
HVL Common Code Base Documentation

Release 0.3.5

Mikolaj Rybiński, David Graber

Feb 18, 2020

Contents:

1	HVL Common Code Base	1
1.1	Features	1
1.2	Credits	2
2	Installation	3
2.1	Stable release	3
2.2	From sources	3
3	Usage	5
4	API Documentation	7
4.1	hvl_ccb package	7
5	Contributing	119
5.1	Types of Contributions	119
5.2	Get Started!	120
5.3	Merge Request Guidelines	121
5.4	Tips	121
5.5	Deploying	121
6	Credits	123
6.1	Development Lead	123
6.2	Contributors	123
7	History	125
7.1	0.3.5 (2020-02-18)	125
7.2	0.3.4 (2019-12-20)	125
7.3	0.3.3 (2019-05-08)	126
7.4	0.3.2 (2019-05-08)	126
7.5	0.3.1 (2019-05-02)	126
7.6	0.3 (2019-05-02)	126
7.7	0.2.1 (2019-04-01)	126
7.8	0.2.0 (2019-03-31)	126
7.9	0.1.0 (2019-02-06)	126
8	Indices and tables	127
	Python Module Index	129

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
- **Documentation:**
 - if you're planning to develop start w/ reading "CONTRIBUTING.rst", otherwise either
 - read [HVL CCB documentation at RTD](#), or
 - install *Sphinx* and *sphinx_rtd_theme* Python packages and locally build docs on Windows in git-bash by running:

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

from a shell with Make installed by running:

```
$ make docs
```

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

1.1 Features

Manage experiments with `ExperimentManager` instance controlling one or more of the following devices:

- a MBW973 SF6 Analyzer / dew point mirror over a serial connection (COM-ports)
- a LabJack (T7-PRO) device using a LabJack LJM Library for communication

- a Schneider Electric ILS2T stepper motor drive over Modbus TCP
- a Elektro-Automatik PSI9000 DC power supply using VISA over TCP for communication
- a Rhode & Schwarz RTO 1024 oscilloscope using VISA interface over TCP : : INSTR
- a state-of-the-art HVL in-house Supercube device variants using an OPC UA client
- a Heinzinger Digital Interface I/II and a Heinzinger PNC power supply over a serial connection
- a passively Q-switched Pulsed Laser and a laser attenuator from CryLas over a serial connection
- a Newport SMC100PP single axis motion controller for 2-phase stepper motors over a serial connection
- a Pfeiffer TPG controller (TPG 25x, TPG 26x and TPG 36x) for Compact pressure Gauges

1.2 Credits

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

2.1 Stable release

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

```
$ pip install hvl_ccb
```

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:

```
$ python setup.py install
```


CHAPTER 3

Usage

To use HVL Common Code Base in a project:

```
import hvl_ccb
```


4.1 hvl_ccb package

4.1.1 Subpackages

`hvl_ccb.comm` package

Submodules

`hvl_ccb.comm.base` module

Module with base classes for communication protocols.

class `hvl_ccb.comm.base.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 = `None`

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

close ()

Close the communication protocol

open ()

Open communication protocol

`hvl_ccb.comm.labjack_ljm` module

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 hvl_ccb.comm.labjack_ljm.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

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, 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 hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig(device_type: Union[str,  
                                                    hvl_ccb.dev.labjack.DeviceType]  
                                                    = 'ANY', connec-  
tion_type: Union[str,  
                                                    hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig  
                                                    = 'ANY', identifier: str =  
                                                    'ANY')
```

Bases: `object`

Configuration dataclass for *LJMCommunication*.

```
class ConnectionType (*args, **kwargs)
    Bases: hvl_ccb.utils.enum.AutoNumberNameEnum

    LabJack connection type.

    ANY = 1

    ETHERNET = 4

    TCP = 3

    USB = 2

    WIFI = 5

class DeviceType (*args, **kwargs)
    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

    get_by_p_id = <bound method DeviceType.get_by_p_id of <aenum 'DeviceType'>>

clean_values () → None
    Performs value checks on device_type and connection_type.

connection_type = 'ANY'
    Can be either string or of enum ConnectionType.

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

is_configdataclass = True

classmethod keys () → Sequence[str]
    Returns a list of all configdataclass fields key-names.

    Returns a list of strings containing all keys.
```

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

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

Returns a list of strings containing all optional keys.

classmethod `required_keys ()` → Sequence[str]

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

Returns a list of strings containing all required keys.

exception `hvl_ccb.comm.labjack_ljm.LJMCommunicationError`

Bases: `Exception`

Errors coming from LJMCommunication.

hvl_ccb.comm.modbus_tcp module

Communication protocol for modbus TCP ports. Makes use of the `pymodbus` library.

class `hvl_ccb.comm.modbus_tcp.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 `hvl_ccb.comm.modbus_tcp.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 = `None`

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 = `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 = `None`

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 `hvl_ccb.comm.modbus_tcp.ModbusTcpConnectionFailedException` (*string=""*)

Bases: `pymodbus.exceptions.ConnectionException`

Exception raised when the connection failed.

hvl_ccb.comm.opc module

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

```
class hvl_ccb.comm.opc.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[str, Iterable[str]], ns_index: int) → None
```

Initialize monitored nodes.

Parameters

- **node_id** – one or more strings of node IDs.
- **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

```
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 hvl_ccb.comm.opc.OpcUaCommunicationConfig (host: str, endpoint_name: str,  
                                              port: int = 4840, sub_handler:  
                                              hvl_ccb.comm.opc.OpcUaSubHandler  
                                              = <hvl_ccb.comm.opc.OpcUaSubHandler  
                                              object>, update_period: int = 500)
```

Bases: object

Configuration dataclass for OPC UA Communciation.

clean_values ()

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

endpoint_name = None

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 = None

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.

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 = 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 object>

object to use for handling subscriptions.

update_period = 500

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

exception hvl_ccb.comm.opc.OpcUaCommunicationIOError

Bases: OSError

OPC-UA communication I/O error.

class `hvl_ccb.comm.opc.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 module

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

class `hvl_ccb.comm.serial.SerialCommunication` (*configuration*)

Bases: `hvl_ccb.comm.base.CommunicationProtocol`

Implements the Communication Protocol for serial ports.

ENCODING = `'utf-8'`

UNICODE_HANDLING = `'replace'`

close ()

Close the serial connection.

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

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 (*size: int = 1*) → bytes

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

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

Parameters **size** – number of bytes to read

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

Raises `SerialCommunicationIOError` – when communication port is not opened

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.

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

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

write_text (*text: str*)

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

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

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

Raises *SerialCommunicationIOError* – when communication port is not opened

class `hvl_ccb.comm.serial.SerialCommunicationBytesize` (**args, **kwargs*)

Bases: *hvl_ccb.utils.enum.ValueEnum*

Serial communication bytesize.

EIGHTBITS = 8

FIVEBITS = 5

SEVENBITS = 7

SIXBITS = 6

class `hvl_ccb.comm.serial.SerialCommunicationConfig` (*port: str, baudrate: int, parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity], stopbits: Union[int, float, hvl_ccb.comm.serial.SerialCommunicationStopbits], bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationBytesize], terminator: bytes = b'rn', time-out: Union[int, float] = 2*)

Bases: object

Configuration dataclass for *SerialCommunication*.

Bytesize

alias of *SerialCommunicationBytesize*

Parity

alias of *SerialCommunicationParity*

Stopbits

alias of *SerialCommunicationStopbits*

baudrate = None

Baudrate of the serial port

bytesize = None

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

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.

parity = None

Parity to be used for the connection.

port = 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 = None

Stopbits setting, can be 1, 1.5 or 2.

terminator = b'\r\n'

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

timeout = 2

Timeout in seconds for the serial port

exception hvl_ccb.comm.serial.SerialCommunicationIOError

Bases: OSError

Serial communication related I/O errors.

class hvl_ccb.comm.serial.SerialCommunicationParity(*args, **kwargs)

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 hvl_ccb.comm.serial.SerialCommunicationStopbits(*args, **kws)
    Bases: hvl_ccb.utils.enum.ValueEnum
    Serial communication stopbits.

    ONE = 1
    ONE_POINT_FIVE = 1.5
    TWO = 2

```

hvl_ccb.comm.visa module

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 hvl_ccb.comm.visa.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) → 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) → 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 hvl_ccb.comm.visa.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**(*args, **kwargs)

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: int = None, board: 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

address

Address string depending on the VISA protocol's configuration.

Returns address string corresponding to current configuration

board = 0

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

chunk_size = 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 = None

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

interface_type = None

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 = 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 = 5025

TCP port, standard is 5025.

read_termination = '\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 = 5000

Timeout for commands in milli seconds.

visa_backend = ''

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 = '\n'

Write termination character.

exception hvl_ccb.comm.visa.VisaCommunicationError

Bases: Exception

Base class for VisaCommunication errors.

Module contents

Communication protocols subpackage.

hvl_ccb.dev package

Subpackages

hvl_ccb.dev.supercube package

Submodules

hvl_ccb.dev.supercube.base module

Base classes for the Supercube device.

class `hvl_ccb.dev.supercube.base.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

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_manual (*number: int*) → bool

TODO: Test get_earthing_manual with device

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*) → `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

get_measurement_ratio (*channel: int*) → float

TODO: test get_measurement_ratio with device

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

get_measurement_voltage (*channel: int*) → float

TODO: test get_measurement_voltage with device

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

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

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

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

TODO: Test set_earthing_manual with device

Set the manual status of an earthing stick. 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)
- **manual** – earthing stick manual status (True or False)

set_remote_control (*state: bool*) → None

TODO: test set_remote_control with device

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 hvl_ccb.dev.supercube.base.SupercubeConfiguration (namespace_index: int = 3)
```

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 = **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.

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 hvl_ccb.dev.supercube.base.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 hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunicationConfig (host:
                                                                    str, end-
                                                                    point_name:
                                                                    str, port:
                                                                    int      =
                                                                    4840,
                                                                    sub_handler:
                                                                    hvl_ccb.comm.opc.OpcUaSubHandler
                                                                    =
                                                                    <hvl_ccb.dev.supercube.base.SupercubeOpcUaSubHandler
                                                                    object>,
                                                                    up-
                                                                    date_period:
                                                                    int = 500)
```

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.dev.supercube.base.SupercubeSubscriptionHandler object>

Subscription handler for data change events

class hvl_ccb.dev.supercube.base.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` module

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

class `hvl_ccb.dev.supercube.constants.AlarmText` (*args, **kws)
 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'
get = <bound method AlarmText.get of <aenum 'AlarmText'>>
not_defined = 'NO ALARM TEXT DEFINED'

class hvl_ccb.dev.supercube.constants.Alarms(*args, **kws)
    Bases: hvl_ccb.dev.supercube.constants._AlarmsBase
    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 hvl_ccb.dev.supercube.constants.BreakdownDetection (*args, **kws)
```

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 hvl_ccb.dev.supercube.constants.Door (*args, **kws)
```

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

Variable NodeID strings for doors.

```
get = <bound method Door.get of <aenum 'Door'>>
```

```
status_1 = ' "DB_Safety_Circuit"."Türe 1"."si_HMI_status_door" '
```

```
status_2 = ' "DB_Safety_Circuit"."Türe 2"."si_HMI_status_door" '
```

```
status_3 = ' "DB_Safety_Circuit"."Türe 3"."si_HMI_status_door" '
```

```

class hvl_ccb.dev.supercube.constants.DoorStatus (*args, **kws)
    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 hvl_ccb.dev.supercube.constants.EarthingStick (*args, **kws)
    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.

    manual = <bound method EarthingStick.manual of <aenum 'EarthingStick'>>
    manual_1 = '"DB_Safety_Circuit"."Erdpeitsche 1"."sx_earthing_manually"'
    manual_2 = '"DB_Safety_Circuit"."Erdpeitsche 2"."sx_earthing_manually"'
    manual_3 = '"DB_Safety_Circuit"."Erdpeitsche 3"."sx_earthing_manually"'
    manual_4 = '"DB_Safety_Circuit"."Erdpeitsche 4"."sx_earthing_manually"'
    manual_5 = '"DB_Safety_Circuit"."Erdpeitsche 5"."sx_earthing_manually"'
    manual_6 = '"DB_Safety_Circuit"."Erdpeitsche 6"."sx_earthing_manually"'
    status = <bound method EarthingStick.status of <aenum 'EarthingStick'>>
    status_1 = '"DB_Safety_Circuit"."Erdpeitsche 1"."si_HMI_Status"'
    status_2 = '"DB_Safety_Circuit"."Erdpeitsche 2"."si_HMI_Status"'
    status_3 = '"DB_Safety_Circuit"."Erdpeitsche 3"."si_HMI_Status"'
    status_4 = '"DB_Safety_Circuit"."Erdpeitsche 4"."si_HMI_Status"'
    status_5 = '"DB_Safety_Circuit"."Erdpeitsche 5"."si_HMI_Status"'
    status_6 = '"DB_Safety_Circuit"."Erdpeitsche 6"."si_HMI_Status"'

class hvl_ccb.dev.supercube.constants.EarthingStickStatus (*args, **kws)
    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 hvl_ccb.dev.supercube.constants.**Errors** (*args, **kws)

Bases: *hvl_ccb.utils.enum.ValueEnum*

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

message = "DB_Meldepuffer"."Hinweis_aktiv"

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

quit = "DB_Meldepuffer"."Quittierbutton"

Writable boolean for the error quit button.

stop = "DB_Meldepuffer"."Stop_aktiv"

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

warning = "DB_Meldepuffer"."Warnung_aktiv"

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

class hvl_ccb.dev.supercube.constants.**GeneralSockets** (*args, **kws)

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 (writeable boolean).

t13_2 = "Qx_Allg_Socket_T13_2"

SEV T13 socket No. 2 (writeable boolean).

t13_3 = "Qx_Allg_Socket_T13_3"

SEV T13 socket No. 3 (writeable boolean).

class hvl_ccb.dev.supercube.constants.**GeneralSupport** (*args, **kws)

Bases: *hvl_ccb.utils.enum.ValueEnum*

NodeID strings for the support inputs and outputs.

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'
input = <bound method GeneralSupport.input of <aenum 'GeneralSupport'>>
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'

output = <bound method GeneralSupport.output of <aenum 'GeneralSupport'>>

```

```

class hvl_ccb.dev.supercube.constants.MeasurementsDividerRatio(*args, **kws)
    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.

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

```

get = <bound method MeasurementsDividerRatio.get of <aenum 'MeasurementsDividerRatio'>>
input_1 = 'Ir_Measure_DividerRatio_1'
input_2 = 'Ir_Measure_DividerRatio_2'
input_3 = 'Ir_Measure_DividerRatio_3'
input_4 = 'Ir_Measure_DividerRatio_4'

```

```

class hvl_ccb.dev.supercube.constants.MeasurementsScaledInput(*args, **kws)
    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*.

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

```

get = <bound method MeasurementsScaledInput.get of <aenum 'MeasurementsScaledInput'>>
input_1 = 'Qr_Measure_Input1_scaledVoltage'
input_2 = 'Qr_Measure_Input2_scaledVoltage'
input_3 = 'Qr_Measure_Input3_scaledVoltage'
input_4 = 'Qr_Measure_Input4_scaledVoltage'

```

```
class hvl_ccb.dev.supercube.constants.OpcControl(*args, **kws)
```

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

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

TODO: this variable nodeID string is not tested and/or correct yet.

```
active = 'Ix_OPC_active'
```

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

```
class hvl_ccb.dev.supercube.constants.Power(*args, **kws)
```

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_actualCurrent'
```

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_maxVoltage'
```

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

```
voltage_primary = 'Qr_Power_FU_actualVoltage'
```

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_TargetVoltage'
```

Target voltage setpoint in V.

```
class hvl_ccb.dev.supercube.constants.PowerSetup(*args, **kws)
```

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 `hvl_ccb.dev.supercube.constants.Safety(*args, **kws)`

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`.

switchto_operate = 'DB_Safety_Circuit"."sx_safe_switchto_operate'

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

switchto_ready = 'DB_Safety_Circuit"."sx_safe_switchto_ready'

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

class `hvl_ccb.dev.supercube.constants.SafetyStatus(*args, **kws)`

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 `hvl_ccb.dev.supercube.constants.SupercubeOpcEndpoint(*args, **kws)`

Bases: `hvl_ccb.utils.enum.ValueEnum`

OPC Server Endpoint strings for the supercube variants.

A = 'Supercube Typ A'

B = 'Supercube Typ B'

```
hvl_ccb.dev.supercube.constants.T13_SOCKET_PORTS = (1, 2, 3)
```

Port numbers of SEV T13 power socket

hvl_ccb.dev.supercube.typ_a module

Supercube Typ A module.

```
class hvl_ccb.dev.supercube.typ_a.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 hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaConfiguration(host: str, end-
                                                                point_name: str
                                                                = 'Supercube
                                                                Typ A', port:
                                                                int = 4840,
                                                                sub_handler:
                                                                hvl_ccb.comm.opc.OpcUaSubHandler
                                                                =
                                                                <hvl_ccb.dev.supercube.base.SupercubeS
                                                                object at
                                                                0x7f2f86c81c50>,
                                                                update_period:
                                                                int = 500)
```

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

```
endpoint_name = 'Supercube Typ A'
```

```
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 `hvl_ccb.dev.supercube.typ_a.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 module

Supercube Typ B module.

class `hvl_ccb.dev.supercube.typ_b.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 `hvl_ccb.dev.supercube.typ_b.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 `hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaConfiguration` (*host: str, end-*

point_name: str

= 'Supercube

Typ B', port:

int = 4840,

sub_handler:

hvl_ccb.comm.opc.OpcUaSubHandler

=

<hvl_ccb.dev.supercube.base.SupercubeS

object at

0x7f2f86c81c50>,

update_period:

int = 500)

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

endpoint_name = 'Supercube Typ B'

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 package

Submodules

hvl_ccb.dev.supercube2015.base module

Base classes for the Supercube device.

exception hvl_ccb.dev.supercube2015.base.InvalidSupercubeStatusError

Bases: Exception

Exception raised when supercube has invalid status.

class hvl_ccb.dev.supercube2015.base.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. :return: 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

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 hvl_ccb.dev.supercube2015.base.**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 = 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 hvl_ccb.dev.supercube2015.base.**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 hvl_ccb.dev.supercube2015.base.**SupercubeOpcUaCommunicationConfig** (host:

str,
end-
point_name:
str,
port:
int
=
4845,
sub_handler:
hvl_ccb.comm.opc.OpcUaSu
=
<hvl_ccb.dev.supercube2015
ob-
ject>,
up-
date_period:
int
=
500)

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 = 4845

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.dev.supercube2015.base.SupercubeSubscriptionHandler object>

Subscription handler for data change events

class `hvl_ccb.dev.supercube2015.base.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 module

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

class `hvl_ccb.dev.supercube2015.constants.AlarmText (*args, **kws)`

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'

get = <bound method AlarmText.get of <aenum 'AlarmText'>>
not_defined = 'NO ALARM TEXT DEFINED'

class hvl_ccb.dev.supercube2015.constants.BreakdownDetection(*args, **kws)
    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 hvl_ccb.dev.supercube2015.constants.DoorStatus(*args, **kws)
    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 hvl_ccb.dev.supercube2015.constants.EarthingStick(*args, **kws)
    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.

    manual = <bound method EarthingStick.manual of <aenum 'EarthingStick'>>

```

```
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'
status_closed = <bound method EarthingStick.status_closed of <aenum 'EarthingStick'>>
status_connected = <bound method EarthingStick.status_connected of <aenum 'EarthingStick'>>
status_open = <bound method EarthingStick.status_open of <aenum 'EarthingStick'>>

class hvl_ccb.dev.supercube2015.constants.EarthingStickStatus(*args, **kws)
    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 hvl_ccb.dev.supercube2015.constants.Errors(*args, **kws)
    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 hvl_ccb.dev.supercube2015.constants.GeneralSockets(*args, **kws)
    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 (writable 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 hvl_ccb.dev.supercube2015.constants.GeneralSupport(*args, **kws)
    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'

    input = <bound method GeneralSupport.input of <aenum 'GeneralSupport'>>

    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'

output = <bound method GeneralSupport.output of <aenum 'GeneralSupport'>>

class hvl_ccb.dev.supercube2015.constants.MeasurementsDividerRatio(*args,
                                                                    **kws)
    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.

    get = <bound method MeasurementsDividerRatio.get of <aenum 'MeasurementsDividerRatio'>>

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

class hvl_ccb.dev.supercube2015.constants.MeasurementsScaledInput(*args,
                                                                    **kws)
    Bases: hvl_ccb.utils.enum.ValueEnum

    Variable NodeID strings for the four analog BNC inputs for measuring voltage. The voltage re-
    turned in these variables is already scaled with the set ratio, which can be read using the variables in
    MeasurementsDividerRatio.

    get = <bound method MeasurementsScaledInput.get of <aenum 'MeasurementsScaledInput'>>

    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 hvl_ccb.dev.supercube2015.constants.Power(*args, **kws)
    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 hvl_ccb.dev.supercube2015.constants.PowerSetup(*args, **kws)
    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 hvl_ccb.dev.supercube2015.constants.Safety(*args, **kws)
    Bases: hvl_ccb.utils.enum.ValueEnum

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

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

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

status_ready_for_red = 'hvl-ipc.WINAC.SYSTEMSTATE.ReadyForRed'
    Status is a read-only integer containing the state number of the supercube-internal state machine. The
    values correspond to numbers in SafetyStatus.

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

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

switchto_operate = 'hvl-ipc.WINAC.SYSTEMSTATE.switchon'
    Writable boolean for switching to Red Operate (locket, HV on) state.

switchto_ready = 'hvl-ipc.WINAC.SYSTEMSTATE.RED_REQUEST'
    Writable boolean for switching to Red Ready (locked, HV off) state.

```

```
class hvl_ccb.dev.supercube2015.constants.SafetyStatus (*args, **kwargs)
    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 hvl_ccb.dev.supercube2015.constants.SupercubeOpcEndpoint (*args, **kwargs)
    Bases: hvl_ccb.utils.enum.ValueEnum

    OPC Server Endpoint strings for the supercube variants.

    A = 'OPC.SimaticNET.S7'
    B = 'OPC.SimaticNET.S7'

hvl_ccb.dev.supercube2015.constants.T13_SOCKET_PORTS = (1, 2, 3)
    Port numbers of SEV T13 power socket
```

hvl_ccb.dev.supercube2015.typ_a module

Supercube Typ A module.

```
class hvl_ccb.dev.supercube2015.typ_a.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 () → hv1_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 hv1_ccb.dev.supercube2015.typ_a.**SupercubeAOpcUaCommunication** (*config*)

Bases: *hv1_ccb.dev.supercube2015.base.SupercubeOpcUaCommunication*

static config_cls ()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

```
class hvl_ccb.dev.supercube2015.typ_a.SupercubeAOpcUaConfiguration (host:
                                                                    str, end-
                                                                    point_name:
                                                                    str =
                                                                    'OPC.SimaticNET.S7',
                                                                    port: int
                                                                    = 4845,
                                                                    sub_handler:
                                                                    hvl_ccb.comm.opc.OpcUaSubHana
                                                                    =
                                                                    <hvl_ccb.dev.supercube2015.base.
                                                                    object at
                                                                    0x7f2f8698fbe0>,
                                                                    up-
                                                                    date_period:
                                                                    int = 500)
```

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

endpoint_name = 'OPC.SimaticNET.S7'

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

Module with base classes for devices.

class `hvl_ccb.dev.base.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

start () → None

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

stop () → None

Stop this Device. To be implemented in the subclass.

exception `hvl_ccb.dev.base.DeviceExistingException`

Bases: `Exception`

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

class `hvl_ccb.dev.base.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` –

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.

stop () → None

Stop all devices in this sequence in their reverse order.

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

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

Base class for devices with a single communication protocol.

com

Get the communication protocol of this device.

Returns an instance of CommunicationProtocol subtype

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 module

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 `hvl_ccb.dev.crylas.CryLasAttenuator` (*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

Device class for the CryLas laser attenuator.

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

transmission

class `hvl_ccb.dev.crylas.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 = 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 = 1

exception hvl_ccb.dev.crylas.CryLasAttenuatorError

Bases: Exception

General error with the CryLas Attenuator.

class hvl_ccb.dev.crylas.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 hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig (port: str,
                                                                    baudrate:
                                                                    int      =
                                                                    9600,
                                                                    parity:
                                                                    Union[str,
                                                                    hvl_ccb.comm.serial.SerialCommu
                                                                    = <Serial-
                                                                    Communi-
                                                                    cationPar-
                                                                    ity.NONE:
                                                                    'N'>,
                                                                    stopbits:
                                                                    Union[int,
                                                                    hvl_ccb.comm.serial.SerialCommu
                                                                    = <Seri-
                                                                    alCom-
                                                                    munica-
                                                                    tionStop-
                                                                    bits.ONE:
                                                                    1>, byte-
                                                                    size:
                                                                    Union[int,
                                                                    hvl_ccb.comm.serial.SerialCommu
                                                                    = <Seri-
                                                                    alCom-
                                                                    munica-
                                                                    tionByte-
                                                                    size.EIGHTBITS:
                                                                    8>, ter-
                                                                    minator:
                                                                    bytes
                                                                    =      b",
                                                                    timeout:
                                                                    Union[int,
                                                                    float]   =
                                                                    3)

```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate = 9600

Baudrate for CryLas attenuator is 9600 baud

bytesize = 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 = '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 = 1

CryLas attenuator uses one stop bit

terminator = b''

No terminator

timeout = 3

use 3 seconds timeout as default

class `hvl_ccb.dev.crylas.CryLasLaser` (*com*, *dev_config=None*)

Bases: `hvl_ccb.dev.base.SingleCommDevice`

CryLas laser controller device class.

class `AnswersShutter` (**args*, ***kws*)

Bases: `aenum.Enum`

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

CLOSED = 'Shutter inaktiv'

OPENED = 'Shutter aktiv'

class `AnswersStatus` (**args*, ***kws*)

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` (**args*, ***kws*)

Bases: `aenum.Enum`

Status of the CryLas laser

READY_INACTIVE = 1

READ_ACTIVE = 2

```

UNREADY_INACTIVE = 0

is_inactive

is_ready

class RepetitionRates(*args, **kws)
    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 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

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 hvl_ccb.dev.crylas.CryLasLaserConfig (calibration_factor: Union[int, float]
                                           = 4.35, polling_period: Union[int,
float] = 5, polling_timeout: Union[int,
float] = 300, on_start_wait_until_ready:
bool = False, auto_laser_on: bool =
True, init_shutter_status: Union[int,
hvl_ccb.dev.crylas.CryLasLaserShutterStatus]
                                           = <CryLasLaserShutterStatus.CLOSED: 0>)
```

Bases: object

Device configuration dataclass for the CryLas laser controller.

ShutterStatus

alias of *CryLasLaserShutterStatus*

auto_laser_on = True

calibration_factor = 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 = 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.

on_start_wait_until_ready = False

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 = 5

polling_timeout = 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 `hvl_ccb.dev.crylas.CryLasLaserError`

Bases: `Exception`

General error with the CryLas Laser.

exception `hvl_ccb.dev.crylas.CryLasLaserNotReadyError`

Bases: `hvl_ccb.dev.crylas.CryLasLaserError`

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

class `hvl_ccb.dev.crylas.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: 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

```
class hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig (port: str, baudrate: int = 19200, parity: Union[str, hvl_ccb.comm.serial.SerialCommunicationParity.NONE: 'N'], stopbits: Union[int, hvl_ccb.comm.serial.SerialCommunicationStopbits.ONE: 1], bytesize: Union[int, hvl_ccb.comm.serial.SerialCommunicationByteSize.EIGHTBITS: 8], terminator: bytes = b'n', timeout: Union[int, float] = 3)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate = 19200

Baudrate for CryLas laser is 19200 baud

bytesize = 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 = '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 = 1

CryLas laser uses one stop bit

terminator = b'\n'

The terminator is LF

timeout = 3

use 3 seconds timeout as default

class hvl_ccb.dev.crylas.**CryLasLaserShutterStatus** (*args, **kws)

Bases: aenum.Enum

Status of the CryLas laser shutter

CLOSED = 0

OPENED = 1

hvl_ccb.dev.ea_psi9000 module

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 hvl_ccb.dev.ea_psi9000.**PSI9000** (com: Union[hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunication, hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig, dict], dev_config: Union[hvl_ccb.dev.ea_psi9000.PSI9000Config, dict, None] = 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: float = None, current_limit: 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: float = None, current_limit: float = None, power_limit: 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 hvl_ccb.dev.ea_psi9000.PSI9000Config (poll_interval: Union[int, float] = 0.5,  
poll_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 = 0.0

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

current_upper_limit = 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 = 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 = 0.0

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

voltage_upper_limit = 10.0

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

wait_sec_initialisation = 2

wait_sec_settings_effect = 1

wait_sec_system_lock = 0.5

exception hvl_ccb.dev.ea_psi9000.PSI9000Error

Bases: Exception

Base error class regarding problems with the PSI 9000 supply.

class hvl_ccb.dev.ea_psi9000.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 hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig (host: str, interface_type: Union[str, hvl_ccb.comm.visa.VisaCommunicationConfig], = <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 = 1

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.heinzinger module

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 hvl_ccb.dev.heinzinger.HeinzingerConfig (default_number_of_recordings:
                                             Union[int,
                                             hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum]
                                             = 1, number_of_decimals: int = 6,
                                             wait_sec_stop_commands: Union[int,
                                             float] = 0.5)
```

Bases: `object`

Device configuration dataclass for Heinzinger power supplies.

```
class RecordingsEnum
```

Bases: `enum.IntEnum`

An enumeration.

```
EIGHT = 8
```

```
FOUR = 4
```

```
ONE = 1
```

```
SIXTEEN = 16
```

```
TWO = 2
```

```
clean_values ()
```

```
default_number_of_recordings = 1
```

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

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

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

Parameters

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

```
is_configdataclass = True
```

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

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

```
number_of_decimals = 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 = 0.5

Time to wait after subsequent commands during stop (in seconds)

class hvl_ccb.dev.heinzinger.**HeinzingerDI** (*com, dev_config=None*)

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

Heinzinger Digital Interface I/II device class

Sends basic SCPI commands and reads the answer. Only the standard instruction set from the manual is implemented.

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.

Raises *SerialCommunicationIOError* – when communication port is not opened

output_on () → None

Switch DC voltage output on.

Raises *SerialCommunicationIOError* – when communication port is not opened

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

start ()

Opens the communication protocol.

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

stop () → None

Stop the device. Closes also the communication protocol.

class hvl_ccb.dev.heinzinger.**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** (*args, **kws)

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

A = 3

UNKNOWN = 1

mA = 2

class **UnitVoltage** (*args, **kws)

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

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

max_current

max_current_hardware

max_voltage

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.

unit_current

unit_voltage

exception *hvl_ccb.dev.heinzinger.HeinzingerPNCDeviceNotRecognizedException*
Bases: *hvl_ccb.dev.heinzinger.HeinzingerPNCError*
Error indicating that the serial number of the device is not recognized.

exception *hvl_ccb.dev.heinzinger.HeinzingerPNCError*
Bases: *Exception*
General error with the Heinzinger PNC voltage source.

exception *hvl_ccb.dev.heinzinger.HeinzingerPNCMaxCurrentExceededException*
Bases: *hvl_ccb.dev.heinzinger.HeinzingerPNCError*
Error indicating that program attempted to set the current to a value exceeding 'max_current'.

exception *hvl_ccb.dev.heinzinger.HeinzingerPNCMaxVoltageExceededException*
Bases: *hvl_ccb.dev.heinzinger.HeinzingerPNCError*
Error indicating that program attempted to set the voltage to a value exceeding 'max_voltage'.

class *hvl_ccb.dev.heinzinger.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

read_text_nonempty (*n_attempts_max: int = 40*) → str
Reads from the serial port, until a non-empty line is found, or the number of attempts is exceeded.

Parameters

- **n_attempts_max** – maximum number of read attempts
- **attempt_interval_sec** – interval between subsequent attempts in seconds

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

```
class hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig (port:      str,
                                                                baudrate:
                                                                int    = 9600,
                                                                parity:
                                                                Union[str,
                                                                hvl_ccb.comm.serial.SerialCommunicationParity.NONE:
                                                                'N'>,
                                                                stopbits:
                                                                Union[int,
                                                                hvl_ccb.comm.serial.SerialCommunicationStopbits.ONE:
                                                                1>,
                                                                bytesize:
                                                                Union[int,
                                                                hvl_ccb.comm.serial.SerialCommunicationByteSize.EIGHTBITS:
                                                                8>,
                                                                terminator: bytes =
                                                                b'n',
                                                                timeout:
                                                                Union[int,
                                                                float]   = 3,
                                                                wait_sec_read_text_nonempty:
                                                                Union[int,
                                                                float]   = 0.5)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate = 9600

Baudrate for Heinzinger power supplies is 9600 baud

bytesize = 8

One byte is eight bits long

clean_values ()

force_value (fieldname, value)

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

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

Parameters

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

classmethod keys () → Sequence[str]

Returns a list of all configdataclass fields key-names.

Returns a list of strings containing all keys.

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

parity = '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 = 1

Heinzinger uses one stop bit

terminator = b'\n'

The terminator is LF

timeout = 3

use 3 seconds timeout as default

wait_sec_read_text_nonempty = 0.5

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

hvl_ccb.dev.labjack module

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 hvl_ccb.dev.labjack.**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** (*args, **kws)

Bases: *hvl_ccb.utils.enum.StrEnumBase*

ONE = 1.0

ONE_HUNDREDTH = 0.01

ONE_TENTH = 0.1

TEN = 10.0

value

```

class CalMicroAmpere(*args, **kws)
    Bases: aenum.Enum

    Pre-defined microampere (uA) values for calibration current source query.

    TEN = '10uA'

    TWO_HUNDRED = '200uA'

class CjcType(*args, **kws)
    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(*args, **kws)
    Bases: aenum.IntEnum

    State of a digital I/O channel.

    HIGH = 1

    LOW = 0

class DeviceType(*args, **kws)
    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

    get_by_p_id = <bound method DeviceType.get_by_p_id of <aenum 'DeviceType'>>

class TemperatureUnit(*args, **kws)
    Bases: hvl_ccb.utils.enum.NameEnum

    Temperature unit (to be returned)

    C = 1

    F = 2

    K = 0

class ThermocoupleType(*args, **kws)
    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) → 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_sbusrh (*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_sbustemp (*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 `hvl_ccb.dev.labjack.LabJackError`

Bases: *Exception*

Errors of the LabJack device.

exception `hvl_ccb.dev.labjack.LabJackIdentifierDIOError`

Bases: *Exception*

Error indicating a wrong DIO identifier

hvl_ccb.dev.mbw973 module

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 hvl_ccb.dev.mbw973.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[CT_co] = <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 `hvl_ccb.dev.mbw973.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 = 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 `hvl_ccb.dev.mbw973.MBW973ControlRunningException`

Bases: *hvl_ccb.dev.mbw973.MBW973Error*

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

exception `hvl_ccb.dev.mbw973.MBW973Error`

Bases: `Exception`

General error with the MBW973 dew point mirror device.

exception `hvl_ccb.dev.mbw973.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 `hvl_ccb.dev.mbw973.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 hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig (port: str, baudrate: int =
    9600, parity: Union[str,
    hvl_ccb.comm.serial.SerialCommunicationParity]
    = <SerialCommunica-
    tionParity.NONE: 'N'>,
    stopbits: Union[int,
    hvl_ccb.comm.serial.SerialCommunicationStopbits]
    = <SerialCommunica-
    tionStopbits.ONE:
    1>, bytesize: Union[int,
    hvl_ccb.comm.serial.SerialCommunicationBytesize]
    = <SerialCom-
    municationByte-
    size.EIGHTBITS: 8>,
    terminator: bytes = b'r',
    timeout: Union[int, float]
    = 3)
```

Bases: `hvl_ccb.comm.serial.SerialCommunicationConfig`

baudrate = 9600

Baudrate for MBW973 is 9600 baud

bytesize = 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 = '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 = 1

MBW973 does use one stop bit

terminator = b'\r'

The terminator is only CR

timeout = 3

use 3 seconds timeout as default

class hvl_ccb.dev.mbw973.**Poller** (*period: float, callback: Callable*)

Bases: object

Wrapper for the threading.Timer class to periodically poll data.

start () → None

Start the polling timer.

stop () → None

Stop the polling timer.

timer_callback () → None

Callback method that is called every time the timer elapses. It calls the specified user callback function and restarts the timer.

hvl_ccb.dev.newport module

Device class for Newport SMC100PP stepper motor controller with serial communication.

The SMC100PP is a single axis motion controller/driver for stepper motors up to 48 VDC at 1.5 A rms. Up to 31 controllers can be networked through the internal RS-485 communication link.

Manufacturer homepage: <https://www.newport.com/f/smc100-single-axis-dc-or-stepper-motion-controller>

class hvl_ccb.dev.newport.**NewportConfigCommands** (*args, **kws)

Bases: *hvl_ccb.utils.enum.NameEnum*

Commands predefined by the communication protocol of the SMC100PP

AC = 'acceleration'

BA = 'backlash_compensation'

BH = 'hysteresis_compensation'

FRM = 'micro_step_per_full_step_factor'

FRS = 'motion_distance_per_full_step'

HT = 'home_search_type'

```

JR = 'jerk_time'
OH = 'home_search_velocity'
OT = 'home_search_timeout'
QIL = 'peak_output_current_limit'
SA = 'rs485_address'
SL = 'negative_software_limit'
SR = 'positive_software_limit'
VA = 'velocity'
VB = 'base_velocity'
ZX = 'stage_configuration'

exception hvl_ccb.dev.newport.NewportControllerError
    Bases: Exception

    Error with the Newport controller.

exception hvl_ccb.dev.newport.NewportMotorError
    Bases: Exception

    Error with the Newport motor.

class hvl_ccb.dev.newport.NewportSMC100PP (com, dev_config=None)
    Bases: hvl_ccb.dev.base.SingleCommDevice

    Device class of the Newport motor controller SMC100PP

    class MotorErrors (*args, **kws)
        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 (*args, **kws)
        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 *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: 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: 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: 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: 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: 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: 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: 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

get_positive_software_limit (*add: 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: 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: 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*: 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*: 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*: 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*: 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*: 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: int = None, config: 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: 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: 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*: 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*: 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

wait_until_move_finished (*add*: int = None) → None

Wait until the motor leaves the MOVING state (at which point it should have arrived to the target 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 hvl_ccb.dev.newport.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_finished_extra_wait_sec:
Union[int, float] = 1, accelera-
tion: Union[int, float] = 10, back-
lash_compensation: Union[int,
float] = 0, hysteresis_compensation:
Union[int, float] = 0.015, mi-
cro_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, posi-
tive_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.EspStageConfi,
=
<EspStageCon-
fig.EnableEspStageCheck: 3>)
```

Bases: object

Configuration dataclass for the Newport motor controller SMC100PP.

```
class EspStageConfig (*args, **kws)
```

Bases: aenum.IntEnum

Different configurations to check or not the motor configuration upon power-up.

```
DisableEspStageCheck = 1
```

```
EnableEspStageCheck = 3
```

```
UpdateEspStageInfo = 2
```

```
class HomeSearch (*args, **kws)
```

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 = 10
    address = 1
    backlash_compensation = 0
    base_velocity = 0
    clean_values ()
    exit_configuration_wait_sec = 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 = 1
home_search_timeout = 27.5
home_search_type = 2
home_search_velocity = 4
hysteresis_compensation = 0.015
is_configdataclass = True
jerk_time = 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 = 100
motion_distance_per_full_step = 0.01

```

motor_config

Gather the configuration parameters of the motor into a dictionary.

Returns dict containing the configuration parameters of the motor

```

move_finished_extra_wait_sec = 1
negative_software_limit = -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 = 0.4
positive_software_limit = 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` = 2

`screw_scaling` = 1

`stage_configuration` = 3

`user_position_offset` = 23.987

`velocity` = 4

class `hvl_ccb.dev.newport.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` (**args*, ***kws*)

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'

static config_cls()

Return the default configdataclass class.

Returns a reference to the default configdataclass class

query (*add: int, cmd: str, param: Union[int, float, str, None] = 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

send_command (*add: int, cmd: str, param: Union[int, float, str, None] = 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 hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig(port: str,  
                                                                    baudrate:  
                                                                    int      =  
                                                                    57600,  
                                                                    parity:  
                                                                    Union[str,  
                                                                    hvl_ccb.comm.serial.SerialCommu  
                                                                    = <Serial-  
                                                                    Communi-  
                                                                    cationPar-  
                                                                    ity.NONE:  
                                                                    'N'>,  
                                                                    stopbits:  
                                                                    Union[int,  
                                                                    hvl_ccb.comm.serial.SerialCommu  
                                                                    = <Seri-  
                                                                    alCom-  
                                                                    munica-  
                                                                    tionStop-  
                                                                    bits.ONE:  
                                                                    1>,  
                                                                    byte-  
                                                                    size:  
                                                                    Union[int,  
                                                                    hvl_ccb.comm.serial.SerialCommu  
                                                                    = <Seri-  
                                                                    alCom-  
                                                                    munica-  
                                                                    tionByte-  
                                                                    size.EIGHTBITS:  
                                                                    8>,  
                                                                    ter-  
                                                                    minator:  
                                                                    bytes    =  
                                                                    b'rn',  
                                                                    timeout:  
                                                                    Union[int,  
                                                                    float]   =  
                                                                    10)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate = 57600

Baudrate for Heinzinger power supplies is 9600 baud

bytesize = 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 = '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 = 1

Heinzinger uses one stop bit

terminator = b'\r\n'

The terminator is CR/LF

timeout = 10

use 10 seconds timeout as default

exception hvl_ccb.dev.newport.**NewportSerialCommunicationError**

Bases: Exception

Communication error with the Newport controller.

class hvl_ccb.dev.newport.**NewportStates** (*args, **kws)

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

States of the Newport controller. Certain commands are allowed only in certain states.

CONFIG = 3

DISABLE = 6

HOMING = 2

JOGGING = 7

MOVING = 5

NO_REF = 1

READY = 4

hvl_ccb.dev.pfeiffer_tpg module

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 hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG (com, dev_config=None)
```

Bases: *hvl_ccb.dev.base.SingleCommDevice*

Pfeiffer TPG control unit device class

```
class PressureUnits (*args, **kws)
```

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
```

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
```

Bases: *enum.Enum*

An enumeration.

CMR = 4

IKR = 2

IKR11 = 2

IKR9 = 2

IMR = 5

```

None = 7
PBR = 6
PKR = 3
TPR = 1
noSENSOR = 7
noSen = 7
static config_cls ()
    Return the default configdataclass class.

    Returns a reference to the default configdataclass class
static default_com_cls ()
    Get the class for the default communication protocol used with this device.

    Returns the type of the standard communication protocol for this device
get_full_scale_mbar () → List[Union[int, float]]
    Get the full scale range of the attached sensors

    Returns full scale range values in mbar, like [0.01, 1, 0.1, 1000, 50000, 10]

    Raises
        • SerialCommunicationIOError – when communication port is not opened
        • PfeifferTPGError – if command fails
get_full_scale_unitless () → List[int]
    Get the full scale range of the attached sensors. See lookup table between command and corresponding
    pressure in the device user manual.

    Returns list of full scale range values, like [0, 1, 3, 3, 2, 0]

    Raises
        • SerialCommunicationIOError – when communication port is not opened
        • PfeifferTPGError – if command fails
identify_sensors () → None
    Send identification request TID to sensors on all channels.

    Raises
        • SerialCommunicationIOError – when communication port is not opened
        • PfeifferTPGError – if command fails
measure (channel: int) → Tuple[str, float]
    Get the status and measurement of one sensor

    Parameters channel – int channel on which the sensor is connected, with
    1 <= channel <= number_of_sensors :return: measured value as float if measurement successful, sensor
    status as string if not :raises SerialCommunicationIOError: when communication port is not opened :raises
    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 :raises PfeifferTPGError: if command fails

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.

unit
The pressure unit of readings is always mbar, regardless of the display unit.

```
class hvl_ccb.dev.pfeiffer_tpg.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 (*args, **kwargs)

Bases: *hvl_ccb.utils.enum.NameEnum*

TPG25xA = {0.1: 8, 1: 0, 10: 1, 100: 2, 1000: 3, 2000: 4, 5000: 5, 10000: 6, 50000: 7, 0.1: 8}

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 = {0.1: 8, 1: 0, 10: 1, 100: 2, 1000: 3, 2000: 4, 5000: 5, 10000: 6, 50000: 7}

classmethod optional_defaults () → Dict[str, object]

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

Returns a list of strings containing all optional keys.

classmethod required_keys () → Sequence[str]

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

Returns a list of strings containing all required keys.

exception hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGError

Bases: Exception

Error with the Pfeiffer TPG Controller.

class hvl_ccb.dev.pfeiffer_tpg.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 hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig(port:  
                                                                    str, baud-  
                                                                    rate:  
                                                                    int      =  
                                                                    9600,  
                                                                    parity:  
                                                                    Union[str,  
                                                                    hvl_ccb.comm.serial.SerialComm  
                                                                    = <Seri-  
                                                                    alCom-  
                                                                    munica-  
                                                                    tionPar-  
                                                                    ity.NONE:  
                                                                    'N'>,  
                                                                    stopbits:  
                                                                    Union[int,  
                                                                    hvl_ccb.comm.serial.SerialComm  
                                                                    = <Seri-  
                                                                    alCom-  
                                                                    munica-  
                                                                    tionStop-  
                                                                    bits.ONE:  
                                                                    1>,  
                                                                    bytesize:  
                                                                    Union[int,  
                                                                    hvl_ccb.comm.serial.SerialComm  
                                                                    = <Seri-  
                                                                    alCom-  
                                                                    munica-  
                                                                    tionByte-  
                                                                    size.EIGHTBITS:  
                                                                    8>, ter-  
                                                                    minator:  
                                                                    bytes   =  
                                                                    b'rn',  
                                                                    timeout:  
                                                                    Union[int,  
                                                                    float]  =  
                                                                    3)
```

Bases: *hvl_ccb.comm.serial.SerialCommunicationConfig*

baudrate = 9600

Baudrate for Pfeiffer TPG controllers is 9600 baud

bytesize = 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 = '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 = 1

Pfeiffer TPG controllers use one stop bit

terminator = b'\r\n'

The terminator is <CR><LF>

timeout = 3

use 3 seconds timeout as default

hvl_ccb.dev.rs_rto1024 module

Python module for the Rhode & Schwarz RTO 1024 oscilloscope. The communication to the device is through VISA, type TCPIP / INSTR.

```
class hvl_ccb.dev.rs_rto1024.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 (*args, **kwargs)
```

Bases: *hvl_ccb.utils.enum.AutoNumberNameEnum*

Enumeration for the three available trigger modes.

AUTO = 1

FREERUN = 3

NORMAL = 2

names = <bound method RTO1024.TriggerModes.names of <aenum 'TriggerModes'>>

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 **RTO1024Error** – if the operation did not complete

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 **RTO1024Error** – if the timestamps are invalid

list_directory (*path: str*) → List[Tuple[str, str, int]]
List the contents of a given directory on the oscilloscope filesystem.

Parameters path – is the path to a folder

Returns a list of filenames in the given folder

load_configuration (*filename: str*) → None
Load current settings from a configuration file. The filename has to be specified without base directory and '.dfl' extension.

Information from the manual *ReCall* calls up the instrument settings from an intermediate memory identified by the specified number. The instrument settings can be stored to this memory using the command **SAV* with the associated number. It also activates the instrument settings which are stored in a file and loaded using *MMEMemory: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.

run_continuous_acquisition () → None

Start acquiring continuously.

run_single_acquisition () → None

Start a single or Nx acquisition.

save_configuration (filename: str) → None

Save the current oscilloscope settings to a file. The filename has to be specified without path and '.dff' extension, the file will be saved to the configured settings directory.

Information from the manual *SAVE* stores the current instrument settings under the specified number in an intermediate memory. The settings can be recalled using the command **RCL* with the associated number. To transfer the stored instrument settings to a file, use *MMEMory:STORe:STAtE* .

Parameters filename – is the name of the settings file without path and extension

save_waveform_history (filename: str, channel: int, waveform: int = 1) → None

Save the history of one channel and one waveform to a .bin file. This function is used after an acquisition using sequence trigger mode (with or without ultra segmentation) was performed.

Parameters

- **filename** – is the name (without extension) of the file
- **channel** – is the channel number
- **waveform** – is the waveform number (typically 1)

Raises *RT01024Error* – if storing waveform times out

set_acquire_length (timerange: float) → None

Defines the time of one acquisition, that is the time across the 10 divisions of the diagram.

- Range: 250E-12 ... 500 [s]
- Increment: 1E-12 [s]
- *RST = 0.5 [s]

Parameters timerange – is the time for one acquisition. Range: 250e-12 ... 500 [s]

set_channel_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 Ω input coupling, the range is 10 mV to 10 V. For 1 M Ω 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 Ω input coupling, the vertical scale (input sensitivity) is 1 mV/div to 1 V/div. For 1 M Ω 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 hvl_ccb.dev.rs_rto1024.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 hvl_ccb.dev.rs_rto1024.RTO1024Error

Bases: Exception

class hvl_ccb.dev.rs_rto1024.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 hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunicationConfig (host:      str,  in-
                                                                interface_type:
                                                                Union[str,
                                                                hvl_ccb.comm.visa.VisaCommunicationConfig
                                                                =      <Interface-
                                                                Type.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 = 2

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 module

Device class for controlling a Schneider Electric ILS2T stepper drive over modbus TCP.

class hvl_ccb.dev.se_ils2t.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

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

Bases: *enum.IntEnum*

ILS2T device modes

JOG = 1

```
PTP = 3

class Ref16Jog
    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

class RegAddr
    Bases: enum.IntEnum

    ILS2T Modbus Register Adresses

    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 RegDatatype(*args, **kws)
    Bases: aenum.Enum

    Modbus Register Datatypes

    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

class State

Bases: `enum.IntEnum`

State machine status values

ON = 6

QUICKSTOP = 7

READY = 4

absolute_position (*position: int*) → None

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.

absolute_position_and_wait (*position: int*) → None

Enable motor, perform absolute position and wait until done, disable.

Parameters position – absolute position of motor in user defined steps.

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 () → None

Disable the driver of the stepper motor and enable the brake.

do_ioscanning_write (***kwargs*) → 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.

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.

relative_step (*steps: int*) → None

Turn the motor the relative amount of steps. This function does not enable or disable the motor automatically. positive numbers -> CW negative numbers -> CCW

Parameters **steps** – Number of steps to turn the motor.

relative_step_and_wait (*steps: int*) → None

Enable motor, perform relative steps and wait until done, disable.

Parameters **steps** – Number of steps.

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*.

```
class hvl_ccb.dev.se_ils2t.ILS2TConfig (rpm_max_init: numbers.Integral = 1500,  
                                         wait_sec_post_enable: Union[float, int] = 1,  
                                         wait_sec_post_relative_step: Union[float, int] = 2,  
                                         wait_sec_post_absolute_position: Union[float, int]  
                                         = 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 = 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_post_absolute_position = 2

wait_sec_post_enable = 1

wait_sec_post_relative_step = 2

exception `hvl_ccb.dev.se_ils2t.ILS2TException`

Bases: Exception

Exception to indicate problems with the SE ILS2T stepper motor.

class `hvl_ccb.dev.se_ils2t.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 `hvl_ccb.dev.se_ils2t.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 = 255

The unit has to be 255 such that IO scanning mode works.

exception `hvl_ccb.dev.se_ils2t.IoScanningModeValueError`

Bases: `hvl_ccb.dev.se_ils2t.ILS2TException`

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

exception `hvl_ccb.dev.se_ils2t.ScalingFactorValueError`

Bases: `hvl_ccb.dev.se_ils2t.ILS2TException`

Exception to indicate that a scaling factor value is invalid.

hvl_ccb.dev.visa module

class `hvl_ccb.dev.visa.VisaDevice` (*com:* `Union[hvl_ccb.comm.visa.VisaCommunication, hvl_ccb.comm.visa.VisaCommunicationConfig, dict],
dev_config: Union[hvl_ccb.dev.visa.VisaDeviceConfig, dict, None] = 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 hvl_ccb.dev.visa.**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.

class hvl_ccb.dev.visa.**VisaStatusPoller** (*target: Callable, interval: float = 0.5, start_delay:*
float = 5)

Bases: threading.Thread

Thread to periodically poll the status byte of a VISA device.

run () → None

Threaded method.

stop () → None

Gracefully stop the poller.

Module contents

Devices subpackage.

hvl_ccb.utils package

Submodules

hvl_ccb.utils.enum module

class hvl_ccb.utils.enum.**AutoNumberNameEnum**(*args, **kws)

Bases: *hvl_ccb.utils.enum.StrEnumBase*, *aenum.AutoNumberEnum*

Auto-numbered enum with names used as string representation, and with lookup and equality based on this representation.

class hvl_ccb.utils.enum.**NameEnum**(*args, **kws)

Bases: *hvl_ccb.utils.enum.StrEnumBase*

Enum with names used as string representation, and with lookup and equality based on this representation.

class hvl_ccb.utils.enum.**StrEnumBase**(*args, **kws)

Bases: *aenum.Enum*

String representation-based equality and lookup.

class hvl_ccb.utils.enum.**ValueEnum**(*args, **kws)

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 module

Additional Python typing module utilities

hvl_ccb.utils.typing.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

hvl_ccb.utils.typing.is_generic(type_)

Check if class is a user-defined generic type, for example *Union[int, float]* but not *List*.

Parameters **type** – type to check

hvl_ccb.utils.typing.is_type_hint(type_)

Check if class is a generic type, for example *Union* or *List[int]*

Parameters **type** – type to check

Module contents

4.1.2 Submodules

hvl_ccb.configuration module

Facilities providing classes for handling configuration for communication protocols and devices.

class `hvl_ccb.configuration.ConfigurationMixin` (*configuration*)

Bases: `abc.ABC`

Mixin providing configuration to a class.

config

ConfigDataclass property.

Returns the configuration

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

`hvl_ccb.configuration.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 module

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 `hvl_ccb.experiment_manager.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 `hvl_ccb.experiment_manager.ExperimentManager` (*devices: Dict[str, hvl_ccb.dev.base.Device]*)

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* –

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()

status

Get experiment status.

Returns experiment status enum code.

stop () → None

Alias for ExperimentManager.finish()

class hvl_ccb.experiment_manager.**ExperimentStatus**

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.

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/
$ python setup.py develop
$ 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
```

- 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 (including an entry in HISTORY.rst). 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.

Optionally, go to https://gitlab.com/ethz_hvl/hvl_ccb/tags/vM.N.P/release/edit and add release notes (e.g. changes lists).

6.1 Development Lead

- Mikołaj Rybiński <mikolaj.rybinski@id.ethz.ch>
- (previously) David Graber <graber@eeh.ee.ethz.ch>

6.2 Contributors

- Henrik Menne <henrik.menne@eeh.ee.ethz.ch>
- Alise Chachereau <chachereau@eeh.ee.ethz.ch>

7.1 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).

7.2 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.

7.3 0.3.3 (2019-05-08)

- Use PyPI labjack-ljm (no external dependencies)

7.4 0.3.2 (2019-05-08)

- INSTALLATION.rst with LJMPython prerequisite info

7.5 0.3.1 (2019-05-02)

- readthedocs.org support

7.6 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)

7.7 0.2.1 (2019-04-01)

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

7.8 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.

7.9 0.1.0 (2019-02-06)

- Communication protocol base and serial communication implementation.
- Device base and MBW973 implementation.

CHAPTER 8

Indices and tables

- `genindex`
- `modindex`
- `search`

h

- [hvl_ccb](#), 117
- [hvl_ccb.comm](#), 19
 - [hvl_ccb.comm.base](#), 7
 - [hvl_ccb.comm.labjack_ljm](#), 7
 - [hvl_ccb.comm.modbus_tcp](#), 10
 - [hvl_ccb.comm.opc](#), 11
 - [hvl_ccb.comm.serial](#), 14
 - [hvl_ccb.comm.visa](#), 17
- [hvl_ccb.configuration](#), 116
- [hvl_ccb.dev](#), 114
 - [hvl_ccb.dev.base](#), 53
 - [hvl_ccb.dev.crylas](#), 55
 - [hvl_ccb.dev.ea_psi9000](#), 64
 - [hvl_ccb.dev.heinzinger](#), 68
 - [hvl_ccb.dev.labjack](#), 74
 - [hvl_ccb.dev.mbw973](#), 79
 - [hvl_ccb.dev.newport](#), 82
 - [hvl_ccb.dev.pfeiffer_tpg](#), 96
 - [hvl_ccb.dev.rs_rto1024](#), 101
 - [hvl_ccb.dev.se_ils2t](#), 107
 - [hvl_ccb.dev.supercube](#), 39
 - [hvl_ccb.dev.supercube.base](#), 20
 - [hvl_ccb.dev.supercube.constants](#), 25
 - [hvl_ccb.dev.supercube.typ_a](#), 36
 - [hvl_ccb.dev.supercube.typ_b](#), 38
 - [hvl_ccb.dev.supercube2015](#), 52
 - [hvl_ccb.dev.supercube2015.base](#), 39
 - [hvl_ccb.dev.supercube2015.constants](#), 44
 - [hvl_ccb.dev.supercube2015.typ_a](#), 50
 - [hvl_ccb.dev.visa](#), 113
- [hvl_ccb.experiment_manager](#), 116
- [hvl_ccb.utils](#), 115
 - [hvl_ccb.utils.enum](#), 115
 - [hvl_ccb.utils.typing](#), 115

A

- A (*hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitCurrent attribute*), 71
- A (*hvl_ccb.dev.supercube.constants.SupercubeOpcEndpoint attribute*), 35
- A (*hvl_ccb.dev.supercube2015.constants.SupercubeOpcEndpoint attribute*), 50
- ABSOLUTE_POSITION (*hvl_ccb.dev.se_ils2t.ILS2T.ActionsPtp attribute*), 107
- `absolute_position()` (*hvl_ccb.dev.se_ils2t.ILS2T method*), 109
- `absolute_position_and_wait()` (*hvl_ccb.dev.se_ils2t.ILS2T method*), 109
- AC (*hvl_ccb.dev.newport.NewportConfigCommands attribute*), 82
- AC_DoubleStage_150kV (*hvl_ccb.dev.supercube.constants.PowerSetup attribute*), 34
- AC_DoubleStage_150kV (*hvl_ccb.dev.supercube2015.constants.PowerSetup attribute*), 49
- AC_DoubleStage_200kV (*hvl_ccb.dev.supercube.constants.PowerSetup attribute*), 34
- AC_DoubleStage_200kV (*hvl_ccb.dev.supercube2015.constants.PowerSetup attribute*), 49
- AC_SingleStage_100kV (*hvl_ccb.dev.supercube.constants.PowerSetup attribute*), 34
- AC_SingleStage_100kV (*hvl_ccb.dev.supercube2015.constants.PowerSetup attribute*), 49
- AC_SingleStage_50kV (*hvl_ccb.dev.supercube.constants.PowerSetup attribute*), 34
- AC_SingleStage_50kV (*hvl_ccb.dev.supercube2015.constants.PowerSetup attribute*), 49
- acceleration (*hvl_ccb.dev.newport.NewportSMC100PPConfig attribute*), 91
- ACCESS_ENABLE (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute*), 108
- access_lock (*hvl_ccb.comm.base.CommunicationProtocol attribute*), 7
- ACTION_JOG_VALUE (*hvl_ccb.dev.se_ils2t.ILS2T attribute*), 107
- activated (*hvl_ccb.dev.supercube.constants.BreakdownDetection attribute*), 30
- activated (*hvl_ccb.dev.supercube2015.constants.BreakdownDetection attribute*), 45
- ACTIVE (*hvl_ccb.dev.crylas.CryLasLaser.AnswersStatus attribute*), 58
- active (*hvl_ccb.dev.supercube.constants.OpcControl attribute*), 34
- `add_device()` (*hvl_ccb.dev.base.DeviceSequenceMixin method*), 53
- `add_device()` (*hvl_ccb.experiment_manager.ExperimentManager method*), 116
- ADDR_INCORRECT (*hvl_ccb.dev.newport.NewportSMC100PPSerialComm attribute*), 92
- address (*hvl_ccb.comm.visa.VisaCommunicationConfig attribute*), 18
- address (*hvl_ccb.dev.newport.NewportSMC100PPConfig attribute*), 91
- `address()` (*hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType method*), 18
- Alarm0 (*hvl_ccb.dev.supercube2015.constants.AlarmText attribute*), 44
- Alarm1 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 26
- Alarm1 (*hvl_ccb.dev.supercube.constants.AlarmText attribute*), 25
- Alarm1 (*hvl_ccb.dev.supercube2015.constants.AlarmText attribute*), 44
- Alarm10 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 26
- Alarm10 (*hvl_ccb.dev.supercube.constants.AlarmText attribute*), 49

[illegible]

[illegible]

- tribute), 26
- Alarm7 (*hvl_ccb.dev.supercube2015.constants.AlarmText attribute*), 45
- Alarm70 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm71 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm72 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm73 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm74 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm75 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm76 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm77 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm78 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm79 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm8 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm8 (*hvl_ccb.dev.supercube.constants.AlarmText attribute*), 26
- Alarm8 (*hvl_ccb.dev.supercube2015.constants.AlarmText attribute*), 45
- Alarm80 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm81 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm82 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 29
- Alarm83 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm84 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm85 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm86 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm87 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm88 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm89 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm9 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm9 (*hvl_ccb.dev.supercube.constants.AlarmText attribute*), 26
- Alarm9 (*hvl_ccb.dev.supercube2015.constants.AlarmText attribute*), 45
- Alarm90 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm91 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm92 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm93 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm94 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm95 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm96 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm97 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm98 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarm99 (*hvl_ccb.dev.supercube.constants.Alarms attribute*), 30
- Alarms (class in *hvl_ccb.dev.supercube.constants*), 26
- AlarmText (class in *hvl_ccb.dev.supercube.constants*), 25
- AlarmText (class in *hvl_ccb.dev.supercube2015.constants*), 44
- ANY (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType attribute*), 9
- ANY (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.DeviceType attribute*), 9
- ANY (*hvl_ccb.dev.labjack.LabJack.DeviceType attribute*), 75
- attenuation (*hvl_ccb.dev.crylas.CryLasAttenuator attribute*), 55
- AUTO (*hvl_ccb.dev.rs_rto1024.RTO1024.TriggerModes attribute*), 101
- auto_laser_on (*hvl_ccb.dev.crylas.CryLasLaserConfig attribute*), 61
- AutoNumberNameEnum (class in *hvl_ccb.utils.enum*), 115
- ## B
- B (*hvl_ccb.dev.supercube.constants.SupercubeOpcEndpoint attribute*), 35
- B (*hvl_ccb.dev.supercube2015.constants.SupercubeOpcEndpoint attribute*), 50
- BA (*hvl_ccb.dev.newport.NewportConfigCommands attribute*), 82
- backlash_compensation (*hvl_ccb.dev.newport.NewportSMC100PPConfig attribute*), 91
- backup_waveform() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 102

```

bar (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits
    attribute), 96
base_velocity (hvl_ccb.dev.newport.NewportSMC100PPConfig
    attribute), 91
baudrate (hvl_ccb.comm.serial.SerialCommunicationConfig
    attribute), 15
baudrate (hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig
    attribute), 57
baudrate (hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig
    attribute), 63
baudrate (hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig
    attribute), 73
baudrate (hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig
    attribute), 81
baudrate (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig
    attribute), 94
baudrate (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig
    attribute), 100
BH (hvl_ccb.dev.newport.NewportConfigCommands
    attribute), 82
board (hvl_ccb.comm.visa.VisaCommunicationConfig
    attribute), 18
BreakdownDetection (class in
    hvl_ccb.dev.supercube.constants), 30
BreakdownDetection (class in
    hvl_ccb.dev.supercube2015.constants), 45
Bytesize (hvl_ccb.comm.serial.SerialCommunicationConfig
    attribute), 15
bytesize (hvl_ccb.comm.serial.SerialCommunicationConfig
    attribute), 15
bytesize (hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig
    attribute), 57
bytesize (hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig
    attribute), 63
bytesize (hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig
    attribute), 73
bytesize (hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig
    attribute), 81
bytesize (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig
    attribute), 94
bytesize (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig
    attribute), 100
cee16 (hvl_ccb.dev.supercube2015.constants.GeneralSockets
    attribute), 47
check_master_slave_config (in module
    hvl_ccb.utils.typing), 115
clean_values (hvl_ccb.dev.ea_psi9000.PSI9000
    method),
clean_values (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig
    method), 9
clean_values (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig
    method), 11
clean_values (hvl_ccb.comm.opc.OpcUaCommunicationConfig
    method), 18
clean_values (hvl_ccb.comm.serial.SerialCommunicationConfig
    method), 18
clean_values (hvl_ccb.comm.visa.VisaCommunicationConfig
    method), 18
clean_values (hvl_ccb.dev.base.EmptyConfig
    method), 54
clean_values (hvl_ccb.dev.crylas.CryLasAttenuatorConfig
    method), 55
clean_values (hvl_ccb.dev.crylas.CryLasLaserConfig
    method), 61
clean_values (hvl_ccb.dev.ea_psi9000.PSI9000Config
    method), 66
clean_values (hvl_ccb.dev.heinzinger.HeinzingerConfig
    method), 69
clean_values (hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig
    method), 73
clean_values (hvl_ccb.dev.mbw973.MBW973Config
    method), 80
clean_values (hvl_ccb.dev.newport.NewportSMC100PPConfig
    method), 91
clean_values (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig
    method), 99
clean_values (hvl_ccb.dev.se_ils2t.ILS2TConfig
    method), 12
clean_values (hvl_ccb.dev.supercube.base.SupercubeConfiguration
    method), 23
clean_values (hvl_ccb.dev.supercube2015.base.SupercubeConfiguration
    method), 42
close (hvl_ccb.comm.base.CommunicationProtocol
    method), 7
close (hvl_ccb.comm.labjack_ljm.LJMCommunication
    method), 8
close (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication
    method), 10
close (hvl_ccb.comm.opc.OpcUaCommunication
    method), 12
close (hvl_ccb.comm.serial.SerialCommunication
    method), 14
close (hvl_ccb.comm.visa.VisaCommunication
    method), 14

```

C

```

C (hvl_ccb.dev.labjack.LabJack.TemperatureUnit
    attribute), 75
C (hvl_ccb.dev.labjack.LabJack.ThermocoupleType
    attribute), 75
calibration_factor (hvl_ccb.dev.crylas.CryLasLaserConfig
    attribute), 61
cee16 (hvl_ccb.dev.supercube.constants.GeneralSockets
    attribute), 32

```


method), 17

close_shutter() (hvl_ccb.dev.crylas.CryLasLaser method), 59

CLOSED (hvl_ccb.dev.crylas.CryLasLaser.AnswersShutter attribute), 58

CLOSED (hvl_ccb.dev.crylas.CryLasLaserShutterStatus attribute), 64

closed (hvl_ccb.dev.supercube.constants.DoorStatus attribute), 31

closed (hvl_ccb.dev.supercube.constants.EarthingStickStatus attribute), 31

closed (hvl_ccb.dev.supercube2015.constants.DoorStatus attribute), 45

closed (hvl_ccb.dev.supercube2015.constants.EarthingStickStatus attribute), 46

CMD_EXEC_ERROR (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_CC (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_CONFIGURATION (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_DISABLE (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_HOMING (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_MOVING (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_NOT_REFERENCED (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_PP (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMD_NOT_ALLOWED_READY (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CMR (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes attribute), 96

CODE_OR_ADDR_INVALID (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

com (hvl_ccb.dev.base.SingleCommDevice attribute), 54

COM_TIMEOUT (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92

CommunicationProtocol (class in hvl_ccb.comm.base), 7

config (hvl_ccb.configuration.ConfigurationMixin attribute), 116

CONFIG (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 83

CONFIG (hvl_ccb.dev.newport.NewportStates attribute), 95

config_cls() (hvl_ccb.comm.labjack_ljm.LJMCommunication static method), 8

config_cls() (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication static method), 10

config_cls() (hvl_ccb.comm.opc.OpcUaCommunication static method), 12

config_cls() (hvl_ccb.comm.serial.SerialCommunication static method), 14

config_cls() (hvl_ccb.comm.visa.VisaCommunication static method), 17

ConfigurationMixin (class in hvl_ccb.configuration), 6

Control (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 53

config_cls() (hvl_ccb.dev.crylas.CryLasAttenuator static method), 55

config_cls() (hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunication static method), 56

Control (hvl_ccb.dev.crylas.CryLasLaser static method), 59

config_cls() (hvl_ccb.dev.crylas.CryLasLaserSerialCommunication static method), 59

config_cls() (hvl_ccb.dev.ea_psi9000.PSI9000 static method), 65

Control (hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunication static method), 67

config_cls() (hvl_ccb.dev.heinzinger.HeinzingerDI static method), 72

config_cls() (hvl_ccb.dev.heinzinger.HeinzingerSerialCommunication static method), 72

Control (hvl_ccb.dev.mbw973.MBW973 static method), 79

config_cls() (hvl_ccb.dev.mbw973.MBW973SerialCommunication static method), 84

Control (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication static method), 92

config_cls() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG static method), 97

config_cls() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunication static method), 97

config_cls() (hvl_ccb.dev.rs_rto1024.RTO1024 static method), 102

config_cls() (hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunication static method), 106

config_cls() (hvl_ccb.dev.se_ils2t.ILS2T static method), 109

config_cls() (hvl_ccb.dev.se_ils2t.ILS2TModbusTcpCommunication static method), 109

static method), 112

config_cls() (hvl_ccb.dev.supercube.base.SupercubeBase static method), 20

config_cls() (hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunication static method), 23

config_cls() (hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaCommunication static method), 36

config_cls() (hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaCommunication static method), 38

config_cls() (hvl_ccb.dev.supercube2015.base.Supercube2015Base static method), 39

config_cls() (hvl_ccb.dev.supercube2015.base.Supercube2015BaseOpcUaCommunication static method), 43

config_cls() (hvl_ccb.dev.supercube2015.typ_a.SupercubeAOpcUaCommunication static method), 51

config_cls() (hvl_ccb.dev.visa.VisaDevice static method), 113

configdataclass() (in module hvl_ccb.configuration), 116

configuration_save_json() (hvl_ccb.configuration.ConfigurationMixin method), 116

ConfigurationMixin (class in hvl_ccb.configuration), 116

connection_type (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig attribute), 9

create_serial_port() (hvl_ccb.comm.serial.SerialCommunicationConfig method), 15

CryLasAttenuator (class in hvl_ccb.dev.crylas), 55

CryLasAttenuatorConfig (class in hvl_ccb.dev.crylas), 55

CryLasAttenuatorError, 56

CryLasAttenuatorSerialCommunication (class in hvl_ccb.dev.crylas), 56

CryLasAttenuatorSerialCommunicationConfig (class in hvl_ccb.dev.crylas), 56

CryLasLaser (class in hvl_ccb.dev.crylas), 58

CryLasLaser.AnswersShutter (class in hvl_ccb.dev.crylas), 58

CryLasLaser.AnswersStatus (class in hvl_ccb.dev.crylas), 58

CryLasLaser.LaserStatus (class in hvl_ccb.dev.crylas), 58

CryLasLaser.RepetitionRates (class in hvl_ccb.dev.crylas), 59

CryLasLaserConfig (class in hvl_ccb.dev.crylas), 61

CryLasLaserError, 62

CryLasLaserNotReadyError, 62

CryLasLaserSerialCommunication (class in hvl_ccb.dev.crylas), 62

CryLasLaserSerialCommunicationConfig (class in hvl_ccb.dev.crylas), 63

CryLasLaserShutterStatus (class in hvl_ccb.dev.crylas), 64

current_lower_limit (hvl_ccb.dev.ea_psi9000.PSI9000Config attribute), 66

current_primary (hvl_ccb.dev.supercube.constants.Power attribute), 74

current_primary (hvl_ccb.dev.supercube2015.constants.Power attribute), 78

current_upper_limit (hvl_ccb.dev.ea_psi9000.PSI9000Config attribute), 66

datachange_notification() (hvl_ccb.comm.opc.OpcUaSubHandler method), 14

datachange_notification() (hvl_ccb.dev.supercube.base.SupercubeSubscriptionHandler method), 24

datachange_notification() (hvl_ccb.dev.supercube2015.base.SupercubeSubscriptionHandler method), 44

DC_DoubleStage_280kV (hvl_ccb.dev.supercube.constants.PowerSetup attribute), 34

DC_DoubleStage_280kV (hvl_ccb.dev.supercube2015.constants.PowerSetup attribute), 49

DC_SingleStage_140kV (hvl_ccb.dev.supercube.constants.PowerSetup attribute), 34

DC_SingleStage_140kV (hvl_ccb.dev.supercube2015.constants.PowerSetup attribute), 49

DC_VOLTAGE_TOO_LOW (hvl_ccb.dev.newport.NewportSMC100PP.MotorErrors attribute), 83

default_com_cls() (hvl_ccb.dev.base.SingleCommDevice static method), 54

default_com_cls() (hvl_ccb.dev.crylas.CryLasAttenuator static method), 55

default_com_cls() (hvl_ccb.dev.crylas.CryLasLaser static method), 59

default_com_cls() (hvl_ccb.dev.ea_psi9000.PSI9000 static method), 65

default_com_cls() (hvl_ccb.dev.heinzinger.HeinzingerDI static method), 70

default_com_cls() (hvl_ccb.dev.labjack.LabJack static method), 76
 default_com_cls() (hvl_ccb.dev.mbw973.MBW973 static method), 79
 default_com_cls() (hvl_ccb.dev.newport.NewportSMC100PP static method), 84
 default_com_cls() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG static method), 97
 default_com_cls() (hvl_ccb.dev.rs_rto1024.RTO1024 static method), 102
 default_com_cls() (hvl_ccb.dev.se_ils2t.ILS2T static method), 109
 default_com_cls() (hvl_ccb.dev.supercube.base.SupercubeBase static method), 20
 default_com_cls() (hvl_ccb.dev.supercube.typ_a.SupercubeWithFU static method), 37
 default_com_cls() (hvl_ccb.dev.supercube.typ_b.SupercubeB static method), 38
 default_com_cls() (hvl_ccb.dev.supercube2015.base.Supercube2015Base static method), 39
 default_com_cls() (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU static method), 50
 default_com_cls() (hvl_ccb.dev.visa.VisaDevice static method), 113
 DEFAULT_IO_SCANNING_CONTROL_VALUES (hvl_ccb.dev.se_ils2t.ILS2T attribute), 107
 default_number_of_recordings (hvl_ccb.dev.heinzinger.HeinzingerConfig attribute), 69
 Device (class in hvl_ccb.dev.base), 53
 device_type (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig attribute), 9
 DeviceExistingException, 53
 DeviceSequenceMixin (class in hvl_ccb.dev.base), 53
 DIOChannel (hvl_ccb.dev.labjack.LabJack attribute), 75
 DISABLE (hvl_ccb.dev.newport.NewportStates attribute), 95
 disable() (hvl_ccb.dev.se_ils2t.ILS2T method), 109
 DISABLE_FROM_JOGGING (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 83
 DISABLE_FROM_MOVING (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 83
 DISABLE_FROM_READY (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 83
 DisableEspStageCheck (hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageConfig attribute), 90
 DISPLACEMENT_OUT_OF_LIMIT (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92
 do_ioscanning_write() (hvl_ccb.dev.se_ils2t.ILS2T method), 109
 Door (class in hvl_ccb.dev.supercube.constants), 30
 DoorStatus (class in hvl_ccb.dev.supercube.constants), 30
 DoorStatus (class in hvl_ccb.dev.supercube2015.constants), 45
E
 E (hvl_ccb.dev.labjack.LabJack.ThermocoupleType attribute), 75
 EarthingStick (class in hvl_ccb.dev.supercube.constants), 31
 EarthingStick (class in hvl_ccb.dev.supercube2015.constants), 45
 EarthingStickStatus (class in hvl_ccb.dev.supercube.constants), 31
 EarthingStickStatus (class in hvl_ccb.dev.supercube2015.constants), 46
 ERROR_ACCESS_ERROR (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication attribute), 92
 EIGHT (hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum attribute), 69
 EIGHTBITS (hvl_ccb.comm.serial.SerialCommunicationBytesize attribute), 15
 EmptyConfig (class in hvl_ccb.dev.base), 54
 enable() (hvl_ccb.dev.se_ils2t.ILS2T method), 109
 EnableEspStageCheck (hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageConfig attribute), 90
 ENCODING (hvl_ccb.comm.serial.SerialCommunication attribute), 14
 EndOfRunSwitch (hvl_ccb.dev.newport.NewportSMC100PPConfig.Home attribute), 90
 EndOfRunSwitch_and_Index (hvl_ccb.dev.newport.NewportSMC100PPConfig.HomeSearch attribute), 90
 endpoint_name (hvl_ccb.comm.opc.OpcUaCommunicationConfig attribute), 13
 endpoint_name (hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaConfig attribute), 36
 endpoint_name (hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaConfig attribute), 38

endpoint_name (hvl_ccb.dev.supercube2015.typ_a.SuperCube2015ConfigExperimentManager.ExperimentStatus attribute), 52
 error (hvl_ccb.dev.supercube.constants.DoorStatus attribute), 31
 error (hvl_ccb.dev.supercube.constants.EarthingStickStatus attribute), 31
 Error (hvl_ccb.dev.supercube.constants.SafetyStatus attribute), 35
 error (hvl_ccb.dev.supercube2015.constants.DoorStatus attribute), 45
 error (hvl_ccb.dev.supercube2015.constants.EarthingStickStatus attribute), 46
 Error (hvl_ccb.dev.supercube2015.constants.SafetyStatus attribute), 50
 ERROR (hvl_ccb.experiment_manager.ExperimentStatus attribute), 117
 Errors (class in hvl_ccb.dev.supercube.constants), 32
 Errors (class in hvl_ccb.dev.supercube2015.constants), 47
 ESP_STAGE_NAME_INVALID (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfigOverErrorComm.serial.SerialCommunicationConfig attribute), 92
 ETHERNET (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType attribute), 9
 EVEN (hvl_ccb.comm.serial.SerialCommunicationParity attribute), 16
 event_notification() (hvl_ccb.comm.opc.OpcUaSubHandler method), 14
 exit_configuration() (hvl_ccb.dev.newport.NewportSMC100PPConfig method), 84
 exit_configuration_wait_sec (hvl_ccb.dev.newport.NewportSMC100PPConfig attribute), 91
 ExperimentError, 116
 ExperimentManager (class in hvl_ccb.experiment_manager), 116
 ExperimentStatus (class in hvl_ccb.experiment_manager), 117
 External (hvl_ccb.dev.supercube.constants.PowerSetup attribute), 34
 External (hvl_ccb.dev.supercube2015.constants.PowerSetup attribute), 49
F
 F (hvl_ccb.dev.labjack.LabJack.TemperatureUnit attribute), 75
 FAST (hvl_ccb.dev.se_ils2t.ILS2T.Ref16Jog attribute), 108
 file_copy() (hvl_ccb.dev.rs_rto1024.RTO1024 method), 102
 finish() (hvl_ccb.experiment_manager.ExperimentManager method), 117
 FINISHING (hvl_ccb.experiment_manager.ExperimentStatus attribute), 117
 FIVEBITS (hvl_ccb.comm.serial.SerialCommunicationBytesize attribute), 15
 FLT_INFO (hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute), 108
 FLT_MEM_DEL (hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute), 108
 FLT_MEM_RESET (hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute), 108
 FOLLOWING_ERROR (hvl_ccb.dev.newport.NewportSMC100PP.MotorError attribute), 83
 force_value() (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig method), 9
 force_value() (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig method), 11
 force_value() (hvl_ccb.comm.opc.OpcUaCommunicationConfig method), 13
 force_value() (hvl_ccb.comm.serial.SerialCommunicationConfig method), 16
 force_value() (hvl_ccb.comm.visa.VisaCommunicationConfig method), 18
 force_value() (hvl_ccb.dev.base.EmptyConfig method), 54
 force_value() (hvl_ccb.dev.crylas.CryLasAttenuatorConfig method), 55
 force_value() (hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig method), 57
 force_value() (hvl_ccb.dev.crylas.CryLasLaserConfig method), 61
 force_value() (hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig method), 63
 force_value() (hvl_ccb.dev.ea_psi9000.PSI9000Config method), 66
 force_value() (hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig method), 68
 force_value() (hvl_ccb.dev.heinzinger.HeinzingerConfig method), 69
 force_value() (hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig method), 73
 force_value() (hvl_ccb.dev.mbw973.MBW973Config method), 80
 force_value() (hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig method), 81
 force_value() (hvl_ccb.dev.newport.NewportSMC100PPConfig method), 91
 force_value() (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig method), 94
 force_value() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig method), 99
 force_value() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig method), 101

`force_value()` (`hvl_ccb.dev.rs_rto1024.RTO1024Config` get (`hvl_ccb.dev.supercube.constants.Door` attribute),
 method), 105 30
`force_value()` (`hvl_ccb.dev.rs_rto1024.RTO1024VisaConfig` get (`hvl_ccb.dev.supercube.constants.MeasurementsDividerRatio`
 method), 107 attribute), 33
`force_value()` (`hvl_ccb.dev.se_ils2t.ILS2TConfig` get (`hvl_ccb.dev.supercube.constants.MeasurementsScaledInput`
 method), 111 attribute), 33
`force_value()` (`hvl_ccb.dev.se_ils2t.ILS2TModbusTcpConfig` get (`hvl_ccb.dev.supercube2015.constants.AlarmText` at-
 method), 112 tribute), 45
`force_value()` (`hvl_ccb.dev.supercube.base.SupercubeConfig` get (`hvl_ccb.dev.supercube2015.constants.MeasurementsDividerRatio`
 method), 23 attribute), 48
`force_value()` (`hvl_ccb.dev.supercube.base.SupercubeConfig` get (`hvl_ccb.dev.supercube2015.constants.MeasurementsScaledInput`
 method), 24 attribute), 48
`force_value()` (`hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaConfiguration` get (`hvl_ccb.dev.newport.NewportSMC100PP`
 method), 36 attribute), 84
`force_value()` (`hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaConfiguration` get (`hvl_ccb.dev.newport.NewportSMC100PP`
 method), 38 attribute), 84
`force_value()` (`hvl_ccb.dev.supercube2015.base.SupercubeConfiguration` get (`hvl_ccb.dev.labjack.LabJack method`), 76
 method), 42 attribute), 9
`force_value()` (`hvl_ccb.dev.supercube2015.base.SupercubeOpcUaConfiguration` get (`hvl_ccb.dev.labjack.LabJack.LabJack.DeviceType`
 method), 43 attribute), 75
`force_value()` (`hvl_ccb.dev.supercube2015.typ_a.SupercubeAOpcUaConfiguration` get (`hvl_ccb.dev.labjack.LabJack method`), 76
 method), 52 attribute), 76
`force_value()` (`hvl_ccb.dev.visa.VisaDeviceConfig` get (`hvl_ccb.dev.supercube.base.SupercubeBase`
 method), 114 attribute), 20
`FOUR` (`hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum` method), 20
 attribute), 69
`FREERUN` (`hvl_ccb.dev.rs_rto1024.RTO1024.TriggerModes` method), 39
 attribute), 101
`frequency` (`hvl_ccb.dev.supercube.constants.Power` at- get (`hvl_ccb.dev.newport.NewportSMC100PP`
 tribute), 34 attribute), 85
`frequency` (`hvl_ccb.dev.supercube2015.constants.Power` method), 85
 attribute), 48
`FRM` (`hvl_ccb.dev.newport.NewportConfigCommands` at- get (`hvl_ccb.dev.heinzinger.HeinzingerDI`
 tribute), 82 method), 70
`from_json()` (`hvl_ccb.configuration.ConfigurationMixin` get (`hvl_ccb.dev.se_ils2t.ILS2T` method),
 class method), 116 109
`FRS` (`hvl_ccb.dev.newport.NewportConfigCommands` at- get (`hvl_ccb.dev.base.DeviceSequenceMixin`
 tribute), 82 method), 53
`fso_reset()` (`hvl_ccb.dev.supercube.typ_a.SupercubeWithFU` method), 53
 method), 37
`fso_reset()` (`hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU` get (`hvl_ccb.dev.labjack.LabJack method`), 76
 method), 50 attribute), 76
 method), 20
 method), 20
 method), 39
 method), 20
 method), 40
 method), 40

G

`GeneralSockets` (class in get (`hvl_ccb.dev.supercube2015.base.Supercube2015Base`
`hvl_ccb.dev.supercube.constants`), 32 method), 39
`GeneralSockets` (class in get (`hvl_ccb.dev.supercube2015.base.Supercube2015Base`
`hvl_ccb.dev.supercube2015.constants`), 47 method), 39
`GeneralSupport` (class in get (`hvl_ccb.dev.supercube.base.SupercubeBase`
`hvl_ccb.dev.supercube.constants`), 32 method), 20
`GeneralSupport` (class in get (`hvl_ccb.dev.supercube2015.base.Supercube2015Base`
`hvl_ccb.dev.supercube2015.constants`), 47 method), 40
`get` (`hvl_ccb.dev.supercube.constants.AlarmText` at- get (`hvl_ccb.dev.supercube2015.base.Supercube2015Base`
 tribute), 26 attribute), 40
 method), 40

```

        (hvl_ccb.dev.supercube.base.SupercubeBase
        method), 20
get_earthing_status()
    (hvl_ccb.dev.supercube2015.base.Supercube2015Base
    method), 40
get_error_code()
    (hvl_ccb.dev.se_ils2t.ILS2T
    method), 109
get_error_queue()
    (hvl_ccb.dev.visa.VisaDevice
    method), 113
get_frequency()
    (hvl_ccb.dev.supercube.typ_a.SupercubeWithFU
    method), 37
get_frequency()
    (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU
    method), 50
get_fso_active()
    (hvl_ccb.dev.supercube.typ_a.SupercubeWithFU
    method), 37
get_fso_active()
    (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU
    method), 50
get_full_scale_mbar()
    (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method),
    97
get_full_scale_unitless()
    (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method),
    97
get_identification()
    (hvl_ccb.dev.visa.VisaDevice method), 113
get_interface_version()
    (hvl_ccb.dev.heinzinger.HeinzingerDI method),
    70
get_max_voltage()
    (hvl_ccb.dev.supercube.typ_a.SupercubeWithFU
    method), 37
get_max_voltage()
    (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU
    method), 51
get_measurement_ratio()
    (hvl_ccb.dev.supercube.base.SupercubeBase
    method), 20
get_measurement_ratio()
    (hvl_ccb.dev.supercube2015.base.Supercube2015Base
    method), 40
get_measurement_voltage()
    (hvl_ccb.dev.supercube.base.SupercubeBase
    method), 21
get_measurement_voltage()
    (hvl_ccb.dev.supercube2015.base.Supercube2015Base
    method), 40
get_motor_configuration()
    (hvl_ccb.dev.newport.NewportSMC100PP
    method), 85
get_move_duration()
    (hvl_ccb.dev.newport.NewportSMC100PP
    method), 85
get_negative_software_limit()
    (hvl_ccb.dev.newport.NewportSMC100PP
    method), 85
get_number_of_recordings()
    (hvl_ccb.dev.heinzinger.HeinzingerDI method),
    70
get_output()
    (hvl_ccb.dev.ea_psi9000.PSI9000
    method), 65
get_position()
    (hvl_ccb.dev.newport.NewportSMC100PP
    method), 85
get_position()
    (hvl_ccb.dev.se_ils2t.ILS2T
    method), 110
get_positive_software_limit()
    (hvl_ccb.dev.newport.NewportSMC100PP
    method), 86
get_power_setup()
    (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU
    method), 51
get_primary_current()
    (hvl_ccb.dev.supercube.typ_a.SupercubeWithFU
    method), 37
get_primary_current()
    (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU
    method), 51
get_primary_voltage()
    (hvl_ccb.dev.supercube.typ_a.SupercubeWithFU
    method), 37
get_primary_voltage()
    (hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU
    method), 51
get_product_id()
    (hvl_ccb.dev.labjack.LabJack
    method), 76
get_product_name()
    (hvl_ccb.dev.labjack.LabJack
    method), 76
get_product_type()
    (hvl_ccb.dev.labjack.LabJack
    method), 76
get_pulse_energy_and_rate()
    (hvl_ccb.dev.crylas.CryLasLaser method),
    59
get_sbus_rh()
    (hvl_ccb.dev.labjack.LabJack
    method), 77
get_sbus_temp()
    (hvl_ccb.dev.labjack.LabJack
    method), 77
get_serial_number()
    (hvl_ccb.dev.heinzinger.HeinzingerDI method),
    70
get_serial_number()
    (hvl_ccb.dev.labjack.LabJack method), 77
get_state()
    (hvl_ccb.dev.newport.NewportSMC100PP
    method), 86
get_status()
    (hvl_ccb.dev.se_ils2t.ILS2T method),
    110
get_status()
    (hvl_ccb.dev.supercube.base.SupercubeBase
    method), 85

```

<code>method</code>), 21	GreenReady (<i>hvl_ccb.dev.supercube2015.constants.SafetyStatus</i> attribute), 35
<code>get_status()</code> (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base</i> attribute), 50	
<code>method</code>), 40	
<code>get_support_input()</code>	H
(<i>hvl_ccb.dev.supercube.base.SupercubeBase</i>	HARDWARE (<i>hvl_ccb.dev.crylas.CryLasLaser.RepetitionRates</i>
<i>method</i>), 21	attribute), 59
<code>get_support_input()</code>	HEAD (<i>hvl_ccb.dev.crylas.CryLasLaser.AnswersStatus</i> at-
(<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base</i>	tribute), 58
<i>method</i>), 40	HeinzingerConfig (class in <i>hvl_ccb.dev.heinzinger</i>),
<code>get_support_output()</code>	69
(<i>hvl_ccb.dev.supercube.base.SupercubeBase</i>	HeinzingerConfig.RecordingsEnum (class in
<i>method</i>), 21	<i>hvl_ccb.dev.heinzinger</i>), 69
<code>get_support_output()</code>	HeinzingerDI (class in <i>hvl_ccb.dev.heinzinger</i>), 70
(<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base</i>	HeinzingerPNC (class in <i>hvl_ccb.dev.heinzinger</i>), 71
<i>method</i>), 40	HeinzingerPNC.UnitCurrent (class in
<code>get_system_lock()</code>	<i>hvl_ccb.dev.heinzinger</i>), 71
(<i>hvl_ccb.dev.ea_psi9000.PSI9000</i> <i>method</i>),	HeinzingerPNC.UnitVoltage (class in
65	<i>hvl_ccb.dev.heinzinger</i>), 71
<code>get_t13_socket()</code> (<i>hvl_ccb.dev.supercube.base.SupercubeBase</i>	HeinzingerPNCDeviceNotRecognizedException,
<i>method</i>), 21	72
<code>get_t13_socket()</code> (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base</i>	HeinzingerPNCError, 72
<i>method</i>), 40	HeinzingerPNCMaxCurrentExceededException,
<code>get_target_voltage()</code>	72
(<i>hvl_ccb.dev.supercube.typ_a.SupercubeWithFU</i>	HeinzingerPNCMaxVoltageExceededException,
<i>method</i>), 37	72
<code>get_target_voltage()</code>	HeinzingerSerialCommunication (class in
(<i>hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU</i>	<i>hvl_ccb.dev.heinzinger</i>), 72
<i>method</i>), 51	HeinzingerSerialCommunicationConfig
<code>get_temperature()</code> (<i>hvl_ccb.dev.se_ils2t.ILS2T</i>	(class in <i>hvl_ccb.dev.heinzinger</i>), 73
<i>method</i>), 110	HIGH (<i>hvl_ccb.dev.labjack.LabJack.DIOSStatus</i> attribute),
<code>get_timestamps()</code> (<i>hvl_ccb.dev.rs_rto1024.RTO1024</i>	75
<i>method</i>), 102	home_search_polling_interval
<code>get_ui_lower_limits()</code>	(<i>hvl_ccb.dev.newport.NewportSMC100PPConfig</i>
(<i>hvl_ccb.dev.ea_psi9000.PSI9000</i> <i>method</i>),	attribute), 91
65	home_search_timeout
<code>get_uip_upper_limits()</code>	(<i>hvl_ccb.dev.newport.NewportSMC100PPConfig</i>
(<i>hvl_ccb.dev.ea_psi9000.PSI9000</i> <i>method</i>),	attribute), 91
65	home_search_type (<i>hvl_ccb.dev.newport.NewportSMC100PPConfig</i>
<code>get_voltage()</code> (<i>hvl_ccb.dev.heinzinger.HeinzingerDI</i>	attribute), 91
<i>method</i>), 70	home_search_velocity
<code>get_voltage_current_setpoint()</code>	(<i>hvl_ccb.dev.newport.NewportSMC100PPConfig</i>
(<i>hvl_ccb.dev.ea_psi9000.PSI9000</i> <i>method</i>),	attribute), 91
65	HOME_STARTED (<i>hvl_ccb.dev.newport.NewportSMC100PPSerialCommun</i>
<code>go_home()</code> (<i>hvl_ccb.dev.newport.NewportSMC100PP</i>	attribute), 92
<i>method</i>), 86	HomeSwitch (<i>hvl_ccb.dev.newport.NewportSMC100PPConfig.HomeSear</i>
<code>go_to_configuration()</code>	attribute), 90
(<i>hvl_ccb.dev.newport.NewportSMC100PP</i>	HomeSwitch_and_Index
<i>method</i>), 86	(<i>hvl_ccb.dev.newport.NewportSMC100PPConfig.HomeSearch</i>
GreenNotReady (<i>hvl_ccb.dev.supercube.constants.SafetyStatus</i>	attribute), 91
attribute), 35	HOMING (<i>hvl_ccb.dev.newport.NewportStates</i> attribute),
GreenNotReady (<i>hvl_ccb.dev.supercube2015.constants.SafetyStatus</i>	95
attribute), 50	HOMING_FROM_RS232
GreenReady (<i>hvl_ccb.dev.supercube.constants.SafetyStatus</i>	(<i>hvl_ccb.dev.newport.NewportSMC100PP.StateMessages</i>
attribute), 35	attribute), 84

HOMING_FROM_SMC (*hvl_ccb.dev.newport.NewportSMC100PP.StateMessages*, 91
 attribute), 84
 HOMING_TIMEOUT (*hvl_ccb.dev.newport.NewportSMC100PP.MotorErrors*
 attribute), 83
 host (*hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig* (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus*
 attribute), 11
 Identification_error
 host (*hvl_ccb.comm.opc.OpcUaCommunicationConfig* (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig*
 attribute), 13
 identifier (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig*
 attribute), 9
 host (*hvl_ccb.comm.visa.VisaCommunicationConfig* *attribute*), 19
 identify_device()
 (*hvl_ccb.dev.heinzinger.HeinzingerPNC*
 method), 71
 hPascal (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits*
 attribute), 96
 identify_sensors()
 (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG* *method*),
 97
 HT (*hvl_ccb.dev.newport.NewportConfigCommands* *attribute*), 82
 hvl_ccb (*module*), 117
 hvl_ccb.comm (*module*), 19
 hvl_ccb.comm.base (*module*), 7
 hvl_ccb.comm.labjack_ljm (*module*), 7
 hvl_ccb.comm.modbus_tcp (*module*), 10
 hvl_ccb.comm.opc (*module*), 11
 hvl_ccb.comm.serial (*module*), 14
 hvl_ccb.comm.visa (*module*), 17
 hvl_ccb.configuration (*module*), 116
 hvl_ccb.dev (*module*), 114
 hvl_ccb.dev.base (*module*), 53
 hvl_ccb.dev.crylas (*module*), 55
 hvl_ccb.dev.ea_psi9000 (*module*), 64
 hvl_ccb.dev.heinzinger (*module*), 68
 hvl_ccb.dev.labjack (*module*), 74
 hvl_ccb.dev.mbw973 (*module*), 79
 hvl_ccb.dev.newport (*module*), 82
 hvl_ccb.dev.pfeiffer_tpg (*module*), 96
 hvl_ccb.dev.rs_rto1024 (*module*), 101
 hvl_ccb.dev.se_ils2t (*module*), 107
 hvl_ccb.dev.supercube (*module*), 39
 hvl_ccb.dev.supercube.base (*module*), 20
 hvl_ccb.dev.supercube.constants (*module*),
 25
 hvl_ccb.dev.supercube.typ_a (*module*), 36
 hvl_ccb.dev.supercube.typ_b (*module*), 38
 hvl_ccb.dev.supercube2015 (*module*), 52
 hvl_ccb.dev.supercube2015.base (*module*),
 39
 hvl_ccb.dev.supercube2015.constants
 (*module*), 44
 hvl_ccb.dev.supercube2015.typ_a (*module*),
 50
 hvl_ccb.dev.visa (*module*), 113
 hvl_ccb.experiment_manager (*module*), 116
 hvl_ccb.utils (*module*), 115
 hvl_ccb.utils.enum (*module*), 115
 hvl_ccb.utils.typing (*module*), 115
 hysteresis_compensation
 (*hvl_ccb.dev.newport.NewportSMC100PPConfig*

[in_3_1 \(hvl_ccb.dev.supercube.constants.GeneralSupport Initializing \(hvl_ccb.dev.supercube.constants.SafetyStatus attribute\), 32](#)
[in_3_1 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport Initializing \(hvl_ccb.dev.supercube2015.constants.SafetyStatus attribute\), 47](#)
[in_3_2 \(hvl_ccb.dev.supercube.constants.GeneralSupport INITIALIZING \(hvl_ccb.experiment_manager.ExperimentStatus attribute\), 32](#)
[in_3_2 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input \(hvl_ccb.dev.supercube.constants.GeneralSupport attribute\), 47](#)
[in_4_1 \(hvl_ccb.dev.supercube.constants.GeneralSupport input \(hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute\), 32](#)
[in_4_1 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input_1 \(hvl_ccb.dev.supercube.constants.MeasurementsDividerRatio attribute\), 47](#)
[in_4_2 \(hvl_ccb.dev.supercube.constants.GeneralSupport input_1 \(hvl_ccb.dev.supercube.constants.MeasurementsScaledInput attribute\), 32](#)
[in_4_2 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input_1 \(hvl_ccb.dev.supercube2015.constants.MeasurementsDividerRatio attribute\), 47](#)
[in_5_1 \(hvl_ccb.dev.supercube.constants.GeneralSupport input_1 \(hvl_ccb.dev.supercube2015.constants.MeasurementsScaledInput attribute\), 32](#)
[in_5_1 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input_2 \(hvl_ccb.dev.supercube.constants.MeasurementsDividerRatio attribute\), 47](#)
[in_5_2 \(hvl_ccb.dev.supercube.constants.GeneralSupport input_2 \(hvl_ccb.dev.supercube.constants.MeasurementsScaledInput attribute\), 32](#)
[in_5_2 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input_2 \(hvl_ccb.dev.supercube2015.constants.MeasurementsScaledInput attribute\), 47](#)
[in_6_1 \(hvl_ccb.dev.supercube.constants.GeneralSupport input_3 \(hvl_ccb.dev.supercube.constants.MeasurementsDividerRatio attribute\), 32](#)
[in_6_1 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input_3 \(hvl_ccb.dev.supercube.constants.MeasurementsScaledInput attribute\), 47](#)
[in_6_2 \(hvl_ccb.dev.supercube.constants.GeneralSupport input_3 \(hvl_ccb.dev.supercube2015.constants.MeasurementsScaledInput attribute\), 32](#)
[in_6_2 \(hvl_ccb.dev.supercube2015.constants.GeneralSupport input_4 \(hvl_ccb.dev.supercube.constants.MeasurementsDividerRatio attribute\), 47](#)
[INACTIVE \(hvl_ccb.dev.crylas.CryLasLaser.AnswersStatus input_4 \(hvl_ccb.dev.supercube.constants.MeasurementsScaledInput attribute\), 58](#)
[inactive \(hvl_ccb.dev.supercube.constants.DoorStatus input_4 \(hvl_ccb.dev.supercube2015.constants.MeasurementsScaledInput attribute\), 31](#)
[inactive \(hvl_ccb.dev.supercube.constants.EarthingStickStatus \(hvl_ccb.dev.se_ils2t.ILS2T.RegDatatype attribute\), 31](#)
[inactive \(hvl_ccb.dev.supercube2015.constants.DoorStatus interface_type \(hvl_ccb.comm.visa.VisaCommunicationConfig attribute\), 45](#)
[inactive \(hvl_ccb.dev.supercube2015.constants.EarthingStickStatus interface_type \(hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunication attribute\), 46](#)
[init_attenuation \(hvl_ccb.dev.crylas.CryLasAttenuatorConfig interface_type \(hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunication attribute\), 56](#)
[init_monitored_nodes \(\) internal \(hvl_ccb.dev.labjack.LabJack.CjcType \(hvl_ccb.comm.opc.OpcUaCommunication attribute\), 12](#)
[init_shutter_status Internal \(hvl_ccb.dev.supercube.constants.PowerSetup attribute\), 35](#)
[initialize \(\) \(hvl_ccb.dev.crylas.CryLasLaserConfig attribute\), 62](#)
[initialize \(\) \(hvl_ccb.dev.newport.NewportSMC100PP InvalidSupercubeStatusError, 39](#)
[INITIALIZED \(hvl_ccb.experiment_manager.ExperimentStatus IO_SCANNING \(hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute\), 117](#)
[IoScanningModeValueError, 113](#)

```

is_configdataclass (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig attribute), 9
is_configdataclass (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig attribute), 11
is_configdataclass (hvl_ccb.comm.opc.OpcUaCommunicationConfig attribute), 13
is_configdataclass (hvl_ccb.comm.serial.SerialCommunicationConfig attribute), 16
is_configdataclass (hvl_ccb.comm.visa.VisaCommunicationConfig attribute), 19
is_configdataclass (hvl_ccb.dev.base.EmptyConfig attribute), 54
is_configdataclass (hvl_ccb.dev.crylas.CryLasAttenuatorConfig attribute), 56
is_configdataclass (hvl_ccb.dev.crylas.CryLasLaserConfig attribute), 62
is_configdataclass (hvl_ccb.dev.heinzinger.HeinzingerConfig attribute), 69
is_configdataclass (hvl_ccb.dev.mbw973.MBW973Config attribute), 80
is_configdataclass (hvl_ccb.dev.newport.NewportSMC100PPConfig attribute), 91
is_configdataclass (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig attribute), 99
is_configdataclass (hvl_ccb.dev.se_ils2t.ILS2TConfig attribute), 111
is_configdataclass (hvl_ccb.dev.supercube.base.SupercubeConfiguration attribute), 23
is_configdataclass (hvl_ccb.dev.supercube2015.base.SupercubeConfiguration attribute), 42
is_done() (hvl_ccb.dev.mbw973.MBW973 method), 79
is_error() (hvl_ccb.experiment_manager.ExperimentManager method), 117
is_finished() (hvl_ccb.experiment_manager.ExperimentManager method), 117
is_generic() (in module hvl_ccb.utils.typing), 115
is_in_range() (hvl_ccb.dev.se_ils2t.ILS2T.RegDatatype method), 109
is_inactive(hvl_ccb.dev.crylas.CryLasLaser.LaserStatus attribute), 59
is_open(hvl_ccb.comm.labjack_ljm.LJMCommunication attribute), 8
is_open(hvl_ccb.comm.opc.OpcUaCommunication attribute), 12
is_open(hvl_ccb.comm.serial.SerialCommunication attribute), 14
is_ready(hvl_ccb.dev.crylas.CryLasLaser.LaserStatus attribute), 59
is_running() (hvl_ccb.experiment_manager.ExperimentManager method), 117
is_type_hint() (in module hvl_ccb.utils.typing), 115
is_valid_scale_range_reversed_str() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model method), 98

J
J (hvl_ccb.dev.labjack.LabJack.ThermocoupleType attribute), 75
jerk_time(hvl_ccb.dev.newport.NewportSMC100PPConfig attribute), 91
JOG(hvl_ccb.dev.se_ils2t.ILS2T.Mode attribute), 107
jog_run() (hvl_ccb.dev.se_ils2t.ILS2T method), 110
jog_stop() (hvl_ccb.dev.se_ils2t.ILS2T method), 110
JOGGING(hvl_ccb.dev.newport.NewportStates attribute), 95
JOGGING_FROM_DISABLE(hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84
JOGGING_FROM_READY(hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84
JOGN_FAST(hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute), 108
JOGN_SLOW(hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute), 108
JR(hvl_ccb.dev.newport.NewportConfigCommands attribute), 82

K
K(hvl_ccb.dev.labjack.LabJack.TemperatureUnit attribute), 75
K(hvl_ccb.dev.labjack.LabJack.ThermocoupleType attribute), 75
keys() (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig class method), 9
keys() (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig class method), 11
keys() (hvl_ccb.comm.opc.OpcUaCommunicationConfig class method), 13
keys() (hvl_ccb.comm.serial.SerialCommunicationConfig class method), 16

```

keys () (*hvl_ccb.comm.visa.VisaCommunicationConfig* class method), 19
 keys () (*hvl_ccb.dev.base.EmptyConfig* class method), 54
 keys () (*hvl_ccb.dev.crylas.CryLasAttenuatorConfig* class method), 56
 keys () (*hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig* class method), 57
 keys () (*hvl_ccb.dev.crylas.CryLasLaserConfig* class method), 62
 keys () (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig* class method), 63
 keys () (*hvl_ccb.dev.ea_psi9000.PSI9000Config* class method), 67
 keys () (*hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig* class method), 68
 keys () (*hvl_ccb.dev.heinzinger.HeinzingerConfig* class method), 69
 keys () (*hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig* class method), 74
 keys () (*hvl_ccb.dev.mbw973.MBW973Config* class method), 80
 keys () (*hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig* class method), 81
 keys () (*hvl_ccb.dev.newport.NewportSMC100PPConfig* class method), 91
 keys () (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig* class method), 95
 keys () (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig* class method), 99
 keys () (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig* class method), 101
 keys () (*hvl_ccb.dev.rs_rto1024.RTO1024Config* class method), 106
 keys () (*hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunicationConfig* class method), 107
 keys () (*hvl_ccb.dev.se_ils2t.ILS2TConfig* class method), 111
 keys () (*hvl_ccb.dev.se_ils2t.ILS2TModbusTcpCommunicationConfig* class method), 112
 keys () (*hvl_ccb.dev.supercube.base.SupercubeConfiguration* class method), 23
 keys () (*hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunicationConfig* class method), 24
 keys () (*hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaConfiguration* class method), 36
 keys () (*hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaConfiguration* class method), 39
 keys () (*hvl_ccb.dev.supercube2015.base.SupercubeConfiguration* class method), 42
 keys () (*hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunicationConfig* class method), 43
 keys () (*hvl_ccb.dev.supercube2015.typ_a.SupercubeAOpcUaConfiguration* class method), 52
 keys () (*hvl_ccb.dev.visa.VisaDeviceConfig* class method), 114
 kV (*hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitVoltage* attribute), 71
L
 LabJack.CalMicroAmpere (class in *hvl_ccb.dev.labjack*), 74
 LabJack.CjcType (class in *hvl_ccb.dev.labjack*), 75
 LabJack.DeviceType (class in *hvl_ccb.dev.labjack*), 75
 LabJack.DIOStatus (class in *hvl_ccb.dev.labjack*), 75
 LabJack.TemperatureUnit (class in *hvl_ccb.dev.labjack*), 75
 LabJack.ThermocoupleType (class in *hvl_ccb.dev.labjack*), 75
 LabJackError, 78
 LabJackIdentifierDIOError, 78
 laser_off () (*hvl_ccb.dev.crylas.CryLasLaser* method), 59
 laser_on () (*hvl_ccb.dev.crylas.CryLasLaser* method), 59
 list_directory () (*hvl_ccb.dev.rs_rto1024.RTO1024* method), 102
 LJMCommunication (class in *hvl_ccb.comm.labjack_ljm*), 8
 LJMCommunicationConfig (class in *hvl_ccb.comm.labjack_ljm*), 8
 LJMCommunicationConfig.ConnectionType (class in *hvl_ccb.comm.labjack_ljm*), 9
 LJMCommunicationConfig.DeviceType (class in *hvl_ccb.comm.labjack_ljm*), 9
 LJMCommunicationError, 10
 lm34 (*hvl_ccb.dev.labjack.LabJack.CjcType* attribute), 75
 local_configuration () (*hvl_ccb.dev.rs_rto1024.RTO1024* method), 102
 local_display () (*hvl_ccb.dev.rs_rto1024.RTO1024* method), 102
 locked (*hvl_ccb.dev.supercube.constants.DoorStatus* attribute), 31
 locked (*hvl_ccb.dev.supercube2015.constants.DoorStatus* attribute), 45
 LOW (*hvl_ccb.dev.labjack.LabJack.DIOStatus* attribute), 75
M
 mA (*hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitCurrent* attribute), 71

manual (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 45
 manual_1 (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual_1 (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 45
 manual_2 (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual_2 (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 46
 manual_3 (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual_3 (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 46
 manual_4 (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual_4 (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 46
 manual_5 (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual_5 (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 46
 manual_6 (*hvl_ccb.dev.supercube.constants.EarthingStick* attribute), 31
 manual_6 (*hvl_ccb.dev.supercube2015.constants.EarthingStick* attribute), 46
 MARK (*hvl_ccb.comm.serial.SerialCommunicationParity* attribute), 16
 max_current (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 72
 max_current_hardware (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 72
 max_voltage (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 72
 max_voltage_hardware (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 72
 mbar (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits* attribute), 96
 MBW973 (class in *hvl_ccb.dev.mbw973*), 79
 MBW973Config (class in *hvl_ccb.dev.mbw973*), 80
 MBW973ControlRunningException, 80
 MBW973Error, 80
 MBW973PumpRunningException, 81
 MBW973SerialCommunication (class in *hvl_ccb.dev.mbw973*), 81
 MBW973SerialCommunicationConfig (class in *hvl_ccb.dev.mbw973*), 81
 measure () (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG* method), 97
 measure_all () (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG* method), 97
 measure_current () (*hvl_ccb.dev.heinzinger.HeinzingerDI* method), 70
 measure_voltage () (*hvl_ccb.dev.heinzinger.HeinzingerDI* method), 70
 measure_voltage_current () (*hvl_ccb.dev.ea_psi9000.PSI9000* method), 65
 MeasurementsDividerRatio (class in *hvl_ccb.dev.supercube.constants*), 33
 MeasurementsDividerRatio (class in *hvl_ccb.dev.supercube2015.constants*), 48
 MeasurementsScaledInput (class in *hvl_ccb.dev.supercube.constants*), 33
 MeasurementsScaledInput (class in *hvl_ccb.dev.supercube2015.constants*), 48
 Message (*hvl_ccb.dev.supercube.constants.Errors* attribute), 32
 micro_step_per_full_step_factor (*hvl_ccb.dev.newport.NewportSMC100PPConfig* attribute), 91
 Micron (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits* attribute), 96
 ModbusTcpCommunication (class in *hvl_ccb.comm.modbus_tcp*), 10
 ModbusTcpCommunicationConfig (class in *hvl_ccb.comm.modbus_tcp*), 11
 ModbusTcpConnectionFailedException, 11
 model (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig* attribute), 99
 motion_distance_per_full_step (*hvl_ccb.dev.newport.NewportSMC100PPConfig* attribute), 91
 motor_config (*hvl_ccb.dev.newport.NewportSMC100PPConfig* attribute), 91
 move_finished_extra_wait_sec (*hvl_ccb.dev.newport.NewportSMC100PPConfig* attribute), 91
 move_to_absolute_position () (*hvl_ccb.dev.newport.NewportSMC100PP* method), 87
 move_to_relative_position () (*hvl_ccb.dev.newport.NewportSMC100PP* method), 87
 MOVING (*hvl_ccb.dev.newport.NewportSMC100PP.StateMessages* attribute), 84
 MOVING (*hvl_ccb.dev.newport.NewportStates* attribute), 95
 MS_NOMINAL_CURRENT (*hvl_ccb.dev.ea_psi9000.PSI9000* attribute), 64
 MS_NOMINAL_VOLTAGE (*hvl_ccb.dev.ea_psi9000.PSI9000* attribute), 64

MULTI_COMMANDS_MAX (hvl_ccb.comm.visa.VisaCommunication attribute), 17

MULTI_COMMANDS_SEPARATOR (hvl_ccb.comm.visa.VisaCommunication attribute), 17

N

NameEnum (class in hvl_ccb.utils.enum), 115

NAMES (hvl_ccb.comm.serial.SerialCommunicationParity attribute), 16

names (hvl_ccb.dev.rs_rto1024.RTO1024.TriggerModes attribute), 102

namespace_index (hvl_ccb.dev.supercube.base.SupercubeConfiguration attribute), 23

namespace_index (hvl_ccb.dev.supercube2015.base.SupercubeConfiguration attribute), 42

NED_END_OF_TURN (hvl_ccb.dev.newport.NewportSMC100PP.MotorErrors attribute), 83

NEG (hvl_ccb.dev.se_ils2t.ILS2T.Ref16Jog attribute), 108

NEG_FAST (hvl_ccb.dev.se_ils2t.ILS2T.Ref16Jog attribute), 108

negative_software_limit (hvl_ccb.dev.newport.NewportSMC100PPConfig attribute), 91

NewportConfigCommands (class in hvl_ccb.dev.newport), 82

NewportControllerError, 83

NewportMotorError, 83

NewportSerialCommunicationError, 95

NewportSMC100PP (class in hvl_ccb.dev.newport), 83

NewportSMC100PP.MotorErrors (class in hvl_ccb.dev.newport), 83

NewportSMC100PP.StateMessages (class in hvl_ccb.dev.newport), 83

NewportSMC100PPConfig (class in hvl_ccb.dev.newport), 89

NewportSMC100PPConfig.EspStageConfig (class in hvl_ccb.dev.newport), 90

NewportSMC100PPConfig.HomeSearch (class in hvl_ccb.dev.newport), 90

NewportSMC100PPSerialCommunication (class in hvl_ccb.dev.newport), 92

NewportSMC100PPSerialCommunication.ControllerError (class in hvl_ccb.dev.newport), 92

NewportSMC100PPSerialCommunicationConfig (class in hvl_ccb.dev.newport), 94

NewportStates (class in hvl_ccb.dev.newport), 95

NO_ERROR (hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication.CommandError attribute), 92

NO_REF (hvl_ccb.dev.newport.NewportStates attribute), 95

NO_REF_ESP_STAGE_ERROR (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 17

attribute), 84

NO_REF_FROM_CONFIG (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

NO_REF_FROM_DISABLED (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

NO_REF_FROM_HOMING (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

NO_REF_FROM_JOGGING (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

NO_REF_FROM_MOVING (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

NO_REF_FROM_READY (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

NO_REF_FROM_RESET (hvl_ccb.dev.newport.NewportSMC100PP.StateMessages attribute), 84

No_sensor (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus attribute), 96

NONE (hvl_ccb.comm.serial.SerialCommunicationParity attribute), 17

NONE (hvl_ccb.dev.labjack.LabJack.ThermocoupleType attribute), 76

None (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes attribute), 96

NONE (hvl_ccb.dev.se_ils2t.ILS2T.Ref16Jog attribute), 108

NoPower (hvl_ccb.dev.supercube.constants.PowerSetup attribute), 35

NORMAL (hvl_ccb.dev.rs_rto1024.RTO1024.TriggerModes attribute), 101

noSen (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes attribute), 97

noSENSOR (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes attribute), 97

not_defined (hvl_ccb.dev.supercube.constants.AlarmText attribute), 26

not_defined (hvl_ccb.dev.supercube2015.constants.AlarmText attribute), 45

number_of_decimals (hvl_ccb.dev.heinzinger.HeinzingerConfig attribute), 69

number_of_sensors (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG attribute), 98

O

ODD (hvl_ccb.comm.serial.SerialCommunicationParity attribute), 17

OH (<i>hvl_ccb.dev.newport.NewportConfigCommands</i> attribute), 83	OPENED (<i>hvl_ccb.dev.crylas.CryLasLaserShutterStatus</i> attribute), 64
Ok (<i>hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus</i> attribute), 96	operate () (<i>hvl_ccb.dev.supercube.base.SupercubeBase</i> method), 21
ON (<i>hvl_ccb.dev.se_ils2t.ILS2T.State</i> attribute), 109	operate () (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base</i> method), 41
on_start_wait_until_ready (<i>hvl_ccb.dev.crylas.CryLasLaserConfig</i> attribute), 62	optional_defaults () (<i>hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig</i> class method), 9
ONE (<i>hvl_ccb.comm.serial.SerialCommunicationStopbits</i> attribute), 17	optional_defaults () (<i>hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig</i> class method), 11
ONE (<i>hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum</i> attribute), 69	optional_defaults () (<i>hvl_ccb.comm.opc.OpcUaCommunicationConfig</i> class method), 13
ONE (<i>hvl_ccb.dev.labjack.LabJack.AInRange</i> attribute), 74	optional_defaults () (<i>hvl_ccb.comm.opc.OpcUaCommunicationConfig</i> class method), 13
ONE_HUNDREDTH (<i>hvl_ccb.dev.labjack.LabJack.AInRange</i> attribute), 74	optional_defaults () (<i>hvl_ccb.comm.serial.SerialCommunicationConfig</i> class method), 16
ONE_POINT_FIVE (<i>hvl_ccb.comm.serial.SerialCommunicationStopbits</i> attribute), 17	optional_defaults () (<i>hvl_ccb.comm.visa.VisaCommunicationConfig</i> class method), 19
ONE_TENTH (<i>hvl_ccb.dev.labjack.LabJack.AInRange</i> attribute), 74	optional_defaults () (<i>hvl_ccb.dev.base.EmptyConfig</i> class method), 54
OpcControl (class in <i>hvl_ccb.dev.supercube.constants</i>), 33	optional_defaults () (<i>hvl_ccb.dev.crylas.CryLasAttenuatorConfig</i> class method), 56
OpcUaCommunication (class in <i>hvl_ccb.comm.opc</i>), 11	optional_defaults () (<i>hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig</i> class method), 58
OpcUaCommunicationConfig (class in <i>hvl_ccb.comm.opc</i>), 12	optional_defaults () (<i>hvl_ccb.dev.crylas.CryLasLaserConfig</i> class method), 62
OpcUaCommunicationIOError, 13	optional_defaults () (<i>hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig</i> class method), 64
OpcUaSubHandler (class in <i>hvl_ccb.comm.opc</i>), 13	optional_defaults () (<i>hvl_ccb.dev.ea_psi9000.PSI9000Config</i> class method), 67
open (<i>hvl_ccb.dev.supercube.constants.DoorStatus</i> attribute), 31	optional_defaults () (<i>hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig</i> class method), 68
open (<i>hvl_ccb.dev.supercube.constants.EarthingStickStatus</i> attribute), 32	optional_defaults () (<i>hvl_ccb.dev.heinzinger.HeinzingerConfig</i> class method), 69
open (<i>hvl_ccb.dev.supercube2015.constants.DoorStatus</i> attribute), 45	optional_defaults () (<i>hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig</i> class method), 74
open (<i>hvl_ccb.dev.supercube2015.constants.EarthingStickStatus</i> attribute), 46	optional_defaults () (<i>hvl_ccb.dev.crylas.CryLasLaser</i> method), 60
open () (<i>hvl_ccb.comm.base.CommunicationProtocol</i> method), 7	optional_defaults () (<i>hvl_ccb.comm.visa.VisaCommunicationConfig</i> method), 19
open () (<i>hvl_ccb.comm.labjack_ljm.LJMCommunication</i> method), 8	optional_defaults () (<i>hvl_ccb.dev.crylas.CryLasLaserAnswersShutter</i> attribute), 58
open () (<i>hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication</i> method), 10	
open () (<i>hvl_ccb.comm.opc.OpcUaCommunication</i> method), 12	
open () (<i>hvl_ccb.comm.serial.SerialCommunication</i> method), 14	
open () (<i>hvl_ccb.comm.visa.VisaCommunication</i> method), 17	
open_shutter () (<i>hvl_ccb.dev.crylas.CryLasLaser</i> method), 60	
open_timeout (<i>hvl_ccb.comm.visa.VisaCommunicationConfig</i> attribute), 19	
OPENED (<i>hvl_ccb.dev.crylas.CryLasLaserAnswersShutter</i> attribute), 58	

optional_defaults()	(hvl_ccb.dev.newport.NewportSMC100PPConfig class method), 91	out_1_2 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
optional_defaults()	(hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig class method), 95	out_1_2 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 47
optional_defaults()	(hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig class method), 99	out_2_1 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 47
optional_defaults()	(hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig class method), 101	out_2_2 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
optional_defaults()	(hvl_ccb.dev.rs_rto1024.RTO1024Config class method), 106	out_2_2 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunicationConfig class method), 107	out_3_1 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
optional_defaults()	(hvl_ccb.dev.se_ils2t.ILS2TConfig class method), 111	out_3_1 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.se_ils2t.ILS2TModbusTcpCommunicationConfig class method), 112	out_3_2 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.supercube.base.SupercubeConfiguration class method), 23	out_4_1 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
optional_defaults()	(hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunicationConfig class method), 24	out_4_1 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaConfiguration class method), 36	out_4_2 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
optional_defaults()	(hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaConfiguration class method), 39	out_4_2 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.supercube2015.base.SupercubeConfiguration class method), 42	out_5_1 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunicationConfig class method), 43	out_5_2 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
optional_defaults()	(hvl_ccb.dev.supercube2015.typ_a.SupercubeAOpcUaConfiguration class method), 52	out_5_2 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
optional_defaults()	(hvl_ccb.dev.visa.VisaDeviceConfig class method), 114	out_6_1 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
OT (hvl_ccb.dev.newport.NewportConfigCommands attribute), 83		out_6_1 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
out_1_1 (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33		output (hvl_ccb.dev.supercube.constants.GeneralSupport attribute), 33
out_1_1 (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 47		output (hvl_ccb.dev.supercube2015.constants.GeneralSupport attribute), 48
		output_off() (hvl_ccb.dev.heinzinger.HeinzingerDI method), 70
		output_on() (hvl_ccb.dev.heinzinger.HeinzingerDI method), 71
		OUTPUT_POWER_EXCEEDED (hvl_ccb.dev.newport.NewportSMC100PP.MotorErrors

attribute), 83

Overrange (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus*
attribute), 96

P

PARAM_MISSING_OR_INVALID

(*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig*
attribute), 92

Parity (*hvl_ccb.comm.serial.SerialCommunicationConfig*
attribute), 15

parity (*hvl_ccb.comm.serial.SerialCommunicationConfig*
attribute), 16

parity (*hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig*
attribute), 58

parity (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig*
attribute), 64

parity (*hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig*
attribute), 74

parity (*hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig*
attribute), 82

parity (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig*
attribute), 95

parity (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig*
attribute), 101

Pascal (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits*
attribute), 96

PBR (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes*
attribute), 97

PEAK_CURRENT_LIMIT

(*hvl_ccb.dev.newport.NewportSMC100PP.MotorError*
attribute), 83

peak_output_current_limit
(*hvl_ccb.dev.newport.NewportSMC100PPConfig*
attribute), 91

PfeifferTPG (class in *hvl_ccb.dev.pfeiffer_tpg*), 96

PfeifferTPG.PressureUnits (class in
hvl_ccb.dev.pfeiffer_tpg), 96

PfeifferTPG.SensorStatus (class in
hvl_ccb.dev.pfeiffer_tpg), 96

PfeifferTPG.SensorTypes (class in
hvl_ccb.dev.pfeiffer_tpg), 96

PfeifferTPGConfig (class in
hvl_ccb.dev.pfeiffer_tpg), 98

PfeifferTPGConfig.Model (class in
hvl_ccb.dev.pfeiffer_tpg), 98

PfeifferTPGError, 99

PfeifferTPGSerialCommunication (class in
hvl_ccb.dev.pfeiffer_tpg), 99

PfeifferTPGSerialCommunicationConfig
(class in *hvl_ccb.dev.pfeiffer_tpg*), 100

PKR (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes*
attribute), 97

Poller (class in *hvl_ccb.dev.mbw973*), 82

polling_interval (*hvl_ccb.dev.mbw973.MBW973Config*
attribute), 80

polling_period (*hvl_ccb.dev.crylas.CryLasLaserConfig*
attribute), 62

polling_timeout (*hvl_ccb.dev.crylas.CryLasLaserConfig*
attribute), 62

Port (*hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig*
attribute), 11

port (*hvl_ccb.comm.opc.OpcUaCommunicationConfig*
attribute), 13

port (*hvl_ccb.comm.serial.SerialCommunicationConfig*
attribute), 16

port (*hvl_ccb.comm.visa.VisaCommunicationConfig*
attribute), 19

port (*hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunicationConfig*
attribute), 44

POS (*hvl_ccb.dev.se_ils2t.ILS2T.Ref16Jog* *attribute*), 108

POS_END_OF_TURN (*hvl_ccb.dev.newport.NewportSMC100PP.MotorError*
attribute), 83

POS_FAST (*hvl_ccb.dev.se_ils2t.ILS2T.Ref16Jog* *attribute*), 108

POSITION (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr* *attribute*), 108

POSITION_OUT_OF_LIMIT

(*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig*
attribute), 92

positive_software_limit
(*hvl_ccb.dev.newport.NewportSMC100PPConfig*
attribute), 91

post_force_value()
(*hvl_ccb.dev.newport.NewportSMC100PPConfig*
method), 91

Power (class in *hvl_ccb.dev.supercube.constants*), 34

Power (class in *hvl_ccb.dev.supercube2015.constants*),
48

power_limit (*hvl_ccb.dev.ea_psi9000.PSI9000Config*
attribute), 67

PowerSetup (class in
hvl_ccb.dev.supercube.constants), 34

PowerSetup (class in
hvl_ccb.dev.supercube2015.constants), 49

prepare_ultra_segmentation()
(*hvl_ccb.dev.rs_rto1024.RTO1024* *method*),
102

PSI9000 (class in *hvl_ccb.dev.ea_psi9000*), 64

PSI9000Config (class in *hvl_ccb.dev.ea_psi9000*), 66

PSI9000Error, 67

PSI9000VisaCommunication (class in
hvl_ccb.dev.ea_psi9000), 67

PSI9000VisaCommunicationConfig (class in
hvl_ccb.dev.ea_psi9000), 67

PT100 (*hvl_ccb.dev.labjack.LabJack.ThermocoupleType*
attribute), 76

PT1000 (*hvl_ccb.dev.labjack.LabJack.ThermocoupleType*

attribute), 76
PT500 (*hvl_ccb.dev.labjack.LabJack.ThermocoupleType*
attribute), 76
PTP (*hvl_ccb.dev.se_ils2t.ILS2T.Mode* attribute), 107

Q

QIL (*hvl_ccb.dev.newport.NewportConfigCommands* at-
tribute), 83
query() (*hvl_ccb.comm.visa.VisaCommunication*
method), 17
query() (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunication*
method), 62
query() (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication*
method), 93
query() (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunication*
method), 99
query_all() (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunication*
method), 63
query_multiple() (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication*
method), 93
QUICKSTOP (*hvl_ccb.dev.se_ils2t.ILS2T.State* attribute),
109
QuickStop (*hvl_ccb.dev.supercube.constants.SafetyStatus*
attribute), 35
QuickStop (*hvl_ccb.dev.supercube2015.constants.SafetyStatus*
attribute), 50
quickstop() (*hvl_ccb.dev.se_ils2t.ILS2T* method),
110
quit (*hvl_ccb.dev.supercube.constants.Errors* attribute),
32
quit (*hvl_ccb.dev.supercube2015.constants.Errors* at-
tribute), 47
quit_error() (*hvl_ccb.dev.supercube.base.SupercubeBase*
method), 21
quit_error() (*hvl_ccb.dev.supercube2015.base.Supercube2015Base*
method), 41

R

R (*hvl_ccb.dev.labjack.LabJack.ThermocoupleType* at-
tribute), 76
RAMP_ACC (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr* at-
tribute), 108
RAMP_DECEL (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr* at-
tribute), 108
RAMP_N_MAX (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr* at-
tribute), 108
RAMP_TYPE (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr*
attribute), 108
read() (*hvl_ccb.comm.opc.OpcUaCommunication*
method), 12
read() (*hvl_ccb.dev.mbw973.MBW973* method), 79
read() (*hvl_ccb.dev.supercube.base.SupercubeBase*
method), 21

read() (*hvl_ccb.dev.supercube2015.base.Supercube2015Base*
method), 41
READ_ACTIVE (*hvl_ccb.dev.crylas.CryLasLaser.LaserStatus*
attribute), 58
read_bytes() (*hvl_ccb.comm.serial.SerialCommunication*
method), 14
read_float() (*hvl_ccb.dev.mbw973.MBW973*
method), 79
read_holding_registers()
(*hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication*
method), 10
read_input_registers()
(*hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication*
method), 10
read_int() (*hvl_ccb.dev.mbw973.MBW973* method),
79
read_measurements()
(*hvl_ccb.dev.mbw973.MBW973* method),
79
read_name() (*hvl_ccb.comm.labjack_ljm.LJMCommunication*
method), 8
read_resistance() (*hvl_ccb.dev.labjack.LabJack*
method), 77
read_termination(*hvl_ccb.comm.visa.VisaCommunicationConfig*
attribute), 19
read_text() (*hvl_ccb.comm.serial.SerialCommunication*
method), 14
read_text_nonempty()
(*hvl_ccb.dev.heinzinger.HeinzingerSerialCommunication*
method), 72
READ_TEXT_SKIP_PREFIXES
(*hvl_ccb.dev.crylas.CryLasLaserSerialCommunication*
attribute), 62
read_thermocouple()
(*hvl_ccb.dev.labjack.LabJack* method), 77
READY (*hvl_ccb.dev.crylas.CryLasLaser.AnswersStatus*
attribute), 58
READY (*hvl_ccb.dev.newport.NewportStates* attribute),
95
READY (*hvl_ccb.dev.se_ils2t.ILS2T.State* attribute), 109
ready() (*hvl_ccb.dev.supercube.base.SupercubeBase*
method), 21
ready() (*hvl_ccb.dev.supercube2015.base.Supercube2015Base*
method), 41
READY_FROM_DISABLE
(*hvl_ccb.dev.newport.NewportSMC100PP.StateMessages*
attribute), 84
READY_FROM_HOMING
(*hvl_ccb.dev.newport.NewportSMC100PP.StateMessages*
attribute), 84
READY_FROM_JOGGING
(*hvl_ccb.dev.newport.NewportSMC100PP.StateMessages*
attribute), 84
READY_FROM_MOVING

`(hvl_ccb.dev.newport.NewportSMC100PP.StateMessage`
`attribute), 84`
`READY_INACTIVE (hvl_ccb.dev.crylas.CryLasLaser.LaserStatus`
`attribute), 58`
`RedOperate (hvl_ccb.dev.supercube.constants.SafetyStatus`
`attribute), 35`
`RedOperate (hvl_ccb.dev.supercube2015.constants.SafetyStatus`
`attribute), 50`
`RedReady (hvl_ccb.dev.supercube.constants.SafetyStatus`
`attribute), 35`
`RedReady (hvl_ccb.dev.supercube2015.constants.SafetyStatus`
`attribute), 50`
`RELATIVE_POSITION_MOTOR`
`(hvl_ccb.dev.se_ils2t.ILS2T.ActionsPtp at-`
`tribute), 107`
`RELATIVE_POSITION_TARGET`
`(hvl_ccb.dev.se_ils2t.ILS2T.ActionsPtp at-`
`tribute), 107`
`relative_step () (hvl_ccb.dev.se_ils2t.ILS2T`
`method), 110`
`relative_step_and_wait ()`
`(hvl_ccb.dev.se_ils2t.ILS2T method), 110`
`remove_device () (hvl_ccb.dev.base.DeviceSequenceMixin`
`method), 53`
`required_keys () (hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig`
`class method), 10`
`required_keys () (hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig`
`class method), 11`
`required_keys () (hvl_ccb.comm.opc.OpcUaCommunicationConfig`
`class method), 13`
`required_keys () (hvl_ccb.comm.serial.SerialCommunicationConfig`
`class method), 16`
`required_keys () (hvl_ccb.comm.visa.VisaCommunicationConfig`
`class method), 19`
`required_keys () (hvl_ccb.dev.base.EmptyConfig`
`class method), 54`
`required_keys () (hvl_ccb.dev.crylas.CryLasAttenuatorConfig`
`class method), 56`
`required_keys () (hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig`
`class method), 58`
`required_keys () (hvl_ccb.dev.crylas.CryLasLaserConfig`
`class method), 62`
`required_keys () (hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig`
`class method), 64`
`required_keys () (hvl_ccb.dev.ea_psi9000.PSI9000Config`
`class method), 67`
`required_keys () (hvl_ccb.dev.ea_psi9000.PSI9000VisaCommunicationConfig`
`class method), 68`
`required_keys () (hvl_ccb.dev.heinzinger.HeinzingerConfig`
`class method), 69`
`required_keys () (hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig`
`class method), 74`
`required_keys () (hvl_ccb.dev.mbw973.MBW973Config`
`class method), 80`
`required_keys () (hvl_ccb.dev.mbw973.MBW973SerialCommunication`
`class method), 82`
`required_keys () (hvl_ccb.dev.newport.NewportSMC100PPConfig`
`class method), 92`
`required_keys () (hvl_ccb.dev.newport.NewportSMC100PPSerialComm`
`class method), 95`
`required_keys () (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig`
`class method), 99`
`required_keys () (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommun`
`class method), 101`
`required_keys () (hvl_ccb.dev.rs_rto1024.RTO1024Config`
`class method), 106`
`required_keys () (hvl_ccb.dev.rs_rto1024.RTO1024VisaCommunication`
`class method), 107`
`required_keys () (hvl_ccb.dev.se_ils2t.ILS2TConfig`
`class method), 111`
`required_keys () (hvl_ccb.dev.se_ils2t.ILS2TModbusTcpCommunication`
`class method), 112`
`required_keys () (hvl_ccb.dev.supercube.base.SupercubeConfiguration`
`class method), 23`
`required_keys () (hvl_ccb.dev.supercube.base.SupercubeOpcUaComm`
`class method), 24`
`required_keys () (hvl_ccb.dev.supercube.typ_a.SupercubeAOpcUaCor`
`class method), 36`
`required_keys () (hvl_ccb.dev.supercube.typ_b.SupercubeBOpcUaCor`
`class method), 39`
`required_keys () (hvl_ccb.dev.supercube2015.base.SupercubeConfig`
`class method), 43`
`required_keys () (hvl_ccb.dev.supercube2015.base.SupercubeOpcUaC`
`class method), 44`
`required_keys () (hvl_ccb.dev.supercube2015.base.SupercubeOpcU`
`class method), 52`
`required_keys () (hvl_ccb.dev.visa.VisaDeviceConfig`
`class method), 114`
`reset (hvl_ccb.dev.supercube.constants.BreakdownDetection`
`attribute), 30`
`reset (hvl_ccb.dev.supercube2015.constants.BreakdownDetection`
`attribute), 45`
`reset (hvl_ccb.dev.newport.NewportSMC100PP`
`method), 87`
`reset () (hvl_ccb.dev.visa.VisaDevice method), 113`
`reset_error () (hvl_ccb.dev.se_ils2t.ILS2T method),`
`reset_interface ()`
`(hvl_ccb.dev.heinzinger.HeinzingerDI method),`
`71`
`time`
`(hvl_ccb.dev.crylas.CryLasAttenuatorConfig`
`attribute), 56`
`RMS_CURRENT_LIMIT`
`(hvl_ccb.dev.newport.NewportSMC100PP.MotorErrors`
`attribute), 83`
`rpm_max_init (hvl_ccb.dev.se_ils2t.ILS2TConfig at-`
`tribute), 112`

rs485_address (*hvl_ccb.dev.newport.NewportSMC100PPConfig* attribute), 92

RTO1024 (*class in hvl_ccb.dev.rs_rto1024*), 101

RTO1024.TriggerModes (*class in hvl_ccb.dev.rs_rto1024*), 101

RTO1024Config (*class in hvl_ccb.dev.rs_rto1024*), 105

RTO1024Error, 106

RTO1024VisaCommunication (*class in hvl_ccb.dev.rs_rto1024*), 106

RTO1024VisaCommunicationConfig (*class in hvl_ccb.dev.rs_rto1024*), 106

run() (*hvl_ccb.dev.visa.VisaStatusPoller method*), 114

run() (*hvl_ccb.experiment_manager.ExperimentManager method*), 117

run_continuous_acquisition() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 102

run_single_acquisition() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 103

RUNNING (*hvl_ccb.experiment_manager.ExperimentStatus attribute*), 117

S

S (*hvl_ccb.dev.labjack.LabJack.ThermocoupleType attribute*), 76

SA (*hvl_ccb.dev.newport.NewportConfigCommands attribute*), 83

Safety (*class in hvl_ccb.dev.supercube.constants*), 35

Safety (*class in hvl_ccb.dev.supercube2015.constants*), 49

SafetyStatus (*class in hvl_ccb.dev.supercube.constants*), 35

SafetyStatus (*class in hvl_ccb.dev.supercube2015.constants*), 49

save_configuration() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 103

save_waveform_history() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 103

SCALE (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr attribute*), 108

ScalingFactorValueError, 113

screw_scaling (*hvl_ccb.dev.newport.NewportSMC100PPConfig attribute*), 92

send_command() (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication method*), 93

send_command() (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunication method*), 99

send_stop() (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunication method*), 93

SerialCommunication_error (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus attribute*), 96

Sensor_off (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus attribute*), 96

SerialCommunication (*class in hvl_ccb.comm.serial*), 14

SerialCommunicationBytesize (*class in hvl_ccb.comm.serial*), 15

SerialCommunicationConfig (*class in hvl_ccb.comm.serial*), 15

SerialCommunicationIOError, 16

SerialCommunicationParity (*class in hvl_ccb.comm.serial*), 16

SerialCommunicationStopbits (*class in hvl_ccb.comm.serial*), 17

set_acceleration() (*hvl_ccb.dev.newport.NewportSMC100PP method*), 87

set_acquire_length() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 103

set_ain_differential() (*hvl_ccb.dev.labjack.LabJack method*), 77

set_ain_range() (*hvl_ccb.dev.labjack.LabJack method*), 77

set_ain_resistance() (*hvl_ccb.dev.labjack.LabJack method*), 77

set_ain_resolution() (*hvl_ccb.dev.labjack.LabJack method*), 78

set_ain_thermocouple() (*hvl_ccb.dev.labjack.LabJack method*), 78

set_attenuation() (*hvl_ccb.dev.crylas.CryLasAttenuator method*), 55

set_cel16_socket() (*hvl_ccb.dev.supercube.base.SupercubeBase method*), 21

set_cel16_socket() (*hvl_ccb.dev.supercube2015.base.Supercube2015Base method*), 41

set_channel_position() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 103

set_channel_range() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 103

set_channel_scale() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 104

set_channel_state() (*hvl_ccb.dev.rs_rto1024.RTO1024 method*), 104

set_current() (*hvl_ccb.dev.heinzinger.HeinzingerDI method*), 71

set_current() (<i>hvl_ccb.dev.heinzinger.HeinzingerPNC method</i>), 72	(<i>hvl_ccb.dev.crylas.CryLasLaser method</i>), 60
set_digital_output() (<i>hvl_ccb.dev.labjack.LabJack method</i>), 78	set_ramp_type() (<i>hvl_ccb.dev.se_ils2t.ILS2T method</i>), 111
set_display_unit() (<i>hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method</i>), 98	set_reference_point() (<i>hvl_ccb.dev.rs_rto1024.RTO1024 method</i>), 104
set_earthing_manual() (<i>hvl_ccb.dev.supercube.base.SupercubeBase method</i>), 22	set_remote_control() (<i>hvl_ccb.dev.supercube.base.SupercubeBase method</i>), 22
set_earthing_manual() (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base method</i>), 41	set_remote_control() (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base method</i>), 41
set_full_scale_mbar() (<i>hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method</i>), 98	set_repetition_rate() (<i>hvl_ccb.dev.crylas.CryLasLaser method</i>), 60
set_full_scale_unitless() (<i>hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method</i>), 98	set_repetitions() (<i>hvl_ccb.dev.rs_rto1024.RTO1024 method</i>), 104
set_init_attenuation() (<i>hvl_ccb.dev.crylas.CryLasAttenuator method</i>), 55	set_slope() (<i>hvl_ccb.dev.supercube.typ_a.SupercubeWithFU method</i>), 37
set_init_shutter_status() (<i>hvl_ccb.dev.crylas.CryLasLaser method</i>), 60	set_slope() (<i>hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU method</i>), 51
set_jog_speed() (<i>hvl_ccb.dev.se_ils2t.ILS2T method</i>), 110	set_support_output() (<i>hvl_ccb.dev.supercube.base.SupercubeBase method</i>), 22
set_lower_limits() (<i>hvl_ccb.dev.ea_psi9000.PSI9000 method</i>), 65	set_support_output() (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base method</i>), 41
set_max_acceleration() (<i>hvl_ccb.dev.se_ils2t.ILS2T method</i>), 110	set_support_output_impulse() (<i>hvl_ccb.dev.supercube.base.SupercubeBase method</i>), 22
set_max_deceleration() (<i>hvl_ccb.dev.se_ils2t.ILS2T method</i>), 110	set_support_output_impulse() (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base method</i>), 41
set_max_rpm() (<i>hvl_ccb.dev.se_ils2t.ILS2T method</i>), 110	set_system_lock() (<i>hvl_ccb.dev.ea_psi9000.PSI9000 method</i>), 65
set_measuring_options() (<i>hvl_ccb.dev.mbw973.MBW973 method</i>), 79	set_t13_socket() (<i>hvl_ccb.dev.supercube.base.SupercubeBase method</i>), 22
set_motor_configuration() (<i>hvl_ccb.dev.newport.NewportSMC100PP method</i>), 88	set_t13_socket() (<i>hvl_ccb.dev.supercube2015.base.Supercube2015Base method</i>), 42
set_negative_software_limit() (<i>hvl_ccb.dev.newport.NewportSMC100PP method</i>), 88	set_target_voltage() (<i>hvl_ccb.dev.supercube.typ_a.SupercubeWithFU method</i>), 38
set_number_of_recordings() (<i>hvl_ccb.dev.heinzinger.HeinzingerDI method</i>), 71	set_target_voltage() (<i>hvl_ccb.dev.supercube2015.typ_a.Supercube2015WithFU method</i>), 51
set_output() (<i>hvl_ccb.dev.ea_psi9000.PSI9000 method</i>), 65	set_transmission() (<i>hvl_ccb.dev.crylas.CryLasAttenuator method</i>), 55
set_positive_software_limit() (<i>hvl_ccb.dev.newport.NewportSMC100PP method</i>), 88	set_trigger_level() (<i>hvl_ccb.dev.rs_rto1024.RTO1024 method</i>), 105
set_pulse_energy()	

set_trigger_mode() (hvl_ccb.dev.rs_rto1024.RTO1024 method), 105	SR (hvl_ccb.dev.newport.NewportConfigCommands attribute), 83
set_trigger_source() (hvl_ccb.dev.rs_rto1024.RTO1024 method), 105	stage_configuration (hvl_ccb.dev.newport.NewportSMC100PPConfig attribute), 92
set_upper_limits() (hvl_ccb.dev.ea_psi9000.PSI9000 method), 65	start() (hvl_ccb.dev.base.Device method), 53
set_voltage() (hvl_ccb.dev.heinzinger.HeinzingerDI method), 71	start() (hvl_ccb.dev.base.DeviceSequenceMixin method), 53
set_voltage() (hvl_ccb.dev.heinzinger.HeinzingerPNC method), 72	start() (hvl_ccb.dev.base.SingleCommDevice method), 54
set_voltage_current() (hvl_ccb.dev.ea_psi9000.PSI9000 method), 66	start() (hvl_ccb.dev.crylas.CryLasAttenuator method), 55
setup (hvl_ccb.dev.supercube.constants.Power attribute), 34	start() (hvl_ccb.dev.crylas.CryLasLaser method), 60
setup (hvl_ccb.dev.supercube2015.constants.Power attribute), 48	start() (hvl_ccb.dev.ea_psi9000.PSI9000 method), 66
SEVENBITS (hvl_ccb.comm.serial.SerialCommunicationBytesize attribute), 15	start() (hvl_ccb.dev.heinzinger.HeinzingerDI method), 71
SHORT_CIRCUIT (hvl_ccb.dev.newport.NewportSMC100PP.MotorError attribute), 83	start() (hvl_ccb.dev.heinzinger.HeinzingerPNC method), 72
SHUTDOWN_CURRENT_LIMIT (hvl_ccb.dev.ea_psi9000.PSI9000 attribute), 64	start() (hvl_ccb.dev.labjack.LabJack method), 78
SHUTDOWN_VOLTAGE_LIMIT (hvl_ccb.dev.ea_psi9000.PSI9000 attribute), 64	start() (hvl_ccb.dev.mbw973.MBW973 method), 79
ShutterStatus (hvl_ccb.dev.crylas.CryLasLaser attribute), 59	start() (hvl_ccb.dev.mbw973.Poller method), 82
ShutterStatus (hvl_ccb.dev.crylas.CryLasLaserConfig attribute), 61	start() (hvl_ccb.dev.newport.NewportSMC100PP method), 88
SingleCommDevice (class in hvl_ccb.dev.base), 54	start() (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method), 98
SIXBITS (hvl_ccb.comm.serial.SerialCommunicationBytesize attribute), 15	start() (hvl_ccb.dev.rs_rto1024.RTO1024 method), 105
SIXTEEN (hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingEnum attribute), 69	start() (hvl_ccb.dev.se_ils2t.ILS2T method), 111
SL (hvl_ccb.dev.newport.NewportConfigCommands attribute), 83	start() (hvl_ccb.dev.supercube.base.SupercubeBase method), 22
SOFTWARE_INTERNAL_SIXTY (hvl_ccb.dev.crylas.CryLasLaser.RepetitionRates attribute), 59	start() (hvl_ccb.dev.supercube2015.base.Supercube2015Base method), 42
SOFTWARE_INTERNAL_TEN (hvl_ccb.dev.crylas.CryLasLaser.RepetitionRates attribute), 59	start() (hvl_ccb.dev.visa.VisaDevice method), 113
SOFTWARE_INTERNAL_TWENTY (hvl_ccb.dev.crylas.CryLasLaser.RepetitionRates attribute), 59	start() (hvl_ccb.experiment_manager.ExperimentManager method), 117
SPACE (hvl_ccb.comm.serial.SerialCommunicationParity attribute), 17	start_control() (hvl_ccb.dev.mbw973.MBW973 method), 80
spoll() (hvl_ccb.comm.visa.VisaCommunication method), 17	STARTING (hvl_ccb.experiment_manager.ExperimentStatus attribute), 117
spoll_handler() (hvl_ccb.dev.visa.VisaDevice method), 113	States (hvl_ccb.dev.newport.NewportSMC100PP attribute), 84
	status (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
	status (hvl_ccb.dev.supercube.constants.Safety attribute), 35
	status (hvl_ccb.experiment_manager.ExperimentManager attribute), 117
	status_1 (hvl_ccb.dev.supercube.constants.Door attribute), 30
	status_1 (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
	status_1_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
	status_1_connected

(hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_1_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_2 (hvl_ccb.dev.supercube.constants.Door attribute), 30
 status_2 (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
 status_2_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_2_connected (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_2_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_3 (hvl_ccb.dev.supercube.constants.Door attribute), 30
 status_3 (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
 status_3_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_3_connected (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_3_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_4 (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
 status_4_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_4_connected (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_4_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_5 (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
 status_5_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_5_connected (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_5_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_6 (hvl_ccb.dev.supercube.constants.EarthingStick attribute), 31
 status_6_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_6_connected (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_6_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_closed (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 15

status_connected (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_error (hvl_ccb.dev.supercube2015.constants.Safety attribute), 49
 status_green (hvl_ccb.dev.supercube2015.constants.Safety attribute), 49
 status_open (hvl_ccb.dev.supercube2015.constants.EarthingStick attribute), 46
 status_ready_for_red (hvl_ccb.dev.supercube2015.constants.Safety attribute), 49
 status_red (hvl_ccb.dev.supercube2015.constants.Safety attribute), 49
 stop (hvl_ccb.dev.supercube.constants.Errors attribute), 32
 stop (hvl_ccb.dev.supercube2015.constants.Errors attribute), 47
 stop () (hvl_ccb.dev.base.Device method), 53
 stop () (hvl_ccb.dev.base.DeviceSequenceMixin method), 54
 stop () (hvl_ccb.dev.base.SingleCommDevice method), 54
 stop () (hvl_ccb.dev.crylas.CryLasLaser method), 60
 stop () (hvl_ccb.dev.ea_psi9000.PSI9000 method), 66
 stop () (hvl_ccb.dev.heinzinger.HeinzingerDI method), 71
 stop () (hvl_ccb.dev.labjack.LabJack method), 78
 stop () (hvl_ccb.dev.mbw973.MBW973 method), 80
 stop () (hvl_ccb.dev.mbw973.Poller method), 82
 stop () (hvl_ccb.dev.newport.NewportSMC100PP method), 88
 stop () (hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG method), 88
 stop () (hvl_ccb.dev.rs_rto1024.RTO1024 method), 105
 stop () (hvl_ccb.dev.se_ils2t.ILS2T method), 111
 stop () (hvl_ccb.dev.supercube.base.SupercubeBase method), 22
 stop () (hvl_ccb.dev.supercube2015.base.Supercube2015Base method), 42
 stop () (hvl_ccb.dev.visa.VisaDevice method), 113
 stop () (hvl_ccb.dev.visa.VisaStatusPoller method), 114
 stop () (hvl_ccb.experiment_manager.ExperimentManager method), 117
 stop_acquisition () (hvl_ccb.dev.rs_rto1024.RTO1024 method), 105
 stop_motion () (hvl_ccb.dev.newport.NewportSMC100PP method), 89
 stop_number (hvl_ccb.dev.supercube2015.constants.Errors attribute), 47
 Stopbits (hvl_ccb.comm.serial.SerialCommunicationConfig attribute), 15

stopbits (*hvl_ccb.comm.serial.SerialCommunicationConfig* attribute), 16
 stopbits (*hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig* attribute), 58
 stopbits (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig* attribute), 64
 stopbits (*hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig* attribute), 74
 stopbits (*hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig* attribute), 82
 stopbits (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig* attribute), 95
 stopbits (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig* attribute), 101
 StrEnumBase (class in *hvl_ccb.utils.enum*), 115
 sub_handler (*hvl_ccb.comm.opc.OpcUaCommunicationConfig* attribute), 13
 sub_handler (*hvl_ccb.dev.supercube.base.SupercubeOpcUaCommunicationConfig* attribute), 24
 sub_handler (*hvl_ccb.dev.supercube2015.base.SupercubeOpcUaCommunicationConfig* attribute), 44
 Supercube2015Base (class in *hvl_ccb.dev.supercube2015.base*), 39
 Supercube2015WithFU (class in *hvl_ccb.dev.supercube2015.typ_a*), 50
 SupercubeAOpcUaCommunication (class in *hvl_ccb.dev.supercube.typ_a*), 36
 SupercubeAOpcUaCommunication (class in *hvl_ccb.dev.supercube2015.typ_a*), 51
 SupercubeAOpcUaConfiguration (class in *hvl_ccb.dev.supercube.typ_a*), 36
 SupercubeAOpcUaConfiguration (class in *hvl_ccb.dev.supercube2015.typ_a*), 51
 SupercubeB (class in *hvl_ccb.dev.supercube.typ_b*), 38
 SupercubeBase (class in *hvl_ccb.dev.supercube.base*), 20
 SupercubeBOpcUaCommunication (class in *hvl_ccb.dev.supercube.typ_b*), 38
 SupercubeBOpcUaConfiguration (class in *hvl_ccb.dev.supercube.typ_b*), 38
 SupercubeConfiguration (class in *hvl_ccb.dev.supercube.base*), 23
 SupercubeConfiguration (class in *hvl_ccb.dev.supercube2015.base*), 42
 SupercubeOpcEndpoint (class in *hvl_ccb.dev.supercube.constants*), 35
 SupercubeOpcEndpoint (class in *hvl_ccb.dev.supercube2015.constants*), 50
 SupercubeOpcUaCommunication (class in *hvl_ccb.dev.supercube.base*), 23
 SupercubeOpcUaCommunication (class in *hvl_ccb.dev.supercube2015.base*), 43
 SupercubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.supercube.base*), 23
 SupercubeOpcUaCommunicationConfig (class in *hvl_ccb.dev.supercube2015.base*), 43
 SupercubeOpcUaCommunicationConfigSubscriptionHandler (class in *hvl_ccb.dev.supercube.base*), 24
 SupercubeOpcUaCommunicationConfigSubscriptionHandler (class in *hvl_ccb.dev.supercube2015.base*), 44
 SupercubeOpcUaCommunicationConfigWithFU (class in *hvl_ccb.dev.supercube.typ_a*), 36
 SupercubeOpcUaCommunicationConfigWithFU_green (class in *hvl_ccb.dev.supercube2015.constants.Safety* attribute), 49
 SupercubeOpcUaCommunicationConfigWithFU_red (class in *hvl_ccb.dev.supercube2015.constants.Safety* attribute), 35
 SupercubeOpcUaCommunicationConfigWithFU_green (class in *hvl_ccb.dev.supercube2015.constants.Safety* attribute), 49
 switchto_ready (*hvl_ccb.dev.supercube.constants.Safety* attribute), 35
 switchto_ready (*hvl_ccb.dev.supercube2015.constants.Safety* attribute), 35
 T (class in *hvl_ccb.dev.labjack.LabJack.ThermocoupleType* attribute), 76
 t13_1 (*hvl_ccb.dev.supercube.constants.GeneralSockets* attribute), 32
 t13_1 (*hvl_ccb.dev.supercube2015.constants.GeneralSockets* attribute), 47
 t13_2 (*hvl_ccb.dev.supercube.constants.GeneralSockets* attribute), 32
 t13_2 (*hvl_ccb.dev.supercube2015.constants.GeneralSockets* attribute), 47
 t13_3 (*hvl_ccb.dev.supercube.constants.GeneralSockets* attribute), 32
 t13_3 (*hvl_ccb.dev.supercube2015.constants.GeneralSockets* attribute), 47
 T13_SOCKET_PORTS (in module *hvl_ccb.dev.supercube.constants*), 35
 T13_SOCKET_PORTS (in module *hvl_ccb.dev.supercube2015.constants*), 50
 T4 (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.DeviceType* attribute), 9
 T4 (*hvl_ccb.dev.labjack.LabJack.DeviceType* attribute), 75
 T7 (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.DeviceType* attribute), 9
 T7 (*hvl_ccb.dev.labjack.LabJack.DeviceType* attribute), 75
 T7_PRO (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.DeviceType* attribute), 9
 T7_PRO (*hvl_ccb.dev.labjack.LabJack.DeviceType* attribute), 75
 target_pulse_energy (*hvl_ccb.dev.crylas.CryLasLaser* attribute), 60

TCP (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType* attribute), 9
 TCP (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorTypes* attribute), 97
 TCP_IP_INSTR (*hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType* attribute), 18
 TCP_IP_INSTR (*hvl_ccb.dev.crylas.CryLasAttenuator* attribute), 55
 TCP_IP_SOCKET (*hvl_ccb.comm.visa.VisaCommunicationConfig.InterfaceType* attribute), 18
 TCP_IP_SOCKET (*hvl_ccb.dev.supercube.constants.BreakdownDetection* attribute), 30
 TEC1 (*hvl_ccb.dev.crylas.CryLasLaser.AnswersStatus* attribute), 58
 triggered (*hvl_ccb.dev.supercube2015.constants.BreakdownDetection* attribute), 45
 TEC2 (*hvl_ccb.dev.crylas.CryLasLaser.AnswersStatus* attribute), 58
 TWO (*hvl_ccb.comm.serial.SerialCommunicationStopbits* attribute), 17
 TEMP (*hvl_ccb.dev.se_ils2t.ILS2T.RegAddr* attribute), 108
 TWO (*hvl_ccb.dev.heinzinger.HeinzingerConfig.RecordingsEnum* attribute), 69
 TEN (*hvl_ccb.dev.labjack.LabJack.AInRange* attribute), 74
 TWO_HUNDRED (*hvl_ccb.dev.labjack.LabJack.CalMicroAmpere* attribute), 75
 TEN (*hvl_ccb.dev.labjack.LabJack.CalMicroAmpere* attribute), 75
 U
 terminator (*hvl_ccb.comm.serial.SerialCommunicationConfig* attribute), 16
 underrange (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.SensorStatus* attribute), 96
 terminator (*hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig* attribute), 58
 UNKNOWN (*hvl_ccb.comm.serial.SerialCommunicationConfig* attribute), 14
 terminator (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig* attribute), 64
 unit (*hvl_ccb.comm.modbus_tcp.ModbusTcpCommunicationConfig* attribute), 11
 terminator (*hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig* attribute), 74
 unit (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG* attribute), 98
 terminator (*hvl_ccb.dev.se_ils2t.ILS2TModbusTcpCommunicationConfig* attribute), 113
 unit (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 2
 terminator (*hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig* attribute), 82
 unit_current (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 2
 terminator (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig* attribute), 95
 unit_voltage (*hvl_ccb.dev.heinzinger.HeinzingerPNC* attribute), 2
 terminator (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig* attribute), 101
 UNKNOWN (*hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitCurrent* attribute), 71
 timeout (*hvl_ccb.comm.serial.SerialCommunicationConfig* attribute), 16
 UNKNOWN (*hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitVoltage* attribute), 71
 timeout (*hvl_ccb.comm.visa.VisaCommunicationConfig* attribute), 19
 UNREADY_INACTIVE (*hvl_ccb.dev.crylas.CryLasLaser.LaserStatus* attribute), 58
 timeout (*hvl_ccb.dev.crylas.CryLasAttenuatorSerialCommunicationConfig* attribute), 58
 update_laser_status () (*hvl_ccb.dev.crylas.CryLasLaser* method), 60
 timeout (*hvl_ccb.dev.crylas.CryLasLaserSerialCommunicationConfig* attribute), 64
 update_period (*hvl_ccb.comm.opc.OpcUaCommunicationConfig* attribute), 13
 timeout (*hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig* attribute), 74
 update_repetition_rate () (*hvl_ccb.dev.crylas.CryLasLaser* method), 61
 timeout (*hvl_ccb.dev.mbw973.MBW973SerialCommunicationConfig* attribute), 82
 update_shutter_status () (*hvl_ccb.dev.crylas.CryLasLaser* method), 61
 timeout (*hvl_ccb.dev.newport.NewportSMC100PPSerialCommunicationConfig* attribute), 95
 timeout (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGSerialCommunicationConfig* attribute), 101
 timer_callback () (*hvl_ccb.dev.mbw973.Poller* method), 82
 update_target_pulse_energy () (*hvl_ccb.dev.crylas.CryLasLaser* method), 61
 Torr (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits* attribute), 96
 UpdateEspStageInfo (*hvl_ccb.dev.newport.NewportSMC100PPConfig.EspStageConfig* attribute), 90
 TPG25xA (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model* attribute), 98
 TPGx6x (*hvl_ccb.dev.pfeiffer_tpg.PfeifferTPGConfig.Model* attribute), 98
 USB (*hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType* attribute), 9

`user_position_offset` (`hvl_ccb.dev.newport.NewportSMC100PPConfig` attribute), 92

`user_steps()` (`hvl_ccb.dev.se_ils2t.ILS2T` method), 111

V

`V` (`hvl_ccb.dev.heinzinger.HeinzingerPNC.UnitVoltage` attribute), 71

`VA` (`hvl_ccb.dev.newport.NewportConfigCommands` attribute), 83

`value` (`hvl_ccb.dev.labjack.LabJack.AInRange` attribute), 74

`ValueEnum` (class in `hvl_ccb.utils.enum`), 115

`VB` (`hvl_ccb.dev.newport.NewportConfigCommands` attribute), 83

`velocity` (`hvl_ccb.dev.newport.NewportSMC100PPConfig` attribute), 92

`visa_backend` (`hvl_ccb.comm.visa.VisaCommunicationConfig` attribute), 19

`VisaCommunication` (class in `hvl_ccb.comm.visa`), 17

`VisaCommunicationConfig` (class in `hvl_ccb.comm.visa`), 18

`VisaCommunicationConfig.InterfaceType` (class in `hvl_ccb.comm.visa`), 18

`VisaCommunicationError`, 19

`VisaDevice` (class in `hvl_ccb.dev.visa`), 113

`VisaDeviceConfig` (class in `hvl_ccb.dev.visa`), 114

`VisaStatusPoller` (class in `hvl_ccb.dev.visa`), 114

`Volt` (`hvl_ccb.dev.pfeiffer_tpg.PfeifferTPG.PressureUnits` attribute), 96

`VOLT` (`hvl_ccb.dev.se_ils2t.ILS2T.RegAddr` attribute), 108

`voltage_lower_limit` (`hvl_ccb.dev.ea_psi9000.PSI9000Config` attribute), 67

`voltage_max` (`hvl_ccb.dev.supercube.constants.Power` attribute), 34

`voltage_max` (`hvl_ccb.dev.supercube2015.constants.Power` attribute), 48

`voltage_primary` (`hvl_ccb.dev.supercube.constants.Power` attribute), 34

`voltage_primary` (`hvl_ccb.dev.supercube2015.constants.Power` attribute), 48

`voltage_slope` (`hvl_ccb.dev.supercube.constants.Power` attribute), 34

`voltage_slope` (`hvl_ccb.dev.supercube2015.constants.Power` attribute), 49

`voltage_target` (`hvl_ccb.dev.supercube.constants.Power` attribute), 34

`voltage_target` (`hvl_ccb.dev.supercube2015.constants.Power` attribute), 49

`voltage_upper_limit`

W

`WAIT_AFTER_WRITE` (`hvl_ccb.comm.visa.VisaCommunication` attribute), 17

`wait_operation_complete()` (`hvl_ccb.dev.visa.VisaDevice` method), 113

`wait_sec_initialisation` (`hvl_ccb.dev.ea_psi9000.PSI9000Config` attribute), 67

`wait_sec_post_absolute_position` (`hvl_ccb.dev.se_ils2t.ILS2TConfig` attribute), 112

`wait_sec_post_enable` (`hvl_ccb.dev.se_ils2t.ILS2TConfig` attribute), 112

`wait_sec_post_relative_step` (`hvl_ccb.dev.se_ils2t.ILS2TConfig` attribute), 112

`wait_sec_read_text_nonempty` (`hvl_ccb.dev.heinzinger.HeinzingerSerialCommunicationConfig` attribute), 74

`wait_sec_settings_effect` (`hvl_ccb.dev.ea_psi9000.PSI9000Config` attribute), 67

`wait_sec_stop_commands` (`hvl_ccb.dev.heinzinger.HeinzingerConfig` attribute), 69

`wait_sec_system_lock` (`hvl_ccb.dev.ea_psi9000.PSI9000Config` attribute), 67

`wait_until_motor_initialized()` (`hvl_ccb.dev.newport.NewportSMC100PP` method), 89

`wait_until_move_finished()` (`hvl_ccb.dev.newport.NewportSMC100PP` method), 89

`wait_until_ready()` (`hvl_ccb.dev.crylas.CryLasLaser` method), 61

`warning` (`hvl_ccb.dev.supercube.constants.Errors` attribute), 32

`write` (`hvl_ccb.comm.labjack_ljm.LJMCommunicationConfig.ConnectionType` attribute), 9

`write()` (`hvl_ccb.comm.opc.OpcUaCommunication` method), 12

`write()` (`hvl_ccb.comm.visa.VisaCommunication` method), 18

`write()` (`hvl_ccb.dev.mbw973.MBW973` method), 80

`write()` (`hvl_ccb.dev.supercube.base.SupercubeBase` method), 22

`write()` (`hvl_ccb.dev.supercube2015.base.Supercube2015Base` method), 42

`write_bytes()` (*hvl_ccb.comm.serial.SerialCommunication*
method), 15

`write_name()` (*hvl_ccb.comm.labjack_ljm.LJMCommunication*
method), 8

`write_names()` (*hvl_ccb.comm.labjack_ljm.LJMCommunication*
method), 8

`write_registers()`
(*hvl_ccb.comm.modbus_tcp.ModbusTcpCommunication*
method), 10

`write_termination`
(*hvl_ccb.comm.visa.VisaCommunicationConfig*
attribute), 19

`write_text()` (*hvl_ccb.comm.serial.SerialCommunication*
method), 15

`WRONG_ESP_STAGE` (*hvl_ccb.dev.newport.NewportSMC100PP.MotorErrors*
attribute), 83

Z

`ZX` (*hvl_ccb.dev.newport.NewportConfigCommands* *at-*
tribute), 83