HtHeatpump Documentation

Release 1.3.2

Daniel Strigl

Jan 13, 2022

Contents

Documentation built using Sphinx Jan 13, 2022 for HtHeatpump version 1.3.2.

Contents:

CHAPTER 1

HtHeatpump

Easy-to-use Python communication module for Heliotherm and Brötje BSW NEO heat pumps.

- GitHub repo: https://github.com/dstrigl/htheatpump
- Documentation: https://htheatpump.readthedocs.io
- Free software: GNU General Public License v3

1.1 Introduction

This library provides a pure Python interface to access Heliotherm and Brötje BSW NEO heat pumps over a serial connection. It's compatible with Python version 3.7 and 3.8.

1.1.1 Features

- read the manufacturer's serial number of the heat pump
- read the software version of the heat pump
- read and write the current date and time of the heat pump
- read the fault list of the heat pump
- query whether the heat pump is malfunctioning
- query for several parameters of the heat pump

- change parameter values of the heat pump
- fast query of MP data points / parameters ("Web-Online")
- read and write the time programs of the heat pump

1.1.2 Tested with*[0]

*

- Heliotherm HP08S10W-WEB, SW 3.0.20
- Heliotherm HP10S12W-WEB, SW 3.0.8
- Heliotherm HP08E-K-BC, SW 3.0.7B
- Heliotherm HP05S07W-WEB, SW 3.0.17 and SW 3.0.37
- Heliotherm HP12L-M-BC, SW 3.0.21
- Heliotherm HP07S08W-WEB, SW 3.0.37
- Heliotherm HP-30-L-M-WEB, SW 3.0.21
- Brötje BSW NEO 8 SW 3.0.38

1.2 Installation

You can install or upgrade `htheatpump` with:

```
$ pip install htheatpump --upgrade
```

Or you can install from source with:

```
$ git clone https://github.com/dstrigl/htheatpump.git
$ cd htheatpump
$ python setup.py install
```

1.3 Getting started

To use `htheatpump` in a project take a look on the following example. After establishing a connection with the Heliotherm heat pump one can interact with it by different functions like reading or writing parameters.

```
from htheatpump import HtHeatpump

hp = HtHeatpump("/dev/ttyUSB0", baudrate=9600)
try:
    hp.open_connection()
    hp.login()
    # query for the outdoor temperature
    temp = hp.get_param("Temp. Aussen")
    print(temp)
    # ...
finally:
```

(continues on next page)

[0] thanks to Kilian, Hans, Alois, Simon, Felix (FelixPetriconi) and Matthias for contribution

(continued from previous page)

```
    hp.logout()  # try to logout for an ordinary cancellation (if possible)
    hp.close_connection()
```

```
from htheatpump import AioHtHeatpump

hp = AioHtHeatpump("/dev/ttyUSB0", baudrate=9600)
try:
    hp.open_connection()
    await hp.login_async()
    # query for the outdoor temperature
    temp = await hp.get_param_async("Temp. Aussen")
    print(temp)
    # ...
finally:
    await hp.logout_async()  # try to logout for an ordinary cancellation (if␣
↪possible)
    hp.close_connection()
```

A full list of supported functions can be found in the `htheatpump` documentation at readthedocs.io.

There are also some sample scripts that are part of the `htheatpump` package and can be run immediately after installation, e.g.:

```
$ htquery --device /dev/ttyUSB1 "Temp. Aussen" "Stoerung"
Stoerung    : False
Temp. Aussen: 5.0
```

1.3.1 Logging

This library uses the `logging` module. To set up logging to standard output, put

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

at the beginning of your script.

1.4 Disclaimer

Warning: Please note that any incorrect or careless usage of this module as well as errors in the implementation can damage your heat pump!

Therefore, the author does not provide any guarantee or warranty concerning to correctness, functionality or performance and does not accept any liability for damage caused by this module, examples or mentioned information.

Thus, use it on your own risk!

1.5 Contributing

Contributions are always welcome. Please review the contribution guidelines to get started. You can also help by reporting bugs.

1.6 Wanna support me?

1.7 Credits

- Created with Cookiecutter and the audreyr/cookiecutter-pypackage project template.
- Project dependencies scanned by PyUp.io.

1.8 License

Distributed under the terms of the GNU General Public License v3.

CHAPTER 2

Installation

For both installation methods described in the following sections the usage of Python virtual environments[1] is highly recommended.

2.1 Stable release

To install or upgrade `htheatpump`, run this command in your terminal:

```
$ pip install htheatpump --upgrade
```

This is the preferred method to install `htheatpump`, 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 `htheatpump` can be downloaded from the Github repo.

You can either clone the public repository:

```
$ git clone https://github.com/dstrigl/htheatpump.git
```

Or download the tarball:

```
$ curl -OL https://github.com/dstrigl/htheatpump/tarball/master
```

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

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

[1] If you need more information about Python virtual environments take a look at this article on RealPython.

CHAPTER 3

Usage

The following example shows how to query for a specific parameter (e.g. “Temp. Aussen”) of the heat pump.

An overview about the available parameters can be found here: *Heliotherm heat pump parameters*

```
from htheatpump import HtHeatpump

hp = HtHeatpump("/dev/ttyUSB0", baudrate=9600)
try:
    hp.open_connection()
    hp.login()
    # query for the outdoor temperature
    temp = hp.get_param("Temp. Aussen")
    print(temp)
    # ...
finally:
    hp.logout()  # try to logout for an ordinary cancellation (if possible)
    hp.close_connection()
```

```
from htheatpump import AioHtHeatpump

hp = AioHtHeatpump("/dev/ttyUSB0", baudrate=9600)
try:
    hp.open_connection()
    await hp.login_async()
    # query for the outdoor temperature
    temp = await hp.get_param_async("Temp. Aussen")
    print(temp)
    # ...
finally:
    await hp.logout_async()  # try to logout for an ordinary cancellation (if␣
↪possible)
    hp.close_connection()
```

Some more examples showing how to use the `htheatpump` module can be found in the *Sample scripts*.

CHAPTER 4

htheatpump package

4.1 htheatpump.htheatpump

This module is responsible for the communication with the Heliotherm heat pump.

class htheatpump.htheatpump.**VerifyAction**
Possible actions for the parameter verification:

- `NAME` Verification of the parameter name.
- `MIN` Verification of the minimal value of the parameter.
- `MAX` Verification of the maximal value of the parameter.
- `VALUE` Verification of the current parameter value.

The above enum entries can be used to specify the steps which should be performed during a parameter verification, e.g.:

```
hp = HtHeatpump("...", verify_param_action = {VerifyAction.NAME, VerifyAction.MAX}
↪)
temp = hp.get_param("Temp. Aussen")
...
```

or:

```
hp = HtHeatpump("/dev/ttyUSB0", baudrate=9600)
hp.verify_param_action = {VerifyAction.NAME, VerifyAction.MAX}
temp = hp.get_param("Temp. Aussen")
...
```

exception htheatpump.htheatpump.**VerificationException**(*message: str*)
Exception which represents a verification error during parameter access.

Parameters **message** (*str*) – A detailed message describing the parameter verification failure.

class htheatpump.htheatpump.**HtHeatpump**(*device: str*, *baudrate: int = 115200*, *bytesize: int = 8*, *parity: str = 'N'*, *stopbits: Union[float, int] = 1*, *timeout: Union[float, int, None] = 5*, *xonxoff: bool = True*, *rtscts: bool = False*, *write_timeout: Union[float, int, None] = None*, *dsrdtr: bool = False*, *inter_byte_timeout: Union[float, int, None] = None*, *exclusive: Optional[bool] = None*, *verify_param_action: Optional[Set[VerifyAction]] = None*, *verify_param_error: bool = False*)

Object which encapsulates the communication with the Heliotherm heat pump.

Parameters

- **device** (*str*) – The serial device to attach to (e.g. `/dev/ttyUSB0`).
- **baudrate** (*int*) – The baud rate to use for the serial device.
- **bytesize** (*int*) – The bytesize of the serial messages.
- **parity** (*str*) – Which kind of parity to use.
- **stopbits** (*float or int*) – The number of stop bits to use.
- **timeout** (*None, float or int*) – The read timeout value. Default is *DEFAULT_SERIAL_TIMEOUT*.
- **xonxoff** (*bool*) – Software flow control enabled.
- **rtscts** (*bool*) – Hardware flow control (RTS/CTS) enabled.
- **write_timeout** (*None, float or int*) – The write timeout value.
- **dsrdtr** (*bool*) – Hardware flow control (DSR/DTR) enabled.
- **inter_byte_timeout** (*None, float or int*) – Inter-character timeout, `None` to disable (default).
- **exclusive** (*bool*) – Exclusive access mode enabled (POSIX only).
- **verify_param_action** (*None or set*) – Parameter verification actions.
- **verify_param_error** (*bool*) – Interpretation of parameter verification failure as error enabled.

Example:

```
hp = HtHeatpump("/dev/ttyUSB0", baudrate=9600)
try:
    hp.open_connection()
    hp.login()
    # query for the outdoor temperature
    temp = hp.get_param("Temp. Aussen")
    print(temp)
    # ...
finally:
    hp.logout()  # try to logout for an ordinary cancellation (if possible)
    hp.close_connection()
```

DEFAULT_LOGIN_RETRIES = 2

Maximum number of retries for a login attempt; 1 regular try + *DEFAULT_LOGIN_RETRIES* retries.

DEFAULT_SERIAL_TIMEOUT = 5

Serial timeout value in seconds; normally no need to change it.

close_connection() → None
Close the serial connection.

fast_query(**args*) → Dict[str, Union[bool, int, float]]
Query for the current values of parameters from the heat pump the fast way.

Note: Only available for parameters representing a "MP" data point and no parameter verification possible!

Parameters **args** (*str*) – The parameter name(s) to request from the heat pump. If not specified all "known" parameters representing a "MP" data point are requested.

Returns

A dict of the requested parameters with their values, e.g.:

```
{ "EQ Pumpe (Ventilator)": False,
  "FWS Stroemungsschalter": False,
  "Frischwasserpumpe": 0,
  "HKR_Sollwert": 26.8,
  # ...
  }
```

Return type dict

Raises

- **KeyError** – Will be raised when the parameter definition for a passed parameter is not found.
- **ValueError** – Will be raised when a passed parameter doesn't represent a "MP" data point.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_date_time() → Tuple[datetime.datetime, int]
Read the current date and time of the heat pump.

Returns

The current date and time of the heat pump as a tuple with 2 elements, where the first element is of type datetime.datetime which represents the current date and time while the second element is the corresponding weekday in form of an int between 1 and 7, inclusive (Monday through Sunday). For example:

```
( datetime.datetime(...), 2 )  # 2 = Tuesday
```

Return type tuple (datetime.datetime, int)

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_fault_list(**args*) → List[Dict[str, object]]
Query for the fault list of the heat pump.

Parameters **args** (*int*) – The index number(s) to request from the fault list (optional). If not specified all fault list entries are requested.

Returns

The requested entries of the fault list as `list`, e.g.:

```
[ { "index"   : 29,                       # fault list index
    "error"   : 20,                       # error code
    "datetime": datetime.datetime(...), # date and time of the entry
    "message" : "EQ_Spreizung",           # error message
    },
  # ...
  ]
```

Return type `list`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_fault_list_size() → int
Query for the fault list size of the heat pump.

Returns The size of the fault list as `int`.

Return type `int`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_last_fault() → Tuple[int, int, datetime.datetime, str]
Query for the last fault message of the heat pump.

Returns

The last fault message of the heat pump as a tuple with 4 elements. The first element of the returned tuple represents the index as `int` of the message inside the fault list. The second element is (probably) the the error code as `int` defined by Heliotherm. The last two elements of the tuple are the date and time when the error occurred (as `datetime.datetime`) and the error message string itself. For example:

```
( 29, 20, datetime.datetime(...), "EQ_Spreizung" )
```

Return type `tuple` (int, int, datetime.datetime, str)

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_param(*name: str*) → Union[bool, int, float]
Query for a specific parameter of the heat pump.

Parameters **name** (*str*) – The parameter name, e.g. `"Betriebsart"`.

Returns Returned value of the requested parameter. The type of the returned value is defined by the csv-table of supported heat pump parameters in `htparams.csv`.

Return type `bool`, `int` or `float`

Raises

- **KeyError** – Will be raised when the parameter definition for the passed parameter is not found.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

- ***VerificationException*** – Will be raised if the parameter verification fails and the property *verify_param_error* is set to `True`. If property *verify_param_error* is set to `False` only a warning message will be emitted. The performed verification steps are defined by the property *verify_param_action*.

For example, the following call

```
temp = hp.get_param("Temp. Aussen")
```

will return the current measured outdoor temperature in °C.

get_serial_number() → int
Query for the manufacturer's serial number of the heat pump.

Returns The manufacturer's serial number of the heat pump as `int` (e.g. `123456`).

Return type `int`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_time_prog(*idx: int*, *with_entries: bool = True*) → htheatpump.httimeprog.TimeProgram
Return a specific time program (specified by their index) together with their time program entries (if desired) from the heat pump.

Parameters

- **idx** (*int*) – The time program index.
- **with_entries** (*bool*) – Determines whether also the single time program entries should be requested or not. Default is `True`.

Returns The requested time program as *TimeProgram*.

Return type `TimeProgram`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_time_prog_entry(*idx: int*, *day: int*, *num: int*) → htheatpump.httimeprog.TimeProgEntry
Return a specific time program entry (specified by time program index, day and entry-of-day) of the heat pump.

Parameters

- **idx** (*int*) – The time program index.
- **day** (*int*) – The day of the time program entry (inside the specified time program).
- **num** (*int*) – The number of the time program entry (of the specified day).

Returns The requested time program entry as *TimeProgEntry*.

Return type `TimeProgEntry`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_time_progs() → List[htheatpump.httimeprog.TimeProgram]
Return a list of all available time programs of the heat pump.

Returns A list of *TimeProgram* instances.

Return type `list`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_version() → Tuple[str, int]
Query for the software version of the heat pump.

Returns

The software version of the heat pump as a tuple with 2 elements. The first element inside the returned tuple represents the software version as a readable string in a common version number format (e.g. `"3.0.20"`). The second element (probably) contains a numerical representation as `int` of the software version returned by the heat pump. For example:

```
( "3.0.20", 2321 )
```

Return type `tuple` (str, int)

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

in_error
Query whether the heat pump is malfunctioning.

Returns `True` if the heat pump is malfunctioning, `False` otherwise.

Return type `bool`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

is_open
Return the state of the serial port, whether it's open or not.

Returns The state of the serial port as `bool`.

Return type `bool`

login(*update_param_limits: bool = False*, *max_retries: int = 2*) → None
Log in the heat pump. If *update_param_limits* is `True` an update of the parameter limits in *HtParams* will be performed. This will be done by requesting the current value together with their limits (MIN and MAX) for all "known" parameters directly after a successful login.

Parameters

- **update_param_limits** (*bool*) – Determines whether an update of the parameter limits in *HtParams* should be done or not. Default is `False`.
- **max_retries** (*int*) – Maximal number of retries for a successful login. One regular try plus `max_retries` retries. Default is *DEFAULT_LOGIN_RETRIES*.

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

logout() → None
Log out from the heat pump session.

open_connection() → None
Open the serial connection with the defined settings.

Raises

- **IOError** – When the serial connection is already open.
- **ValueError** – Will be raised when parameter are out of range, e.g. baudrate, bytesize.

- **SerialException** – In case the device can not be found or can not be configured.

query(**args*) → Dict[str, Union[bool, int, float]]
Query for the current values of parameters from the heat pump.

Parameters **args** (*str*) – The parameter name(s) to request from the heat pump. If not specified all "known" parameters are requested.

Returns

A dict of the requested parameters with their values, e.g.:

```
{ "HKR Soll_Raum": 21.0,
  "Stoerung": False,
  "Temp. Aussen": 8.8,
  # ...
  }
```

Return type dict

Raises

- **KeyError** – Will be raised when the parameter definition for a passed parameter is not found.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).
- ***VerificationException*** – Will be raised if the parameter verification fails and the property *verify_param_error* is set to True. If property *verify_param_error* is set to False only a warning message will be emitted. The performed verification steps are defined by the property *verify_param_action*.

read_response() → str
Read the response message from the heat pump.

Returns The returned response message of the heat pump as str.

Return type str

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid (or unknown) response (e.g. broken data stream, unknown header, invalid checksum, ...).

Note: There is a little bit strange behavior how the heat pump sometimes replies on some requests:

A response from the heat pump normally consists of the following header (the first 6 bytes) b"\x02\xfd\xe0\xd0\x00\x00" together with the payload and a computed checksum. But sometimes the heat pump replies with a different header (b"\x02\xfd\xe0\xd0\x04\x00" or b"\x02\xfd\xe0\xd0\x08\x00") together with the payload and a *fixed* value of 0x0 for the checksum (regardless of the content).

We have no idea about the reason for this behavior. But after analysing the communication between the Heliotherm home control Windows application and the heat pump, which simply accepts this kind of responses, we also decided to handle it as a valid answer to a request.

Furthermore, we have noticed another behavior, which is not fully explainable: For some response messages from the heat pump (e.g. for the error message "ERR,INVALID IDX") the transmitted payload length in the protocol is zero (0 bytes), although some payload follows. In this case we read until we will found the trailing b"\r\n" at the end of the payload to determine the payload of the message.

Additionally to the upper described facts, for some of the answers of the heat pump the payload length must be corrected (for the checksum computation) so that the received checksum fits with the computed one (e.g. for `b"\x02\xfd\xe0\xd0\x01\x00"` and `b"\x02\xfd\xe0\xd0\x02\x00"`).

---

reconnect() → None
Perform a reconnect of the serial connection. Flush the output and input buffer, close the serial connection and open it again.

send_request(*cmd: str*) → None
Send a request to the heat pump.

Parameters **cmd** (*str*) – Command to send to the heat pump.

Raises **IOError** – Will be raised when the serial connection is not open.

set_date_time(*dt: Optional[datetime.datetime] = None*) → Tuple[datetime.datetime, int]
Set the current date and time of the heat pump.

Parameters **dt** (*datetime.datetime*) – The date and time to set. If `None` current date and time of the host will be used.

Returns A 2-elements tuple composed of a `datetime.datetime` which represents the sent date and time and an `int` between 1 and 7, inclusive, for the corresponding weekday (Monday through Sunday).

Return type `tuple` (datetime.datetime, int)

Raises

- **TypeError** – Raised for an invalid type of argument `dt`. Must be `None` or of type `datetime.datetime`.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

set_param(*name: str, val: Union[bool, int, float], ignore_limits: bool = False*) → Union[bool, int, float]
Set the value of a specific parameter of the heat pump. If `ignore_limits` is `False` and the passed value is beyond the parameter limits a `ValueError` will be raised.

Parameters

- **name** (*str*) – The parameter name, e.g. `"Betriebsart"`.
- **val** (*bool, int or float*) – The value to set.
- **ignore_limits** (*bool*) – Indicates if the parameter limits should be ignored or not.

Returns Returned value of the parameter set request. In case of success this value should be the same as the one passed to the function. The type of the returned value is defined by the csv-table of supported heat pump parameters in `htparams.csv`.

Return type `bool`, `int` or `float`

Raises

- **KeyError** – Will be raised when the parameter definition for the passed parameter is not found.
- **ValueError** – Will be raised if the passed value is beyond the parameter limits and argument `ignore_limits` is set to `False`.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

- ***VerificationException*** – Will be raised if the parameter verification fails and the property *verify_param_error* is set to `True`. If property *verify_param_error* is set to `False` only a warning message will be emitted. The performed verification steps are defined by the property *verify_param_action*.

For example, the following call

```
hp.set_param("HKR Soll_Raum", 21.5)
```

will set the desired room temperature of the heating circuit to 21.5 °C.

set_time_prog(*time_prog: htheatpump.httimeprog.TimeProgram*) → htheatpump.httimeprog.TimeProgram

Set all time program entries of a specific time program. Any non-specified entry (which is `None`) in the time program will be requested from the heat pump. The returned *TimeProgram* instance includes therefore all entries of this time program.

Parameters **time_prog** (`TimeProgram`) – The given time program as *TimeProgram*.

Returns The time program as *TimeProgram* including all time program entries.

Return type `TimeProgram`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

set_time_prog_entry(*idx: int*, *day: int*, *num: int*, *entry: htheatpump.httimeprog.TimeProgEntry*) → htheatpump.httimeprog.TimeProgEntry

Set a specific time program entry (specified by time program index, day and entry-of-day) of the heat pump.

Parameters

- **idx** (*int*) – The time program index.
- **day** (*int*) – The day of the time program entry (inside the specified time program).
- **num** (*int*) – The number of the time program entry (of the specified day).
- **entry** (`TimeProgEntry`) – The new time program entry as *TimeProgEntry*.

Returns The changed time program entry *TimeProgEntry*.

Return type `TimeProgEntry`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

update_param_limits() → List[str]

Perform an update of the parameter limits in *HtParams* by requesting the limit values of all "known" parameters directly from the heat pump.

Returns The list of updated (changed) parameters.

Return type `list`

Raises ***VerificationException*** – Will be raised if the parameter verification fails and the property *verify_param_error* is set to `True`. If property *verify_param_error* is set to `False` only a warning message will be emitted. The performed verification steps are defined by the property *verify_param_action*.

verify_param_action

Property to specify the actions which should be performed during the parameter verification.

The possible actions for the parameter verification can be found in the enum *VerifyAction*. The default includes just a verification of the parameter name.

Param A set of *VerifyAction* enum values, which specify the actions which should be performed during the parameter verification, e.g. `{VerifyAction.NAME}`.

Returns The set of *VerifyAction* enum values, which specify the parameter verification actions.

Return type `set`

verify_param_error

Property to get or set whether a parameter verification failure should result in an error or not.

If `True` a failed parameter verification will result in an *VerificationException* exception. If `False` (default) only a warning message will be emitted.

Param Boolean value which indicates whether a parameter verification failure should result in an error or not.

Returns `True` if a verification failure should result in an error, `False` otherwise.

Return type `bool`

4.2 htheatpump.aiohtheatpump

This module provides an asynchronous communication with the Heliotherm heat pump.

class htheatpump.aiohtheatpump.**AioHtHeatpump**(*device: str*, *baudrate: int = 115200*, *bytesize: int = 8*, *parity: str = 'N'*, *stopbits: Union[float, int] = 1*, *timeout: Union[float, int, None] = 5*, *xonxoff: bool = True*, *rtscts: bool = False*, *write_timeout: Union[float, int, None] = None*, *dsrdtr: bool = False*, *inter_byte_timeout: Union[float, int, None] = None*, *exclusive: Optional[bool] = None*, *verify_param_action: Optional[Set[VerifyAction]] = None*, *verify_param_error: bool = False*, *loop: Optional[asyncio.events.AbstractEventLoop] = None*, *cancel_read_timeout: int = 1*, *cancel_write_timeout: int = 1*)

Object which encapsulates the asynchronous communication with the Heliotherm heat pump.

Parameters

- **device** (*str*) – The serial device to attach to (e.g. `/dev/ttyUSB0`).
- **baudrate** (*int*) – The baud rate to use for the serial device.
- **bytesize** (*int*) – The bytesize of the serial messages.
- **parity** (*str*) – Which kind of parity to use.
- **stopbits** (*float or int*) – The number of stop bits to use.
- **timeout** (*None, float or int*) – The read timeout value. Default is *DEFAULT_SERIAL_TIMEOUT*.
- **xonxoff** (*bool*) – Software flow control enabled.

- **rtscts** (*bool*) – Hardware flow control (RTS/CTS) enabled.
- **write_timeout** (*None, float or int*) – The write timeout value.
- **dsrdtr** (*bool*) – Hardware flow control (DSR/DTR) enabled.
- **inter_byte_timeout** (*None, float or int*) – Inter-character timeout, None to disable (default).
- **exclusive** (*bool*) – Exclusive access mode enabled (POSIX only).
- **verify_param_action** (*None or set*) – Parameter verification actions.
- **verify_param_error** (*bool*) – Interpretation of parameter verification failure as error enabled.
- **loop** (*None or asyncio.AbstractEventLoop*) – The event loop, None for the currently running event loop (default).
- **cancel_read_timeout** (*int*) – TODO
- **cancel_write_timeout** (*int*) – TODO

Example:

```
hp = AioHtHeatpump("/dev/ttyUSB0", baudrate=9600)
try:
    hp.open_connection()
    await hp.login_async()
    # query for the outdoor temperature
    temp = await hp.get_param_async("Temp. Aussen")
    print(temp)
    # ...
finally:
    await hp.logout_async()  # try to logout for an ordinary cancellation (if␣
↪possible)
    hp.close_connection()
```

fast_query_async(**args*) → Dict[str, Union[bool, int, float]]

Query for the current values of parameters from the heat pump the fast way.

Note: Only available for parameters representing a "MP" data point and no parameter verification possible!

Parameters **args** (*str*) – The parameter name(s) to request from the heat pump. If not specified all "known" parameters representing a "MP" data point are requested.

Returns

A dict of the requested parameters with their values, e.g.:

```
{ "EQ Pumpe (Ventilator)": False,
  "FWS Stroemungsschalter": False,
  "Frischwasserpumpe": 0,
  "HKR_Sollwert": 26.8,
  # ...
  }
```

Return type dict

Raises

- **KeyError** – Will be raised when the parameter definition for a passed parameter is not found.
- **ValueError** – Will be raised when a passed parameter doesn't represent a "MP" data point.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_date_time_async() → Tuple[datetime.datetime, int]
Read the current date and time of the heat pump.

Returns

The current date and time of the heat pump as a tuple with 2 elements, where the first element is of type `datetime.datetime` which represents the current date and time while the second element is the corresponding weekday in form of an `int` between 1 and 7, inclusive (Monday through Sunday). For example:

```
( datetime.datetime(...), 2 )   # 2 = Tuesday
```

Return type `tuple` (datetime.datetime, int)

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_fault_list_async(**args*) → List[Dict[str, object]]
Query for the fault list of the heat pump.

Parameters **args** (*int*) – The index number(s) to request from the fault list (optional). If not specified all fault list entries are requested.

Returns

The requested entries of the fault list as `list`, e.g.:

```
[ { "index"   : 29,                       # fault list index
    "error"   : 20,                       # error code
    "datetime": datetime.datetime(...), # date and time of the entry
    "message" : "EQ_Spreizung",           # error message
    },
  # ...
  ]
```

Return type `list`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_fault_list_size_async() → int
Query for the fault list size of the heat pump.

Returns The size of the fault list as `int`.

Return type `int`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_last_fault_async() → Tuple[int, int, datetime.datetime, str]
Query for the last fault message of the heat pump.

Returns

The last fault message of the heat pump as a tuple with 4 elements. The first element of the returned tuple represents the index as `int` of the message inside the fault list. The second element is (probably) the the error code as `int` defined by Heliotherm. The last two elements of the tuple are the date and time when the error occurred (as `datetime.datetime`) and the error message string itself. For example:

```
( 29, 20, datetime.datetime(...), "EQ_Spreizung" )
```

Return type `tuple` (int, int, datetime.datetime, str)

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_param_async(*name: str*) → Union[bool, int, float]

Query for a specific parameter of the heat pump.

Parameters **name** (*str*) – The parameter name, e.g. `"Betriebsart"`.

Returns Returned value of the requested parameter. The type of the returned value is defined by the csv-table of supported heat pump parameters in `htparams.csv`.

Return type `bool`, `int` or `float`

Raises

- **KeyError** – Will be raised when the parameter definition for the passed parameter is not found.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).
- ***VerificationException*** – Will be raised if the parameter verification fails and the property `verify_param_error` is set to `True`. If property `verify_param_error` is set to `False` only a warning message will be emitted. The performed verification steps are defined by the property `verify_param_action`.

For example, the following call

```
temp = await hp.get_param_async("Temp. Aussen")
```

will return the current measured outdoor temperature in °C.

get_serial_number_async() → int

Query for the manufacturer's serial number of the heat pump.

Returns The manufacturer's serial number of the heat pump as `int` (e.g. `123456`).

Return type `int`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_time_prog_async(*idx: int*, *with_entries: bool = True*) → htheatpump.httimeprog.TimeProgram

Return a specific time program (specified by their index) together with their time program entries (if desired) from the heat pump.

Parameters

- **idx** (*int*) – The time program index.

- **with_entries** (*bool*) – Determines whether also the single time program entries should be requested or not. Default is `True`.

Returns The requested time program as *TimeProgram*.

Return type `TimeProgram`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_time_prog_entry_async(*idx: int*, *day: int*, *num: int*) → htheatpump.httimeprog.TimeProgEntry

Return a specific time program entry (specified by time program index, day and entry-of-day) of the heat pump.

Parameters

- **idx** (*int*) – The time program index.
- **day** (*int*) – The day of the time program entry (inside the specified time program).
- **num** (*int*) – The number of the time program entry (of the specified day).

Returns The requested time program entry as *TimeProgEntry*.

Return type `TimeProgEntry`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_time_progs_async() → List[htheatpump.httimeprog.TimeProgram]

Return a list of all available time programs of the heat pump.

Returns A list of *TimeProgram* instances.

Return type `list`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

get_version_async() → Tuple[str, int]

Query for the software version of the heat pump.

Returns

The software version of the heat pump as a tuple with 2 elements. The first element inside the returned tuple represents the software version as a readable string in a common version number format (e.g. `"3.0.20"`). The second element (probably) contains a numerical representation as `int` of the software version returned by the heat pump. For example:

```
( "3.0.20", 2321 )
```

Return type `tuple` (str, int)

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

in_error_async

Query whether the heat pump is malfunctioning.

Returns `True` if the heat pump is malfunctioning, `False` otherwise.

Return type `bool`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

login_async(*update_param_limits: bool = False*, *max_retries: int = 2*) → None
Log in the heat pump. If `update_param_limits` is `True` an update of the parameter limits in *HtParams* will be performed. This will be done by requesting the current value together with their limits (MIN and MAX) for all "known" parameters directly after a successful login.

Parameters

- **update_param_limits** (*bool*) – Determines whether an update of the parameter limits in *HtParams* should be done or not. Default is `False`.
- **max_retries** (*int*) – Maximal number of retries for a successful login. One regular try plus `max_retries` retries. Default is *DEFAULT_LOGIN_RETRIES*.

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

logout_async() → None
Log out from the heat pump session.

open_connection() → None
Open the serial connection with the defined settings.

Raises

- **IOError** – When the serial connection is already open.
- **ValueError** – Will be raised when parameter are out of range, e.g. baudrate, bytesize.
- **SerialException** – In case the device can not be found or can not be configured.

query_async(**args*) → Dict[str, Union[bool, int, float]]
Query for the current values of parameters from the heat pump.

Parameters **args** (*str*) – The parameter name(s) to request from the heat pump. If not specified all "known" parameters are requested.

Returns

A dict of the requested parameters with their values, e.g.:

```
{ "HKR Soll_Raum": 21.0,
  "Stoerung": False,
  "Temp. Aussen": 8.8,
  # ...
  }
```

Return type `dict`

Raises

- **KeyError** – Will be raised when the parameter definition for a passed parameter is not found.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).
- ***VerificationException*** – Will be raised if the parameter verification fails and the property `verify_param_error` is set to `True`. If property `verify_param_error` is set to `False` only a warning message will be emitted. The performed verification steps are defined by the property `verify_param_action`.

read_response_async() → str
Read the response message from the heat pump.

Returns The returned response message of the heat pump as `str`.

Return type str

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid (or unknown) response (e.g. broken data stream, unknown header, invalid checksum, ...).

Note: There is a little bit strange behavior how the heat pump sometimes replies on some requests:

A response from the heat pump normally consists of the following header (the first 6 bytes) b"\x02\xfd\xe0\xd0\x00\x00" together with the payload and a computed checksum. But sometimes the heat pump replies with a different header (b"\x02\xfd\xe0\xd0\x04\x00" or b"\x02\xfd\xe0\xd0\x08\x00") together with the payload and a *fixed* value of 0x0 for the checksum (regardless of the content).

We have no idea about the reason for this behavior. But after analysing the communication between the Heliotherm home control Windows application and the heat pump, which simply accepts this kind of responses, we also decided to handle it as a valid answer to a request.

Furthermore, we have noticed another behavior, which is not fully explainable: For some response messages from the heat pump (e.g. for the error message "ERR,INVALID IDX") the transmitted payload length in the protocol is zero (0 bytes), although some payload follows. In this case we read until we will found the trailing b"\r\n" at the end of the payload to determine the payload of the message.

Additionally to the upper described facts, for some of the answers of the heat pump the payload length must be corrected (for the checksum computation) so that the received checksum fits with the computed one (e.g. for b"\x02\xfd\xe0\xd0\x01\x00" and b"\x02\xfd\xe0\xd0\x02\x00").

---

send_request_async(*cmd: str*) → None

Send a request to the heat pump.

Parameters **cmd** (*str*) – Command to send to the heat pump.

Raises **IOError** – Will be raised when the serial connection is not open.

set_date_time_async(*dt: Optional[datetime.datetime] = None*) → Tuple[datetime.datetime, int]

Set the current date and time of the heat pump.

Parameters **dt** (*datetime.datetime*) – The date and time to set. If None current date and time of the host will be used.

Returns A 2-elements tuple composed of a datetime.datetime which represents the sent date and time and an int between 1 and 7, inclusive, for the corresponding weekday (Monday through Sunday).

Return type tuple (datetime.datetime, int)

Raises

- **TypeError** – Raised for an invalid type of argument dt. Must be None or of type datetime.datetime.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

set_param_async(*name: str, val: Union[bool, int, float], ignore_limits: bool = False*) → Union[bool, int, float]

Set the value of a specific parameter of the heat pump. If ignore_limits is False and the passed value is beyond the parameter limits a ValueError will be raised.

Parameters

- **name** (*str*) – The parameter name, e.g. "Betriebsart".

- **val** (*bool, int or float*) – The value to set.
- **ignore_limits** (*bool*) – Indicates if the parameter limits should be ignored or not.

Returns Returned value of the parameter set request. In case of success this value should be the same as the one passed to the function. The type of the returned value is defined by the csv-table of supported heat pump parameters in `htparams.csv`.

Return type `bool`, `int` or `float`

Raises

- **KeyError** – Will be raised when the parameter definition for the passed parameter is not found.
- **ValueError** – Will be raised if the passed value is beyond the parameter limits and argument `ignore_limits` is set to `False`.
- **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).
- ***VerificationException*** – Will be raised if the parameter verification fails and the property `verify_param_error` is set to `True`. If property `verify_param_error` is set to `False` only a warning message will be emitted. The performed verification steps are defined by the property `verify_param_action`.

For example, the following call

```
await hp.set_param_async("HKR Soll_Raum", 21.5)
```

will set the desired room temperature of the heating circuit to 21.5 °C.

set_time_prog_async(*time_prog: htheatpump.httimeprog.TimeProgram*) → htheatpump.httimeprog.TimeProgram

Set all time program entries of a specific time program. Any non-specified entry (which is `None`) in the time program will be requested from the heat pump. The returned *TimeProgram* instance includes therefore all entries of this time program.

Parameters **time_prog** (`TimeProgram`) – The given time program as *TimeProgram*.

Returns The time program as *TimeProgram* including all time program entries.

Return type `TimeProgram`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

set_time_prog_entry_async(*idx: int, day: int, num: int, entry: htheatpump.httimeprog.TimeProgEntry*) → htheatpump.httimeprog.TimeProgEntry

Set a specific time program entry (specified by time program index, day and entry-of-day) of the heat pump.

Parameters

- **idx** (*int*) – The time program index.
- **day** (*int*) – The day of the time program entry (inside the specified time program).
- **num** (*int*) – The number of the time program entry (of the specified day).
- **entry** (`TimeProgEntry`) – The new time program entry as *TimeProgEntry*.

Returns The changed time program entry *TimeProgEntry*.

Return type `TimeProgEntry`

Raises **IOError** – Will be raised when the serial connection is not open or received an incomplete/invalid response (e.g. broken data stream, invalid checksum).

update_param_limits_async() → List[str]
Perform an update of the parameter limits in *HtParams* by requesting the limit values of all "known" parameters directly from the heat pump.

Returns The list of updated (changed) parameters.

Return type list

Raises ***VerificationException*** – Will be raised if the parameter verification fails and the property verify_param_error is set to True. If property verify_param_error is set to False only a warning message will be emitted. The performed verification steps are defined by the property verify_param_action.

4.3 htheatpump.httimeprog

Classes representing the time programs of the Heliotherm heat pump.

class htheatpump.httimeprog.**TimeProgPeriod**(*start_hour: int*, *start_minute: int*, *end_hour: int*, *end_minute: int*)
Representation of a time program period defined by start- and end-time (HH:MM).

Parameters

- **start_hour** (*int*) – The hour value of the start-time (HH).
- **start_minute** (*int*) – The minute value of the start-time (MM).
- **end_hour** (*int*) – The hour value of the end-time (HH).
- **end_minute** (*int*) – The minute value of the end-time (MM).

Raises **ValueError** – Will be raised for any invalid argument.

as_dict() → Dict[str, object]
Create a dict representation of this time program period.

Returns A dict representing this time program period.

Return type dict

as_json() → Dict[str, object]
Create a json-readable dict representation of this time program period.

Returns A json-readable dict representing this time program period.

Return type dict

end
Return the end-time of this time program period as a tuple with 2 elements, where the first element represents the hours and the second one the minutes.

Returns

The end-time of this time program period as tuple. For example:

```
( 16, 45 )   # -> 16:45
```

Return type tuple (int, int)

end_hour
Return the hour value of the end-time of this time program period.

Returns The hour value of the end-time of this time program period.

Return type int

end_minute
Return the minute value of the end-time of this time program period.

Returns The minute value of the end-time of this time program period.

Return type int

end_str
Return the end-time of this time program period as str.

Returns

The end-time of this time program period as str. For example:

```
'16:45'
```

Return type str

classmethod from_json (*json_dict: Dict[str, str]*) → TimeProgPeriodT
Create a *TimeProgPeriod* instance from a JSON representation.

Parameters **json_dict** (*dict*) – The JSON representation of the time program period as dict.

Return type TimeProgPeriod

Raises **ValueError** – Will be raised for any invalid argument.

classmethod from_str (*start_str: str*, *end_str: str*) → TimeProgPeriodT
Create a *TimeProgPeriod* instance from string representations of the start- and end-time.

Parameters

- **start_str** (*str*) – The start-time of the time program entry as str.
- **end_str** (*str*) – The end-time of the time program entry as str.

Returns A *TimeProgPeriod* instance with the given properties.

Return type TimeProgPeriod

Raises **ValueError** – Will be raised for any invalid argument.

set (*start_hour: int*, *start_minute: int*, *end_hour: int*, *end_minute: int*) → None
Set the start- and end-time of this time program period.

Parameters

- **start_hour** (*int*) – The hour value of the start-time.
- **start_minute** (*int*) – The minute value of the start-time.
- **end_hour** (*int*) – The hour value of the end-time.
- **end_minute** (*int*) – The minute value of the end-time.

Raises **ValueError** – Will be raised for any invalid argument.

start
Return the start-time of this time program period as a tuple with 2 elements, where the first element represents the hours and the second one the minutes.

Returns

The start-time of this time program period as tuple. For example:

```
( 11, 0 )  # -> 11:00
```

Return type tuple (int, int)

start_hour
Return the hour value of the start-time of this time program period.

Returns The hour value of the start-time of this time program period.

Return type int

start_minute
Return the minute value of the start-time of this time program period.

Returns The minute value of the start-time of this time program period.

Return type int

start_str
Return the start-time of this time program period as str.

Returns

The start-time of this time program period as str. For example:

```
'11:00'
```

Return type str

class htheatpump.httimeprog.**TimeProgEntry**(*state: int*, *period: htheatpump.httimeprog.TimeProgPeriod*)
Representation of a single time program entry.

Parameters

- **state** (*int*) – The state of the time program entry.
- **period** (TimeProgPeriod) – The period of the time program entry.

as_dict() → Dict[str, object]
Create a dict representation of this time program entry.

Returns A dict representing this time program entry.

Return type dict

as_json() → Dict[str, object]
Create a json-readable dict representation of this time program entry.

Returns A json-readable dict representing this time program entry.

Return type dict

classmethod from_json(*json_dict: Dict[str, Any]*) → TimeProgEntryT
Create a *TimeProgEntry* instance from a JSON representation.

Parameters **json_dict** (*dict*) – The JSON representation of the time program entry as dict.

Return type TimeProgEntry

Raises **ValueError** – Will be raised for any invalid argument.

classmethod from_str(*state: str*, *start_str: str*, *end_str: str*) → TimeProgEntryT
Create a *TimeProgEntry* instance from string representations of the state, start- and end-time.

Parameters

- **state** (*str*) – The state of the time program entry as `str`.
- **start_str** (*str*) – The start-time of the time program entry as `str`.
- **end_str** (*str*) – The end-time of the time program entry as `str`.

Returns A *TimeProgEntry* instance with the given properties.

Return type `TimeProgEntry`

period
Property to get or set the period of this time program entry.

Param The new period of the time program entry as *TimeProgPeriod*.

Returns A copy of the current period of the time program entry as *TimeProgPeriod*.

Return type `TimeProgPeriod`

set(*state: int*, *period: htheatpump.httimeprog.TimeProgPeriod*) → None
Set the state and period of this time program entry.

Parameters

- **state** (*int*) – The state of the time program entry.
- **period** (`TimeProgPeriod`) – The period of the time program entry.

state
Property to get or set the state of this time program entry.

Param The new state of the time program entry.

Returns The current state of the time program entry.

Return type `int`

class htheatpump.httimeprog.**TimeProgram**(*idx: int*, *name: str*, *ead: int*, *nos: int*, *ste: int*, *nod: int*)
Representation of a time program of the Heliotherm heat pump.

Parameters

- **idx** (*int*) – The time program index.
- **name** (*str*) – The name of the time program (e.g. “Warmwasser”).
- **ead** (*int*) – The number of entries a day of the time program.
- **nos** (*int*) – The number of states of the time program.
- **ste** (*int*) – The step-size (in minutes) of the start- and end-times of the time program entries.
- **nod** (*int*) – The number of days of the time program.

as_dict(*with_entries: bool = True*) → Dict[str, object]
Create a dict representation of this time program.

Parameters **with_entries** (*bool*) – Determines whether the single time program entries should be included or not. Default is `True`.

Returns A dict representing this time program.

Return type dict

as_json (*with_entries: bool = True*) → Dict[str, object]
Create a json-readable dict representation of this time program.

Parameters **with_entries** (*bool*) – Determines whether the single time program entries should be included or not. Default is True.

Returns A json-readable dict representing this time program.

Return type dict

entries_a_day
Return the number of entries a day of this time program.

Returns The number of entries a day of this time program.

Return type int

entries_of_day (*day: int*) → List[Optional[htheatpump.httimeprog.TimeProgEntry]]
Return a list of copies of time program entries of a specific day.

Parameters **day** (*int*) – The day of the time program entries.

Returns A list of *TimeProgEntry* instances or None if not set.

Return type list (TimeProgEntry)

entry (*day: int*, *num: int*) → Optional[htheatpump.httimeprog.TimeProgEntry]
Return a copy of a specific time program entry.

Parameters

- **day** (*int*) – The day of the time program entry.
- **num** (*int*) – The number of the time program entry.

Returns The time program entry as instance of *TimeProgEntry* or None if not set.

Return type TimeProgEntry

classmethod from_json (*json_dict: Dict[str, Any]*) → TimeProgramT
Create a *TimeProgram* instance from a JSON representation.

Parameters **json_dict** (*dict*) – The JSON representation of the time program as dict.

Return type TimeProgram

Raises **ValueError** – Will be raised for any invalid argument.

index
Return the index of this time program.

Returns The index of this time program.

Return type int

name
Return the name of this time program.

Returns The name of this time program.

Return type int

number_of_days
Return the number of days of this time program.

Returns The number of days of this time program.

Return type int

number_of_states
Return the number of states of this time program.

Returns The number of states of this time program.

Return type int

set_entry(*day: int*, *num: int*, *entry: htheatpump.httimeprog.TimeProgEntry*) → None
Set the properties of a given time program entry of the heat pump.

Parameters

- **day** (*int*) – The day of the time program entry.
- **num** (*int*) – The number of the time program entry.
- **entry** (TimeProgEntry) – The time program entry itself.

Raises **ValueError** – Will be raised if any property of the given entry is out of the specification of this time program.

step_size
Return the step-size (in minutes) of the start- and end-times of this time program entries.

Returns The step-size (in minutes) of the start- and end-times of this time program entries.

Return type int

4.4 htheatpump.htparams

Definition of the Heliotherm heat pump parameters together with their

- data point type ("MP", "SP"),
- data point number,
- access rights,
- data type,
- minimal value and
- maximal value.

class htheatpump.htparams.**HtDataTypes**
Supported data types of the Heliotherm heat pump:

- BOOL The value of the parameter is given as **boolean** (e.g. on/off, yes/no, enabled/disabled).
- INT The value of the parameter is given as **integer**.
- FLOAT The value of the parameter is given as **floating point number**.

class htheatpump.htparams.**HtParam**(*dp_type: str*, *dp_number: int*, *acl: str*, *data_type: htheatpump.htparams.HtDataTypes*, *min_val: Union[bool, int, float, None] = None*, *max_val: Union[bool, int, float, None] = None*)
Representation of a specific heat pump parameter.

Parameters

- **dp_type** (*str*) – The data point type ("MP", "SP").

- **dp_number** (*int*) – The data point number.
- **acl** (*str*) – The access rights (`'r'` = read, `'w'` = write).
- **data_type** (`HtDataTypes`) – The data type, see *HtDataTypes*.
- **min_val** (*bool, int, float or None*) – The minimal value (default `None`, which means "doesn't matter").
- **max_val** (*bool, int, float or None*) – The maximal value (default `None`, which means "doesn't matter").

Raises **TypeError** – Will be raised if the passed minimal or maximal value has an invalid type.

check_value_type(*arg: Union[bool, int, float, htheatpump.htparams.HtDataTypes]*) → None

Check the type of the passed value against the given parameter data type.

This method can be called as a *static method*, e.g.:

```
s = HtParam.check_value_type(123, HtDataTypes.FLOAT)
```

or as a *member method* of *HtParam*, e.g.:

```
param = HtParams["Temp. Aussen"]
s = param.check_value_type(3.2)
```

If the method is called as a member method of *HtParam*, the data type of the passed value don't have to be specified. It will be automatically determined from the *HtParam* instance.

Raises **TypeError** – Will be raised if the passed value has an invalid type.

cmd() → str

Return the command string, based on the data point type and number of the parameter.

Returns The command string.

Return type `str`

from_str(*arg: Union[str, htheatpump.htparams.HtDataTypes], strict: bool = True*) → Union[bool, int, float]

Convert the passed value (in form of a string) to the expected data type.

This method can be called as a *static method*, e.g.:

```
val = HtParam.from_str("123", HtDataTypes.INT)
```

or as a *member method* of *HtParam*, e.g.:

```
param = HtParams["Temp. Aussen"]
val = param.from_str(s, strict=False)
```

If the method is called as a member method of *HtParam*, the expected data type don't have to be specified. It will be automatically determined from the *HtParam* instance.

Parameters **strict** – Determines whether the conversion to `float` should be strict (if `False` also integers are accepted, e.g. `'328'`).

Returns The passed value which data type matches the expected one.

Return type `bool`, `int` or `float`

Raises

- **TypeError** – Will be raised if the passed value has an invalid type.

- **ValueError** – Will be raised if the passed value could not be converted to the expected data type.

in_limits(*val: Union[bool, int, float, None]*) → bool
Determine whether the passed value is in between the parameter limits or not.

Parameters **val** (*bool, int or float*) – The value to check against the parameter limits.

Returns `True` if the passed value is in between the limits, `False` otherwise.

Return type `bool`

Raises **TypeError** – Will be raised if the passed value has an invalid type.

set_limits(*min_val: Union[bool, int, float, None] = None*, *max_val: Union[bool, int, float, None] = None*) → bool
Set the limits of the parameter and return whether the passed limit values differed from the old one.

Parameters

- **min_val** (*bool, int, float or None*) – The minimal value (default `None`, which means "doesn't matter").
- **max_val** (*bool, int, float or None*) – The maximal value (default `None`, which means "doesn't matter").

Returns `True` if the passed min- and/or max-value differed from the old one, `False` otherwise.

Return type `bool`

Raises **TypeError** – Will be raised if the passed minimal or maximal value has an invalid type.

to_str(*arg: Union[bool, int, float, htheatpump.htparams.HtDataTypes]*) → str
Convert the passed value to a string.

This method can be called as a *static method*, e.g.:

```
s = HtParam.to_str(123, HtDataTypes.FLOAT)
```

or as a *member method* of *HtParam*, e.g.:

```
param = HtParams["Temp. Aussen"]
s = param.to_str(3.2)
```

If the method is called as a member method of *HtParam*, the data type of the passed value don't have to be specified. It will be automatically determined from the *HtParam* instance.

Returns The string representation of the passed value.

Return type `str`

class htheatpump.htparams.**HtParams**
Dictionary of the supported Heliotherm heat pump parameters.*[0]

Note: The supported parameters and their definitions are loaded from the CSV file `htparams.csv` in this package, but the user can create his own user specific CSV file under `~/.htheatpump/htparams.csv`.

[0] Most of the supported heat pump parameters were found by "sniffing" the serial communication of the Heliotherm home control Windows application (http://homecontrol.heliotherm.com) during a refresh! ;-)

4.5 htheatpump.protocol

Protocol constants and functions for the Heliotherm heat pump communication.

htheatpump.protocol.**add_checksum**(*s: bytes*) → bytes
Add a checksum at the end of the provided bytes array.

Parameters **s** (*bytes*) – The provided byte array.

Returns Byte array with the added checksum.

Return type bytes

Raises **ValueError** – Will be raised for an invalid byte array with length less than 1 byte.

htheatpump.protocol.**calc_checksum**(*s: bytes*) → int
Function that calculates the checksum of a provided bytes array.

Parameters **s** (*bytes*) – Byte array from which the checksum should be computed.

Returns The computed checksum as int.

Return type int

htheatpump.protocol.**create_request**(*cmd: str*) → bytes
Create a specified request command for the heat pump.

Parameters **cmd** (*str*) – The command string.

Returns The request string for the specified command as byte array.

Return type bytes

Raises **ValueError** – Will be raised for an invalid byte array with length greater than 253 byte.

htheatpump.protocol.**verify_checksum**(*s: bytes*) → bool
Verify if the provided bytes array is terminated with a valid checksum.

Parameters **s** (*bytes*) – The byte array including the checksum.

Returns True if valid, False otherwise.

Return type bool

Raises **ValueError** – Will be raised for an invalid byte array with length less than 2 bytes.

4.6 htheatpump.utils

Some useful helper classes and methods.

class htheatpump.utils.**Singleton**
Singleton base class.

Example:

```
>>> class MySingleton(Singleton):
...     def __init__(self, v):
...         self._val = v
...     def __str__(self):
...         return str(self._val)
```

```
>>> s1 = MySingleton(1)
>>> print(str(s1))
1
>>> s2 = MySingleton(2)
>>> print(str(s2))
2
>>> print(str(s1))
2
```

See also:

https://mail.python.org/pipermail/python-list/2007-July/431423.html

class htheatpump.utils.**Timer**

Context manager for execution time measurement.

Example:

```
>>> with Timer() as timer:
...     s = "-".join(str(n) for n in range(1000))
...
>>> exec_time = timer.elapsed
```

elapsed

Return the elapsed time (in seconds).

Returns The elapsed time in seconds.

Return type float

CHAPTER 5

Sample scripts

Warning: Please note that any incorrect or careless usage of this module as well as errors in the implementation can damage your heat pump!

Therefore, the author does not provide any guarantee or warranty concerning to correctness, functionality or performance and does not accept any liability for damage caused by this module, examples or mentioned information.

Thus, use it on your own risk!

5.1 htdatetime

Command line tool to get and set date and time on the Heliotherm heat pump.

To change date and/or time on the heat pump the date and time has to be passed in ISO 8601 format (`YYYY-MM-DDTHH:MM:SS`) to the program. It is also possible to pass an empty string, therefore the current date and time of the host will be used. If nothing is passed to the program the current date and time on the heat pump will be returned.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htdatetime.py

Example:

```
$ htdatetime --device /dev/ttyUSB1 --baudrate 9600
Tuesday, 2017-11-21T21:48:04
```

```
$ htdatetime -d /dev/ttyUSB1 -b 9600 "2008-09-03T20:56:35"
Wednesday, 2008-09-03T20:56:35
```

5.2 htshell

Command shell tool to send raw commands to the Heliotherm heat pump.

For commands which deliver more than one response from the heat pump the expected number of responses can be defined by the argument `-r` or `--responses`.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htshell.py

Example:

```
$ htshell --device /dev/ttyUSB1 "AR,28,29,30" -r 3
> 'AR,28,29,30'
< 'AA,28,19,14.09.14-02:08:56,EQ_Spreizung'
< 'AA,29,20,14.09.14-11:52:08,EQ_Spreizung'
< 'AA,30,65534,15.09.14-09:17:12,Keine Stoerung'
```

5.3 htquery

Command line tool to query for parameters of the Heliotherm heat pump.

If the `-j`, `--json` option is used, the output will be in JSON format.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htquery.py

Example:

```
$ htquery --device /dev/ttyUSB1 "Temp. Aussen" "Stoerung"
Stoerung    : False
Temp. Aussen: 5.0
```

```
$ htquery --json "Temp. Aussen" "Stoerung"
{
    "Stoerung": false,
    "Temp. Aussen": 3.2
}
```

5.4 htset

Command line tool to set the value of a specific parameter of the heat pump.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htset.py

Example:

```
$ htset --device /dev/ttyUSB1 "HKR Soll_Raum" "21.5"
21.5
```

5.5 htfaultlist

Command line tool to query for the fault list of the heat pump.

The option `-c`, `--csv` and `-j`, `--json` can be used to write the fault list to a specified CSV or JSON file.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htfaultlist.py

Example:

```
$ htfaultlist --device /dev/ttyUSB1 --baudrate 9600
#000 [2000-01-01T00:00:00]: 65534, Keine Stoerung
#001 [2000-01-01T00:00:00]: 65286, Info: Programmupdate 1
#002 [2000-01-01T00:00:00]: 65285, Info: Initialisiert
#003 [2000-01-01T00:00:16]: 00009, HD Schalter
#004 [2000-01-01T00:00:20]: 00021, EQ Motorschutz
#005 [2014-08-06T13:25:54]: 65289, Info: Manueller Init
#006 [2014-08-06T13:26:10]: 65534, Keine Stoerung
#007 [2014-08-06T13:26:10]: 65287, Info: Programmupdate 2
#008 [2014-08-06T13:26:10]: 65285, Info: Initialisiert
#009 [2014-08-06T13:26:37]: 65298, Info: L.I.D. geaendert
#010 [2014-08-06T13:28:23]: 65534, Keine Stoerung
#011 [2014-08-06T13:28:27]: 65534, Keine Stoerung
```

5.6 htbackup

Command line tool to create a backup of the Heliotherm heat pump data points.

The option `-c`, `--csv` and `-j`, `--json` can be used to write the read data point values to a specified CSV or JSON file.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htbackup.py

Example:

```
$ htbackup --baudrate 9600 --csv backup.csv
'SP,NR=0' [Language]: VAL='0', MIN='0', MAX='4'
'SP,NR=1' [TBF_BIT]: VAL='0', MIN='0', MAX='1'
'SP,NR=2' [Rueckruferlaubnis]: VAL='1', MIN='0', MAX='1'
...
'MP,NR=0' [Temp. Aussen]: VAL='-7.0', MIN='-20.0', MAX='40.0'
'MP,NR=1' [Temp. Aussen verzoegert]: VAL='-6.9', MIN='-20.0', MAX='40.0'
'MP,NR=2' [Temp. Brauchwasser]: VAL='45.7', MIN='0.0', MAX='70.0'
...
```

5.7 hthttp

Simple HTTP server which provides the possibility to access the Heliotherm heat pump via URL requests.

Supported URL requests:

- **http://ip:port/datetime/sync** synchronize the system time of the heat pump with the current time
- **http://ip:port/datetime** query for the current system time of the heat pump
- **http://ip:port/faultlist/last** query for the last fault message of the heat pump
- **http://ip:port/faultlist** query for the whole fault list of the heat pump
- **http://ip:port/timeprog** query for the list of available time programs of the heat pump
- **http://ip:port/timeprog/<idx>** query for a specific time program of the heat pump
- **http://ip:port/param/?Param1&Param2&Param3=Value&Param4=Value ...** query and/or set specific parameter values of the heat pump
- **http://ip:port/param/** query for all "known" parameter values of the heat pump

- **http://ip:port/** query for some properties of the connected heat pump

The result in the HTTP response is given in JSON format.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/hthttp.py

Example:

```
$ hthttp start --device /dev/ttyUSB1 --ip 192.168.1.80 --port 8080
hthttp started with PID 1234

$ tail /tmp/hthttp-daemon.log
[2020-03-29 16:21:48,012][INFO    ][__main__|run]: === HtHttpDaemon.run() ␣
↪========================================
[2020-03-29 16:21:48,034][INFO    ][htheatpump.htheatpump|open_connection]: Serial
↪<id=0xb6020f50, open=True>(...)
[2020-03-29 16:21:48,083][INFO    ][htheatpump.htheatpump|login]: login successfully
[2020-03-29 16:21:48,116][INFO    ][__main__|run]: Connected successfully to heat␣
↪pump with serial number: 123456
[2020-03-29 16:21:48,156][INFO    ][__main__|run]: Software version: 3.0.20 (273)
[2020-03-29 16:21:48,203][INFO    ][htheatpump.htheatpump|logout]: logout successfully
[2020-03-29 16:21:48,400][INFO    ][__main__|run]: Starting server at: ('192.168.1.80
↪', 8080)
...

$ hthttp stop
```

5.8 htfastquery

Command line tool to query for parameters of the Heliotherm heat pump the fast way.

Note: Only parameters representing a "MP" data point are supported!

If the `-j, --json` option is used, the output will be in JSON format.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htfastquery.py

Example:

```
$ htfastquery --device /dev/ttyUSB1 "Temp. Vorlauf" "Temp. Ruecklauf"
Temp. Ruecklauf [MP,04]: 25.2
Temp. Vorlauf   [MP,03]: 25.3
```

```
$ htfastquery --json "Temp. Vorlauf" "Temp. Ruecklauf"
{
    "Temp. Ruecklauf": 25.2,
    "Temp. Vorlauf": 25.3
}
```

5.9 httimeprog

Command line tool to query for the time programs of the heat pump.

The option `-c`, `--csv` and `-j`, `--json` can be used to write the time program properties to a specified CSV or JSON file.

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/httimeprog.py

Example:

```
$ httimeprog --device /dev/ttyUSB1 --csv timeprog.csv 1 1
[idx=1]: idx=1, name='Zirkulationspumpe', ead=7, nos=2, ste=15, nod=7, entries=[...]
[day=1, entry=0]: state=0, time=00:00-06:00
[day=1, entry=1]: state=1, time=06:00-08:00
[day=1, entry=2]: state=0, time=08:00-11:30
[day=1, entry=3]: state=1, time=11:30-14:00
[day=1, entry=4]: state=0, time=14:00-18:00
[day=1, entry=5]: state=1, time=18:00-20:00
[day=1, entry=6]: state=0, time=20:00-24:00
```

5.10 htcomplparams

Command line tool to create a complete list of all Heliotherm heat pump parameters.

The option `-c` or `--csv` can be used to write the determined data to a CSV file. If no filename is specified an automatic one, consisting of serial number an software version, will be used (e.g. `htparams-123456-3_0_20-273.csv`).

This script can be used to create the basis for your own user specific parameter definition file, which can than be placed under `~/.htheatpump/htparams.csv` (see also *HtParams*).

Source: https://github.com/dstrigl/htheatpump/blob/master/htheatpump/scripts/htcomplparams.py

Example:

```
$ htcomplparams --device /dev/ttyUSB1 --baudrate 9600 --csv
connected successfully to heat pump with serial number 123456
software version = 3.0.20 (273)
'SP,NR=0' [Language]: VAL=0, MIN=0, MAX=4 (dtype=INT)
'SP,NR=1' [TBF_BIT]: VAL=0, MIN=0, MAX=1 (dtype=BOOL)
'SP,NR=2' [Rueckruferlaubnis]: VAL=1, MIN=0, MAX=1 (dtype=BOOL)
...
write data to: /home/pi/prog/htheatpump/htparams-123456-3_0_20-273.csv
```

CHAPTER 6

Heliotherm heat pump parameters

Tested with:

- Heliotherm HP08S10W-WEB, SW 3.0.20
- Heliotherm HP10S12W-WEB, SW 3.0.8
- Heliotherm HP08E-K-BC, SW 3.0.7B
- Heliotherm HP05S07W-WEB, SW 3.0.17 and SW 3.0.37
- Heliotherm HP12L-M-BC, SW 3.0.21
- Heliotherm HP-30-L-M-WEB, SW 3.0.21

6.1 SP Data Points

Parameter Name	Data Point	ACL	Data Type	Min Value	Max Value
Liegenschaft	SP,NR=3	r-	INT	0	2147483647
WP_System	SP,NR=4	r-	INT	1	2
WW Type	SP,NR=5	r-	INT	0	4
FWS Type	SP,NR=6	r-	BOOL	0	1
Puffer Type	SP,NR=7	r-	BOOL	0	1
Softwareversion	SP,NR=9	r-	INT	0	0
Verdichter_Status	SP,NR=10	r-	INT	0	11
Verdichter laeuft seit	SP,NR=11	r-	INT	10	100000
Verdichter Einschaltverz.(sec)	SP,NR=12	r-	INT	0	10800
Betriebsart[1]	SP,NR=13	r-	INT	0	7

Continued on next page

Table 1 – continued from previous page

Parameter Name	Data Point	ACL	Data Type	Min Value	Max Value
HKR Soll_Raum	SP,NR=69	r-	FLOAT	10.0	25.0
HKR Aufheiztemp. (K)	SP,NR=71	r-	INT	1	10
HKR Absenk-temp. (K)	SP,NR=72	r-	INT	-10	-1
HKR Heiz-grenze	SP,NR=76	r-	INT	0	45
HKR RLT Soll_oHG (Heizkurve)	SP,NR=80	r-	FLOAT	15.0	40.0
HKR RLT Soll_0 (Heizkurve)	SP,NR=81	r-	FLOAT	20.0	50.0
HKR RLT Soll_uHG (Heizkurve)	SP,NR=82	r-	FLOAT	25.0	60.0
WW Normal-temp.	SP,NR=83	r-	INT	10	75
WW Hysterese Normaltemp.	SP,NR=84	r-	INT	1	10
WW Minimal-temp.	SP,NR=85	r-	INT	5	45
WW Hysterese Minimaltemp.	SP,NR=86	r-	INT	1	10
BSZ HKP Be-triebsstunden	SP,NR=153	r-	INT	0	100000
BSZ HKP Schaltung	SP,NR=155	r-	INT	0	100000
BSZ EQ Be-triebsstunden	SP,NR=162	r-	INT	0	100000
BSZ EQ Schal-tungen	SP,NR=164	r-	INT	0	100000
BSZ WWV Be-triebsstunden	SP,NR=165	r-	INT	0	100000
BSZ WWV Schaltungen	SP,NR=167	r-	INT	0	100000
BSZ ZIPWW Betriebsstunden	SP,NR=168	r-	INT	0	100000
BSZ ZIPWW Schaltungen	SP,NR=170	r-	INT	0	100000
BSZ Verdichter Betriebsst. WW	SP,NR=171	r-	INT	0	100000
BSZ Verdichter Betriebsst. HKR	SP,NR=172	r-	INT	0	100000
BSZ Verdichter Betriebsst. ges	SP,NR=173	r-	INT	0	100000

Continued on next page

Table 1 – continued from previous page

Parameter Name	Data Point	ACL	Data Type	Min Value	Max Value
BSZ Verdichter akt. Laufzeit	SP,NR=174	r-	INT	0	100000
MKR2 Aktiviert	SP,NR=222	r-	INT	0	2
Energiezaehler	SP,NR=263	r-	INT	0	2
BSZ Verdichter Schaltung WW	SP,NR=375	r-	INT	0	100000

6.2 MP Data Points

Parameter Name	Data Point	ACL	Data Type	Min Value	Max Value
Temp. Aussen	MP,NR=0	r-	FLOAT	-20.0	40.0
Temp. Aussen verzoegert	MP,NR=1	r-	FLOAT	-20.0	40.0
Temp. Brauchwasser	MP,NR=2	r-	FLOAT	0.0	70.0
Temp. Vorlauf	MP,NR=3	r-	FLOAT	0.0	70.0
Temp. Ruecklauf	MP,NR=4	r-	FLOAT	0.0	70.0
Temp. Pufferspeicher	MP,NR=5	r-	FLOAT	0.0	70.0
Temp. EQ_Eintritt	MP,NR=6	r-	FLOAT	-20.0	30.0
Temp. EQ_Austritt	MP,NR=7	r-	FLOAT	-20.0	30.0
Temp. Sauggas	MP,NR=9	r-	FLOAT	-20.0	30.0
Temp. Frischwasser_Istwert	MP,NR=11	r-	FLOAT	0.0	70.0
Temp. Verdampfung	MP,NR=12	r-	FLOAT	-50.0	30.0
Temp. Kondensation	MP,NR=13	r-	FLOAT	-50.0	60.0
Temp. Heissgas	MP,NR=15	r-	FLOAT	0.0	150.0
Niederdruck (bar)	MP,NR=20	r-	FLOAT	0.0	18.0
Hochdruck (bar)	MP,NR=21	r-	FLOAT	0.0	40.0
Heizkreispumpe	MP,NR=22	r-	BOOL	False	True
EQ Pumpe (Ventilator)	MP,NR=24	r-	BOOL	False	True
Warmwasservorrang	MP,NR=25	r-	BOOL	False	True
Zirkulationspumpe WW	MP,NR=29	r-	BOOL	False	True
Verdichter	MP,NR=30	r-	BOOL	False	True
Stoerung	MP,NR=31	r-	BOOL	False	True
Hauptschalter	MP,NR=36	r-	BOOL	False	True
FWS Stroemungsschalter	MP,NR=38	r-	BOOL	False	True
BSZ Verdichter Schaltungen	MP,NR=41	r-	INT	0	100000
Frischwasserpumpe	MP,NR=50	r-	INT	0	100
Verdichteranforderung	MP,NR=56	r-	INT	0	5
HKR_Sollwert	MP,NR=57	r-	FLOAT	0.0	50.0

[1] **Betriebsart:**

- 0 = Aus
- 1 = Automatik
- 2 = Kühlen
- 3 = Sommer
- 4 = Dauerbetrieb
- 5 = Absenkung
- 6 = Urlaub
- 7 = Party
- [8 = Ausheizen]
- [9 = EVU-Sperre]
- [10 = Hauptschalter]

CHAPTER 7

Contributing

Contributions are welcome, and they are greatly appreciated! Every little bit helps, and credit will always be given.

You can contribute in many ways:

7.1 Types of Contributions

7.1.1 Report Bugs

Report bugs at https://github.com/dstrigl/htheatpump/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.

7.1.2 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with "bug" and "help wanted" is open to whoever wants to implement it.

7.1.3 Implement Features

Look through the GitHub issues for features. Anything tagged with "enhancement" and "help wanted" is open to whoever wants to implement it.

7.1.4 Write Documentation

htheatpump could always use more documentation, whether as part of the official htheatpump docs, in docstrings, or even on the web in blog posts, articles, and such.

7.1.5 Submit Feedback

The best way to send feedback is to file an issue at https://github.com/dstrigl/htheatpump/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 :)

7.2 Get Started!

Ready to contribute? Here's how to set up htheatpump for local development.

1. Fork the htheatpump repository on GitHub.
2. Clone your fork locally:

```
$ git clone git@github.com:your_name_here/htheatpump.git
```

3. Install your local copy into a virtualenv[1]. Assuming you have virtualenvwrapper installed, this is how you set up your fork for local development under Python 3.7:

```
$ mkvirtualenv hthp-py37 -p python3.7
$ cd htheatpump/
$ python setup.py develop
```

4. Install all project dependencies for local development (and testing):

```
$ pip install -r requirements/develop.pip
$ pip install -r requirements/test.pip
```

5. Create a branch for local development:

```
$ git checkout -b name-of-your-bugfix-or-feature
```

   Now you can make your changes locally.

6. When you're done making changes, check that your changes pass flake8 and the *tests* (using pytest), including testing other Python versions with tox:

```
$ flake8 htheatpump tests samples setup.py
$ pytest
$ tox
```

   There are also a few tests which only run if a heat pump is connected. These can be executed by passing the argument --connected to the test commands:

[1] If you need more information about Python virtual environments take a look at this article on RealPython.

```
$ pytest --connected
$ tox -- --connected
```

To change the default device (`/dev/ttyUSB0`) and baudrate (115200) use the arguments `--device` and `--baudrate`:

```
$ pytest --connected --device /dev/ttyUSB1 --baudrate 9600
$ tox -- --connected --device /dev/ttyUSB1 --baudrate 9600
```

7. Commit your changes and push your branch to GitHub:

```
$ git add .
$ git commit -m "A description of your changes."
$ git push origin name-of-your-bugfix-or-feature
```

8. Submit a pull request through the GitHub website.

7.3 Pull Request Guidelines

Before you submit a pull request, check that it meets these guidelines:

1. The pull request should include tests.
2. If the pull 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 pull request should work for Python 3.7 and 3.8. Check https://travis-ci.org/dstrigl/htheatpump/pull_requests and make sure that the tests pass for all supported Python versions.

7.4 Tips

To run a subset of tests:

```
$ pytest tests/test_htparams.py
```

CHAPTER 8

Credits

8.1 Development Lead

- Daniel Strigl (dstrigl)

8.2 Contributors

Thanks to

- Kilian,
- Hans,
- Alois,
- Simon and
- Felix (FelixPetriconi)

for their contribution with testing the functionality of this module.

CHAPTER 9

History

9.1 1.3.2 (2022-01-13)

- updated copyright statements
- some minor improvements in the tests
- fix: removed log of loaded definition file path (`HTHEATPUMP: load parameter definitions ...`) to get a clean JSON output in the sample scripts (e.g. `htquery.py`)

9.2 1.3.1 (2021-01-20)

- replaced Travis CI by GitHub Actions
- added async version of console scripts
- updated copyright statements
- some minor cleanup and improvements

9.3 1.3.0 (2020-12-28)

- added new class `AioHtHeatpump` for asynchronous communication (async/await) with the heat pump
- Python code reformatting using *Black* and *isort*
- moved protocol related constants and functions to `protocol.py`
- dropped support for Python 3.5 and 3.6

9.4 1.2.4 (2020-04-20)

- added support for Python 3.8
- some minor cleanup and improvements
- changed log statements to the form with the preferred and well-known `%s` (and `%d`, `%f`, etc.) string formatting indicators (due to performance reasons)
- added additional heat pump parameter (data points) `Hauptschalter` in `htparams.csv`

9.5 1.2.3 (2020-03-31)

- changed behaviour of `HtHeatpump.reconnect()`, which will now also establish a connection if still not connected
- added sample scripts (e.g. `htcomplparams`, `htquery`, etc.) to be part of the `htheatpump` package
- clean-up of `setup.py` and `MANIFEST.in`

9.6 1.2.2 (2020-03-29)

- added sample file `htparams-xxxxxx-3_0_20-273.csv` with a complete list of all heat pump parameters from a Heliotherm heat pump with SW 3.0.20
- added new sample script `htcomplparams.py` to create a complete list of all heat pump parameters
- added some more heat pump parameters (data points) in `htparams.csv`
- Python code reformatting using *Black*
- changed package requirements structure; some changes in `setup.py`, `setup.cfg`, `tox.ini`, etc.

9.7 1.2.1 (2020-02-07)

- updated copyright statements
- added factory function `from_json` to classes `TimeProgPeriod`, `TimeProgEntry` and `TimeProgram`
- fixed issue with fault lists with larger number of entries (in `HtHeatpump.get_fault_list()`); thanks to Alois for reporting
- added new function `HtParam.check_value_type` to verify the correct type of a passed value; the type of a passed value to `HtHeatpump.set_param()` will now be verified
- fixed issue with passing a larger number of indices to `HtHeatpump.fast_query()`

9.8 1.2.0 (2019-06-10)

- added support for Python's "with" statement for the `HtHeatpump` class
- added some more unit-tests (especially for the time program functions)
- extended the sample scripts `hthttp.py` to query for time programs of the heat pump

- added new sample `samples/httimeprog.py` to read the time programs of the heat pump
- added new functions to write/change time program entries of the heat pump (see `HtHeatpump.set_time_prog...`)
- added new functions to read the time program of the heat pump (see `HtHeatpump.get_time_prog...`)
- added type annotations and hints for static type checking (using *mypy*)
- splitted up property `HtHeatpump.verify_param` to `HtHeatpump.verify_param_action` and `HtHeatpump.verify_param_error`
- renamed exception `ParamVerificationException` to `VerificationException`
- added support for Python 3.7
- dropped support for Python 3.4
- added some more heat pump parameters (data points) in `htparams.csv`

9.9 1.1.0 (2019-02-23)

- added some more heat pump parameters (data points) in `htparams.csv`
- extended sample script `htfaultlist.py` by the possibility to write a JSON/CSV file
- added new sample scripts `hthttp.py` and `htfastquery.py`
- fixed some formatting (flake8) errors
- some improvement for the reconnect in the `login()` method of class `HtHeatpump`
- changed return type of `HtHeatpump.get_fault_list()` from `dict` to `list`
- added support for Python 3.6
- added support for a user specific parameter definition file under `~/.htheatpump/htparams.csv`
- extended sample `htbackup.py` to store also the limits (MIN and MAX) of each data point
- added method to verify the parameter definitions in `htparams.csv` during a `HtHeatpump.get_param()`, `HtHeatpump.set_param()` or `HtHeatpump.query()`; this is just for safety to be sure that the parameter definitions in `HtParams` are correct (deactivated by default, but can be activated by setting the property `HtHeatpump.verify_param` to `True`)
- added new method `HtHeatpump.fast_query()` to retrieve "MP" data point values in a faster way ("Web-Online")
- extended the `HtHeatpump.login()` method to perform an update of the parameter limits if desired

9.10 1.0.0 (2018-01-12)

- First release on PyPI.

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

h

Index

H

I

L

N

O