wants to implement it.

5.1.3 Implement Features

Look through the GitLab issues for features. Anything tagged with “enhancement” and “help wanted” is open to whoever wants to implement it.

5.1.4 Write Documentation

HVL Common Code Base could always use more documentation, whether as part of the official HVL Common Code Base docs, in docstrings, or even on the web in blog posts, articles, and such.

5.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://gitlab.com/ethz_hvl/hvl_ccb/issues.

If you are proposing a feature:

- Explain in detail how it would work.
- Keep the scope as narrow as possible, to make it easier to implement.
- Remember that this is a volunteer-driven project, and that contributions are welcome :)

5.2 Get Started!

Ready to contribute? Here's how to set up *hvl_ccb* for local development.

1. Clone *hvl_ccb* repo from GitLab.

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

2. Install your local copy into a virtualenv. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development:

```
$ mkvirtualenv hvl_ccb
$ cd hvl_ccb/
$ pip install -e .[all]
$ pip install -r requirements_dev.txt
```

3. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

Now you can make your changes locally.

4. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 hvl_ccb tests
$ python setup.py test or py.test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv. You can also use the provided make-like shell script to run flake8 and tests:

```
$ ./make.sh lint
$ ./make.sh test
```

5. Commit your changes and push your branch to GitLab:

```
$ git add .
$ git commit -m "Your detailed description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

6. Submit a merge request through the GitLab website.

5.3 Merge Request Guidelines

Before you submit a merge request, check that it meets these guidelines:

1. The merge request should include tests.
2. If the merge request adds functionality, the docs should be updated. Put your new functionality into a function with a docstring, and add the feature to the list in README.rst.
3. The merge request should work for Python 3.7. Check https://gitlab.com/ethz_hvl/hvl_ccb/merge_requests and make sure that the tests pass for all supported Python versions.

5.4 Tips

- To run tests from a single file:

```
$ py.test tests/test_hvl_ccb.py
```

or a single test function:

```
$ py.test tests/test_hvl_ccb.py::test_command_line_interface
```

- If your tests are slow, profile them using the pytest-profiling plugin:

```
$ py.test tests/test_hvl_ccb.py --profile
```

or for a graphical overview (you need a SVG image viewer):

```
$ py.test tests/test_hvl_ccb.py --profile-svg
$ open prof/combined.svg
```

- To add dependency, edit appropriate `*requirements` variable in the `setup.py` file and re-run:

```
$ python setup.py develop
```

- To generate a PDF version of the Sphinx documentation instead of HTML use:

```
$ rm -rf docs/hvl_ccb.rst docs/modules.rst docs/_build && sphinx-apidoc -o docs/
↪hvl_ccb && python -msphinx -M latexpdf docs/ docs/_build
```

This command can also be run through the make-like shell script:

```
$ ./make.sh docs-pdf
```

This requires a local installation of a LaTeX distribution, e.g. MikTeX.

5.5 Deploying

A reminder for the maintainers on how to deploy. Create `release-N.M.K` branch. Make sure all your changes are committed. Update or create entry in `HISTORY.rst` file, update features list in `README.rst` file and update API docs:

```
$ make docs
```

Commit all of the above and then run:

```
$ bumpversion patch # possible: major / minor / patch
$ git push
$ git push --tags
$ make release
```

Merge the release branch into master and devel branches with `--no-ff` flag and delete the release branch:

```
$ git checkout master
$ git merge --no-ff release-N.M.K
$ git checkout devel
$ git merge --no-ff release-N.M.K
$ git push --delete origin release-N.M.K
$ git branch --delete release-N.M.K
```

Finally, go to https://gitlab.com/ethz_hvl/hvl_ccb/tags/, select the latest release tag, press “Edit release notes” and add release notes (corresponding entry from `HISTORY.rst` file, but consider also additional brief header or synopsis if needed).

CREDITS

6.1 Maintainers

- Mikołaj Rybiński <mikolaj.rybinski@id.ethz.ch>

6.2 Authors

- Mikołaj Rybiński <mikolaj.rybinski@id.ethz.ch>
- David Graber <dev@davidgraber.ch>
- Henrik Menne <henrik.menne@eeh.ee.ethz.ch>
- Alise Chachereau <chachereau@eeh.ee.ethz.ch>
- Henning Janssen <janssen@eeh.ee.ethz.ch>

6.3 Contributors

- Luca Nembrini <lucane@student.ethz.ch>
- Maria Pukhlyakova <mariia.pukhliakova@id.ethz.ch>
- Raphael Faerber <raphael.ferber@eeh.ee.ethz.ch>
- Ruben Stadler <rstadler@student.ethz.ch>

HISTORY

7.1 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.

7.2 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`.

7.3 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_luminos.Luminos`: 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`.

7.4 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.

7.5 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).

7.6 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.

7.7 0.3.3 (2019-05-08)

- Use PyPI labjack-ljm (no external dependencies)

7.8 0.3.2 (2019-05-08)

- INSTALLATION.rst with LJMPython prerequisite info

7.9 0.3.1 (2019-05-02)

- readthedocs.org support

7.10 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)

7.11 0.2.1 (2019-04-01)

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

7.12 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.

7.13 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](#), 168
- [hvl_ccb.comm](#), 29
 - [hvl_ccb.comm.base](#), 9
 - [hvl_ccb.comm.labjack_ljm](#), 13
 - [hvl_ccb.comm.modbus_tcp](#), 16
 - [hvl_ccb.comm.opc](#), 18
 - [hvl_ccb.comm.serial](#), 21
 - [hvl_ccb.comm.telnet](#), 24
 - [hvl_ccb.comm.visa](#), 26
- [hvl_ccb.configuration](#), 165
- [hvl_ccb.dev](#), 164
 - [hvl_ccb.dev.base](#), 69
 - [hvl_ccb.dev.crylas](#), 71
 - [hvl_ccb.dev.ea_psi9000](#), 81
 - [hvl_ccb.dev.fug](#), 85
 - [hvl_ccb.dev.heinzinger](#), 96
 - [hvl_ccb.dev.labjack](#), 102
 - [hvl_ccb.dev.mbw973](#), 107
 - [hvl_ccb.dev.newport](#), 111
 - [hvl_ccb.dev.pfeiffer_tpg](#), 124
 - [hvl_ccb.dev.rs_rto1024](#), 130
 - [hvl_ccb.dev.se_ils2t](#), 137
 - [hvl_ccb.dev.sst_luminos](#), 144
 - [hvl_ccb.dev.supercube](#), 53
 - [hvl_ccb.dev.supercube.base](#), 29
 - [hvl_ccb.dev.supercube.constants](#), 35
 - [hvl_ccb.dev.supercube.typ_a](#), 49
 - [hvl_ccb.dev.supercube.typ_b](#), 52
 - [hvl_ccb.dev.supercube2015](#), 68
 - [hvl_ccb.dev.supercube2015.base](#), 53
 - [hvl_ccb.dev.supercube2015.constants](#), 59
 - [hvl_ccb.dev.supercube2015.typ_a](#), 66
 - [hvl_ccb.dev.technix](#), 148
 - [hvl_ccb.dev.tiepie](#), 153
 - [hvl_ccb.dev.utils](#), 161
 - [hvl_ccb.dev.visa](#), 162
- [hvl_ccb.experiment_manager](#), 167
- [hvl_ccb.utils](#), 165
 - [hvl_ccb.utils.enum](#), 164
 - [hvl_ccb.utils.typing](#), 165

A

- A (*HeinzingerPNC.UnitCurrent attribute*), 99
- A (*SupercubeOpcEndpoint attribute*), 48, 66
- ABSOLUTE (*TiePieOscilloscopeTriggerLevelMode attribute*), 160
- ABSOLUTE_POSITION (*ILS2T.ActionsPtp attribute*), 138
- AC (*NewportConfigCommands attribute*), 111
- AC_DoubleStage_150kV (*PowerSetup attribute*), 47, 64
- AC_DoubleStage_200kV (*PowerSetup attribute*), 47, 64
- AC_SingleStage_100kV (*PowerSetup attribute*), 47, 64
- AC_SingleStage_50kV (*PowerSetup attribute*), 47, 64
- ACA (*TiePieOscilloscopeChannelCoupling attribute*), 158
- acceleration (*NewportSMC100PPConfig attribute*), 119
- ACCESS_ENABLE (*ILS2TRegAddr attribute*), 143
- access_lock (*CommunicationProtocol attribute*), 12
- ACTION_JOG_VALUE (*ILS2T attribute*), 138
- activate_measurements() (*RTO1024 method*), 130
- activate_output() (*Luminex method*), 144
- activated (*BreakdownDetection attribute*), 40, 60
- ACTIVE (*CryLasLaser.AnswersStatus attribute*), 74
- active (*OpcControl attribute*), 46
- actualsetvalue() (*FuGProbusVSetRegisters property*), 93
- ACV (*TiePieOscilloscopeChannelCoupling attribute*), 159
- adc_mode() (*FuGProbusVMonitorRegisters property*), 92
- add_device() (*DeviceSequenceMixin method*), 69
- add_device() (*ExperimentManager method*), 167
- ADDR_INCORRECT (*NewportSMC100PPSerialCommunication.ControllerErrors attribute*), 120
- address (*NewportSMC100PPConfig attribute*), 119
- address() (*VisaCommunicationConfig property*), 27
- address() (*VisaCommunicationConfig.InterfaceType method*), 27
- ADMODE (*FuGProbusIVCommands attribute*), 90
- Alarm0 (*AlarmText attribute*), 59
- Alarm1 (*Alarms attribute*), 36
- Alarm1 (*AlarmText attribute*), 35, 59
- Alarm10 (*Alarms attribute*), 36
- Alarm10 (*AlarmText attribute*), 35, 59
- Alarm100 (*Alarms attribute*), 36
- Alarm101 (*Alarms attribute*), 36
- Alarm102 (*Alarms attribute*), 36
- Alarm103 (*Alarms attribute*), 36
- Alarm104 (*Alarms attribute*), 36
- Alarm105 (*Alarms attribute*), 36
- Alarm106 (*Alarms attribute*), 36
- Alarm107 (*Alarms attribute*), 36
- Alarm108 (*Alarms attribute*), 36
- Alarm109 (*Alarms attribute*), 36
- Alarm11 (*Alarms attribute*), 37
- Alarm11 (*AlarmText attribute*), 35, 59
- Alarm110 (*Alarms attribute*), 37
- Alarm111 (*Alarms attribute*), 37
- Alarm112 (*Alarms attribute*), 37
- Alarm113 (*Alarms attribute*), 37
- Alarm114 (*Alarms attribute*), 37
- Alarm115 (*Alarms attribute*), 37
- Alarm116 (*Alarms attribute*), 37
- Alarm117 (*Alarms attribute*), 37
- Alarm118 (*Alarms attribute*), 37
- Alarm119 (*Alarms attribute*), 37
- Alarm12 (*Alarms attribute*), 37
- Alarm12 (*AlarmText attribute*), 35, 59
- Alarm120 (*Alarms attribute*), 37
- Alarm121 (*Alarms attribute*), 37
- Alarm122 (*Alarms attribute*), 37
- Alarm123 (*Alarms attribute*), 37
- Alarm124 (*Alarms attribute*), 37
- Alarm125 (*Alarms attribute*), 37
- Alarm126 (*Alarms attribute*), 37
- Alarm127 (*Alarms attribute*), 37
- Alarm128 (*Alarms attribute*), 37
- Alarm129 (*Alarms attribute*), 37

Alarm13 (*Alarms attribute*), 37
Alarm13 (*AlarmText attribute*), 35, 59
Alarm130 (*Alarms attribute*), 37
Alarm131 (*Alarms attribute*), 37
Alarm132 (*Alarms attribute*), 37
Alarm133 (*Alarms attribute*), 37
Alarm134 (*Alarms attribute*), 37
Alarm135 (*Alarms attribute*), 37
Alarm136 (*Alarms attribute*), 37
Alarm137 (*Alarms attribute*), 37
Alarm138 (*Alarms attribute*), 37
Alarm139 (*Alarms attribute*), 37
Alarm14 (*Alarms attribute*), 37
Alarm14 (*AlarmText attribute*), 35, 59
Alarm140 (*Alarms attribute*), 37
Alarm141 (*Alarms attribute*), 37
Alarm142 (*Alarms attribute*), 38
Alarm143 (*Alarms attribute*), 38
Alarm144 (*Alarms attribute*), 38
Alarm145 (*Alarms attribute*), 38
Alarm146 (*Alarms attribute*), 38
Alarm147 (*Alarms attribute*), 38
Alarm148 (*Alarms attribute*), 38
Alarm149 (*Alarms attribute*), 38
Alarm15 (*Alarms attribute*), 38
Alarm15 (*AlarmText attribute*), 35
Alarm150 (*Alarms attribute*), 38
Alarm151 (*Alarms attribute*), 38
Alarm16 (*Alarms attribute*), 38
Alarm16 (*AlarmText attribute*), 35
Alarm17 (*Alarms attribute*), 38
Alarm17 (*AlarmText attribute*), 35, 59
Alarm18 (*Alarms attribute*), 38
Alarm18 (*AlarmText attribute*), 35
Alarm19 (*Alarms attribute*), 38
Alarm19 (*AlarmText attribute*), 35, 59
Alarm2 (*Alarms attribute*), 38
Alarm2 (*AlarmText attribute*), 35, 59
Alarm20 (*Alarms attribute*), 38
Alarm20 (*AlarmText attribute*), 35, 59
Alarm21 (*Alarms attribute*), 38
Alarm21 (*AlarmText attribute*), 35, 59
Alarm22 (*Alarms attribute*), 38
Alarm22 (*AlarmText attribute*), 35, 59
Alarm23 (*Alarms attribute*), 38
Alarm23 (*AlarmText attribute*), 35
Alarm24 (*Alarms attribute*), 38
Alarm24 (*AlarmText attribute*), 35
Alarm25 (*Alarms attribute*), 38
Alarm25 (*AlarmText attribute*), 35
Alarm26 (*Alarms attribute*), 38
Alarm26 (*AlarmText attribute*), 36
Alarm27 (*Alarms attribute*), 38
Alarm28 (*Alarms attribute*), 38
Alarm29 (*Alarms attribute*), 38
Alarm3 (*Alarms attribute*), 38
Alarm3 (*AlarmText attribute*), 36, 59
Alarm30 (*Alarms attribute*), 38
Alarm31 (*Alarms attribute*), 38
Alarm32 (*Alarms attribute*), 38
Alarm33 (*Alarms attribute*), 38
Alarm34 (*Alarms attribute*), 38
Alarm35 (*Alarms attribute*), 38
Alarm36 (*Alarms attribute*), 38
Alarm37 (*Alarms attribute*), 38
Alarm38 (*Alarms attribute*), 38
Alarm39 (*Alarms attribute*), 39
Alarm4 (*Alarms attribute*), 39
Alarm4 (*AlarmText attribute*), 36, 59
Alarm40 (*Alarms attribute*), 39
Alarm41 (*Alarms attribute*), 39
Alarm41 (*AlarmText attribute*), 36
Alarm42 (*Alarms attribute*), 39
Alarm42 (*AlarmText attribute*), 36
Alarm43 (*Alarms attribute*), 39
Alarm43 (*AlarmText attribute*), 36
Alarm44 (*Alarms attribute*), 39
Alarm44 (*AlarmText attribute*), 36
Alarm45 (*Alarms attribute*), 39
Alarm45 (*AlarmText attribute*), 36
Alarm46 (*Alarms attribute*), 39
Alarm46 (*AlarmText attribute*), 36
Alarm47 (*Alarms attribute*), 39
Alarm47 (*AlarmText attribute*), 36
Alarm48 (*Alarms attribute*), 39
Alarm48 (*AlarmText attribute*), 36
Alarm49 (*Alarms attribute*), 39
Alarm5 (*Alarms attribute*), 39
Alarm5 (*AlarmText attribute*), 36, 59
Alarm50 (*Alarms attribute*), 39
Alarm51 (*Alarms attribute*), 39
Alarm52 (*Alarms attribute*), 39
Alarm53 (*Alarms attribute*), 39
Alarm54 (*Alarms attribute*), 39
Alarm55 (*Alarms attribute*), 39
Alarm56 (*Alarms attribute*), 39
Alarm57 (*Alarms attribute*), 39
Alarm58 (*Alarms attribute*), 39
Alarm59 (*Alarms attribute*), 39
Alarm6 (*Alarms attribute*), 39
Alarm6 (*AlarmText attribute*), 36, 59
Alarm60 (*Alarms attribute*), 39
Alarm61 (*Alarms attribute*), 39
Alarm62 (*Alarms attribute*), 39
Alarm63 (*Alarms attribute*), 39
Alarm64 (*Alarms attribute*), 39
Alarm65 (*Alarms attribute*), 39
Alarm66 (*Alarms attribute*), 39

- Alarm67 (*Alarms attribute*), 39
 Alarm68 (*Alarms attribute*), 39
 Alarm69 (*Alarms attribute*), 39
 Alarm7 (*Alarms attribute*), 39
 Alarm7 (*AlarmText attribute*), 36, 59
 Alarm70 (*Alarms attribute*), 39
 Alarm71 (*Alarms attribute*), 40
 Alarm72 (*Alarms attribute*), 40
 Alarm73 (*Alarms attribute*), 40
 Alarm74 (*Alarms attribute*), 40
 Alarm75 (*Alarms attribute*), 40
 Alarm76 (*Alarms attribute*), 40
 Alarm77 (*Alarms attribute*), 40
 Alarm78 (*Alarms attribute*), 40
 Alarm79 (*Alarms attribute*), 40
 Alarm8 (*Alarms attribute*), 40
 Alarm8 (*AlarmText attribute*), 36, 59
 Alarm80 (*Alarms attribute*), 40
 Alarm81 (*Alarms attribute*), 40
 Alarm82 (*Alarms attribute*), 40
 Alarm83 (*Alarms attribute*), 40
 Alarm84 (*Alarms attribute*), 40
 Alarm85 (*Alarms attribute*), 40
 Alarm86 (*Alarms attribute*), 40
 Alarm87 (*Alarms attribute*), 40
 Alarm88 (*Alarms attribute*), 40
 Alarm89 (*Alarms attribute*), 40
 Alarm9 (*Alarms attribute*), 40
 Alarm9 (*AlarmText attribute*), 36, 59
 Alarm90 (*Alarms attribute*), 40
 Alarm91 (*Alarms attribute*), 40
 Alarm92 (*Alarms attribute*), 40
 Alarm93 (*Alarms attribute*), 40
 Alarm94 (*Alarms attribute*), 40
 Alarm95 (*Alarms attribute*), 40
 Alarm96 (*Alarms attribute*), 40
 Alarm97 (*Alarms attribute*), 40
 Alarm98 (*Alarms attribute*), 40
 Alarm99 (*Alarms attribute*), 40
 Alarms (*class in hvl_ccb.dev.supercube.constants*), 36
 AlarmText (*class in hvl_ccb.dev.supercube.constants*), 35
 AlarmText (*class in hvl_ccb.dev.supercube2015.constants*), 59
 ALL (*TiePieOscilloscopeAutoResolutionModes attribute*), 157
 all_measurements (*LuminoxMeasurementType attribute*), 146
 all_measurements_types() (*LuminoxMeasurementType class method*), 146
 amplitude() (*TiePieGeneratorConfig property*), 155
 analog_control() (*FuGProbusVDIRegisters property*), 92
 ANY (*LabJack.DeviceType attribute*), 103
 ANY (*LJMCommunicationConfig.ConnectionType attribute*), 14
 ANY (*LJMCommunicationConfig.DeviceType attribute*), 14
 ANY (*TiePieOscilloscopeTriggerKind attribute*), 160
 ARBITRARY (*TiePieGeneratorSignalType attribute*), 155
 AsyncCommunicationProtocol (*class in hvl_ccb.comm.base*), 9
 AsyncCommunicationProtocolConfig (*class in hvl_ccb.comm.base*), 11
 attenuation() (*CryLasAttenuator property*), 72
 auto (*EarthingStickOperatingStatus attribute*), 43
 AUTO (*RTO1024.TriggerModes attribute*), 130
 auto_laser_on (*CryLasLaserConfig attribute*), 78
 auto_resolution_mode() (*TiePieOscilloscopeConfig property*), 159
 AutoNumberNameEnum (*class in hvl_ccb.utils.enum*), 164
- ## B
- B (*SupercubeOpcEndpoint attribute*), 48, 66
 BA (*NewportConfigCommands attribute*), 111
 backlash_compensation (*NewportSMC100PPConfig attribute*), 119
 backup_waveform() (*RTO1024 method*), 131
 bar (*PfeifferTPG.PressureUnits attribute*), 124
 barometric_pressure (*LuminoxMeasurementType attribute*), 146
 base_velocity (*NewportSMC100PPConfig attribute*), 119
 baudrate (*CryLasAttenuatorSerialCommunicationConfig attribute*), 73
 baudrate (*CryLasLaserSerialCommunicationConfig attribute*), 80
 baudrate (*FuGSerialCommunicationConfig attribute*), 95
 baudrate (*HeinzingerSerialCommunicationConfig attribute*), 101
 baudrate (*LuminoxSerialCommunicationConfig attribute*), 147
 baudrate (*MBW973SerialCommunicationConfig attribute*), 110
 baudrate (*NewportSMC100PPSerialCommunicationConfig attribute*), 122
 baudrate (*PfeifferTPGSerialCommunicationConfig attribute*), 129
 baudrate (*SerialCommunicationConfig attribute*), 22
 BH (*NewportConfigCommands attribute*), 111
 board (*VisaCommunicationConfig attribute*), 27
 BreakdownDetection (*class in hvl_ccb.dev.supercube.constants*), 40
 BreakdownDetection (*class in hvl_ccb.dev.supercube2015.constants*), 60

- bytesize (*CryLasAttenuatorSerialCommunicationConfig* attribute), 73
 bytesize (*CryLasLaserSerialCommunicationConfig* attribute), 80
 bytesize (*FuGSerialCommunicationConfig* attribute), 95
 bytesize (*HeinzingerSerialCommunicationConfig* attribute), 101
 bytesize (*LuminoxSerialCommunicationConfig* attribute), 147
 bytesize (*MBW973SerialCommunicationConfig* attribute), 110
 bytesize (*NewportSMC100PPSerialCommunicationConfig* attribute), 123
 bytesize (*PfeifferTPGSerialCommunicationConfig* attribute), 129
 Bytesize (*SerialCommunicationConfig* attribute), 22
 bytesize (*SerialCommunicationConfig* attribute), 22
- ## C
- C (*LabJack.TemperatureUnit* attribute), 104
 C (*LabJack.ThermocoupleType* attribute), 104
 calibration_factor (*CryLasLaserConfig* attribute), 78
 calibration_mode() (*FuGProbusVDIRegisters* property), 92
 cc_mode() (*FuGProbusVDIRegisters* property), 92
 ceel6 (*GeneralSockets* attribute), 44, 62
 channels_enabled() (*TiePieOscilloscope* property), 156
 check_for_error() (*NewportSMC100PPSerialCommunication* method), 120
 check_generic_type() (in module *hvl_ccb.utils.typing*), 165
 check_master_slave_config() (*PSI9000* method), 81
 chunk_size (*VisaCommunicationConfig* attribute), 27
 clean_amplitude() (*TiePieGeneratorConfig* method), 155
 clean_auto_resolution_mode() (*TiePieOscilloscopeConfig* static method), 159
 clean_coupling() (*TiePieOscilloscopeChannelConfig* static method), 158
 clean_enabled() (*TiePieGeneratorConfig* static method), 155
 clean_enabled() (*TiePieOscilloscopeChannelConfig* static method), 158
 clean_frequency() (*TiePieGeneratorConfig* method), 155
 clean_input_range() (*TiePieOscilloscopeChannelConfig* method), 158
 clean_offset() (*TiePieGeneratorConfig* method), 155
 clean_pre_sample_ratio() (*TiePieOscilloscopeConfig* method), 159
 clean_probe_offset() (*TiePieOscilloscopeChannelConfig* method), 158
 clean_record_length() (*TiePieOscilloscopeConfig* method), 159
 clean_resolution() (*TiePieOscilloscopeConfig* static method), 159
 clean_safe_ground_enabled() (*TiePieOscilloscopeChannelConfig* static method), 158
 clean_sample_frequency() (*TiePieOscilloscopeConfig* method), 159
 clean_signal_type() (*TiePieGeneratorConfig* static method), 155
 clean_trigger_enabled() (*TiePieOscilloscopeChannelConfig* static method), 158
 clean_trigger_hysteresis() (*TiePieOscilloscopeChannelConfig* method), 158
 clean_trigger_kind() (*TiePieOscilloscopeChannelConfig* static method), 158
 clean_trigger_level() (*TiePieOscilloscopeChannelConfig* method), 158
 clean_trigger_level_mode() (*TiePieOscilloscopeChannelConfig* static method), 158
 clean_trigger_timeout() (*TiePieOscilloscopeConfig* method), 159
 clean_values() (*AsyncCommunicationProtocolConfig* method), 11
 clean_values() (*CryLasAttenuatorConfig* method), 72
 clean_values() (*CryLasLaserConfig* method), 78
 clean_values() (*EmptyConfig* method), 70, 166
 clean_values() (*FuGConfig* method), 87
 clean_values() (*HeinzingerConfig* method), 97
 clean_values() (*ILS2TConfig* method), 141
 clean_values() (*LJMCommunicationConfig* method), 15
 clean_values() (*LuminoxConfig* method), 145
 clean_values() (*MBW973Config* method), 109
 clean_values() (*ModbusTcpCommunicationConfig* method), 17
 clean_values() (*NewportSMC100PPConfig* method), 119
 clean_values() (*OpcUaCommunicationConfig* method), 19
 clean_values() (*PfeifferTPGConfig* method), 127
 clean_values() (*PSI9000Config* method), 83
 clean_values() (*SerialCommunicationConfig* method), 22
 clean_values() (*SupercubeConfiguration* method), 33, 57
 clean_values() (*TechnixConfig* method), 150
 clean_values() (*TelnetCommunicationConfig* method), 25

`clean_values()` (*TiePieDeviceConfig* method), 154
`clean_values()` (*VisaCommunicationConfig* method), 28
`close` (*EarthingStickOperation* attribute), 43
`close()` (*CommunicationProtocol* method), 12
`close()` (*LJMCommunication* method), 13
`close()` (*ModbusTcpCommunication* method), 16
`close()` (*NullCommunicationProtocol* method), 12
`close()` (*OpcUaCommunication* method), 18
`close()` (*SerialCommunication* method), 21
`close()` (*TelnetCommunication* method), 24
`close()` (*VisaCommunication* method), 26
`close_shutter()` (*CryLasLaser* method), 75
`CLOSED` (*CryLasLaser.AnswersShutter* attribute), 74
`CLOSED` (*CryLasLaserShutterStatus* attribute), 81
`closed` (*DoorStatus* attribute), 41, 60
`closed` (*EarthingStickStatus* attribute), 43, 61
`CMD_EXEC_ERROR` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_CC` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_CONFIGURATION` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_DISABLE` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_HOMING` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_MOVING` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_NOT_REFERENCED` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_PP` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMD_NOT_ALLOWED_READY` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`CMR` (*PfeifferTPG.SensorTypes* attribute), 125
`CODE_OR_ADDR_INVALID` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`collect_measurement_data()` (*TiePieOscilloscope* method), 156
`com()` (*SingleCommDevice* property), 71
`COM_TIMEOUT` (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
`command()` (*FuGProbusIV* method), 90
`command()` (*LuminoxMeasurementType* property), 146
`communication_channel` (*TechnixConfig* attribute), 150
`CommunicationProtocol` (class in *hvl_ccb.comm.base*), 12
`CONFIG` (*FuGProbusVRegisterGroups* attribute), 93
`CONFIG` (*NewportSMC100PP.StateMessages* attribute), 112
`CONFIG` (*NewportStates* attribute), 123
`config()` (*ConfigurationMixin* property), 165
`config_cls()` (*AsyncCommunicationProtocol* static method), 9
`config_cls()` (*ConfigurationMixin* static method), 166
`config_cls()` (*CryLasAttenuator* static method), 72
`config_cls()` (*CryLasAttenuatorSerialCommunication* static method), 73
`config_cls()` (*CryLasLaser* static method), 75
`config_cls()` (*CryLasLaserSerialCommunication* static method), 79
`config_cls()` (*Device* static method), 69
`config_cls()` (*FuGProbusIV* static method), 90
`config_cls()` (*FuGSerialCommunication* static method), 94
`config_cls()` (*HeinzingerDI* static method), 97
`config_cls()` (*HeinzingerSerialCommunication* static method), 100
`config_cls()` (*ILS2T* static method), 138
`config_cls()` (*ILS2TModbusTcpCommunication* static method), 142
`config_cls()` (*LJMCommunication* static method), 13
`config_cls()` (*Luminox* static method), 144
`config_cls()` (*LuminoxSerialCommunication* static method), 147
`config_cls()` (*MBW973* static method), 107
`config_cls()` (*MBW973SerialCommunication* static method), 109
`config_cls()` (*ModbusTcpCommunication* static method), 16
`config_cls()` (*NewportSMC100PP* static method), 113
`config_cls()` (*NewportSMC100PPSerialCommunication* static method), 121
`config_cls()` (*NullCommunicationProtocol* static method), 12
`config_cls()` (*OpcUaCommunication* static method), 18
`config_cls()` (*PfeifferTPG* static method), 125
`config_cls()` (*PfeifferTPGSerialCommunication* static method), 125

- static method*), 128
- `config_cls()` (*PSI9000 static method*), 81
- `config_cls()` (*PSI9000VisaCommunication static method*), 84
- `config_cls()` (*RTO1024 static method*), 131
- `config_cls()` (*RTO1024VisaCommunication static method*), 136
- `config_cls()` (*SerialCommunication static method*), 21
- `config_cls()` (*Supercube2015Base static method*), 54
- `config_cls()` (*SupercubeAOpcUaCommunication static method*), 49, 67
- `config_cls()` (*SupercubeBase static method*), 29
- `config_cls()` (*SupercubeBOpcUaCommunication static method*), 52
- `config_cls()` (*SupercubeOpcUaCommunication static method*), 34, 57
- `config_cls()` (*SyncCommunicationProtocol static method*), 12
- `config_cls()` (*Technix static method*), 148
- `config_cls()` (*TechnixSerialCommunication static method*), 151
- `config_cls()` (*TechnixTelnetCommunication static method*), 152
- `config_cls()` (*TelnetCommunication static method*), 24
- `config_cls()` (*TiePieOscilloscope static method*), 156
- `config_cls()` (*VisaCommunication static method*), 26
- `config_cls()` (*VisaDevice static method*), 162
- `config_gen` (*TiePieGeneratorMixin attribute*), 155
- `config_gen` (*TiePieHS5 attribute*), 156
- `config_i2c` (*TiePieI2CHostMixin attribute*), 156
- `config_i2c` (*TiePieWS5 attribute*), 160
- `config_osc` (*TiePieOscilloscope attribute*), 157
- `config_osc_channel_dict` (*TiePieOscilloscope attribute*), 157
- `config_status()` (*FuG property*), 86
- `configdataclass()` (in *module hvl_ccb.configuration*), 166
- `configuration_save_json()` (*Configuration-Mixin method*), 166
- `ConfigurationMixin` (class in *hvl_ccb.configuration*), 165
- `connection_type` (*LJMCommunicationConfig attribute*), 15
- `contact_range()` (*GeneralSupport class method*), 44
- `coupling()` (*TiePieOscilloscopeChannelConfig property*), 158
- `CR` (*FuGTerminators attribute*), 96
- `create_serial_port()` (*SerialCommunication-Config method*), 22
- `create_telnet()` (*TelnetCommunicationConfig method*), 25
- `CRLF` (*FuGTerminators attribute*), 96
- `CryLasAttenuator` (class in *hvl_ccb.dev.crylas*), 72
- `CryLasAttenuatorConfig` (class in *hvl_ccb.dev.crylas*), 72
- `CryLasAttenuatorError`, 73
- `CryLasAttenuatorSerialCommunication` (class in *hvl_ccb.dev.crylas*), 73
- `CryLasAttenuatorSerialCommunicationConfig` (class in *hvl_ccb.dev.crylas*), 73
- `CryLasLaser` (class in *hvl_ccb.dev.crylas*), 74
- `CryLasLaser.AnswersShutter` (class in *hvl_ccb.dev.crylas*), 74
- `CryLasLaser.AnswersStatus` (class in *hvl_ccb.dev.crylas*), 74
- `CryLasLaser.LaserStatus` (class in *hvl_ccb.dev.crylas*), 75
- `CryLasLaser.RepetitionRates` (class in *hvl_ccb.dev.crylas*), 75
- `CryLasLaserConfig` (class in *hvl_ccb.dev.crylas*), 77
- `CryLasLaserError`, 78
- `CryLasLaserNotReadyError`, 78
- `CryLasLaserPoller` (class in *hvl_ccb.dev.crylas*), 78
- `CryLasLaserSerialCommunication` (class in *hvl_ccb.dev.crylas*), 79
- `CryLasLaserSerialCommunicationConfig` (class in *hvl_ccb.dev.crylas*), 79
- `CryLasLaserShutterStatus` (class in *hvl_ccb.dev.crylas*), 81
- `CURRENT` (*FuGProbusIVCommands attribute*), 90
- `CURRENT` (*FuGReadbackChannels attribute*), 94
- `current()` (*FuG property*), 86
- `current()` (*Technix property*), 148
- `current_lower_limit` (*PSI9000Config attribute*), 83
- `current_monitor()` (*FuG property*), 86
- `current_primary` (*Power attribute*), 47, 64
- `current_upper_limit` (*PSI9000Config attribute*), 83
- `CurrentPosition` (*New-portSMC100PPConfig.HomeSearch attribute*), 118
- `cv_mode()` (*FuGProbusVDIRegisters property*), 92
- D**
- `datachange_notification()` (*OpcUaSubHandler method*), 20
- `datachange_notification()` (*SupercubeSubscriptionHandler method*), 35, 58
- `date_of_manufacture` (*LuminosMeasurementType attribute*), 146
- `DC` (*TiePieGeneratorSignalType attribute*), 155

DC_DoubleStage_280kV (*PowerSetup* attribute), 47, 64
 DC_SingleStage_140kV (*PowerSetup* attribute), 47, 65
 DC_VOLTAGE_TOO_LOW (*NewportSMC100PP.MotorErrors* attribute), 112
 DCA (*TiePieOscilloscopeChannelCoupling* attribute), 159
 DCV (*TiePieOscilloscopeChannelCoupling* attribute), 159
 default_com_cls() (*CryLasAttenuator* static method), 72
 default_com_cls() (*CryLasLaser* static method), 75
 default_com_cls() (*FuGProbusIV* static method), 90
 default_com_cls() (*HeinzingerDI* static method), 97
 default_com_cls() (*ILS2T* static method), 139
 default_com_cls() (*LabJack* static method), 104
 default_com_cls() (*Luminos* static method), 144
 default_com_cls() (*MBW973* static method), 107
 default_com_cls() (*NewportSMC100PP* static method), 113
 default_com_cls() (*PfeifferTPG* static method), 125
 default_com_cls() (*PSI9000* static method), 82
 default_com_cls() (*RTO1024* static method), 131
 default_com_cls() (*SingleCommDevice* static method), 71
 default_com_cls() (*Supercube2015Base* static method), 54
 default_com_cls() (*Supercube2015WithFU* static method), 66
 default_com_cls() (*SupercubeB* static method), 52
 default_com_cls() (*SupercubeBase* static method), 29
 default_com_cls() (*SupercubeWithFU* static method), 50
 default_com_cls() (*Technix* method), 149
 default_com_cls() (*TiePieOscilloscope* static method), 157
 default_com_cls() (*VisaDevice* static method), 162
 DEFAULT_IO_SCANNING_CONTROL_VALUES (*ILS2T* attribute), 138
 default_n_attempts_read_text_nonempty (*AsyncCommunicationProtocolConfig* attribute), 11
 default_n_attempts_read_text_nonempty (*FuGSerialCommunicationConfig* attribute), 95
 default_n_attempts_read_text_nonempty (*HeinzingerSerialCommunicationConfig* attribute), 101
 default_number_of_recordings (*HeinzingerConfig* attribute), 97
 Device (class in *hvl_ccb.dev.base*), 69
 device_type (*LJMCommunicationConfig* attribute), 15
 DeviceExistingException, 69
 DeviceFailuresException, 69
 devices_failed_start (*DeviceSequenceMixin* attribute), 70
 devices_failed_start (*ExperimentManager* attribute), 167
 devices_failed_stop (*DeviceSequenceMixin* attribute), 70
 devices_failed_stop (*ExperimentManager* attribute), 167
 DeviceSequenceMixin (class in *hvl_ccb.dev.base*), 69
 di() (*FuG* property), 86
 digital_control() (*FuGProbusVDIRegisters* property), 92
 DIOChannel (*LabJack* attribute), 103
 DISABLE (*NewportStates* attribute), 123
 disable() (*ILS2T* method), 139
 DISABLE_FROM_JOGGING (*NewportSMC100PP.StateMessages* attribute), 112
 DISABLE_FROM_MOVING (*NewportSMC100PP.StateMessages* attribute), 112
 DISABLE_FROM_READY (*NewportSMC100PP.StateMessages* attribute), 112
 DISABLED (*TiePieOscilloscopeAutoResolutionModes* attribute), 157
 DisableEspStageCheck (*NewportSMC100PPConfig.EspStageConfig* attribute), 118
 DISPLACEMENT_OUT_OF_LIMIT (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
 display_message_board() (*SupercubeBase* method), 29
 display_status_board() (*SupercubeBase* method), 29
 do_ioscanning_write() (*ILS2T* method), 139
 Door (class in *hvl_ccb.dev.supercube.constants*), 41
 DoorStatus (class in *hvl_ccb.dev.supercube.constants*), 41
 DoorStatus (class in *hvl_ccb.dev.supercube2015.constants*), 60

E

E (*LabJack.ThermocoupleType* attribute), 104

E0 (*FuErrorcodes* attribute), 88

E1 (*FuErrorcodes* attribute), 88

E10 (*FuErrorcodes* attribute), 88

E100 (*FuErrorcodes* attribute), 88

E106 (*FuErrorcodes* attribute), 88

E11 (*FuErrorcodes* attribute), 88

E115 (*FuErrorcodes* attribute), 88

E12 (*FuErrorcodes* attribute), 88

E125 (*FuErrorcodes* attribute), 88

E13 (*FuErrorcodes* attribute), 88

E135 (*FuErrorcodes* attribute), 88

E14 (*FuErrorcodes* attribute), 88

E145 (*FuErrorcodes* attribute), 88

E15 (*FuErrorcodes* attribute), 88

E155 (*FuErrorcodes* attribute), 88

E16 (*FuErrorcodes* attribute), 88

E165 (*FuErrorcodes* attribute), 89

E2 (*FuErrorcodes* attribute), 89

E206 (*FuErrorcodes* attribute), 89

E306 (*FuErrorcodes* attribute), 89

E4 (*FuErrorcodes* attribute), 89

E5 (*FuErrorcodes* attribute), 89

E504 (*FuErrorcodes* attribute), 89

E505 (*FuErrorcodes* attribute), 89

E6 (*FuErrorcodes* attribute), 89

E666 (*FuErrorcodes* attribute), 89

E7 (*FuErrorcodes* attribute), 89

E8 (*FuErrorcodes* attribute), 89

E9 (*FuErrorcodes* attribute), 89

EarthingRod (class
hvl_ccb.dev.supercube.constants), 41

EarthingRodStatus (class
hvl_ccb.dev.supercube.constants), 41

EarthingStick (class
hvl_ccb.dev.supercube.constants), 41

EarthingStick (class
hvl_ccb.dev.supercube2015.constants), 60

EarthingStickMeta (class
hvl_ccb.dev.supercube.constants), 43

EarthingStickOperatingStatus (class
hvl_ccb.dev.supercube.constants), 43

EarthingStickOperation (class
hvl_ccb.dev.supercube.constants), 43

EarthingStickStatus (class
hvl_ccb.dev.supercube.constants), 43

EarthingStickStatus (class
hvl_ccb.dev.supercube2015.constants), 61

EEPROM_ACCESS_ERROR (New-
portSMC100PPSerialCommunication.ControllerErrors
attribute), 120

EIGHT (*HeinzingerConfig.RecordingsEnum* attribute),
96

EIGHT_BIT (*TiePieOscilloscopeResolution* attribute),
159

EIGHT_HUNDRED_MILLI_VOLT (*TiePieOscilloscop-
eRange* attribute), 159

EIGHT_VOLT (*TiePieOscilloscopeRange* attribute), 159

EIGHTBITS (*SerialCommunicationBytesize* attribute),
22

EIGHTY_VOLT (*TiePieOscilloscopeRange* attribute),
159

EmptyConfig (class in hvl_ccb.configuration), 166

EmptyConfig (class in hvl_ccb.dev.base), 70

enable() (*ILS2T* method), 139

enabled() (*TiePieGeneratorConfig* property), 155

enabled() (*TiePieOscilloscopeChannelConfig* prop-
erty), 158

EnableEspStageCheck (New-
portSMC100PPConfig.EspStageConfig at-
tribute), 118

encoding (*AsyncCommunicationProtocolConfig*
attribute), 11

encoding_error_handling (*AsyncCommunica-
tionProtocolConfig* attribute), 11

EndOfRunSwitch (New-
portSMC100PPConfig.HomeSearch attribute),
118

EndOfRunSwitch_and_Index (New-
portSMC100PPConfig.HomeSearch attribute),
118

endpoint_name (*OpcUaCommunicationConfig* at-
tribute), 19

endpoint_name (*SupercubeAOpcUaConfiguration* at-
tribute), 49, 68

endpoint_name (*SupercubeBOpcUaConfiguration* at-
tribute), 52

error (*DoorStatus* attribute), 41, 60

error (*EarthingStickStatus* attribute), 43, 61

ERROR (*ExperimentStatus* attribute), 168

Error (*SafetyStatus* attribute), 48, 65

errorcode (*FuGError* attribute), 88

Errors (class in hvl_ccb.dev.supercube.constants), 44

Errors (class in hvl_ccb.dev.supercube2015.constants),
62

ESP_STAGE_NAME_INVALID (New-
portSMC100PPSerialCommunication.ControllerErrors
attribute), 120

ETHERNET (*LJMCommunicationConfig.ConnectionType*
attribute), 14

EVEN (*SerialCommunicationParity* attribute), 23

event_notification() (*OpcUaSubHandler*
method), 20

EXECUTE (*FuGProbusIVCommands* attribute), 90

execute_absolute_position() (*ILS2T* method),
139

execute_on_x() (*FuGProbusVConfigRegisters* prop-

- erty), 91
- execute_relative_step() (*ILS2T method*), 139
- EXECUTEONX (*FuGProbusIVCommands attribute*), 90
- exit_configuration() (*NewportSMC100PP method*), 113
- exit_configuration_wait_sec (*NewportSMC100PPConfig attribute*), 119
- experiment_blocked (*EarthingRodStatus attribute*), 41
- experiment_ready (*EarthingRodStatus attribute*), 41
- ExperimentError, 167
- ExperimentManager (class *hvl_ccb.experiment_manager*), 167
- ExperimentStatus (class *hvl_ccb.experiment_manager*), 168
- External (*PowerSetup attribute*), 47, 65
- ## F
- F (*LabJack.TemperatureUnit attribute*), 104
- failures (*DeviceFailuresException attribute*), 69
- FALLING (*TiePieOscilloscopeTriggerKind attribute*), 160
- FAST (*ILS2T.Ref16Jog attribute*), 138
- file_copy() (*RTO1024 method*), 131
- finish() (*ExperimentManager method*), 167
- FINISHED (*ExperimentStatus attribute*), 168
- FINISHING (*ExperimentStatus attribute*), 168
- FIRMWARE (*FuGReadbackChannels attribute*), 94
- FIVEBITS (*SerialCommunicationBytesize attribute*), 22
- FLT_INFO (*ILS2TRegAddr attribute*), 143
- FLT_MEM_DEL (*ILS2TRegAddr attribute*), 143
- FLT_MEM_RESET (*ILS2TRegAddr attribute*), 143
- FOLLOWING_ERROR (*NewportSMC100PP.MotorErrors attribute*), 112
- FOLLOWRAMP (*FuGRampModes attribute*), 94
- force_trigger() (*TiePieOscilloscope method*), 157
- force_value() (*AsyncCommunicationProtocolConfig method*), 11
- force_value() (*CryLasAttenuatorConfig method*), 72
- force_value() (*CryLasAttenuatorSerialCommunicationConfig method*), 74
- force_value() (*CryLasLaserConfig method*), 78
- force_value() (*CryLasLaserSerialCommunicationConfig method*), 80
- force_value() (*EmptyConfig method*), 70, 166
- force_value() (*FuGConfig method*), 87
- force_value() (*FuGSerialCommunicationConfig method*), 95
- force_value() (*HeinzingerConfig method*), 97
- force_value() (*HeinzingerSerialCommunicationConfig method*), 101
- force_value() (*ILS2TConfig method*), 141
- force_value() (*ILS2TModbusTcpCommunicationConfig method*), 142
- force_value() (*LJMCommunicationConfig method*), 15
- force_value() (*LuminosConfig method*), 145
- force_value() (*LuminosSerialCommunicationConfig method*), 147
- force_value() (*MBW973Config method*), 109
- force_value() (*MBW973SerialCommunicationConfig method*), 110
- force_value() (*ModbusTcpCommunicationConfig method*), 17
- in force_value() (*NewportSMC100PPConfig method*), 119
- in force_value() (*NewportSMC100PPSerialCommunicationConfig method*), 123
- force_value() (*OpcUaCommunicationConfig method*), 19
- force_value() (*PfeifferTPGConfig method*), 127
- force_value() (*PfeifferTPGSerialCommunicationConfig method*), 129
- force_value() (*PSI9000Config method*), 83
- force_value() (*PSI9000VisaCommunicationConfig method*), 84
- force_value() (*RTO1024Config method*), 136
- force_value() (*RTO1024VisaCommunicationConfig method*), 137
- force_value() (*SerialCommunicationConfig method*), 22
- force_value() (*SupercubeAOpcUaConfiguration method*), 49, 68
- force_value() (*SupercubeBOpcUaConfiguration method*), 52
- force_value() (*SupercubeConfiguration method*), 33, 57
- force_value() (*SupercubeOpcUaCommunicationConfig method*), 34, 57
- force_value() (*TechnixCommunicationConfig method*), 149
- force_value() (*TechnixConfig method*), 150
- force_value() (*TechnixSerialCommunicationConfig method*), 151
- force_value() (*TechnixTelnetCommunicationConfig method*), 152
- force_value() (*TelnetCommunicationConfig method*), 25
- force_value() (*TiePieDeviceConfig method*), 154
- force_value() (*VisaCommunicationConfig method*), 28
- force_value() (*VisaDeviceConfig method*), 163
- FORTY_VOLT (*TiePieOscilloscopeRange attribute*), 159
- FOUR (*HeinzingerConfig.RecordingsEnum attribute*), 96
- FOUR_HUNDRED_MILLI_VOLT (*TiePieOscilloscope*

- eRange* attribute), 159
- FOUR_VOLT (*TiePieOscilloscopeRange* attribute), 159
- FOURTEEN_BIT (*TiePieOscilloscopeResolution* attribute), 160
- FREERUN (*RTO1024.TriggerModes* attribute), 130
- frequency (*Power* attribute), 47, 64
- frequency () (*TiePieGeneratorConfig* property), 155
- FRM (*NewportConfigCommands* attribute), 111
- from_json () (*ConfigurationMixin* class method), 166
- FRS (*NewportConfigCommands* attribute), 111
- fso_reset () (*Supercube2015WithFU* method), 67
- fso_reset () (*SupercubeWithFU* method), 50
- FuG (class in *hvl_ccb.dev.fug*), 85
- FuGConfig (class in *hvl_ccb.dev.fug*), 87
- FuGDigitalVal (class in *hvl_ccb.dev.fug*), 88
- FuGError, 88
- FuGErrorcodes (class in *hvl_ccb.dev.fug*), 88
- FuGMonitorModes (class in *hvl_ccb.dev.fug*), 89
- FuGPolarities (class in *hvl_ccb.dev.fug*), 89
- FuGProbusIV (class in *hvl_ccb.dev.fug*), 89
- FuGProbusIVCommands (class in *hvl_ccb.dev.fug*), 90
- FuGProbusV (class in *hvl_ccb.dev.fug*), 91
- FuGProbusVConfigRegisters (class in *hvl_ccb.dev.fug*), 91
- FuGProbusVDIRegisters (class in *hvl_ccb.dev.fug*), 92
- FuGProbusVDORegisters (class in *hvl_ccb.dev.fug*), 92
- FuGProbusVMonitorRegisters (class in *hvl_ccb.dev.fug*), 92
- FuGProbusVRegisterGroups (class in *hvl_ccb.dev.fug*), 93
- FuGProbusVSetRegisters (class in *hvl_ccb.dev.fug*), 93
- FuGRampModes (class in *hvl_ccb.dev.fug*), 94
- FuGReadbackChannels (class in *hvl_ccb.dev.fug*), 94
- FuGSerialCommunication (class in *hvl_ccb.dev.fug*), 94
- FuGSerialCommunicationConfig (class in *hvl_ccb.dev.fug*), 94
- FuGTerminators (class in *hvl_ccb.dev.fug*), 96
- G**
- GeneralSockets (class in *hvl_ccb.dev.supercube.constants*), 44
- GeneralSockets (class in *hvl_ccb.dev.supercube2015.constants*), 62
- GeneralSupport (class in *hvl_ccb.dev.supercube.constants*), 44
- GeneralSupport (class in *hvl_ccb.dev.supercube2015.constants*), 62
- GeneralSupportMeta (class in *hvl_ccb.dev.supercube.constants*), 45
- GENERATOR (*TiePieDeviceType* attribute), 154
- generator_start () (*TiePieGeneratorMixin* method), 155
- generator_stop () (*TiePieGeneratorMixin* method), 155
- GeneratorParameterLimits (class in *hvl_ccb.dev.tiepie*), 153
- get () (*AlarmText* class method), 36, 59
- get () (*MeasurementsDividerRatio* class method), 63
- get () (*MeasurementsScaledInput* class method), 64
- get_acceleration () (*NewportSMC100PP* method), 113
- get_acquire_length () (*RTO1024* method), 131
- get_ain () (*LabJack* method), 104
- get_by_p_id () (*LabJack.DeviceType* class method), 103
- get_by_p_id () (*LJMCommunicationConfig.DeviceType* class method), 15
- get_cal_current_source () (*LabJack* method), 104
- get_ceel6_socket () (*Supercube2015Base* method), 54
- get_ceel6_socket () (*SupercubeBase* method), 29
- get_channel_offset () (*RTO1024* method), 131
- get_channel_position () (*RTO1024* method), 131
- get_channel_range () (*RTO1024* method), 131
- get_channel_scale () (*RTO1024* method), 131
- get_channel_state () (*RTO1024* method), 132
- get_controller_information () (*NewportSMC100PP* method), 113
- get_current () (*HeinzingerDI* method), 98
- get_dc_volt () (*ILS2T* method), 139
- get_device () (*DeviceSequenceMixin* method), 70
- get_device_by_serial_number () (in module *hvl_ccb.dev.tiepie*), 160
- get_devices () (*DeviceSequenceMixin* method), 70
- get_digital_input () (*LabJack* method), 104
- get_door_status () (*Supercube2015Base* method), 54
- get_door_status () (*SupercubeBase* method), 29
- get_earthing_manual () (*Supercube2015Base* method), 54
- get_earthing_rod_status () (*SupercubeBase* method), 30
- get_earthing_status () (*Supercube2015Base* method), 54
- get_earthing_stick_manual () (*SupercubeBase* method), 30
- get_earthing_stick_operating_status () (*SupercubeBase* method), 30
- get_earthing_stick_status () (*SupercubeBase*

`method`), 30
`get_error_code()` (*ILS2T method*), 139
`get_error_queue()` (*VisaDevice method*), 162
`get_frequency()` (*Supercube2015WithFU method*), 67
`get_frequency()` (*SupercubeWithFU method*), 50
`get_fso_active()` (*Supercube2015WithFU method*), 67
`get_fso_active()` (*SupercubeWithFU method*), 50
`get_full_scale_mbar()` (*PfeifferTPG method*), 125
`get_full_scale_unitless()` (*PfeifferTPG method*), 125
`get_identification()` (*VisaDevice method*), 162
`get_interface_version()` (*HeinzingerDI method*), 98
`get_max_voltage()` (*Supercube2015WithFU method*), 67
`get_max_voltage()` (*SupercubeWithFU method*), 50
`get_measurement_ratio()` (*Supercube2015Base method*), 54
`get_measurement_ratio()` (*SupercubeBase method*), 30
`get_measurement_voltage()` (*Supercube2015Base method*), 54
`get_measurement_voltage()` (*SupercubeBase method*), 30
`get_motor_configuration()` (*NewportSMC100PP method*), 114
`get_move_duration()` (*NewportSMC100PP method*), 114
`get_negative_software_limit()` (*NewportSMC100PP method*), 114
`get_number_of_recordings()` (*HeinzingerDI method*), 98
`get_output()` (*PSI9000 method*), 82
`get_position()` (*ILS2T method*), 139
`get_position()` (*NewportSMC100PP method*), 114
`get_positive_software_limit()` (*NewportSMC100PP method*), 114
`get_power_setup()` (*Supercube2015WithFU method*), 67
`get_power_setup()` (*SupercubeWithFU method*), 50
`get_primary_current()` (*Supercube2015WithFU method*), 67
`get_primary_current()` (*SupercubeWithFU method*), 50
`get_primary_voltage()` (*Supercube2015WithFU method*), 67
`get_primary_voltage()` (*SupercubeWithFU method*), 51
`get_product_id()` (*LabJack method*), 105
`get_product_name()` (*LabJack method*), 105
`get_product_type()` (*LabJack method*), 105
`get_pulse_energy_and_rate()` (*CryLasLaser method*), 75
`get_reference_point()` (*RTO1024 method*), 132
`get_register()` (*FuGProbusV method*), 91
`get_repetitions()` (*RTO1024 method*), 132
`get_sbus_rh()` (*LabJack method*), 105
`get_sbus_temp()` (*LabJack method*), 105
`get_serial_number()` (*HeinzingerDI method*), 98
`get_serial_number()` (*LabJack method*), 105
`get_state()` (*NewportSMC100PP method*), 115
`get_status()` (*ILS2T method*), 140
`get_status()` (*Supercube2015Base method*), 55
`get_status()` (*SupercubeBase method*), 30
`get_status_byte()` (*Technix method*), 149
`get_support_input()` (*Supercube2015Base method*), 55
`get_support_input()` (*SupercubeBase method*), 30
`get_support_output()` (*Supercube2015Base method*), 55
`get_support_output()` (*SupercubeBase method*), 31
`get_system_lock()` (*PSI9000 method*), 82
`get_t13_socket()` (*Supercube2015Base method*), 55
`get_t13_socket()` (*SupercubeBase method*), 31
`get_target_voltage()` (*Supercube2015WithFU method*), 67
`get_target_voltage()` (*SupercubeWithFU method*), 51
`get_temperature()` (*ILS2T method*), 140
`get_timestamps()` (*RTO1024 method*), 132
`get_ui_lower_limits()` (*PSI9000 method*), 82
`get_ui_upper_limits()` (*PSI9000 method*), 82
`get_voltage()` (*HeinzingerDI method*), 98
`get_voltage_current_setpoint()` (*PSI9000 method*), 82
`go_home()` (*NewportSMC100PP method*), 115
`go_to_configuration()` (*NewportSMC100PP method*), 115
`GreenNotReady` (*SafetyStatus attribute*), 48, 65
`GreenReady` (*SafetyStatus attribute*), 48, 65

H

HARDWARE (*CryLasLaser.RepetitionRates attribute*), 75
`has_safe_ground()` (*TiePieOscilloscopeChannel-Config property*), 158
HEAD (*CryLasLaser.AnswersStatus attribute*), 74
HeinzingerConfig (*class in hvl_ccb.dev.heinzinger*), 96
HeinzingerConfig.RecordingsEnum (*class in hvl_ccb.dev.heinzinger*), 96

HeinzingerDI (*class in hvl_ccb.dev.heinzinger*), 97
HeinzingerDI.OutputStatus (*class in hvl_ccb.dev.heinzinger*), 97
HeinzingerPNC (*class in hvl_ccb.dev.heinzinger*), 99
HeinzingerPNC.UnitCurrent (*class in hvl_ccb.dev.heinzinger*), 99
HeinzingerPNC.UnitVoltage (*class in hvl_ccb.dev.heinzinger*), 99
HeinzingerPNCDeviceNotRecognizedException, 100
HeinzingerPNCError, 100
HeinzingerPNCMaxCurrentExceededException, 100
HeinzingerPNCMaxVoltageExceededException, 100
HeinzingerSerialCommunication (*class in hvl_ccb.dev.heinzinger*), 100
HeinzingerSerialCommunicationConfig (*class in hvl_ccb.dev.heinzinger*), 100
HIGH (*LabJack.DIOStatus attribute*), 103
high_resolution() (*FuGProbusVSetRegisters property*), 93
home_search_polling_interval (*NewportSMC100PPConfig attribute*), 119
home_search_timeout (*NewportSMC100PPConfig attribute*), 119
home_search_type (*NewportSMC100PPConfig attribute*), 119
home_search_velocity (*NewportSMC100PPConfig attribute*), 119
HOME_STARTED (*NewportSMC100PPSerialCommunication.ControllerErrors attribute*), 120
HomeSwitch (*NewportSMC100PPConfig.HomeSearch attribute*), 118
HomeSwitch_and_Index (*NewportSMC100PPConfig.HomeSearch attribute*), 118
HOMING (*NewportStates attribute*), 123
HOMING_FROM_RS232 (*NewportSMC100PP.StateMessages attribute*), 112
HOMING_FROM_SMC (*NewportSMC100PP.StateMessages attribute*), 112
HOMING_TIMEOUT (*NewportSMC100PP.MotorErrors attribute*), 112
horn (*Safety attribute*), 65
horn() (*Supercube2015Base method*), 55
host (*ModbusTcpCommunicationConfig attribute*), 17
host (*OpcUaCommunicationConfig attribute*), 19
host (*TelnetCommunicationConfig attribute*), 25
host (*VisaCommunicationConfig attribute*), 28
hPascal (*PfeifferTPG.PressureUnits attribute*), 124
HT (*NewportConfigCommands attribute*), 111
hv() (*Technix property*), 149
hvl_ccb
 module, 168
hvl_ccb.comm
 module, 29
hvl_ccb.comm.base
 module, 9
hvl_ccb.comm.labjack_ljm
 module, 13
hvl_ccb.comm.modbus_tcp
 module, 16
hvl_ccb.comm.opc
 module, 18
hvl_ccb.comm.serial
 module, 21
hvl_ccb.comm.telnet
 module, 24
hvl_ccb.comm.visa
 module, 26
hvl_ccb.configuration
 module, 165
hvl_ccb.dev
 module, 164
hvl_ccb.dev.base
 module, 69
hvl_ccb.dev.crylas
 module, 71
hvl_ccb.dev.ea_psi9000
 module, 81
hvl_ccb.dev.fug
 module, 85
hvl_ccb.dev.heinzinger
 module, 96
hvl_ccb.dev.labjack
 module, 102
hvl_ccb.dev.mbw973
 module, 107
hvl_ccb.dev.newport
 module, 111
hvl_ccb.dev.pfeiffer_tpg
 module, 124
hvl_ccb.dev.rs_rto1024
 module, 130
hvl_ccb.dev.se_ils2t
 module, 137
hvl_ccb.dev.sst_luminos
 module, 144
hvl_ccb.dev.supercube
 module, 53
hvl_ccb.dev.supercube.base
 module, 29
hvl_ccb.dev.supercube.constants
 module, 35

- `hvl_ccb.dev.supercube.typ_a`
module, 49
- `hvl_ccb.dev.supercube.typ_b`
module, 52
- `hvl_ccb.dev.supercube2015`
module, 68
- `hvl_ccb.dev.supercube2015.base`
module, 53
- `hvl_ccb.dev.supercube2015.constants`
module, 59
- `hvl_ccb.dev.supercube2015.typ_a`
module, 66
- `hvl_ccb.dev.technix`
module, 148
- `hvl_ccb.dev.tiepie`
module, 153
- `hvl_ccb.dev.utils`
module, 161
- `hvl_ccb.dev.visa`
module, 162
- `hvl_ccb.experiment_manager`
module, 167
- `hvl_ccb.utils`
module, 165
- `hvl_ccb.utils.enum`
module, 164
- `hvl_ccb.utils.typing`
module, 165
- `hysteresis_compensation` (NewportSMC100PPPConfig attribute), 119
- I**
- `I2C` (TiePieDeviceType attribute), 154
- `I2CHostParameterLimits` (class in `hvl_ccb.dev.tiepie`), 153
- `ID` (FuGProbusIVCommands attribute), 90
- `Identification_error` (PfeifferTPG.SensorStatus attribute), 125
- `identifier` (LJMCommunicationConfig attribute), 15
- `identify_device()` (FuG method), 86
- `identify_device()` (HeinzingerPNC method), 99
- `identify_sensors()` (PfeifferTPG method), 126
- `IKR` (PfeifferTPG.SensorTypes attribute), 125
- `IKR11` (PfeifferTPG.SensorTypes attribute), 125
- `IKR9` (PfeifferTPG.SensorTypes attribute), 125
- `ILS2T` (class in `hvl_ccb.dev.se_ils2t`), 137
- `ILS2T.ActionsPtp` (class in `hvl_ccb.dev.se_ils2t`), 138
- `ILS2T.Mode` (class in `hvl_ccb.dev.se_ils2t`), 138
- `ILS2T.Ref16Jog` (class in `hvl_ccb.dev.se_ils2t`), 138
- `ILS2T.State` (class in `hvl_ccb.dev.se_ils2t`), 138
- `ILS2TConfig` (class in `hvl_ccb.dev.se_ils2t`), 141
- `ILS2TException`, 142
- `ILS2TModbusTcpCommunication` (class in `hvl_ccb.dev.se_ils2t`), 142
- `ILS2TModbusTcpCommunicationConfig` (class in `hvl_ccb.dev.se_ils2t`), 142
- `ILS2TRegAddr` (class in `hvl_ccb.dev.se_ils2t`), 143
- `ILS2TRegDatatype` (class in `hvl_ccb.dev.se_ils2t`), 143
- `IMMEDIATELY` (FuGRampModes attribute), 94
- `IMR` (PfeifferTPG.SensorTypes attribute), 125
- `in_1_1` (GeneralSupport attribute), 44, 62
- `in_1_2` (GeneralSupport attribute), 44, 62
- `in_2_1` (GeneralSupport attribute), 44, 62
- `in_2_2` (GeneralSupport attribute), 44, 62
- `in_3_1` (GeneralSupport attribute), 44, 62
- `in_3_2` (GeneralSupport attribute), 44, 62
- `in_4_1` (GeneralSupport attribute), 44, 62
- `in_4_2` (GeneralSupport attribute), 44, 62
- `in_5_1` (GeneralSupport attribute), 44, 62
- `in_5_2` (GeneralSupport attribute), 44, 62
- `in_6_1` (GeneralSupport attribute), 45, 62
- `in_6_2` (GeneralSupport attribute), 45, 62
- `INACTIVE` (CryLasLaser.AnswersStatus attribute), 75
- `inactive` (DoorStatus attribute), 41, 60
- `inactive` (EarthingStickStatus attribute), 43, 61
- `inhibit()` (Technix property), 149
- `init_attenuation` (CryLasAttenuatorConfig attribute), 72
- `init_monitored_nodes()` (OpcUaCommunication method), 18
- `init_shutter_status` (CryLasLaserConfig attribute), 78
- `initialize()` (NewportSMC100PPP method), 115
- `INITIALIZED` (ExperimentStatus attribute), 168
- `INITIALIZING` (ExperimentStatus attribute), 168
- `Initializing` (SafetyStatus attribute), 48, 65
- `INPUT` (FuGProbusVRegisterGroups attribute), 93
- `input()` (GeneralSupport class method), 45, 63
- `input_1` (MeasurementsDividerRatio attribute), 45, 63
- `input_1` (MeasurementsScaledInput attribute), 46, 64
- `input_2` (MeasurementsDividerRatio attribute), 46
- `input_2` (MeasurementsScaledInput attribute), 46, 64
- `input_3` (MeasurementsDividerRatio attribute), 46
- `input_3` (MeasurementsScaledInput attribute), 46, 64
- `input_4` (MeasurementsDividerRatio attribute), 46
- `input_4` (MeasurementsScaledInput attribute), 46, 64
- `input_range()` (TiePieOscilloscopeChannelConfig property), 158
- `INT32` (ILS2TRegDatatype attribute), 143
- `interface_type` (PSI9000VisaCommunicationConfig attribute), 85
- `interface_type` (RTO1024VisaCommunicationConfig attribute), 137
- `interface_type` (VisaCommunicationConfig attribute), 28

internal (*LabJack.CjcType attribute*), 103
 Internal (*PowerSetup attribute*), 47, 65
 InvalidSupercubeStatusError, 53
 IO_SCANNING (*ILS2TRegAddr attribute*), 143
 IoScanningModeValueError, 143
 is_configdataclass (*AsyncCommunicationProtocolConfig attribute*), 11
 is_configdataclass (*CryLasAttenuatorConfig attribute*), 73
 is_configdataclass (*CryLasLaserConfig attribute*), 78
 is_configdataclass (*EmptyConfig attribute*), 71, 166
 is_configdataclass (*FuGConfig attribute*), 87
 is_configdataclass (*HeinzingerConfig attribute*), 97
 is_configdataclass (*ILS2TConfig attribute*), 141
 is_configdataclass (*LJMCommunicationConfig attribute*), 15
 is_configdataclass (*LuminoxConfig attribute*), 145
 is_configdataclass (*MBW973Config attribute*), 109
 is_configdataclass (*ModbusTcpCommunicationConfig attribute*), 17
 is_configdataclass (*NewportSMC100PPConfig attribute*), 119
 is_configdataclass (*OpcUaCommunicationConfig attribute*), 19
 is_configdataclass (*PfeifferTPGConfig attribute*), 127
 is_configdataclass (*SupercubeConfiguration attribute*), 33, 57
 is_configdataclass (*TechnixConfig attribute*), 150
 is_configdataclass (*TiePieDeviceConfig attribute*), 154
 is_configdataclass (*VisaCommunicationConfig attribute*), 28
 is_data_ready_polling_interval_sec (*TiePieDeviceConfig attribute*), 154
 is_done() (*MBW973 method*), 108
 is_error() (*ExperimentManager method*), 168
 is_finished() (*ExperimentManager method*), 168
 is_generic_type_hint() (in module *hvl_ccb.utils.typing*), 165
 is_in_range() (*ILS2TRegDatatype method*), 143
 is_inactive() (*CryLasLaser.LaserStatus property*), 75
 is_measurement_data_ready() (*TiePieOscilloscope method*), 157
 is_open() (*LJMCommunication property*), 13
 is_open() (*OpcUaCommunication property*), 18
 is_open() (*SerialCommunication property*), 21
 is_open() (*TelnetCommunication property*), 24

is_polling() (*Poller method*), 161
 is_ready() (*CryLasLaser.LaserStatus property*), 75
 is_running() (*ExperimentManager method*), 168
 is_triggered() (*TiePieOscilloscope method*), 157
 is_valid_scale_range_reversed_str() (*PfeifferTPGConfig.Model method*), 127

J

J (*LabJack.ThermocoupleType attribute*), 104
 jerk_time (*NewportSMC100PPConfig attribute*), 119
 JOG (*ILS2T.Mode attribute*), 138
 jog_run() (*ILS2T method*), 140
 jog_stop() (*ILS2T method*), 140
 JOGGING (*NewportStates attribute*), 123
 JOGGING_FROM_DISABLE (*NewportSMC100PP.StateMessages attribute*), 112
 JOGGING_FROM_READY (*NewportSMC100PP.StateMessages attribute*), 112
 JOGN_FAST (*ILS2TRegAddr attribute*), 143
 JOGN_SLOW (*ILS2TRegAddr attribute*), 143
 JR (*NewportConfigCommands attribute*), 111

K

K (*LabJack.TemperatureUnit attribute*), 104
 K (*LabJack.ThermocoupleType attribute*), 104
 keys() (*AsyncCommunicationProtocolConfig class method*), 11
 keys() (*CryLasAttenuatorConfig class method*), 73
 keys() (*CryLasAttenuatorSerialCommunicationConfig class method*), 74
 keys() (*CryLasLaserConfig class method*), 78
 keys() (*CryLasLaserSerialCommunicationConfig class method*), 80
 keys() (*EmptyConfig class method*), 71, 166
 keys() (*FuGConfig class method*), 87
 keys() (*FuGSerialCommunicationConfig class method*), 95
 keys() (*HeinzingerConfig class method*), 97
 keys() (*HeinzingerSerialCommunicationConfig class method*), 101
 keys() (*ILS2TConfig class method*), 141
 keys() (*ILS2TModbusTcpCommunicationConfig class method*), 142
 keys() (*LJMCommunicationConfig class method*), 15
 keys() (*LuminoxConfig class method*), 145
 keys() (*LuminoxSerialCommunicationConfig class method*), 147
 keys() (*MBW973Config class method*), 109
 keys() (*MBW973SerialCommunicationConfig class method*), 110
 keys() (*ModbusTcpCommunicationConfig class method*), 17

- keys () (*NewportSMC100PPConfig* class method), 119
 keys () (*NewportSMC100PPSerialCommunicationConfig* class method), 123
 keys () (*OpcUaCommunicationConfig* class method), 19
 keys () (*PfeifferTPGConfig* class method), 127
 keys () (*PfeifferTPGSerialCommunicationConfig* class method), 129
 keys () (*PSI9000Config* class method), 84
 keys () (*PSI9000VisaCommunicationConfig* class method), 85
 keys () (*RTO1024Config* class method), 136
 keys () (*RTO1024VisaCommunicationConfig* class method), 137
 keys () (*SerialCommunicationConfig* class method), 23
 keys () (*SupercubeAOpcUaConfiguration* class method), 50, 68
 keys () (*SupercubeBOpcUaConfiguration* class method), 53
 keys () (*SupercubeConfiguration* class method), 33, 57
 keys () (*SupercubeOpcUaCommunicationConfig* class method), 34, 58
 keys () (*TechnixCommunicationConfig* class method), 149
 keys () (*TechnixConfig* class method), 150
 keys () (*TechnixSerialCommunicationConfig* class method), 151
 keys () (*TechnixTelnetCommunicationConfig* class method), 152
 keys () (*TelnetCommunicationConfig* class method), 25
 keys () (*TiePieDeviceConfig* class method), 154
 keys () (*VisaCommunicationConfig* class method), 28
 keys () (*VisaDeviceConfig* class method), 163
 kV (*HeinzingerPNC.UnitVoltage* attribute), 99
- ## L
- LabJack (class in *hvl_ccb.dev.labjack*), 102
 LabJack.AInRange (class in *hvl_ccb.dev.labjack*), 102
 LabJack.CalMicroAmpere (class in *hvl_ccb.dev.labjack*), 103
 LabJack.CjcType (class in *hvl_ccb.dev.labjack*), 103
 LabJack.DeviceType (class in *hvl_ccb.dev.labjack*), 103
 LabJack.DIOStatus (class in *hvl_ccb.dev.labjack*), 103
 LabJack.TemperatureUnit (class in *hvl_ccb.dev.labjack*), 103
 LabJack.ThermocoupleType (class in *hvl_ccb.dev.labjack*), 104
 LabJackError, 107
 LabJackIdentifierDIOError, 107
 laser_off () (*CryLasLaser* method), 76
 laser_on () (*CryLasLaser* method), 76
 LF (*FuGTerminators* attribute), 96
 LFCR (*FuGTerminators* attribute), 96
 line_1 (*MessageBoard* attribute), 46
 line_10 (*MessageBoard* attribute), 46
 line_11 (*MessageBoard* attribute), 46
 line_12 (*MessageBoard* attribute), 46
 line_13 (*MessageBoard* attribute), 46
 line_14 (*MessageBoard* attribute), 46
 line_15 (*MessageBoard* attribute), 46
 line_2 (*MessageBoard* attribute), 46
 line_3 (*MessageBoard* attribute), 46
 line_4 (*MessageBoard* attribute), 46
 line_5 (*MessageBoard* attribute), 46
 line_6 (*MessageBoard* attribute), 46
 line_7 (*MessageBoard* attribute), 46
 line_8 (*MessageBoard* attribute), 46
 line_9 (*MessageBoard* attribute), 46
 list_devices () (*TiePieOscilloscope* static method), 157
 list_directory () (*RTO1024* method), 132
 live (*OpcControl* attribute), 46
 LJMCommunication (class in *hvl_ccb.comm.labjack_ljm*), 13
 LJMCommunicationConfig (class in *hvl_ccb.comm.labjack_ljm*), 14
 LJMCommunicationConfig.ConnectionType (class in *hvl_ccb.comm.labjack_ljm*), 14
 LJMCommunicationConfig.DeviceType (class in *hvl_ccb.comm.labjack_ljm*), 14
 LJMCommunicationError, 15
 lm34 (*LabJack.CjcType* attribute), 103
 load_configuration () (*RTO1024* method), 132
 local_display () (*RTO1024* method), 132
 locked (*DoorStatus* attribute), 41, 60
 log_set () (in module *hvl_ccb.dev.tiepie*), 161
 LOW (*LabJack.DIOStatus* attribute), 103
 Luminos (class in *hvl_ccb.dev.sst_luminos*), 144
 LuminosConfig (class in *hvl_ccb.dev.sst_luminos*), 145
 LuminosMeasurementType (class in *hvl_ccb.dev.sst_luminos*), 146
 LuminosMeasurementTypeDict (in module *hvl_ccb.dev.sst_luminos*), 146
 LuminosMeasurementTypeError, 146
 LuminosMeasurementTypeValue (in module *hvl_ccb.dev.sst_luminos*), 146
 LuminosOutputMode (class in *hvl_ccb.dev.sst_luminos*), 146
 LuminosOutputModeError, 147
 LuminosSerialCommunication (class in *hvl_ccb.dev.sst_luminos*), 147
 LuminosSerialCommunicationConfig (class in *hvl_ccb.dev.sst_luminos*), 147

M

- mA (*HeinzingerPNC.UnitCurrent* attribute), 99
- manual (*EarthingStickOperatingStatus* attribute), 43
- manual() (*EarthingStick* class method), 42, 60
- manual_1 (*EarthingStick* attribute), 42, 60
- manual_2 (*EarthingStick* attribute), 42, 60
- manual_3 (*EarthingStick* attribute), 42, 60
- manual_4 (*EarthingStick* attribute), 42, 60
- manual_5 (*EarthingStick* attribute), 42, 60
- manual_6 (*EarthingStick* attribute), 42, 60
- manuals() (*EarthingStick* class method), 42
- MARK (*SerialCommunicationParity* attribute), 23
- max_current (*TechnixConfig* attribute), 150
- max_current() (*FuG* property), 86
- max_current() (*HeinzingerPNC* property), 99
- max_current() (*Technix* property), 149
- max_current_hardware() (*FuG* property), 86
- max_current_hardware() (*HeinzingerPNC* property), 99
- max_timeout_retry_nr (*OpcUaCommunicationConfig* attribute), 19
- max_voltage (*TechnixConfig* attribute), 150
- max_voltage() (*FuG* property), 86
- max_voltage() (*HeinzingerPNC* property), 99
- max_voltage() (*Technix* property), 149
- max_voltage_hardware() (*FuG* property), 86
- max_voltage_hardware() (*HeinzingerPNC* property), 100
- mbar (*PfeifferTPG.PressureUnits* attribute), 124
- MBW973 (class in *hvl_ccb.dev.mbw973*), 107
- MBW973Config (class in *hvl_ccb.dev.mbw973*), 108
- MBW973ControlRunningException, 109
- MBW973Error, 109
- MBW973PumpRunningException, 109
- MBW973SerialCommunication (class in *hvl_ccb.dev.mbw973*), 109
- MBW973SerialCommunicationConfig (class in *hvl_ccb.dev.mbw973*), 109
- measure() (*PfeifferTPG* method), 126
- measure_all() (*PfeifferTPG* method), 126
- measure_current() (*HeinzingerDI* method), 98
- measure_voltage() (*HeinzingerDI* method), 98
- measure_voltage_current() (*PSI9000* method), 82
- MeasurementsDividerRatio (class in *hvl_ccb.dev.supercube.constants*), 45
- MeasurementsDividerRatio (class in *hvl_ccb.dev.supercube2015.constants*), 63
- MeasurementsScaledInput (class in *hvl_ccb.dev.supercube.constants*), 46
- MeasurementsScaledInput (class in *hvl_ccb.dev.supercube2015.constants*), 63
- message (*Errors* attribute), 44
- MessageBoard (class in *hvl_ccb.dev.supercube.constants*), 46
- micro_step_per_full_step_factor (*NewportSMC100PPConfig* attribute), 119
- Micron (*PfeifferTPG.PressureUnits* attribute), 124
- ModbusTcpCommunication (class in *hvl_ccb.comm.modbus_tcp*), 16
- ModbusTcpCommunicationConfig (class in *hvl_ccb.comm.modbus_tcp*), 17
- ModbusTcpConnectionFailedException, 17
- model (*PfeifferTPGConfig* attribute), 127
- module
 - hvl_ccb*, 168
 - hvl_ccb.comm*, 29
 - hvl_ccb.comm.base*, 9
 - hvl_ccb.comm.labjack_ljm*, 13
 - hvl_ccb.comm.modbus_tcp*, 16
 - hvl_ccb.comm.opc*, 18
 - hvl_ccb.comm.serial*, 21
 - hvl_ccb.comm.telnet*, 24
 - hvl_ccb.comm.visa*, 26
 - hvl_ccb.configuration*, 165
 - hvl_ccb.dev*, 164
 - hvl_ccb.dev.base*, 69
 - hvl_ccb.dev.crylas*, 71
 - hvl_ccb.dev.ea_psi9000*, 81
 - hvl_ccb.dev.fug*, 85
 - hvl_ccb.dev.heinzinger*, 96
 - hvl_ccb.dev.labjack*, 102
 - hvl_ccb.dev.mbw973*, 107
 - hvl_ccb.dev.newport*, 111
 - hvl_ccb.dev.pfeiffer_tpg*, 124
 - hvl_ccb.dev.rs_rto1024*, 130
 - hvl_ccb.dev.se_ils2t*, 137
 - hvl_ccb.dev.sst_luminos*, 144
 - hvl_ccb.dev.supercube*, 53
 - hvl_ccb.dev.supercube.base*, 29
 - hvl_ccb.dev.supercube.constants*, 35
 - hvl_ccb.dev.supercube.typ_a*, 49
 - hvl_ccb.dev.supercube.typ_b*, 52
 - hvl_ccb.dev.supercube2015*, 68
 - hvl_ccb.dev.supercube2015.base*, 53
 - hvl_ccb.dev.supercube2015.constants*, 59
 - hvl_ccb.dev.supercube2015.typ_a*, 66
 - hvl_ccb.dev.technix*, 148
 - hvl_ccb.dev.tiepie*, 153
 - hvl_ccb.dev.utils*, 161
 - hvl_ccb.dev.visa*, 162
 - hvl_ccb.experiment_manager*, 167
 - hvl_ccb.utils*, 165
 - hvl_ccb.utils.enum*, 164
 - hvl_ccb.utils.typing*, 165
- MONITOR_I (*FuGProbusVRegisterGroups* attribute), 93

MONITOR_V (*FuGProbusVRegisterGroups* attribute), 93
 most_recent_error() (*FuGProbusVConfigRegisters* property), 91
 motion_distance_per_full_step (*NewportSMC100PPConfig* attribute), 119
 motor_config() (*NewportSMC100PPConfig* property), 119
 move_to_absolute_position() (*NewportSMC100PP* method), 115
 move_to_relative_position() (*NewportSMC100PP* method), 116
 move_wait_sec (*NewportSMC100PPConfig* attribute), 119
 MOVING (*NewportSMC100PP.StateMessages* attribute), 112
 MOVING (*NewportStates* attribute), 123
 MS_NOMINAL_CURRENT (*PSI9000* attribute), 81
 MS_NOMINAL_VOLTAGE (*PSI9000* attribute), 81
 msb_first() (*TechnixStatusByte* method), 152
 MULTI_COMMANDS_MAX (*VisaCommunication* attribute), 26
 MULTI_COMMANDS_SEPARATOR (*VisaCommunication* attribute), 26

N

n_channels() (*TiePieOscilloscope* property), 157
 n_max_try_get_device (*TiePieDeviceConfig* attribute), 154
 NameEnum (*class in hvl_ccb.utils.enum*), 164
 NAMES (*SerialCommunicationParity* attribute), 23
 names() (*RTO1024.TriggerModes* class method), 130
 namespace_index (*SupercubeConfiguration* attribute), 33, 57
 NATIVEONLY (*TiePieOscilloscopeAutoResolutionModes* attribute), 158
 NED_END_OF_TURN (*NewportSMC100PP.MotorErrors* attribute), 112
 NEG (*ILS2T.Ref16Jog* attribute), 138
 NEG_FAST (*ILS2T.Ref16Jog* attribute), 138
 NEGATIVE (*FuGPolarities* attribute), 89
 negative_software_limit (*NewportSMC100PPConfig* attribute), 119
 NewportConfigCommands (*class in hvl_ccb.dev.newport*), 111
 NewportControllerError, 111
 NewportMotorError, 111
 NewportMotorPowerSupplyWasCutError, 111
 NewportSerialCommunicationError, 123
 NewportSMC100PP (*class in hvl_ccb.dev.newport*), 112
 NewportSMC100PP.MotorErrors (*class in hvl_ccb.dev.newport*), 112
 NewportSMC100PP.StateMessages (*class in hvl_ccb.dev.newport*), 112
 NewportSMC100PPConfig (*class in hvl_ccb.dev.newport*), 118
 NewportSMC100PPConfig.EspStageConfig (*class in hvl_ccb.dev.newport*), 118
 NewportSMC100PPConfig.HomeSearch (*class in hvl_ccb.dev.newport*), 118
 NewportSMC100PPSerialCommunication (*class in hvl_ccb.dev.newport*), 120
 NewportSMC100PPSerialCommunication.ControllerErrors (*class in hvl_ccb.dev.newport*), 120
 NewportSMC100PPSerialCommunicationConfig (*class in hvl_ccb.dev.newport*), 122
 NewportStates (*class in hvl_ccb.dev.newport*), 123
 NewportUncertainPositionError, 124
 NO (*FuGDigitalVal* attribute), 88
 NO_ERROR (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
 NO_REF (*NewportStates* attribute), 123
 NO_REF_ESP_STAGE_ERROR (*NewportSMC100PP.StateMessages* attribute), 112
 NO_REF_FROM_CONFIG (*NewportSMC100PP.StateMessages* attribute), 112
 NO_REF_FROM_DISABLED (*NewportSMC100PP.StateMessages* attribute), 112
 NO_REF_FROM_HOMING (*NewportSMC100PP.StateMessages* attribute), 112
 NO_REF_FROM_JOGGING (*NewportSMC100PP.StateMessages* attribute), 112
 NO_REF_FROM_MOVING (*NewportSMC100PP.StateMessages* attribute), 112
 NO_REF_FROM_READY (*NewportSMC100PP.StateMessages* attribute), 113
 NO_REF_FROM_RESET (*NewportSMC100PP.StateMessages* attribute), 113
 No_sensor (*PfeifferTPG.SensorStatus* attribute), 125
 NOISE (*TiePieGeneratorSignalType* attribute), 155
 NONE (*ILS2T.Ref16Jog* attribute), 138
 NONE (*LabJack.ThermocoupleType* attribute), 104
 None (*PfeifferTPG.SensorTypes* attribute), 125
 NONE (*SerialCommunicationParity* attribute), 23
 NoPower (*PowerSetup* attribute), 48
 NORMAL (*RTO1024.TriggerModes* attribute), 130
 noSen (*PfeifferTPG.SensorTypes* attribute), 125
 noSENSOR (*PfeifferTPG.SensorTypes* attribute), 125
 not_defined (*AlarmText* attribute), 36, 59
 nr_trials_activate (*LuminoxConfig* attribute),

- 145
- NullCommunicationProtocol (class in *hvl_ccb.comm.base*), 12
- Number (in module *hvl_ccb.utils.typing*), 165
- number() (*EarthingStick* property), 42
- number_of_decimals (*HeinzingerConfig* attribute), 97
- number_of_sensors() (*PfeifferTPG* property), 126
- ## O
- ODD (*SerialCommunicationParity* attribute), 23
- OFF (*FuGDigitalVal* attribute), 88
- OFF (*HeinzingerDI.OutputStatus* attribute), 97
- offset() (*TiePieGeneratorConfig* property), 155
- OH (*NewportConfigCommands* attribute), 111
- Ok (*PfeifferTPG.SensorStatus* attribute), 125
- ON (*FuGDigitalVal* attribute), 88
- ON (*HeinzingerDI.OutputStatus* attribute), 97
- ON (*ILS2T.State* attribute), 138
- on() (*FuG* property), 86
- on() (*FuGProbusVDIRegisters* property), 92
- ONE (*HeinzingerConfig.RecordingsEnum* attribute), 96
- ONE (*LabJack.AInRange* attribute), 102
- ONE (*SerialCommunicationStopbits* attribute), 23
- ONE_HUNDREDTH (*LabJack.AInRange* attribute), 102
- ONE_POINT_FIVE (*SerialCommunicationStopbits* attribute), 24
- ONE_TENTH (*LabJack.AInRange* attribute), 103
- ONLYUPWARDSOFFTOZERO (*FuGRampModes* attribute), 94
- OpcControl (class in *hvl_ccb.dev.supercube.constants*), 46
- OpcUaCommunication (class in *hvl_ccb.comm.opc*), 18
- OpcUaCommunicationConfig (class in *hvl_ccb.comm.opc*), 19
- OpcUaCommunicationIOError, 20
- OpcUaCommunicationTimeoutError, 20
- OpcUaSubHandler (class in *hvl_ccb.comm.opc*), 20
- open (*DoorStatus* attribute), 41, 60
- open (*EarthingStickOperation* attribute), 43
- open (*EarthingStickStatus* attribute), 43, 62
- open() (*CommunicationProtocol* method), 12
- open() (*LJMCommunication* method), 13
- open() (*ModbusTcpCommunication* method), 16
- open() (*NullCommunicationProtocol* method), 12
- open() (*OpcUaCommunication* method), 18
- open() (*SerialCommunication* method), 21
- open() (*TelnetCommunication* method), 24
- open() (*VisaCommunication* method), 26
- open_shutter() (*CryLasLaser* method), 76
- open_timeout (*VisaCommunicationConfig* attribute), 28
- OPENED (*CryLasLaser.AnswersShutter* attribute), 74
- OPENED (*CryLasLaserShutterStatus* attribute), 81
- operate() (*Supercube2015Base* method), 55
- operate() (*SupercubeBase* method), 31
- operate_earthing_stick() (*SupercubeBase* method), 31
- operating_status() (*EarthingStick* class method), 42
- operating_status_1 (*EarthingStick* attribute), 42
- operating_status_2 (*EarthingStick* attribute), 42
- operating_status_3 (*EarthingStick* attribute), 42
- operating_status_4 (*EarthingStick* attribute), 42
- operating_status_5 (*EarthingStick* attribute), 42
- operating_status_6 (*EarthingStick* attribute), 42
- operating_statuses() (*EarthingStick* class method), 42
- optional_defaults() (*AsyncCommunicationProtocolConfig* class method), 11
- optional_defaults() (*CryLasAttenuatorConfig* class method), 73
- optional_defaults() (*CryLasAttenuatorSerialCommunicationConfig* class method), 74
- optional_defaults() (*CryLasLaserConfig* class method), 78
- optional_defaults() (*CryLasLaserSerialCommunicationConfig* class method), 80
- optional_defaults() (*EmptyConfig* class method), 71, 166
- optional_defaults() (*FuGConfig* class method), 87
- optional_defaults() (*FuGSerialCommunicationConfig* class method), 95
- optional_defaults() (*HeinzingerConfig* class method), 97
- optional_defaults() (*HeinzingerSerialCommunicationConfig* class method), 101
- optional_defaults() (*ILS2TConfig* class method), 141
- optional_defaults() (*ILS2TModbusTcpCommunicationConfig* class method), 142
- optional_defaults() (*LJMCommunicationConfig* class method), 15
- optional_defaults() (*LuminoxConfig* class method), 145
- optional_defaults() (*LuminoxSerialCommunicationConfig* class method), 147
- optional_defaults() (*MBW973Config* class method), 109
- optional_defaults() (*MBW973SerialCommunicationConfig* class method), 110
- optional_defaults() (*ModbusTcpCommunicationConfig* class method), 17
- optional_defaults() (*NewportSMC100PPConfig*

class method), 119
 optional_defaults() (NewportSMC100PPSerialCommunicationConfig *class method*), 123
 optional_defaults() (OpcUaCommunicationConfig *class method*), 19
 optional_defaults() (PfeifferTPGConfig *class method*), 127
 optional_defaults() (PfeifferTPGSerialCommunicationConfig *class method*), 129
 optional_defaults() (PSI9000Config *class method*), 84
 optional_defaults() (PSI9000VisaCommunicationConfig *class method*), 85
 optional_defaults() (RTO1024Config *class method*), 136
 optional_defaults() (RTO1024VisaCommunicationConfig *class method*), 137
 optional_defaults() (SerialCommunicationConfig *class method*), 23
 optional_defaults() (SupercubeAOpcUaConfiguration *class method*), 50, 68
 optional_defaults() (SupercubeBOpcUaConfiguration *class method*), 53
 optional_defaults() (SupercubeConfiguration *class method*), 33, 57
 optional_defaults() (SupercubeOpcUaCommunicationConfig *class method*), 34, 58
 optional_defaults() (TechnixCommunicationConfig *class method*), 149
 optional_defaults() (TechnixConfig *class method*), 150
 optional_defaults() (TechnixSerialCommunicationConfig *class method*), 151
 optional_defaults() (TechnixTelnetCommunicationConfig *class method*), 152
 optional_defaults() (TelnetCommunicationConfig *class method*), 25
 optional_defaults() (TiePieDeviceConfig *class method*), 154
 optional_defaults() (VisaCommunicationConfig *class method*), 28
 optional_defaults() (VisaDeviceConfig *class method*), 163
 OSCILLOSCOPE (TiePieDeviceType *attribute*), 154
 OscilloscopeChannelParameterLimits (*class in hvl_ccb.dev.tiepie*), 153
 OscilloscopeParameterLimits (*class in hvl_ccb.dev.tiepie*), 153
 OT (NewportConfigCommands *attribute*), 111
 out() (FuGProbusVDORegisters *property*), 92
 out_1_1 (GeneralSupport *attribute*), 45, 63
 out_1_2 (GeneralSupport *attribute*), 45, 63
 out_2_1 (GeneralSupport *attribute*), 45, 63
 out_2_2 (GeneralSupport *attribute*), 45, 63
 out_3_1 (GeneralSupport *attribute*), 45, 63
 out_3_2 (GeneralSupport *attribute*), 45, 63
 out_4_1 (GeneralSupport *attribute*), 45, 63
 out_4_2 (GeneralSupport *attribute*), 45, 63
 out_5_1 (GeneralSupport *attribute*), 45, 63
 out_5_2 (GeneralSupport *attribute*), 45, 63
 out_6_1 (GeneralSupport *attribute*), 45, 63
 out_6_2 (GeneralSupport *attribute*), 45, 63
 OUTPUT (FuGProbusIVCommands *attribute*), 90
 output() (GeneralSupport *class method*), 45, 63
 output_off() (FuGProbusIV *method*), 90
 output_off() (HeinzingerDI *method*), 98
 output_on() (HeinzingerDI *method*), 98
 OUTPUT_POWER_EXCEEDED (NewportSMC100PP.MotorErrors *attribute*), 112
 output_status() (HeinzingerDI *property*), 98
 OUTPUTONCMD (FuGProbusVRegisterGroups *attribute*), 93
 OUTPUTX0 (FuGProbusVRegisterGroups *attribute*), 93
 OUTPUTX1 (FuGProbusVRegisterGroups *attribute*), 93
 OUTPUTX2 (FuGProbusVRegisterGroups *attribute*), 93
 OUTPUTXCMD (FuGProbusVRegisterGroups *attribute*), 93
 outX0() (FuG *property*), 86
 outX1() (FuG *property*), 86
 outX2() (FuG *property*), 87
 outXCMD() (FuG *property*), 87
 Overrange (PfeifferTPG.SensorStatus *attribute*), 125

P

PARAM_MISSING_OR_INVALID (NewportSMC100PPSerialCommunication.ControllerErrors *attribute*), 120
 parity (CryLasAttenuatorSerialCommunicationConfig *attribute*), 74
 parity (CryLasLaserSerialCommunicationConfig *attribute*), 80
 parity (FuGSerialCommunicationConfig *attribute*), 95
 parity (HeinzingerSerialCommunicationConfig *attribute*), 101
 parity (LuminosSerialCommunicationConfig *attribute*), 147
 parity (MBW973SerialCommunicationConfig *attribute*), 110
 parity (NewportSMC100PPSerialCommunicationConfig *attribute*), 123
 parity (PfeifferTPGSerialCommunicationConfig *attribute*), 129
 Parity (SerialCommunicationConfig *attribute*), 22
 parity (SerialCommunicationConfig *attribute*), 23

- parse_read_measurement_value() (*LuminosMeasurementType* method), 146
 partial_pressure_o2 (*LuminosMeasurementType* attribute), 146
 Pascal (*PfeifferTPG.PressureUnits* attribute), 124
 PBR (*PfeifferTPG.SensorTypes* attribute), 125
 PEAK_CURRENT_LIMIT (*NewportSMC100PP.MotorErrors* attribute), 112
 peak_output_current_limit (*NewportSMC100PPConfig* attribute), 119
 percent_o2 (*LuminosMeasurementType* attribute), 146
 PfeifferTPG (class in *hvl_ccb.dev.pfeiffer_tpg*), 124
 PfeifferTPG.PressureUnits (class in *hvl_ccb.dev.pfeiffer_tpg*), 124
 PfeifferTPG.SensorStatus (class in *hvl_ccb.dev.pfeiffer_tpg*), 124
 PfeifferTPG.SensorTypes (class in *hvl_ccb.dev.pfeiffer_tpg*), 125
 PfeifferTPGConfig (class in *hvl_ccb.dev.pfeiffer_tpg*), 127
 PfeifferTPGConfig.Model (class in *hvl_ccb.dev.pfeiffer_tpg*), 127
 PfeifferTPGError, 128
 PfeifferTPGSerialCommunication (class in *hvl_ccb.dev.pfeiffer_tpg*), 128
 PfeifferTPGSerialCommunicationConfig (class in *hvl_ccb.dev.pfeiffer_tpg*), 128
 PKR (*PfeifferTPG.SensorTypes* attribute), 125
 POLARITY (*FuGProbusIVCommands* attribute), 90
 Poller (class in *hvl_ccb.dev.utils*), 161
 polling (*LuminosOutputMode* attribute), 146
 polling_delay_sec (*SupercubeConfiguration* attribute), 33
 polling_interval (*MBW973Config* attribute), 109
 polling_interval_sec (*SupercubeConfiguration* attribute), 33
 polling_interval_sec (*TechnixConfig* attribute), 150
 polling_period (*CryLasLaserConfig* attribute), 78
 polling_timeout (*CryLasLaserConfig* attribute), 78
 port (*ModbusTcpCommunicationConfig* attribute), 17
 port (*OpcUaCommunicationConfig* attribute), 20
 port (*SerialCommunicationConfig* attribute), 23
 port (*SupercubeOpcUaCommunicationConfig* attribute), 58
 port (*TechnixTelnetCommunicationConfig* attribute), 152
 port (*TelnetCommunicationConfig* attribute), 25
 port (*VisaCommunicationConfig* attribute), 28
 port_range() (*GeneralSupport* class method), 45
 POS (*ILS2T.Ref16Jog* attribute), 138
 POS_END_OF_TURN (*NewportSMC100PP.MotorErrors* attribute), 112
 POS_FAST (*ILS2T.Ref16Jog* attribute), 138
 POSITION (*ILS2TRegAddr* attribute), 143
 POSITION_OUT_OF_LIMIT (*NewportSMC100PPSerialCommunication.ControllerErrors* attribute), 120
 POSITIVE (*FuGPolarities* attribute), 89
 positive_software_limit (*NewportSMC100PPConfig* attribute), 119
 post_force_value() (*NewportSMC100PPConfig* method), 119
 post_stop_pause_sec (*TechnixConfig* attribute), 150
 Power (class in *hvl_ccb.dev.supercube.constants*), 47
 Power (class in *hvl_ccb.dev.supercube2015.constants*), 64
 power_limit (*PSI9000Config* attribute), 84
 PowerSetup (class in *hvl_ccb.dev.supercube.constants*), 47
 PowerSetup (class in *hvl_ccb.dev.supercube2015.constants*), 64
 pre_sample_ratio() (*TiePieOscilloscopeConfig* property), 159
 prepare_ultra_segmentation() (*RTO1024* method), 133
 probe_offset() (*TiePieOscilloscopeChannelConfig* property), 158
 PSI9000 (class in *hvl_ccb.dev.ea_psi9000*), 81
 PSI9000Config (class in *hvl_ccb.dev.ea_psi9000*), 83
 PSI9000Error, 84
 PSI9000VisaCommunication (class in *hvl_ccb.dev.ea_psi9000*), 84
 PSI9000VisaCommunicationConfig (class in *hvl_ccb.dev.ea_psi9000*), 84
 PT100 (*LabJack.ThermocoupleType* attribute), 104
 PT1000 (*LabJack.ThermocoupleType* attribute), 104
 PT500 (*LabJack.ThermocoupleType* attribute), 104
 PTP (*ILS2T.Mode* attribute), 138
 PublicPropertiesReprMixin (class in *hvl_ccb.dev.tiepie*), 153
 PULSE (*TiePieGeneratorSignalType* attribute), 155
- ## Q
- QIL (*NewportConfigCommands* attribute), 111
 QUERY (*FuGProbusIVCommands* attribute), 90
 query() (*CryLasLaserSerialCommunication* method), 79
 query() (*FuGSerialCommunication* method), 94
 query() (*NewportSMC100PPSerialCommunication* method), 121
 query() (*PfeifferTPGSerialCommunication* method), 128
 query() (*SyncCommunicationProtocol* method), 12
 query() (*TechnixCommunication* method), 149

[query\(\)](#) (*VisaCommunication method*), 27
[query_all\(\)](#) (*CryLasLaserSerialCommunication method*), 79
[query_multiple\(\)](#) (*NewportSMC100PPSerialCommunication method*), 121
[query_polling\(\)](#) (*Luminos method*), 144
[QUICKSTOP](#) (*ILS2T.State attribute*), 138
[QuickStop](#) (*SafetyStatus attribute*), 48, 65
[quickstop\(\)](#) (*ILS2T method*), 140
[quit](#) (*Errors attribute*), 44, 62
[quit_error\(\)](#) (*Supercube2015Base method*), 55
[quit_error\(\)](#) (*SupercubeBase method*), 31

R

[R](#) (*LabJack.ThermocoupleType attribute*), 104
[raise_\(\)](#) (*FuGErrorcodes method*), 89
[RAMP_ACC](#) (*ILS2TRegAddr attribute*), 143
[RAMP_DECEL](#) (*ILS2TRegAddr attribute*), 143
[RAMP_N_MAX](#) (*ILS2TRegAddr attribute*), 143
[RAMP_TYPE](#) (*ILS2TRegAddr attribute*), 143
[rampmode\(\)](#) (*FuGProbusVSetRegisters property*), 93
[ramprate\(\)](#) (*FuGProbusVSetRegisters property*), 93
[rampstate\(\)](#) (*FuGProbusVSetRegisters property*), 93
[RAMPUPWARDS](#) (*FuGRampModes attribute*), 94
[range\(\)](#) (*EarthingStick class method*), 42
[RATEDCURRENT](#) (*FuGReadbackChannels attribute*), 94
[RATEDVOLTAGE](#) (*FuGReadbackChannels attribute*), 94
[read\(\)](#) (*AsyncCommunicationProtocol method*), 9
[read\(\)](#) (*CryLasLaserSerialCommunication method*), 79
[read\(\)](#) (*MBW973 method*), 108
[read\(\)](#) (*OpcUaCommunication method*), 18
[read\(\)](#) (*Supercube2015Base method*), 55
[read\(\)](#) (*SupercubeBase method*), 31
[read_all\(\)](#) (*AsyncCommunicationProtocol method*), 10
[read_bytes\(\)](#) (*AsyncCommunicationProtocol method*), 10
[read_bytes\(\)](#) (*SerialCommunication method*), 21
[read_bytes\(\)](#) (*TelnetCommunication method*), 24
[read_float\(\)](#) (*MBW973 method*), 108
[read_holding_registers\(\)](#) (*ModbusTcpCommunication method*), 16
[read_input_registers\(\)](#) (*ModbusTcpCommunication method*), 16
[read_int\(\)](#) (*MBW973 method*), 108
[read_measurement\(\)](#) (*RTO1024 method*), 133
[read_measurements\(\)](#) (*MBW973 method*), 108
[read_name\(\)](#) (*LJMCommunication method*), 13
[read_nonempty\(\)](#) (*AsyncCommunicationProtocol method*), 10
[read_resistance\(\)](#) (*LabJack method*), 105

[read_single_bytes\(\)](#) (*SerialCommunication method*), 21
[read_streaming\(\)](#) (*Luminos method*), 145
[read_termination](#) (*VisaCommunicationConfig attribute*), 28
[read_text\(\)](#) (*AsyncCommunicationProtocol method*), 10
[read_text\(\)](#) (*NewportSMC100PPSerialCommunication method*), 121
[read_text_nonempty\(\)](#) (*AsyncCommunicationProtocol method*), 10
[READ_TEXT_SKIP_PREFIXES](#) (*CryLasLaserSerialCommunication attribute*), 79
[read_thermocouple\(\)](#) (*LabJack method*), 105
[readback_data\(\)](#) (*FuGProbusVConfigRegisters property*), 91
[READBACKCHANNEL](#) (*FuGProbusIVCommands attribute*), 90
[READY](#) (*CryLasLaser.AnswersStatus attribute*), 75
[READY](#) (*ILS2T.State attribute*), 138
[READY](#) (*NewportStates attribute*), 123
[ready\(\)](#) (*Supercube2015Base method*), 55
[ready\(\)](#) (*SupercubeBase method*), 31
[READY_ACTIVE](#) (*CryLasLaser.LaserStatus attribute*), 75
[READY_FROM_DISABLE](#) (*NewportSMC100PP.StateMessages attribute*), 113
[READY_FROM_HOMING](#) (*NewportSMC100PP.StateMessages attribute*), 113
[READY_FROM_JOGGING](#) (*NewportSMC100PP.StateMessages attribute*), 113
[READY_FROM_MOVING](#) (*NewportSMC100PP.StateMessages attribute*), 113
[READY_INACTIVE](#) (*CryLasLaser.LaserStatus attribute*), 75
[record_length\(\)](#) (*TiePieOscilloscopeConfig property*), 159
[RedOperate](#) (*SafetyStatus attribute*), 48, 66
[RedReady](#) (*SafetyStatus attribute*), 48, 66
[reg_3\(\)](#) (*FuGProbusVDIRegisters property*), 92
[RegAddr](#) (*ILS2T attribute*), 138
[RegDatatype](#) (*ILS2T attribute*), 138
[register_pulse_time](#) (*TechnixConfig attribute*), 151
[RELATIVE](#) (*TiePieOscilloscopeTriggerLevelMode attribute*), 160
[RELATIVE_POSITION_MOTOR](#) (*ILS2T.ActionsPtp attribute*), 138
[RELATIVE_POSITION_TARGET](#) (*ILS2T.ActionsPtp attribute*), 138

`remote()` (*Technix* property), 149
`remove_device()` (*DeviceSequenceMixin* method), 70
`require_block_measurement_support` (*TiePieDeviceConfig* attribute), 154
`required_keys()` (*AsyncCommunicationProtocolConfig* class method), 12
`required_keys()` (*CryLasAttenuatorConfig* class method), 73
`required_keys()` (*CryLasAttenuatorSerialCommunicationConfig* class method), 74
`required_keys()` (*CryLasLaserConfig* class method), 78
`required_keys()` (*CryLasLaserSerialCommunicationConfig* class method), 80
`required_keys()` (*EmptyConfig* class method), 71, 166
`required_keys()` (*FuGConfig* class method), 87
`required_keys()` (*FuGSerialCommunicationConfig* class method), 95
`required_keys()` (*HeinzingerConfig* class method), 97
`required_keys()` (*HeinzingerSerialCommunicationConfig* class method), 101
`required_keys()` (*ILS2TConfig* class method), 141
`required_keys()` (*ILS2TModbusTcpCommunicationConfig* class method), 142
`required_keys()` (*LJMCommunicationConfig* class method), 15
`required_keys()` (*LuminoxConfig* class method), 145
`required_keys()` (*LuminoxSerialCommunicationConfig* class method), 148
`required_keys()` (*MBW973Config* class method), 109
`required_keys()` (*MBW973SerialCommunicationConfig* class method), 110
`required_keys()` (*ModbusTcpCommunicationConfig* class method), 17
`required_keys()` (*NewportSMC100PPConfig* class method), 119
`required_keys()` (*NewportSMC100PPSerialCommunicationConfig* class method), 123
`required_keys()` (*OpcUaCommunicationConfig* class method), 20
`required_keys()` (*PfeifferTPGConfig* class method), 128
`required_keys()` (*PfeifferTPGSerialCommunicationConfig* class method), 129
`required_keys()` (*PSI9000Config* class method), 84
`required_keys()` (*PSI9000VisaCommunicationConfig* class method), 85
`required_keys()` (*RTO1024Config* class method), 136
`required_keys()` (*RTO1024VisaCommunicationConfig* class method), 137
`required_keys()` (*SerialCommunicationConfig* class method), 23
`required_keys()` (*SupercubeAOpcUaConfiguration* class method), 50, 68
`required_keys()` (*SupercubeBOpcUaConfiguration* class method), 53
`required_keys()` (*SupercubeConfiguration* class method), 34, 57
`required_keys()` (*SupercubeOpcUaCommunicationConfig* class method), 34, 58
`required_keys()` (*TechnixCommunicationConfig* class method), 150
`required_keys()` (*TechnixConfig* class method), 151
`required_keys()` (*TechnixSerialCommunicationConfig* class method), 152
`required_keys()` (*TechnixTelnetCommunicationConfig* class method), 152
`required_keys()` (*TelnetCommunicationConfig* class method), 25
`required_keys()` (*TiePieDeviceConfig* class method), 154
`required_keys()` (*VisaCommunicationConfig* class method), 28
`required_keys()` (*VisaDeviceConfig* class method), 163
`reset` (*BreakdownDetection* attribute), 41, 60
`RESET` (*FuGProbusIVCommands* attribute), 90
`reset()` (*FuGProbusIV* method), 90
`reset()` (*NewportSMC100PP* method), 116
`reset()` (*VisaDevice* method), 162
`reset_error()` (*ILS2T* method), 140
`reset_interface()` (*HeinzingerDI* method), 98
`resolution()` (*TiePieOscilloscopeConfig* property), 159
`response_sleep_time` (*CryLasAttenuatorConfig* attribute), 73
`RISING` (*TiePieOscilloscopeTriggerKind* attribute), 160
`RISING_OR_FALLING` (*TiePieOscilloscopeTriggerKind* attribute), 160
`RMS_CURRENT_LIMIT` (*NewportSMC100PP.MotorErrors* attribute), 112
`rpm_max_init` (*ILS2TConfig* attribute), 142
`rs485_address` (*NewportSMC100PPConfig* attribute), 120
`RTO1024` (class in *hvl_ccb.dev.rs_rto1024*), 130
`RTO1024.TriggerModes` (class in *hvl_ccb.dev.rs_rto1024*), 130
`RTO1024Config` (class in *hvl_ccb.dev.rs_rto1024*), 136
`RTO1024Error`, 136

RTO1024VisaCommunication (class in *hvl_ccb.dev.rs_rto1024*), 136
 RTO1024VisaCommunicationConfig (class in *hvl_ccb.dev.rs_rto1024*), 136
 run() (*ExperimentManager* method), 168
 run_continuous_acquisition() (*RTO1024* method), 133
 run_single_acquisition() (*RTO1024* method), 133
 RUNNING (*ExperimentStatus* attribute), 168

S

S (*LabJack.ThermocoupleType* attribute), 104
 SA (*NewportConfigCommands* attribute), 111
 safe_ground_enabled() (*TiePieOscilloscopeChannelConfig* property), 158
 Safety (class in *hvl_ccb.dev.supercube.constants*), 48
 Safety (class in *hvl_ccb.dev.supercube2015.constants*), 65
 SafetyStatus (class in *hvl_ccb.dev.supercube.constants*), 48
 SafetyStatus (class in *hvl_ccb.dev.supercube2015.constants*), 65
 sample_frequency() (*TiePieOscilloscopeConfig* property), 159
 save_configuration() (*RTO1024* method), 133
 save_waveform_history() (*RTO1024* method), 133
 SCALE (*ILS2TRegAddr* attribute), 143
 ScalingFactorValueError, 143
 screw_scaling (*NewportSMC100PPConfig* attribute), 120
 send_command() (*NewportSMC100PPSerialCommunication* method), 122
 send_command() (*PfeifferTPGSerialCommunication* method), 128
 send_stop() (*NewportSMC100PPSerialCommunication* method), 122
 Sensor_error (*PfeifferTPG.SensorStatus* attribute), 125
 Sensor_off (*PfeifferTPG.SensorStatus* attribute), 125
 sensor_status (*LuminoxMeasurementType* attribute), 146
 serial_number (*LuminoxMeasurementType* attribute), 146
 serial_number (*TiePieDeviceConfig* attribute), 154
 SerialCommunication (class in *hvl_ccb.comm.serial*), 21
 SerialCommunicationBytesize (class in *hvl_ccb.comm.serial*), 22
 SerialCommunicationConfig (class in *hvl_ccb.comm.serial*), 22
 SerialCommunicationIOError, 23
 SerialCommunicationParity (class in *hvl_ccb.comm.serial*), 23
 SerialCommunicationStopbits (class in *hvl_ccb.comm.serial*), 23
 set_acceleration() (*NewportSMC100PP* method), 116
 set_acquire_length() (*RTO1024* method), 133
 set_ain_differential() (*LabJack* method), 105
 set_ain_range() (*LabJack* method), 106
 set_ain_resistance() (*LabJack* method), 106
 set_ain_resolution() (*LabJack* method), 106
 set_ain_thermocouple() (*LabJack* method), 106
 set_attenuation() (*CryLasAttenuator* method), 72
 set_ceel6_socket() (*Supercube2015Base* method), 55
 set_ceel6_socket() (*SupercubeBase* method), 31
 set_channel_offset() (*RTO1024* method), 133
 set_channel_position() (*RTO1024* method), 134
 set_channel_range() (*RTO1024* method), 134
 set_channel_scale() (*RTO1024* method), 134
 set_channel_state() (*RTO1024* method), 134
 set_current() (*HeinzingerDI* method), 98
 set_current() (*HeinzingerPNC* method), 100
 set_digital_output() (*LabJack* method), 107
 set_display_unit() (*PfeifferTPG* method), 126
 set_earthing_manual() (*Supercube2015Base* method), 55
 set_full_scale_mbar() (*PfeifferTPG* method), 126
 set_full_scale_unitless() (*PfeifferTPG* method), 126
 set_init_attenuation() (*CryLasAttenuator* method), 72
 set_init_shutter_status() (*CryLasLaser* method), 76
 set_jog_speed() (*ILS2T* method), 140
 set_lower_limits() (*PSI9000* method), 82
 set_max_acceleration() (*ILS2T* method), 140
 set_max_deceleration() (*ILS2T* method), 140
 set_max_rpm() (*ILS2T* method), 140
 set_measuring_options() (*MBW973* method), 108
 set_message_board() (*SupercubeBase* method), 32
 set_motor_configuration() (*NewportSMC100PP* method), 116
 set_negative_software_limit() (*NewportSMC100PP* method), 117
 set_number_of_recordings() (*HeinzingerDI* method), 99
 set_output() (*PSI9000* method), 82
 set_positive_software_limit() (*NewportSMC100PP* method), 117

- set_pulse_energy() (*CryLasLaser* method), 76
 set_ramp_type() (*ILS2T* method), 140
 set_reference_point() (*RTO1024* method), 135
 set_register() (*FuGProbusV* method), 91
 set_remote_control() (*Supercube2015Base* method), 56
 set_remote_control() (*SupercubeBase* method), 32
 set_repetition_rate() (*CryLasLaser* method), 76
 set_repetitions() (*RTO1024* method), 135
 set_slope() (*Supercube2015WithFU* method), 67
 set_slope() (*SupercubeWithFU* method), 51
 set_status_board() (*SupercubeBase* method), 32
 set_support_output() (*Supercube2015Base* method), 56
 set_support_output() (*SupercubeBase* method), 32
 set_support_output_impulse() (*Supercube2015Base* method), 56
 set_support_output_impulse() (*SupercubeBase* method), 32
 set_system_lock() (*PSI9000* method), 82
 set_t13_socket() (*Supercube2015Base* method), 56
 set_t13_socket() (*SupercubeBase* method), 33
 set_target_voltage() (*Supercube2015WithFU* method), 67
 set_target_voltage() (*SupercubeWithFU* method), 51
 set_transmission() (*CryLasAttenuator* method), 72
 set_trigger_level() (*RTO1024* method), 135
 set_trigger_mode() (*RTO1024* method), 135
 set_trigger_source() (*RTO1024* method), 135
 set_upper_limits() (*PSI9000* method), 82
 set_voltage() (*HeinzingerDI* method), 99
 set_voltage() (*HeinzingerPNC* method), 100
 set_voltage_current() (*PSI9000* method), 83
 SETCURRENT (*FuGProbusVRegisterGroups* attribute), 93
 setup (*Power* attribute), 47, 64
 setvalue() (*FuGProbusVSetRegisters* property), 94
 SETVOLTAGE (*FuGProbusVRegisterGroups* attribute), 93
 SEVENBITS (*SerialCommunicationBytesize* attribute), 22
 SHORT_CIRCUIT (*NewportSMC100PP.MotorErrors* attribute), 112
 SHUTDOWN_CURRENT_LIMIT (*PSI9000* attribute), 81
 SHUTDOWN_VOLTAGE_LIMIT (*PSI9000* attribute), 81
 ShutterStatus (*CryLasLaser* attribute), 75
 ShutterStatus (*CryLasLaserConfig* attribute), 77
 signal_type() (*TiePieGeneratorConfig* property), 155
 SINE (*TiePieGeneratorSignalType* attribute), 155
 SingleCommDevice (*class in hvl_ccb.dev.base*), 71
 SIXBITS (*SerialCommunicationBytesize* attribute), 22
 SIXTEEN (*HeinzingerConfig.RecordingsEnum* attribute), 96
 SIXTEEN_BIT (*TiePieOscilloscopeResolution* attribute), 160
 SL (*NewportConfigCommands* attribute), 111
 SN (*FuGReadbackChannels* attribute), 94
 SOFTWARE_INTERNAL_SIXTY (*CryLasLaser.RepetitionRates* attribute), 75
 SOFTWARE_INTERNAL_TEN (*CryLasLaser.RepetitionRates* attribute), 75
 SOFTWARE_INTERNAL_TWENTY (*CryLasLaser.RepetitionRates* attribute), 75
 software_revision (*LuminosMeasurementType* attribute), 146
 SPACE (*SerialCommunicationParity* attribute), 23
 SPECIALRAMPUPWARDS (*FuGRampModes* attribute), 94
 spoll() (*VisaCommunication* method), 27
 spoll_handler() (*VisaDevice* method), 162
 SQUARE (*TiePieGeneratorSignalType* attribute), 155
 SR (*NewportConfigCommands* attribute), 111
 srq_mask() (*FuGProbusVConfigRegisters* property), 91
 srq_status() (*FuGProbusVConfigRegisters* property), 91
 stage_configuration (*NewportSMC100PPConfig* attribute), 120
 start() (*CryLasAttenuator* method), 72
 start() (*CryLasLaser* method), 76
 start() (*Device* method), 69
 start() (*DeviceSequenceMixin* method), 70
 start() (*ExperimentManager* method), 168
 start() (*FuG* method), 87
 start() (*FuGProbusIV* method), 90
 start() (*HeinzingerDI* method), 99
 start() (*HeinzingerPNC* method), 100
 start() (*ILS2T* method), 140
 start() (*LabJack* method), 107
 start() (*Luminos* method), 145
 start() (*MBW973* method), 108
 start() (*NewportSMC100PP* method), 117
 start() (*PfeifferTPG* method), 127
 start() (*PSI9000* method), 83
 start() (*RTO1024* method), 136
 start() (*SingleCommDevice* method), 71
 start() (*Supercube2015Base* method), 56
 start() (*SupercubeBase* method), 33
 start() (*Technix* method), 149
 start() (*TiePieGeneratorMixin* method), 155
 start() (*TiePieI2CHostMixin* method), 156

`start()` (*TiePieOscilloscope* method), 157
`start()` (*VisaDevice* method), 163
`start_control()` (*MBW973* method), 108
`start_measurement()` (*TiePieOscilloscope* method), 157
`start_polling()` (*Poller* method), 161
`STARTING` (*ExperimentStatus* attribute), 168
`States` (*NewportSMC100PP* attribute), 113
`status` (*Safety* attribute), 48
`status()` (*EarthingStick* class method), 42
`status()` (*ExperimentManager* property), 168
`status()` (*FuGProbusVConfigRegisters* property), 91
`status()` (*FuGProbusVDORegisters* property), 92
`status_1` (*Door* attribute), 41
`status_1` (*EarthingRod* attribute), 41
`status_1` (*EarthingStick* attribute), 43
`status_1_closed` (*EarthingStick* attribute), 60
`status_1_connected` (*EarthingStick* attribute), 61
`status_1_open` (*EarthingStick* attribute), 61
`status_2` (*Door* attribute), 41
`status_2` (*EarthingRod* attribute), 41
`status_2` (*EarthingStick* attribute), 43
`status_2_closed` (*EarthingStick* attribute), 61
`status_2_connected` (*EarthingStick* attribute), 61
`status_2_open` (*EarthingStick* attribute), 61
`status_3` (*Door* attribute), 41
`status_3` (*EarthingRod* attribute), 41
`status_3` (*EarthingStick* attribute), 43
`status_3_closed` (*EarthingStick* attribute), 61
`status_3_connected` (*EarthingStick* attribute), 61
`status_3_open` (*EarthingStick* attribute), 61
`status_4` (*EarthingStick* attribute), 43
`status_4_closed` (*EarthingStick* attribute), 61
`status_4_connected` (*EarthingStick* attribute), 61
`status_4_open` (*EarthingStick* attribute), 61
`status_5` (*EarthingStick* attribute), 43
`status_5_closed` (*EarthingStick* attribute), 61
`status_5_connected` (*EarthingStick* attribute), 61
`status_5_open` (*EarthingStick* attribute), 61
`status_6` (*EarthingStick* attribute), 43
`status_6_closed` (*EarthingStick* attribute), 61
`status_6_connected` (*EarthingStick* attribute), 61
`status_6_open` (*EarthingStick* attribute), 61
`status_closed()` (*EarthingStick* class method), 61
`status_connected()` (*EarthingStick* class method), 61
`status_error` (*Safety* attribute), 65
`status_green` (*Safety* attribute), 65
`status_open()` (*EarthingStick* class method), 61
`status_ready_for_red` (*Safety* attribute), 65
`status_red` (*Safety* attribute), 65
`STATUSBYTE` (*FuGReadbackChannels* attribute), 94
`statuses()` (*EarthingStick* class method), 43
`stop` (*Errors* attribute), 44, 62
`stop()` (*CryLasLaser* method), 77
`stop()` (*Device* method), 69
`stop()` (*DeviceSequenceMixin* method), 70
`stop()` (*ExperimentManager* method), 168
`stop()` (*FuGProbusIV* method), 90
`stop()` (*HeinzingerDI* method), 99
`stop()` (*ILS2T* method), 140
`stop()` (*LabJack* method), 107
`stop()` (*Luminos* method), 145
`stop()` (*MBW973* method), 108
`stop()` (*NewportSMC100PP* method), 117
`stop()` (*PfeifferTPG* method), 127
`stop()` (*PSI9000* method), 83
`stop()` (*RTO1024* method), 136
`stop()` (*SingleCommDevice* method), 71
`stop()` (*Supercube2015Base* method), 56
`stop()` (*SupercubeBase* method), 33
`stop()` (*Technix* method), 149
`stop()` (*TiePieGeneratorMixin* method), 155
`stop()` (*TiePieI2CHostMixin* method), 156
`stop()` (*TiePieOscilloscope* method), 157
`stop()` (*VisaDevice* method), 163
`stop_acquisition()` (*RTO1024* method), 136
`stop_motion()` (*NewportSMC100PP* method), 117
`stop_number` (*Errors* attribute), 62
`stop_polling()` (*Poller* method), 161
`stopbits` (*CryLasAttenuatorSerialCommunicationConfig* attribute), 74
`stopbits` (*CryLasLaserSerialCommunicationConfig* attribute), 80
`stopbits` (*FuGSerialCommunicationConfig* attribute), 95
`stopbits` (*HeinzingerSerialCommunicationConfig* attribute), 101
`stopbits` (*LuminosSerialCommunicationConfig* attribute), 148
`stopbits` (*MBW973SerialCommunicationConfig* attribute), 110
`stopbits` (*NewportSMC100PPSerialCommunicationConfig* attribute), 123
`stopbits` (*PfeifferTPGSerialCommunicationConfig* attribute), 129
`Stopbits` (*SerialCommunicationConfig* attribute), 22
`stopbits` (*SerialCommunicationConfig* attribute), 23
`streaming` (*LuminosOutputMode* attribute), 146
`StrEnumBase` (class in *hvl_ccb.utils.enum*), 164
`sub_handler` (*OpcUaCommunicationConfig* attribute), 20
`sub_handler` (*SupercubeOpcUaCommunicationConfig* attribute), 34, 58
`suitable_range()` (*TiePieOscilloscopeRange* static method), 159
`Supercube2015Base` (class in *hvl_ccb.dev.supercube2015.base*), 54

Supercube2015WithFU (class in *hvl_ccb.dev.supercube2015.typ_a*), 66

SupercubeAopcUaCommunication (class in *hvl_ccb.dev.supercube.typ_a*), 49

SupercubeAopcUaCommunication (class in *hvl_ccb.dev.supercube2015.typ_a*), 67

SupercubeAopcUaConfiguration (class in *hvl_ccb.dev.supercube.typ_a*), 49

SupercubeAopcUaConfiguration (class in *hvl_ccb.dev.supercube2015.typ_a*), 67

SupercubeB (class in *hvl_ccb.dev.supercube.typ_b*), 52

SupercubeBase (class in *hvl_ccb.dev.supercube.base*), 29

SupercubeBopcUaCommunication (class in *hvl_ccb.dev.supercube.typ_b*), 52

SupercubeBopcUaConfiguration (class in *hvl_ccb.dev.supercube.typ_b*), 52

SupercubeConfiguration (class in *hvl_ccb.dev.supercube.base*), 33

SupercubeConfiguration (class in *hvl_ccb.dev.supercube2015.base*), 56

SupercubeEarthingStickOperationError, 34

SupercubeOpcEndpoint (class in *hvl_ccb.dev.supercube.constants*), 48

SupercubeOpcEndpoint (class in *hvl_ccb.dev.supercube2015.constants*), 66

SupercubeOpcUaCommunication (class in *hvl_ccb.dev.supercube.base*), 34

SupercubeOpcUaCommunication (class in *hvl_ccb.dev.supercube2015.base*), 57

SupercubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.supercube.base*), 34

SupercubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.supercube2015.base*), 57

SupercubeSubscriptionHandler (class in *hvl_ccb.dev.supercube.base*), 34

SupercubeSubscriptionHandler (class in *hvl_ccb.dev.supercube2015.base*), 58

SupercubeWithFU (class in *hvl_ccb.dev.supercube.typ_a*), 50

switch_to_operate (Safety attribute), 48

switch_to_ready (Safety attribute), 48

switchto_green (Safety attribute), 65

switchto_operate (Safety attribute), 65

switchto_ready (Safety attribute), 65

SyncCommunicationProtocol (class in *hvl_ccb.comm.base*), 12

SyncCommunicationProtocolConfig (class in *hvl_ccb.comm.base*), 12

t13_2 (GeneralSockets attribute), 44, 62

t13_3 (GeneralSockets attribute), 44, 62

T13_SOCKET_PORTS (in module *hvl_ccb.dev.supercube.constants*), 48

T13_SOCKET_PORTS (in module *hvl_ccb.dev.supercube2015.constants*), 66

T1MS (FuGMonitorModes attribute), 89

T200MS (FuGMonitorModes attribute), 89

T20MS (FuGMonitorModes attribute), 89

T256US (FuGMonitorModes attribute), 89

T4 (LabJack.DeviceType attribute), 103

T4 (LJMCommunicationConfig.DeviceType attribute), 14

T40MS (FuGMonitorModes attribute), 89

T4MS (FuGMonitorModes attribute), 89

T7 (LabJack.DeviceType attribute), 103

T7 (LJMCommunicationConfig.DeviceType attribute), 14

T7_PRO (LabJack.DeviceType attribute), 103

T7_PRO (LJMCommunicationConfig.DeviceType attribute), 15

T800MS (FuGMonitorModes attribute), 89

T80MS (FuGMonitorModes attribute), 89

target_pulse_energy() (CryLasLaser property), 77

TCP (LJMCommunicationConfig.ConnectionType attribute), 14

TCPIP_INSTR (VisaCommunicationConfig.InterfaceType attribute), 27

TCPIP_SOCKET (VisaCommunicationConfig.InterfaceType attribute), 27

TEC1 (CryLasLaser.AnswersStatus attribute), 75

TEC2 (CryLasLaser.AnswersStatus attribute), 75

Technix (class in *hvl_ccb.dev.technix*), 148

TechnixCommunication (class in *hvl_ccb.dev.technix*), 149

TechnixCommunicationConfig (class in *hvl_ccb.dev.technix*), 149

TechnixConfig (class in *hvl_ccb.dev.technix*), 150

TechnixError, 151

TechnixSerialCommunication (class in *hvl_ccb.dev.technix*), 151

TechnixSerialCommunicationConfig (class in *hvl_ccb.dev.technix*), 151

TechnixStatusByte (class in *hvl_ccb.dev.technix*), 152

TechnixTelnetCommunication (class in *hvl_ccb.dev.technix*), 152

TechnixTelnetCommunicationConfig (class in *hvl_ccb.dev.technix*), 152

TelnetCommunication (class in *hvl_ccb.comm.telnet*), 24

TelnetCommunicationConfig (class in *hvl_ccb.comm.telnet*), 25

TelnetError, 25

TEMP (ILS2TRegAddr attribute), 143

T

T (LabJack.ThermocoupleType attribute), 104

t13_1 (GeneralSockets attribute), 44, 62

temperature_sensor (*LuminosMeasurementType attribute*), 146

TEN (*LabJack.AInRange attribute*), 103

TEN (*LabJack.CalMicroAmpere attribute*), 103

terminator (*AsyncCommunicationProtocolConfig attribute*), 12

terminator (*CryLasAttenuatorSerialCommunicationConfig attribute*), 74

terminator (*CryLasLaserSerialCommunicationConfig attribute*), 80

TERMINATOR (*FuGProbusIVCommands attribute*), 90

terminator (*FuGSerialCommunicationConfig attribute*), 95

terminator (*HeinzingerSerialCommunicationConfig attribute*), 101

terminator (*LuminosSerialCommunicationConfig attribute*), 148

terminator (*MBW973SerialCommunicationConfig attribute*), 110

terminator (*NewportSMC100PPSerialCommunicationConfig attribute*), 123

terminator (*PfeifferTPGSerialCommunicationConfig attribute*), 129

terminator (*TechnixCommunicationConfig attribute*), 150

terminator() (*FuGProbusVConfigRegisters property*), 91

terminator_str() (*SerialCommunicationConfig method*), 23

TiePieDeviceConfig (*class in hvl_ccb.dev.tiepie*), 153

TiePieDeviceType (*class in hvl_ccb.dev.tiepie*), 154

TiePieError, 154

TiePieGeneratorConfig (*class in hvl_ccb.dev.tiepie*), 154

TiePieGeneratorMixin (*class in hvl_ccb.dev.tiepie*), 155

TiePieGeneratorSignalType (*class in hvl_ccb.dev.tiepie*), 155

TiePieHS5 (*class in hvl_ccb.dev.tiepie*), 155

TiePieHS6 (*class in hvl_ccb.dev.tiepie*), 156

TiePieI2CHostConfig (*class in hvl_ccb.dev.tiepie*), 156

TiePieI2CHostMixin (*class in hvl_ccb.dev.tiepie*), 156

TiePieOscilloscope (*class in hvl_ccb.dev.tiepie*), 156

TiePieOscilloscopeAutoResolutionModes (*class in hvl_ccb.dev.tiepie*), 157

TiePieOscilloscopeChannelConfig (*class in hvl_ccb.dev.tiepie*), 158

TiePieOscilloscopeChannelCoupling (*class in hvl_ccb.dev.tiepie*), 158

TiePieOscilloscopeConfig (*class in hvl_ccb.dev.tiepie*), 159

TiePieOscilloscopeRange (*class in hvl_ccb.dev.tiepie*), 159

TiePieOscilloscopeResolution (*class in hvl_ccb.dev.tiepie*), 159

TiePieOscilloscopeTriggerKind (*class in hvl_ccb.dev.tiepie*), 160

TiePieOscilloscopeTriggerLevelMode (*class in hvl_ccb.dev.tiepie*), 160

TiePieWS5 (*class in hvl_ccb.dev.tiepie*), 160

timeout (*CryLasAttenuatorSerialCommunicationConfig attribute*), 74

timeout (*CryLasLaserSerialCommunicationConfig attribute*), 80

timeout (*FuGSerialCommunicationConfig attribute*), 95

timeout (*HeinzingerSerialCommunicationConfig attribute*), 102

timeout (*LuminosSerialCommunicationConfig attribute*), 148

timeout (*MBW973SerialCommunicationConfig attribute*), 110

timeout (*NewportSMC100PPSerialCommunicationConfig attribute*), 123

timeout (*PfeifferTPGSerialCommunicationConfig attribute*), 129

timeout (*SerialCommunicationConfig attribute*), 23

timeout (*TelnetCommunicationConfig attribute*), 25

timeout (*VisaCommunicationConfig attribute*), 28

Torr (*PfeifferTPG.PressureUnits attribute*), 124

TPG25xA (*PfeifferTPGConfig.Model attribute*), 127

TPGx6x (*PfeifferTPGConfig.Model attribute*), 127

TPR (*PfeifferTPG.SensorTypes attribute*), 125

transmission() (*CryLasAttenuator property*), 72

TRIANGLE (*TiePieGeneratorSignalType attribute*), 155

trigger_enabled() (*TiePieOscilloscopeChannelConfig property*), 158

trigger_hysteresis() (*TiePieOscilloscopeChannelConfig property*), 158

trigger_kind() (*TiePieOscilloscopeChannelConfig property*), 158

trigger_level() (*TiePieOscilloscopeChannelConfig property*), 158

trigger_level_mode() (*TiePieOscilloscopeChannelConfig property*), 158

trigger_timeout() (*TiePieOscilloscopeConfig property*), 159

triggered (*BreakdownDetection attribute*), 41, 60

TWELVE_BIT (*TiePieOscilloscopeResolution attribute*), 160

TWENTY_VOLT (*TiePieOscilloscopeRange attribute*), 159

TWO (*HeinzingerConfig.RecordingsEnum attribute*), 96

TWO (*SerialCommunicationStopbits attribute*), 24

TWO_HUNDRED (*LabJack.CalMicroAmpere* attribute), 103

TWO_HUNDRED_MILLI_VOLT (*TiePieOscilloscopeRange* attribute), 159

TWO_VOLT (*TiePieOscilloscopeRange* attribute), 159

U

Underrange (*PfeifferTPG.SensorStatus* attribute), 125

unit (*ILS2TModbusTcpCommunicationConfig* attribute), 142

unit (*ModbusTcpCommunicationConfig* attribute), 17

unit() (*PfeifferTPG* property), 127

unit_current() (*HeinzingerPNC* property), 100

unit_voltage() (*HeinzingerPNC* property), 100

UNKNOWN (*HeinzingerDI.OutputStatus* attribute), 97

UNKNOWN (*HeinzingerPNC.UnitCurrent* attribute), 99

UNKNOWN (*HeinzingerPNC.UnitVoltage* attribute), 99

UNKNOWN (*TiePieGeneratorSignalType* attribute), 155

UNKNOWN (*TiePieOscilloscopeAutoResolutionModes* attribute), 158

UNKNOWN (*TiePieOscilloscopeTriggerLevelMode* attribute), 160

UNREADY_INACTIVE (*CryLasLaser.LaserStatus* attribute), 75

update_laser_status() (*CryLasLaser* method), 77

update_period (*OpcUaCommunicationConfig* attribute), 20

update_repetition_rate() (*CryLasLaser* method), 77

update_shutter_status() (*CryLasLaser* method), 77

update_target_pulse_energy() (*CryLasLaser* method), 77

UpdateEspStageInfo (*NewportSMC100PPConfig.EspStageConfig* attribute), 118

USB (*LJMCommunicationConfig.ConnectionType* attribute), 14

user_position_offset (*NewportSMC100PPConfig* attribute), 120

user_steps() (*ILS2T* method), 141

V

V (*HeinzingerPNC.UnitVoltage* attribute), 99

VA (*NewportConfigCommands* attribute), 111

value() (*FuGProbusVMonitorRegisters* property), 92

value() (*LabJack.AInRange* property), 103

value_raw() (*FuGProbusVMonitorRegisters* property), 93

ValueEnum (*class in hvl_ccb.utils.enum*), 164

VB (*NewportConfigCommands* attribute), 111

velocity (*NewportSMC100PPConfig* attribute), 120

visa_backend (*VisaCommunicationConfig* attribute), 28

VisaCommunication (*class in hvl_ccb.comm.visa*), 26

VisaCommunicationConfig (*class in hvl_ccb.comm.visa*), 27

VisaCommunicationConfig.InterfaceType (*class in hvl_ccb.comm.visa*), 27

VisaCommunicationError, 28

VisaDevice (*class in hvl_ccb.dev.visa*), 162

VisaDeviceConfig (*class in hvl_ccb.dev.visa*), 163

VOLT (*ILS2TRegAddr* attribute), 143

Volt (*PfeifferTPG.PressureUnits* attribute), 124

VOLTAGE (*FuGProbusIVCommands* attribute), 90

VOLTAGE (*FuGReadbackChannels* attribute), 94

voltage() (*FuG* property), 87

voltage() (*Technix* property), 149

voltage_lower_limit (*PSI9000Config* attribute), 84

voltage_max (*Power* attribute), 47, 64

voltage_monitor() (*FuG* property), 87

voltage_primary (*Power* attribute), 47, 64

voltage_regulation() (*Technix* property), 149

voltage_slope (*Power* attribute), 47, 64

voltage_target (*Power* attribute), 47, 64

voltage_upper_limit (*PSI9000Config* attribute), 84

W

WAIT_AFTER_WRITE (*VisaCommunication* attribute), 26

wait_for_polling_result() (*Poller* method), 161

wait_operation_complete() (*VisaDevice* method), 163

wait_sec_initialisation (*PSI9000Config* attribute), 84

wait_sec_max_disable (*ILS2TConfig* attribute), 142

wait_sec_post_absolute_position (*ILS2TConfig* attribute), 142

wait_sec_post_activate (*LuminosConfig* attribute), 145

wait_sec_post_cannot_disable (*ILS2TConfig* attribute), 142

wait_sec_post_enable (*ILS2TConfig* attribute), 142

wait_sec_post_relative_step (*ILS2TConfig* attribute), 142

wait_sec_read_text_nonempty (*AsyncCommunicationProtocolConfig* attribute), 12

wait_sec_read_text_nonempty (*FuGSerialCommunicationConfig* attribute), 96

wait_sec_read_text_nonempty (*HeinzingerSerialCommunicationConfig* attribute), 102
 wait_sec_retry_get_device (*TiePieDeviceConfig* attribute), 154
 wait_sec_settings_effect (*PSI9000Config* attribute), 84
 wait_sec_stop_commands (*FuGConfig* attribute), 88
 wait_sec_stop_commands (*HeinzingerConfig* attribute), 97
 wait_sec_system_lock (*PSI9000Config* attribute), 84
 wait_sec_trials_activate (*LuminoxConfig* attribute), 145
 wait_timeout_retry_sec (*OpcUaCommunicationConfig* attribute), 20
 wait_until_motor_initialized() (*NewportSMC100PP* method), 118
 wait_until_ready() (*CryLasLaser* method), 77
 warning (*Errors* attribute), 44
 WIFI (*LJMCommunicationConfig.ConnectionType* attribute), 14
 wrap_libtiepie_exception() (in module *hvl_ccb.dev.tiepie*), 161
 write() (*AsyncCommunicationProtocol* method), 10
 write() (*MBW973* method), 108
 write() (*OpcUaCommunication* method), 19
 write() (*Supercube2015Base* method), 56
 write() (*SupercubeBase* method), 33
 write() (*VisaCommunication* method), 27
 write_absolute_position() (*ILS2T* method), 141
 write_bytes() (*AsyncCommunicationProtocol* method), 11
 write_bytes() (*SerialCommunication* method), 21
 write_bytes() (*TelnetCommunication* method), 25
 write_name() (*LJMCommunication* method), 14
 write_names() (*LJMCommunication* method), 14
 write_registers() (*ModbusTcpCommunication* method), 16
 write_relative_step() (*ILS2T* method), 141
 write_termination (*VisaCommunicationConfig* attribute), 28
 write_text() (*AsyncCommunicationProtocol* method), 11
 WRONG_ESP_STAGE (*NewportSMC100PP.MotorErrors* attribute), 112

X

x_stat() (*FuGProbusVDIRegisters* property), 92
 XOUTPUTS (*FuGProbusIVCommands* attribute), 91

Y

YES (*FuGDigitalVal* attribute), 88

Z

ZX (*NewportConfigCommands* attribute), 111