
MinimalModbus Documentation

Release 1.0.2

Jonas Berg

Aug 11, 2019

Contents

1	MinimalModbus	3
1.1	Web resources	3
1.2	Features	4
2	Installation	5
2.1	Dependencies	5
2.2	Alternate installation on Linux	5
2.3	Alternate installation on Windows	6
2.4	If everything else fails	6
3	Usage	7
3.1	General on Modbus protocol	7
3.2	Typical hardware	7
3.3	Typical usage	8
3.4	Default values	8
3.5	Confusing Modbus register addresses	9
3.6	Using multiple instruments	9
3.7	Closing serial port after each call	10
3.8	Handling communication errors	10
3.9	Byte order for floating point values and long integers	10
3.10	Subclassing	10
3.11	Extending	10
4	API for MinimalModbus	11
5	Modbus details	19
5.1	Modbus data types	19
5.2	Some extensions not covered by the official standard	20
5.3	Implemented functions	20
5.4	Byte order for data stored in serveral registers	21
5.5	Modbus implementation details	22
5.6	Reading individual bits from a 16-bit register	24
5.7	Known deviations from the standard	24
5.8	MODBUS ASCII format	24
5.9	Manual testing of Modbus equipment	25
6	Serial communication	27

6.1	Timing of the serial communications	27
6.2	RS-485 introduction	27
6.3	Controlling the RS485 transmitter	28
6.4	Controlling the RS-485 transceiver from userspace	29
7	Debug mode	31
7.1	Debug mode	31
8	Trouble shooting	35
8.1	No communication	35
8.2	Local echo	36
8.3	Empty bytes added in the beginning or the end on the received message	36
8.4	Serial adaptors not recognized	36
8.5	Known issues	36
8.6	Issues when running under Windows	36
8.7	Support	37
9	Detailed usage documentation	39
9.1	Interactive usage	39
9.2	Making drivers for specific instruments	40
9.3	Using this module as part of a measurement system	42
9.4	Handling extra 0xFE byte after some messages	43
9.5	Install or uninstalling a distribution	43
9.6	Setting the PYTHONPATH	44
9.7	Including MinimalModbus in a Yocto build	45
10	Developer documentation	47
10.1	Design considerations	47
10.2	General driver structure	48
10.3	Number conversion to and from bytestrings	48
10.4	Unit testing	48
10.5	Making sure that error messages are informative for the user	49
10.6	Recording communication data for unittesting	50
10.7	Using the dummy serial port	51
10.8	Data encoding in Python2 and Python3	52
10.9	Extending MinimalModbus	53
10.10	Other useful internal functions	53
10.11	Generate documentation	54
10.12	Webpage	54
10.13	Travis CI	54
10.14	Codecov.io	54
10.15	Notes on distribution	54
10.16	Preparation for release	55
10.17	Useful development tools	57
10.18	Git usage	57
10.19	Sphinx usage	58
10.20	Unittest coverage measurement using coverage.py	60
10.21	Using the flake8 style checker tool	61
10.22	TODO	61
11	Contributing / Report bugs	63
11.1	Types of Contributions	63
11.2	Get Started!	64
11.3	Pull Request Guidelines	65

12 Credits	67
12.1 Development Lead	67
12.2 Contributors	67
13 Related software	69
14 History	71
14.1 Release 1.0.2 (2019-08-11)	71
14.2 Release 1.0.1 (2019-08-10)	71
14.3 Release 1.0.0 (2019-08-10)	71
14.4 Release 0.7 (2015-07-30)	72
14.5 Release 0.6 (2014-06-22)	72
14.6 Release 0.5 (2014-03-23)	72
14.7 Release 0.4 (2012-09-08)	73
14.8 Release 0.3.2 (2012-01-25)	73
14.9 Release 0.3.1 (2012-01-24)	73
14.10 Release 0.3 (2012-01-23)	73
14.11 Release 0.2 (2011-08-19)	74
14.12 Release 0.1 (2011-06-16)	74
15 Internal documentation	75
15.1 Documentation for dummy_serial (which is a serial port mock)	75
15.2 Internal documentation for MinimalModbus	76
15.3 Internal documentation for unit testing of MinimalModbus	97
15.4 Internal documentation for hardware testing of MinimalModbus using DTB4824	111
16 Indices and tables	115
Python Module Index	117
Index	119

Documentation built using Sphinx Aug 11, 2019 for MinimalModbus version 1.0.2.

Contents:

Easy-to-use Modbus RTU and Modbus ASCII implementation for Python.

1.1 Web resources

- **Documentation:** <https://minimalmodbus.readthedocs.io>
- Source code on **GitHub:** <https://github.com/pyhys/minimalmodbus>
- Python package index (PyPI) with download: <https://pypi.org/project/MinimalModbus/>

Other web pages:

- Readthedocs project page: <https://readthedocs.org/projects/minimalmodbus/>
- Travis CI build status page: <https://travis-ci.org/pyhys/minimalmodbus>
- codecov.io project page: <https://codecov.io/github/pyhys/minimalmodbus>

Obsolete web pages:

- Old Sourceforge documentation page: <http://minimalmodbus.sourceforge.net/>
- Old Sourceforge project page: <https://sourceforge.net/projects/minimalmodbus>
- Old Sourceforge repository: <https://sourceforge.net/p/minimalmodbus/code/HEAD/tree/>

1.2 Features

MinimalModbus is an easy-to-use Python module for talking to instruments (slaves) from a computer (master) using the Modbus protocol, and is intended to be running on the master. The only dependence is the pySerial module (also pure Python).

There are convenience functions to handle floats, strings and long integers (in different byte orders).

This software supports the ‘Modbus RTU’ and ‘Modbus ASCII’ serial communication versions of the protocol, and is intended for use on Linux, OS X and Windows platforms. It is open source, and has the Apache License, Version 2.0.

Tested with Python 2.7, 3.5, 3.6 and 3.7.

Support for Python2 will be dropped in 2020.

This package uses semantic versioning.

At the command line:

```
pip3 install -U minimalmodbus
```

or possibly:

```
sudo pip3 install -U minimalmodbus
```

For legacy Python (2.7), use `pip` instead of `pip3`.

2.1 Dependencies

Python versions 2.7 and higher are supported (including 3.x). Tested with Python 2.7, 3.5, 3.6 and 3.7. This module is pure Python.

This module relies on `pySerial` (also pure Python) to do the heavy lifting, and it is the only dependency. It is BSD-3-Clause licensed. You can find it at the Python package index: <https://pypi.org/project/pyserial> The version of `pyserial` should be 3.0 or later.

Note: Since `MinimalModbus` 1.0 you need to use `pySerial` version at least 3.0

2.2 Alternate installation on Linux

You can also manually download the compressed source files from <https://pypi.org/project/MinimalModbus/>. In that case you first need to manually install `pySerial` from <https://pypi.org/project/pyserial>.

There are compressed source files for Unix/Linux (.tar.gz) and Windows (.zip). To install a manually downloaded file, uncompress it and run (from within the directory):

```
python3 setup.py install
```

or possibly:

```
sudo python3 setup.py install
```

If using Python2, use `python` instead of `python3`.

To make sure it is installed properly, print the `_get_diagnostic_string()` message. See the [Support](#) section for instructions.

You can also download the source directly from Linux command line:

```
wget https://pypi.python.org/packages/source/M/MinimalModbus/MinimalModbus-0.7.tar.gz  
TODO
```

Change version number to the appropriate value.

Downloading from GitHub:

```
wget https://github.com/pyhys/minimalmodbus/archive/master.zip  
unzip master.zip
```

This will create a directory 'minimalmodbus-master'.

2.3 Alternate installation on Windows

Install from GitHub, using pip:

```
C:\Python34\Scripts>pip3.4 install https://github.com/pyhys/minimalmodbus/archive/  
↪master.zip  
TODO
```

It will be installed in:

```
C:\Python34\Lib\site-packages  
TODO
```

In order to run Python from command line, you might need:

```
set PATH=%PATH%;C:\Python34  
TODO
```

2.4 If everything else fails

You can download the raw `minimalmodbus.py` file from GitHub, and put it in the same directory as your other code. Note that you must have `pySerial` installed.

3.1 General on Modbus protocol

Modbus is a serial communications protocol published by Modicon in 1979, according to <https://en.wikipedia.org/wiki/Modbus>. It is often used to communicate with industrial electronic devices.

There are several types of Modbus protocols:

Modbus RTU A serial protocol that uses binary representation of the data. **Supported by this software.**

Modbus ASCII A serial protocol that uses ASCII representation of the data. **Supported by this software.**

Modbus TCP, and variants A protocol for communication over TCP/IP networks. Not supported by this software.

For full documentation on the Modbus protocol, see www.modbus.com.

Two important documents are:

- [Modbus application protocol V1.1b](#)
- [Modbus over serial line specification and implementation guide V1.02](#)

Note that the computer (master) actually is a client, and the instruments (slaves) are servers.

3.2 Typical hardware

The application for which I wrote this software is to read and write data from Eurotherm process controllers. These come with different types of communication protocols, but the controllers I prefer use the Modbus RTU protocol. MinimalModbus is intended for general communication using the Modbus RTU protocol (using a serial link), so there should be lots of applications.

There can be several instruments (slaves, nodes) on a single bus, and the slaves have addresses in the range 1 to 247. In the Modbus RTU protocol, only the master can initiate communication. The physical layer is most often the serial bus RS485, which is described at <https://en.wikipedia.org/wiki/RS-485>.

To connect your computer to the RS485 bus, a serial port is required. There are direct USB-to-RS485 converters, but I use a USB-to-RS232 converter together with an industrial RS232-to-RS485 converter (Westermo MDW-45). This has the advantage that the latter is galvanically isolated using opto-couplers, and has transient suppression.

3.3 Typical usage

The instrument is typically connected via a serial port, and a USB-to-serial adaptor should be used on most modern computers. How to configure such a serial port is described on the pySerial page: <https://pythonhosted.org/pyserial/>

For example, consider an instrument (slave) with Modbus RTU mode and address number 1 to which we are to communicate via a serial port with the name `/dev/ttyUSB1`. The instrument stores the measured temperature in register 289. For this instrument a temperature of 77.2 C is stored as (the integer) 772, why we use 1 decimal. To read this data from the instrument:

```
#!/usr/bin/env python3
import minimalmodbus

instrument = minimalmodbus.Instrument('/dev/ttyUSB1', 1) # port name, slave address,
↳(in decimal)

## Read temperature (PV = ProcessValue) ##
temperature = instrument.read_register(289, 1) # Registernumber, number of decimals
print(temperature)

## Change temperature setpoint (SP) ##
NEW_TEMPERATURE = 95
instrument.write_register(24, NEW_TEMPERATURE, 1) # Registernumber, value, number of,
↳decimals for storage
```

The valid slave address range for normal usage is 1 to 247.

The full API for MinimalModbus is available in *API for MinimalModbus*.

Correspondingly for Modbus ASCII mode:

```
instrument = minimalmodbus.Instrument('/dev/ttyUSB1', 1, minimalmodbus.MODE_ASCII)
```

More on the usage of MinimalModbus is found in *Detailed usage documentation*.

3.4 Default values

Most of the serial port parameters have the default values defined in the Modbus standard (19200 8N1):

```
instrument.serial.port # this is the serial port name
instrument.serial.baudrate = 19200 # Baud
instrument.serial.bytesize = 8
instrument.serial.parity = serial.PARITY_NONE
instrument.serial.stopbits = 1
instrument.serial.timeout = 0.05 # seconds

instrument.address # this is the slave address number
instrument.mode = minimalmodbus.MODE_RTU # rtu or ascii mode
instrument.clear_buffers_before_each_transaction = True
```

Change the values like this:

```
instrument.serial.timeout = 0.2
```

To see which settings you actually are using:

```
print(instrument)
```

For details on the allowed parity values, see https://pyserial.readthedocs.io/en/latest/pyserial_api.html#constants

To change the parity setting, use:

```
import serial
instrument.serial.parity = serial.PARITY_EVEN
```

or alternatively (to avoid import of serial):

```
instrument.serial.parity = minimalmodbus.serial.PARITY_EVEN
```

Warning: The module level constants `minimalmodbus.BAUDRATE` etc were removed in version 1.0

3.5 Confusing Modbus register addresses

Sometimes “entity numbers” are used in documentation for Modbus instruments. These numbers are typically five or six digits long, and contains info about both the register type and the register address. The first digit describes the register type, for example 4 is a holding register. The rest of the digits describes the address plus one (yes, very confusing).

According to the example on <https://en.wikipedia.org/wiki/Modbus>, an entity number of 40100 describes a holding register with address 99.

More details on different types of Modbus registers are found in *Modbus details*.

3.6 Using multiple instruments

Use a single script for talking to all your instruments (if connected via the same serial port). Create several instrument objects like:

```
instrumentA = minimalmodbus.Instrument('/dev/ttyUSB1', 1)
instrumentA.serial.baudrate = 9600
instrumentA.serial.timeout = 0.2
instrumentA.mode = minimalmodbus.MODE_RTU

instrumentB = minimalmodbus.Instrument('/dev/ttyUSB1', 2)
instrumentB.mode = minimalmodbus.MODE_ASCII

instrumentC = minimalmodbus.Instrument('/dev/ttyUSB2', 1)
```

The instruments sharing the same serial port share the same serial Python object, so `instrumentB` will have the same baudrate and timeout as `instrumentA`.

You can use instruments on different serial ports in the same script, but running several scripts using the same port will give problems.

3.7 Closing serial port after each call

In some cases (mostly on Windows) the serial port must be closed after each call.

Enable that behavior with:

```
instrument.close_port_after_each_call = True
```

This will slow down the port considerably.

3.8 Handling communication errors

Your top-level code should be able to handle communication errors. Glitches in the serial communication might happen now and then.

Instead of running:

```
print(instrument.read_register(4143))
```

Use:

```
try:
    print(instrument.read_register(4143))
except IOError:
    print("Failed to read from instrument")
```

Different types of errors should be handled separately.

Errors related to wrong argument to functions raises `TypeError` or `ValueError`. When there is communication problems etc the exceptions raised are `ModbusException` (with subclasses) and `serial.serialutil.SerialException`, which both are inheriting from `IOError`.

Note that in Python3 the `IOError` is an alias for `OSError`.

3.9 Byte order for floating point values and long integers

The byte order used by manufacturers differ. See *Byte order for data stored in several registers*.

3.10 Subclassing

It is better to put the details on registers etc in a driver for the specific instrument. See *Making drivers for specific instruments*.

3.11 Extending

It is pretty easy to extend this module to support other functioncodes or special cases. See *Extending MinimalModbus*.

API for MinimalModbus

MinimalModbus: A Python driver for Modbus RTU/ASCII via serial port (via USB, RS485 or RS232).

```
class minimalmodbus.Instrument (port, slaveaddress, mode='rtu',  
                                close_port_after_each_call=False, debug=False)
```

Instrument class for talking to instruments (slaves).

Uses the Modbus RTU or ASCII protocols (via RS485 or RS232).

Args:

- **port** (str): The serial port name, for example `/dev/ttyUSB0` (Linux), `/dev/tty.usbserial` (OS X) or `COM4` (Windows).
- **slaveaddress** (int): Slave address in the range 1 to 247 (use decimal numbers, not hex). Address 0 is for broadcast, and 248-255 are reserved.
- **mode** (str): Mode selection. Can be `MODE_RTU` or `MODE_ASCII`.
- **close_port_after_each_call** (bool): If the serial port should be closed after each call to the instrument.
- **debug** (bool): Set this to `True` to print the communication details

address = None

Slave address (int). Most often set by the constructor (see the class documentation).

mode = None

Slave mode (str), can be `MODE_RTU` or `MODE_ASCII`. Most often set by the constructor (see the class documentation).

Changing this will not affect how other instruments use the same serial port.

New in version 0.6.

precalculate_read_size = None

If this is `False`, the serial port reads until timeout instead of just reading a specific number of bytes. Defaults to `True`.

Changing this will not affect how other instruments use the same serial port.

New in version 0.5.

debug = None

Set this to `True` to print the communication details. Defaults to `False`.

Most often set by the constructor (see the class documentation).

Changing this will not affect how other instruments use the same serial port.

clear_buffers_before_each_transaction = None

If this is `True`, the serial port read and write buffers are cleared before each request to the instrument, to avoid cumulative byte sync errors across multiple messages. Defaults to `True`.

Changing this will not affect how other instruments use the same serial port.

New in version 1.0.

close_port_after_each_call = None

If this is `True`, the serial port will be closed after each call. Defaults to `False`.

Changing this will not affect how other instruments use the same serial port.

Most often set by the constructor (see the class documentation).

handle_local_echo = None

Set to `True` if your RS-485 adaptor has local echo enabled. Then the transmitted message will immediately appear at the receive line of the RS-485 adaptor. `MinimalModbus` will then read and discard this data, before reading the data from the slave. Defaults to `False`.

Changing this will not affect how other instruments use the same serial port.

New in version 0.7.

serial = None

The serial port object as defined by the `pySerial` module. Created by the constructor.

Attributes that could be changed after initialisation:

- **port (str): Serial port name.**
 - Most often set by the constructor (see the class documentation).
- **baudrate (int): Baudrate in Baud.**
 - Defaults to 19200.
- **parity (probably int): Parity. See the `pySerial` module for documentation.**
 - Defaults to `serial.PARITY_NONE`.
- **bytesize (int): Bytesize in bits.**
 - Defaults to 8.
- **stopbits (int): The number of stopbits.**
 - Defaults to 1.
- **timeout (float): Read timeout value in seconds.**
 - Defaults to 0.05 s.
- **write_timeout (float): Write timeout value in seconds.**
 - Defaults to 2.0 s.

read_bit (*registeraddress*, *functioncode=2*)

Read one bit from the slave (instrument).

This is for a bit that has its individual address in the instrument.

Args:

- `registeraddress` (int): The slave register address (use decimal numbers, not hex).
- `functioncode` (int): Modbus function code. Can be 1 or 2.

Returns: The bit value 0 or 1 (int).

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

`write_bit` (*registeraddress, value, functioncode=5*)

Write one bit to the slave (instrument).

This is for a bit that has its individual address in the instrument.

Args:

- `registeraddress` (int): The slave register address (use decimal numbers, not hex).
- `value` (int or bool): 0 or 1, or `True` or `False`
- `functioncode` (int): Modbus function code. Can be 5 or 15.

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

`read_bits` (*registeraddress, number_of_bits, functioncode=2*)

Read multiple bits from the slave (instrument).

This is for bits that have individual addresses in the instrument.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `number_of_bits` (int): Number of bits to read
- `functioncode` (int): Modbus function code. Can be 1 or 2.

Returns: A list of bit values 0 or 1 (int). The first value in the list is for the bit at the given address.

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

`write_bits` (*registeraddress, values*)

Write multiple bits to the slave (instrument).

This is for bits that have individual addresses in the instrument.

Uses Modbus functioncode 15.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `values` (list of int or bool): 0 or 1, or `True` or `False`. The first value in the list is for the bit at the given address.

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

`read_register` (*registeraddress, number_of_decimals=0, functioncode=3, signed=False*)

Read an integer from one 16-bit register in the slave, possibly scaling it.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- `registeraddress` (int): The slave register address (use decimal numbers, not hex).

- `number_of_decimals` (int): The number of decimals for content conversion.
- `functioncode` (int): Modbus function code. Can be 3 or 4.
- `signed` (bool): Whether the data should be interpreted as unsigned or signed.

Note: The parameter `number_of_decimals` was named `numberOfDecimals` before MinimalModbus 1.0

If a value of 77.0 is stored internally in the slave register as 770, then use `number_of_decimals=1` which will divide the received data by 10 before returning the value.

Similarly `number_of_decimals=2` will divide the received data by 100 before returning the value.

Some manufacturers allow negative values for some registers. Instead of an allowed integer range 0 to 65535, a range -32768 to 32767 is allowed. This is implemented as any received value in the upper range (32768 to 65535) is interpreted as negative value (in the range -32768 to -1).

Use the parameter `signed=True` if reading from a register that can hold negative values. Then upper range data will be automatically converted into negative return values (two's complement).

<code>signed</code>	Data type in slave	Alternative name	Range
False	Unsigned INT16	Unsigned short	0 to 65535
True	INT16	Short	-32768 to 32767

Returns: The register data in numerical value (int or float).

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

write_register (*registeraddress*, *value*, *number_of_decimals=0*, *functioncode=16*, *signed=False*)

Write an integer to one 16-bit register in the slave, possibly scaling it.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- `registeraddress` (int): The slave register address (use decimal numbers, not hex).
- `value` (int or float): The value to store in the slave register (might be scaled before sending).
- `number_of_decimals` (int): The number of decimals for content conversion.
- `functioncode` (int): Modbus function code. Can be 6 or 16.
- `signed` (bool): Whether the data should be interpreted as unsigned or signed.

Note: The parameter `number_of_decimals` was named `numberOfDecimals` before MinimalModbus 1.0

To store for example `value=77.0`, use `number_of_decimals=1` if the slave register will hold it as 770 internally. This will multiply `value` by 10 before sending it to the slave register.

Similarly `number_of_decimals=2` will multiply `value` by 100 before sending it to the slave register.

As the largest number that can be written to a register is `0xFFFF = 65535`, the `value` and `number_of_decimals` should max be 65535 when combined. So when using `number_of_decimals=3` the maximum value is 65.535.

For discussion on negative values, the range and on alternative names, see `read_register()`.

Use the parameter `signed=True` if writing to a register that can hold negative values. Then negative input will be automatically converted into upper range data (two's complement).

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

read_long (*registeraddress*, *functioncode=3*, *signed=False*, *byteorder=0*)

Read a long integer (32 bits) from the slave.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *functioncode* (int): Modbus function code. Can be 3 or 4.
- *signed* (bool): Whether the data should be interpreted as unsigned or signed.
- *byteorder* (int): How multi-register data should be interpreted. Defaults to BYTEORDER_BIG.

signed	Data type in slave	Alternative name	Range
False	Unsigned INT32	Unsigned long	0 to 4294967295
True	INT32	Long	-2147483648 to 2147483647

Returns: The numerical value (int).

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

write_long (*registeraddress*, *value*, *signed=False*, *byteorder=0*)

Write a long integer (32 bits) to the slave.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Uses Modbus function code 16.

For discussion on number of bits, number of registers, the range and on alternative names, see [read_long\(\)](#).

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *value* (int or long): The value to store in the slave.
- *signed* (bool): Whether the data should be interpreted as unsigned or signed.
- *byteorder* (int): How multi-register data should be interpreted. Defaults to BYTEORDER_BIG.

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

read_float (*registeraddress*, *functioncode=3*, *number_of_registers=2*, *byteorder=0*)

Read a floating point number from the slave.

Floats are stored in two or more consecutive 16-bit registers in the slave. The encoding is according to the standard IEEE 754.

There are differences in the byte order used by different manufacturers. A floating point value of 1.0 is encoded (in single precision) as 3f800000 (hex). In this implementation the data will be sent as '\x3f\x80' and '\x00\x00' to two consecutive registers by default. Make sure to test that it makes sense for your instrument. If not, change the `byteorder` argument.

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).

- `functioncode` (int): Modbus function code. Can be 3 or 4.
- `number_of_registers` (int): The number of registers allocated for the float. Can be 2 or 4.
- `byteorder` (int): How multi-register data should be interpreted. Defaults to `BYTEORDER_BIG`.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Type of floating point number in slave	Size	Registers	Range
Single precision (binary32)	32 bits (4 bytes)	2 registers	1.4E-45 to 3.4E38
Double precision (binary64)	64 bits (8 bytes)	4 registers	5E-324 to 1.8E308

Returns: The numerical value (float).

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

write_float (*registeraddress*, *value*, *number_of_registers=2*, *byteorder=0*)

Write a floating point number to the slave.

Floats are stored in two or more consecutive 16-bit registers in the slave.

Uses Modbus function code 16.

For discussion on precision, number of registers and on byte order, see `read_float()`.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `value` (float or int): The value to store in the slave
- `number_of_registers` (int): The number of registers allocated for the float. Can be 2 or 4.
- `byteorder` (int): How multi-register data should be interpreted. Defaults to `BYTEORDER_BIG`.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

read_string (*registeraddress*, *number_of_registers=16*, *functioncode=3*)

Read an ASCII string from the slave.

Each 16-bit register in the slave are interpreted as two characters (each 1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

International characters (Unicode/UTF-8) are not supported.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `number_of_registers` (int): The number of registers allocated for the string.
- `functioncode` (int): Modbus function code. Can be 3 or 4.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Returns: The string (str).

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

write_string (*registeraddress*, *textstring*, *number_of_registers=16*)

Write an ASCII string to the slave.

Each 16-bit register in the slave are interpreted as two characters (each 1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

Uses Modbus function code 16.

International characters (Unicode/UTF-8) are not supported.

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *textstring* (str): The string to store in the slave, must be ASCII.
- *number_of_registers* (int): The number of registers allocated for the string.

Note: The parameter *number_of_registers* was named *numberOfRegisters* before MinimalModbus 1.0

If the *textstring* is longer than the $2 * \text{number_of_registers}$, an error is raised. Shorter strings are padded with spaces.

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

read_registers (*registeraddress*, *number_of_registers*, *functioncode=3*)

Read integers from 16-bit registers in the slave.

The slave registers can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *number_of_registers* (int): The number of registers to read, max 125 registers.
- *functioncode* (int): Modbus function code. Can be 3 or 4.

Note: The parameter *number_of_registers* was named *numberOfRegisters* before MinimalModbus 1.0

Any scaling of the register data, or converting it to negative number (two’s complement) must be done manually.

Returns: The register data (a list of int). The first value in the list is for the register at the given address.

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

write_registers (*registeraddress*, *values*)

Write integers to 16-bit registers in the slave.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Uses Modbus function code 16.

The number of registers that will be written is defined by the length of the *values* list.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `values` (list of int): The values to store in the slave registers, max 123 values. The first value in the list is for the register at the given address.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Any scaling of the register data, or converting it to negative number (two's complement) must be done manually.

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

exception `minimalmodbus.ModbusException`

Bases: `exceptions.IOError`

Base class for Modbus communication exceptions.

Inherits from `IOError`, which is an alias for `OSError` in Python3.

exception `minimalmodbus.SlaveReportedException`

Bases: `minimalmodbus.ModbusException`

Base class for exceptions that the slave (instrument) reports.

exception `minimalmodbus.SlaveDeviceBusyError`

Bases: `minimalmodbus.SlaveReportedException`

The slave is busy processing some command.

exception `minimalmodbus.NegativeAcknowledgeError`

Bases: `minimalmodbus.SlaveReportedException`

The slave can not fulfil the programming request.

This typically happens when using function code 13 or 14 decimal.

exception `minimalmodbus.IllegalRequestError`

Bases: `minimalmodbus.SlaveReportedException`

The slave has received an illegal request.

exception `minimalmodbus.MasterReportedException`

Bases: `minimalmodbus.ModbusException`

Base class for exceptions that the master (computer) detects.

exception `minimalmodbus.NoResponseError`

Bases: `minimalmodbus.MasterReportedException`

No response from the slave.

exception `minimalmodbus.LocalEchoError`

Bases: `minimalmodbus.MasterReportedException`

There is some problem with the local echo.

exception `minimalmodbus.InvalidResponseError`

Bases: `minimalmodbus.MasterReportedException`

The response does not fulfill the Modbus standard, for example wrong checksum.

5.1 Modbus data types

The Modbus standard defines storage in:

- Bits. Each bit has its own address.
- Registers (16-bit). Each register has its own address. Can hold integers in the range 0 to 65535 (dec), which is 0 to ffff (hex). Also called ‘unsigned INT16’ or ‘unsigned short’.

Modbus defines “table” names dependent on whether the storage is in a single bit or in a 16-bit register, and whether it is possible to write to the storage.

Storage in	Access	Modbus “table”	Example use on instrument
Bits	Read only	Discrete inputs	A digital input
Bits	Read and write	Coils	A digital output
16-bit register	Read only	Input registers	Several digital inputs, or an analog input
16-bit register	Read and write	Holding registers	Several digital outputs, or a setting parameter

Function codes are used to describe the read or write operations (shown in decimal in the table below)

Storage	Read		Write	
	Single	Multiple	Single	Multiple
Read-and-write bits (coils)		1	5	15
Read-only bits (discrete inputs)		2	None	None
Read-and-write registers (holding registers)		3, 23	6	16, 23
Read-only registers (input registers)		4, 23	None	None

Note that function code 23 not is implemented by this software (it is for reading and writing in same request).

Function codes 128 and larger are used by slaves to indicate errors.

5.2 Some extensions not covered by the official standard

Scaling of register values Some manufacturers store a temperature value of 77.0 C as 770 in the register, to allow room for one decimal.

Negative numbers (INT16 = short) Some manufacturers allow negative values for some registers. Instead of an allowed integer range 0-65535, a range -32768 to 32767 is allowed. This is implemented as any received value in the upper range (32768-65535) is interpreted as negative value (in the range -32768 to -1). This is two's complement and is described at https://en.wikipedia.org/wiki/Two%27s_complement. Help functions to calculate the two's complement value (and back) are provided in MinimalModbus.

Long integers ('Unsigned INT32' or 'INT32') These require 32 bits, and are implemented as two consecutive 16-bit registers. The range is 0 to 4294967295, which is called 'unsigned INT32'. Alternatively negative values can be stored if the instrument is defined that way, and is then called 'INT32' which has the range -2147483648 to 2147483647. Unfortunately the byte order might differ between manufacturers of Modbus instruments.

Floats (single or double precision) Single precision floating point values (binary32) are defined by 32 bits (4 bytes), and are implemented as two consecutive 16-bit registers. Correspondingly, double precision floating point values (binary64) use 64 bits (8 bytes) and are implemented as four consecutive 16-bit registers. How to convert from the bit values to the floating point value is described in the standard IEEE 754, as seen in https://en.wikipedia.org/wiki/Floating_point. Unfortunately the byte order might differ between manufacturers of Modbus instruments.

Strings Each register (16 bits) is interpreted as two ASCII characters (each 1 byte = 8 bits). Often 16 consecutive registers are used, allowing 32 characters in the string. Unicode/UTF-8 is typically not supported.

8-bit registers For example Danfoss use 8-bit registers for storage of some settings internally in the instruments. The data is nevertheless transmitted as 16 bit over the serial link, so you can read and write like normal (but with values limited to the range 0-255).

32-bit registers "Enron Modbus" allows larger registers where you can store 32 bits in a single register (instead of two consecutive 16 bit registers). Not supported by this software.

Bit fields in integers Some manufacturers store multiple bits in a 16-bit register, instead of as individually adressable bits. This is also known as flag registers. See below for how to use them with this software.

JBUS addressing From Eurotherm Modbus/ProfibusCommunications Handbook: "The JBUS protocol is identical in all respects but one to the Modbus protocol. The one difference concerns the parameter or register address. Both use a numeric index but the JBUS index starts at 0 while the Modbus index starts at 1."

Slave addresses larger than 255 Sometimes the slave address is encoded in two bytes to allow values larger than 255. Not supported by this software.

5.3 Implemented functions

These are the functions to use for reading and writing registers and bits of your instrument. Study the documentation of your instrument to find which Modbus function code to use. The function codes (F code) are given in decimal in this table.

Data type in slave	Read	F code	Write	F code
Bit	<code>read_bit()</code>	2 [or 1]	<code>write_bit()</code>	5 [or 15]
Bits Simultaneous reading	<code>read_bits()</code>	2 [or 1]	<code>write_bits()</code>	15
Register Integer, possibly scaled	<code>read_register()</code>	3 [or 4]	<code>write_register()</code>	16 [or 6]
Long integer (32 bits = 2 registers)	<code>read_long()</code>	3 [or 4]	<code>write_long()</code>	16
Float (32 or 64 bits = 2 or 4 registers)	<code>read_float()</code>	3 [or 4]	<code>write_float()</code>	16
String 2 characters per register	<code>read_string()</code>	3 [or 4]	<code>write_string()</code>	16
Registers Integers	<code>read_registers()</code>	3 [or 4]	<code>write_registers()</code>	16

See the API for MinimalModbus: *API for MinimalModbus*.

5.4 Byte order for data stored in serveral registers

Floats and long integers does not fit in a single 16-bit register, so typically consecutive registers are used. However different manufacturers store the bytes in different order.

The functions handling floats and long integers have a parameter for changing which byte order that is used.

Name	Description	Use	Example
Big endian (Motorola)	High order byte first	BYTEORDER_BIG	ABCD
?	Big endian with byte swap	BYTEORDER_BIG_SWAP	BADC
?	Little endian with byte swap	BYTEORDER_LITTLE_SWAP	CDAB
Little endian (Intel)	Low order byte first	BYTEORDER_LITTLE	DCBA

The example column show how the bytes are ordered on the wire (assuming byte A is the most significant byte).

Read more on Modbus byte ordering in these articles:

- <https://store.chipkin.com/articles/how-real-floating-point-and-32-bit-data-is-encoded-in-modbus-rtu-messages>
- https://www.modbustools.com/poll_display_formats.html
- <https://www.simplymodbus.ca/FAQ.htm#Ext>

5.5 Modbus implementation details

In Modbus RTU, the request message is sent from the master in this format:

- Slave address [1 Byte]
- Function code [1 Byte]. Allowed range is 1 to 127 (in decimal).
- Payload data [0 to 252 Bytes]
- CRC [2 Bytes]. It is a Cyclic Redundancy Check code, for error checking of the message

The response from the client is similar, but with other payload data.

Function code (in decimal)	Payload data to slave (Request)	Payload data from slave (Response)
1 Read bits (coils)	Start address [2 Bytes] Number of coils [2 Bytes]	Byte count [1 Byte] Value [k Bytes]
2 Read discrete inputs	Start address [2 Bytes] Number of inputs [2 Bytes]	Byte count [1 Byte] Value [k Bytes]
3 Read holding registers	Start address [2 Bytes] Number of registers [2 Bytes]	Byte count [1 Byte] Value [n*2 Bytes]
4 Read input registers	Start address [2 Bytes] Number of registers [2 Bytes]	Byte count [1 Byte] Value [n*2 Bytes]
5 Write single bit (coil)	Output address [2 Bytes] Value [2 Bytes]	Output address [2 Bytes] Value [2 Bytes]
6 Write single register	Register address [2 Bytes] Value [2 Bytes]	Register address [2 Bytes] Value [2 Bytes]
15 Write multiple bits (coils)	Start address [2 Bytes] Number of outputs [2 Bytes] Byte count [1 Byte] Value [k Bytes]	Start address [2 Bytes] Number of outputs [2 Bytes]
16 Write multiple registers	Start address [2 Bytes] Number of registers [2 Bytes] Byte count [1 Byte] Value [n*2 Bytes]	Start address [2 Bytes] Number of regist [2 Bytes]
23 Read and write multiple registers	?	?

For function code 5, the only valid values are 0000 (hex) or FF00 (hex), representing OFF and ON respectively.

It is seen in the table above that the request and response messages are similar for function code 1 to 4. The same can be said about function code 5 and 6, and also about 15 and 16.

For finding how the k Bytes for the value relates to the number of registers etc (n), see the Modbus documents referred to above.

5.6 Reading individual bits from a 16-bit register

Some manufacturers use 16-bit registers to store individual boolean values (bits), so with a single read from a single address, 16 booleans could be retrieved. This is sometimes called a flag register.

You need to read the register as an integer, and then extract the bit you are interested in. For example to extract the third bit from right:

```
registervalue = instrument.read_register(4143)
is_my_bit_set = (registervalue & 0b0000000000000100) > 0
```

or if using hexadecimal numbers in your code instead:

```
is_my_bit_set = (registervalue & 0x0004) > 0
```

More information on bit manipulation in Python, see the “Single bits” section of <https://wiki.python.org/moin/BitManipulation>

5.7 Known deviations from the standard

Some instruments:

- sets more than one bit in the response when one bit is requested.
- add an extra 0xFE byte after some messages.

5.8 MODBUS ASCII format

This driver also supports Modbus ASCII mode.

Basically, a byte with value 0-255 in Modbus RTU mode will in Modbus ASCII mode be sent as two characters corresponding to the hex value of that byte.

For example a value of 76 (dec) = 4C (hex) is sent as the byte 0x4C in Modbus RTU mode. This byte happens to correspond to the character ‘L’ in the ASCII encoding. Thus for Modbus RTU this is sent: ‘\x4C’, which is a string of length 1 and will print as ‘L’.

The same value will in Modbus ASCII be sent as the string ‘4C’, which has a length of 2.

The frame format is slightly different for Modbus ASCII. The request message is sent from the master in this format:

- Start [1 character]. It is the colon (:).
- Slave Address [2 characters]
- Function code [2 characters]
- Payload data [0 to 2*252 characters]

- LRC [2 characters]. The LRC is a Longitudinal Redundancy Check code, for error checking of the message.
- Stop [2 characters]. The stop characters are carriage return ('`\r`' = '`\x0D`') and line feed ('`\n`' = '`\x0A`').

5.9 Manual testing of Modbus equipment

Look in your equipment's manual to find working communication examples.

You can make a small Python program to test the communication:

```

TODO: Change this to a RTU example

import serial
ser = serial.Serial('/dev/ttyUSB0', 19200, timeout=1)
print(ser)

ser.write(':010310010001EA\r\n')
print(repr(ser.read(1000))) # Read 1000 bytes, or wait for timeout

```

It should print something like:

```

Serial<id=0x9faa08c, open=True>(port='/dev/ttyUSB0', baudrate=19200, bytesize=8,
↳parity='N', stopbits=1, timeout=1, xonxoff=False, rtscts=False, dsrdtr=False)
:0103020136C3

```

Correspondingly for Modbus ASCII, change the write command to for example:

```

TODO: Verify

ser.write(':010310010001EA\r\n')

```

It should then print something like:

```

Serial<id=0x9faa08c, open=True>(port='/dev/ttyUSB0', baudrate=19200, bytesize=8,
↳parity='N', stopbits=1, timeout=1, xonxoff=False, rtscts=False, dsrdtr=False)
:0103020136C3

```

It is also easy to test Modbus ASCII equipment from Linux command line. First must the appropriate serial port be set up properly:

- Print port settings: `stty -F /dev/ttyUSB0`
- Print all settings for a port: `stty -F /dev/ttyUSB0 -a`
- Reset port to default values: `stty -F /dev/ttyUSB0 sane`
- Change port to raw behavior: `stty -F /dev/ttyUSB0 raw`
- and: `stty -F /dev/ttyUSB0 -echo -echoe -echok`
- Change port baudrate: `stty -F /dev/ttyUSB0 19200`

To send out a Modbus ASCII request (read register 0x1001 on slave 1), and print out the response:

```

cat /dev/ttyUSB0 &
echo -e ":010310010001EA\r\n" > /dev/ttyUSB0

```

The response will be something like:

```
:0103020136C3
```


6.1 Timing of the serial communications

The Modbus RTU standard prescribes a silent period corresponding to 3.5 characters between each message, to be able to figure out where one message ends and the next one starts.

The silent period after the message to the slave is the responsibility of the slave.

The silent period after the message from the slave has previously been implemented in MinimalModbus by setting a generous timeout value, and let the serial `read()` function wait for timeout.

The character time corresponds to 11 bit times, according to <https://www.automation.com/library/articles-white-papers/fieldbus-serial-bus-io-networks/introduction-to-modbus>.

According to the Modbus RTU standard, the minimum silent period should be 1.75 ms regardless of the baud rate.

Baud rate	Bit rate	Bit time	Character time	3.5 character times
2400	2400 bits/s	417 us	4.6 ms	16 ms
4800	4800 bits/s	208 us	2.3 ms	8.0 ms
9600	9600 bits/s	104 us	1.2 ms	4.0 ms
19200	19200 bits/s	52 us	573 us	2.0 ms
38400	38400 bits/s	26 us	286 us	1.75 ms (1.0 ms)
115200	115200 bit/s	8.7 us	95 us	1.75 ms (0.33 ms)

6.2 RS-485 introduction

Several nodes (instruments) can be connected to one RS485 bus. The bus consists of two lines, A and B, carrying differential voltages. In both ends of the bus, a 120 Ohm termination resistor is connected between line A and B. Most often a common ground line is connected between the nodes as well.

At idle, both line A and B rest at the same voltage (or almost the same voltage). When a logic 1 is transmitted, line A is pulled towards lower voltage and line B is pulled towards higher voltage. Note that the A/B naming is sometimes

mixed up by some manufacturers.

Each node uses a transceiver chip, containing a transmitter (sender) and a receiver. Only one transmitter can be active on the bus simultaneously.

Pins on the RS485 bus side of the transceiver chip:

- A: inverting line
- B: non-inverting line
- GND

Pins on the microcontroller side of the transceiver chip:

- TX: Data to be transmitted
- TXENABLE: For enabling/disabling the transmitter
- RX: Received data
- RXENABLE: For enabling/disabling the receiver

If the receiver is enabled simultaneously with the transmitter, the sent data is echoed back to the microcontroller. This echo functionality is sometimes useful, but most often the TXENABLE and RXENABLE pins are connected in such a way that the receiver is disabled when the transmitter is active.

For detailed information, see <https://en.wikipedia.org/wiki/RS-485>.

6.3 Controlling the RS485 transmitter

Controlling the TXENABLE pin on the transceiver chip is the tricky part when it comes to RS485 communication. There are some options:

Using a USB-to-serial conversion chip that is capable of setting the TXENABLE pin properly See for example the FTDI chip [FT232RL](#), which has a separate output for this purpose (TXDEN in their terminology). The Sparkfun breakout board [BOB-09822](#) combines this FTDI chip with a RS485 transceiver chip. The TXDEN output from the FTDI chip is high (+5 V) when the transmitter is to be activated. The FTDI chip calculates when the transmitter should be activated, so you do not have to do anything in your application software.

Using a RS232-to-RS485 converter capable of figuring out this by itself This typically requires a microcontroller in the converter, and that you configure the baud rate, stop bits etc. This is a straight-forward and easy-to-use alternative, as you can use it together with a standard USB-to-RS232 cable and nothing needs to be done in your application software. One example of this type of converter is [Westermo MDW-45](#), which I have been using with great success.

Using a converter where the TXENABLE pin is controlled by the TX pin, sometimes via some timer circuit I am not convinced that it is a good idea to control the TXENABLE pin by the TX pin, as only one of the logic levels are actively driving the bus voltage. If using a timer circuit, the hardware needs to be adjusted to the baudrate.

Have the transmitter constantly enabled Some users have been reporting on success for this strategy. The problem is that the master and slaves have their transmitters enabled simultaneously. I guess for certain situations (and being lucky with the transceiver chip) it might work. Note that you will receive your own transmitted message (local echo). See [minimalmodbus.Instrument](#) for echo details.

Controlling a separate GPIO pin from kernelspace software on embedded Linux machines See for example <https://blog.savoirfairelinux.com/en-ca/2013/rs-485-for-beaglebone-a-quick-peek-at-the-omap-uart/> This is a very elegant solution, as the TXENABLE pin is controlled by the kernel driver and you don't have to worry about it in your application program. Unfortunately this is not available for all boards, for example the standard distribution for Beaglebone (September 2014).

Controlling a separate GPIO pin from userspace software on embedded Linux machines This will give large time delays, but might be acceptable for low speeds. It will probably take 1-3 ms to turn off the transceiver. For this to fit in the 3.5 character time before the slave starts transmitting, max speed can be 9600 bps.

Controlling the RTS pin in the RS232 interface (from userspace), and connecting it to the TXENABLE pin of the transceiver
This will give large time delays, but might be acceptable for low speeds.

6.4 Controlling the RS-485 transceiver from userspace

As described above, this should be avoided. Nevertheless, for low speeds (maybe up to 9600 bits/s) it might be useful.

This can be done from userspace, but will then lead to large time delays. I have tested this with a 3.3V FTDI USB-to-serial cable using pySerial on a Linux laptop. The cable has a RTS output, but no TXDEN output. Note that the RTS output is +3.3 V at idle, and 0 V when RTS is set to True. The delay time is around 1 ms, as measured with an oscilloscope. This corresponds to approx 100 bit times when running at 115200 bps, but this value also includes delays caused by the Python interpreter.

7.1 Debug mode

To switch on the debug mode, where the communication details are printed:

```
#!/usr/bin/env python3
import minimalmodbus

instrument = minimalmodbus.Instrument('/dev/ttyUSB1', 1, debug = True)
print(instrument.read_register(289, 1))
```

With this you can easily see what is sent to and from your instrument, and immediately see what is wrong. This is very useful also if developing your own Modbus compatible electronic instruments.

Similar in interactive mode:

```
>>> instrument.read_register(4097,1)
MinimalModbus debug mode. Writing to instrument: '\n\x03\x10\x01\x00\x01\xd0q'
MinimalModbus debug mode. Response from instrument: '\n\x03\x02\x07\xd0\x1e)'
200.0
```

The data is stored internally in this driver as byte strings (representing byte values). For example a byte with value 18 (dec) = 12 (hex) = 00010010 (bin) is stored in a string of length one. This can be created using the function `chr(18)`, or by simply typing the string `'\x12'` (which is a string of length 1). See https://docs.python.org/2/reference/lexical_analysis.html#string-literals for details on escape sequences.

For more information about hexadecimal numbers, see <https://en.wikipedia.org/wiki/Hexadecimal>.

Note that the letter A has the hexadecimal ASCII code 41, why the string `'\x41'` prints 'A'. The Latin-1 encoding is used (on most installations?), and the conversion table is found on https://en.wikipedia.org/wiki/Latin_1.

The byte strings can look pretty strange when printed, as values 0 to 31 (dec) are ASCII control signs (not corresponding to any letter). For example 'vertical tab' and 'line feed' are among those. To make the output easier to understand, print the representation, `repr()`. Use:

```
print(repr(bytestringname))
```

Registers are 16 bit wide (2 bytes), and the data is sent with the most significant byte (MSB) before the least significant byte (LSB). This is called big-endian byte order. To find the register data value, multiply the MSB by 256 (dec) and add the LSB.

Error checking is done using CRC (cyclic redundancy check), and the result is two bytes.

7.1.1 Example

We use this example in debug mode. It reads one register (number 5) and interpret the data as having 1 decimal. The slave has address 1 (as set when creating the `instrument` instance), and we are using MODBUS function code 3 (the default value for `read_register()`):

```
>>> instrument.read_register(5,1)
```

This will be displayed:

```
MinimalModbus debug mode. Writing to instrument: '\x01\x03\x00\x05\x00\x01\x94\x0b'
```

In the section ‘Modbus implementation details’ above, the request message structure is described. See the table entry for function code 3.

Interpret the request message (8 bytes) as:

Displayed	Hex	Dec	Description
\x01	01	1	Slave address (here 1)
\x03	03	3	Function code (here 3 = read registers)
\x00	00	0	Start address MSB
\x05	05	5	Start address LSB
\x00	00	0	Number of registers MSB
\x01	01	1	Number of registers LSB
\x94	94	148	CRC LSB
\x0b	0b	11	CRC MSB

So the data in the request is:

- Start address: $0 * 256 + 5 = 5$ (dec)
- Number of registers: $0 * 256 + 1 = 1$ (dec)

The response will be displayed as:

```
MinimalModbus debug mode. Response from instrument: '\x01\x03\x02\x00°9÷'
```

Interpret the response message (7 bytes) as:

Displayed	Hex	Dec	Description
\x01	01	1	Slave address (here 1)
\x03	03	3	Function code (here 3 = read registers)
\x02	02	2	Byte count
\x00	00	0	Value MSB
°	ba	186	Value LSB
9	37	57	CRC LSB
÷	f7	247	CRC MSB

Out of the response, this is the payload part: `\x02\x00°` (3 bytes)

So the data in the response is:

- Byte count: 2 (dec)
- Register value: $0 * 256 + 186 = 186$ (dec)

We know since earlier that this instrument stores a temperature of 18.6 C as 186. We provide this information as the second argument in the function call `read_register(5, 1)`, why it automatically divides the register data by 10 and returns 18.6.

7.1.2 Special characters

Some ASCII control characters have representations like `\n`, and their meanings are described in this table:

<code>repr()</code> shows as	Can be written as	ASCII hex	ASCII dec	Description
<code>\t</code>	<code>\x09</code>	09	9	Horizontal Tab (TAB)
<code>\n</code>	<code>\x0a</code>	0a	10	Linefeed (LF)
<code>\r</code>	<code>\x0d</code>	0d	13	Carriage Return (CR)

It is also possible to write for example ASCII Bell (BEL, hex = 07, dec = 7) as `\a`, but its `repr()` will still print `\x07`.

More about ASCII control characters is found on <https://en.wikipedia.org/wiki/ASCII>.

8.1 No communication

If there is no communication, make sure that the settings on your instrument are OK:

- Wiring is correct
- Communication module is set for digital communication
- Correct protocol (Modbus, and the RTU or ASCII version)
- Baud rate
- Parity
- Delay (most often not necessary)
- Address

The corresponding settings should also be used in `MinimalModbus`. Check also your:

- Port name

For troubleshooting, it is recommended to use interactive mode with debug enabled. See *Interactive usage*.

If there is no response from your instrument, you can try using a lower baud rate, or to adjust the timeout setting.

See also the pySerial pages: <https://pyserial.readthedocs.io>

To make sure you are sending something valid, start with the examples in the users manual of your instrument. Use `MinimalModbus` in debug mode and make sure that each sent byte is correct.

The termination resistors of the RS-485 bus must be set correctly. Use a multimeter to verify that there is termination in the appropriate nodes of your RS-485 bus.

To troubleshoot the communication in more detail, an oscilloscope can be very useful to verify transmitted data.

8.2 Local echo

Local echo of the USB-to-RS485 adaptor can also be the cause of some problems, and give rise to strange error messages (like “CRC error” or “wrong number of bytes error” etc). Switch on the debug mode to see the request and response messages. If the full request message can be found as the first part of the response, then local echo is likely the cause.

Make a test to remove the adaptor from the instrument (but still connected to the computer), and see if you still have a response.

Most adaptors have switches to select echo ON/OFF. Turning off the local echo can be done in a number of ways:

- A DIP-switch inside the plastic cover.
- A jumper inside the plastic cover.
- Shorting two of the pins in the 9-pole D-SUB connector turns off the echo for some models.
- If based on a FTDI chip, some special program can be used to change a chip setting for disabling echo.

To handle local echo, see *minimalmodbus.Instrument*.

8.3 Empty bytes added in the beginning or the end on the received message

This is due to interference. Use biasing of modbus lines, by connecting resistors to GND and Vcc from the the two lines. This is sometimes named “failsafe”.

8.4 Serial adaptors not recognized

There have been reports on problems with serial adaptors on some platforms, for example Raspberry Pi. It seems to lack kernel drives for some chips, like PL2303. Serial adaptors based on FTDI FT232RL are known to work.

Make sure to run the `dmesg` command before and after plugging in your serial adaptor, to verify that the proper kernel driver is loaded.

8.5 Known issues

See GitHub page.

8.6 Issues when running under Windows

Since MinimalModbus version 0.5, the handling of several instruments on the same serial port has been improved for Windows.

8.7 Support

Ask a question on <https://stackoverflow.com/>. Use the tags “modbus”, “python” and “minimalmodbus”.

Describe the problem in detail, and include any error messages.

Note that it can be very helpful to switch on the debug mode, where the communication details are printed. See *Debug mode*.

Please also include the output after running:

```
>>> import minimalmodbus
>>> print(minimalmodbus._get_diagnostic_string())
```

Describe which instrument model you are using, and possibly a link to online PDF documentation for it.

Detailed usage documentation

For introductory usage documentation, see *Usage*.

9.1 Interactive usage

To use interactive mode, start the Python interpreter and import `minimalmodbus`:

```
>>> import minimalmodbus
>>> instr = minimalmodbus.Instrument('/dev/ttyUSB0', 1)
>>> instr
minimalmodbus.Instrument<id=0xb7437b2c, address=1, close_port_after_each_call=False,
↳ debug=False, serial=Serial<id=0xb7437b6c, open=True>(port='/dev/ttyUSB0',
↳ baudrate=19200, bytesize=8, parity='N', stopbits=1, timeout=0.05, xonxoff=False,
↳ rtscts=False, dsrdtr=False)>
>>> instr.read_register(24, 1)
5.0
>>> instr.write_register(24, 450, 1)
>>> instr.read_register(24, 1)
450.0
```

Note that when you call a function, in interactive mode the representation of the return value is printed. The representation is kind of a debug information, like seen here for the returned string (example from Omega CN7500 driver, which previously was included in this package):

```
>>> instrument.get_all_pattern_variables(0)
'SP0: 10.0 Time0: 10\nSP1: 20.0 Time1: 20\nSP2: 30.0 Time2: 30\nSP3: 333.3 Time3:
↳ 45\nSP4: 50.0 Time4: 50\nSP5: 60.0 Time5: 60\nSP6: 70.0 Time6: 70\nSP7: 80.0
↳ Time7: 80\nActual step: 7\nAdditional cycles: 4\nLinked pattern: 1\n'
```

To see how the string look when printed, use instead:

```
>>> print(instrument.get_all_pattern_variables(0))
SP0: 10.0 Time0: 10
```

(continues on next page)

(continued from previous page)

```

SP1: 20.0  Time1: 20
SP2: 30.0  Time2: 30
SP3: 333.3  Time3: 45
SP4: 50.0  Time4: 50
SP5: 60.0  Time5: 60
SP6: 70.0  Time6: 70
SP7: 80.0  Time7: 80
Actual step:      7
Additional cycles: 4
Linked pattern:   1

```

It is possible to show the representation also when printing, if you use the function `repr()`:

```

>>> print(repr(instrument.get_all_pattern_variables(0)))
'SP0: 10.0  Time0: 10\nSP1: 20.0  Time1: 20\nSP2: 30.0  Time2: 30\nSP3: 333.3  Time3:
↪45\nSP4: 50.0  Time4: 50\nSP5: 60.0  Time5: 60\nSP6: 70.0  Time6: 70\nSP7: 80.0
↪Time7: 80\nActual step:      7\nAdditional cycles:  4\nLinked pattern:    1\n'

```

In case of problems using MinimalModbus, it is useful to switch on the debug mode to see the communication details:

```

>>> instr.debug = True
>>> instr.read_register(24, 1)
MinimalModbus debug mode. Writing to instrument: '\x01\x03\x00\x18\x00\x01\x04\r'
MinimalModbus debug mode. Response from instrument: '\x01\x03\x02\x11\x94µ'
450.0

```

9.2 Making drivers for specific instruments

With proper instrument drivers you can use commands like `getTemperatureCenter()` in your code instead of `read_register(289, 1)`. So the driver is a basically a collection of numerical constants to make your code more readable.

This segment is part of the example driver `eurotherm3500` which previously was included in this distribution:

```

import minimalmodbus

class Eurotherm3500( minimalmodbus.Instrument ):
    """Instrument class for Eurotherm 3500 process controller.

    Args:
        * portname (str): port name
        * slaveaddress (int): slave address in the range 1 to 247

    """

    def __init__(self, portname, slaveaddress):
        minimalmodbus.Instrument.__init__(self, portname, slaveaddress)

    def get_pv_loop1(self):
        """Return the process value (PV) for loop1."""
        return self.read_register(289, 1)

    def is_manual_loop1(self):
        """Return True if loop1 is in manual mode."""

```

(continues on next page)

(continued from previous page)

```

    return self.read_register(273, 1) > 0

def get_sptarget_loop1(self):
    """Return the setpoint (SP) target for loop1."""
    return self.read_register(2, 1)

def get_sp_loop1(self):
    """Return the (working) setpoint (SP) for loop1."""
    return self.read_register(5, 1)

def set_sp_loop1(self, value):
    """Set the SP1 for loop1.

    Note that this is not necessarily the working setpoint.

    Args:
        value (float): Setpoint (most often in degrees)
    """
    self.write_register(24, value, 1)

def disable_sprate_loop1(self):
    """Disable the setpoint (SP) change rate for loop1. """
    VALUE = 1
    self.write_register(78, VALUE, 0)

```

To get the process value (PV from loop1):

```

#!/usr/bin/env python3
import eurotherm3500

heatercontroller = eurotherm3500.Eurotherm3500('/dev/ttyUSB1', 1) # port name, slave_
↳address

## Read temperature (PV) ##
temperature = heatercontroller.get_pv_loop1()
print(temperature)

## Change temperature setpoint (SP) ##
NEW_TEMPERATURE = 95.0
heatercontroller.set_sp_loop1(NEW_TEMPERATURE)

```

Note that I have one additional driver layer on top of `eurotherm3500` (which is one layer on top of `minimalmodbus`). I use this process controller to run a heater, so I have a driver `heater.py` in which all my settings are done.

The idea is that `minimalmodbus` should be useful to most Modbus users, and `eurotherm3500` should be useful to most users of that controller type. So my `heater.py` driver has functions like `getTemperatureCenter()` and `getTemperatureEdge()`, and there I also define resistance values etc.

Here is a part of `heater.py`:

```

"""Driver for the heater in the CVD system. Talks to the heater controller and the_
↳heater policeman.

Implemented with the modules :mod:`eurotherm3500` and :mod:`eurotherm3216i`.

"""

```

(continues on next page)

(continued from previous page)

```

import eurotherm3500
import eurotherm3216i

class heater():
    """Class for the heater in the CVD system. Talks to the heater controller and the
    ↪heater policeman.

    """

    ADDRESS_HEATERCONTROLLER = 1
    """Modbus address for the heater controller."""

    ADDRESS_POLICEMAN = 2
    """Modbus address for the heater over-temperature protection unit."""

    SUPPLY_VOLTAGE = 230
    """Supply voltage (V)."""

    def __init__(self, port):
        self.heatercontroller = eurotherm3500.Eurotherm3500( port, self.ADDRESS_
        ↪HEATERCONTROLLER)
        self.policeman = eurotherm3216i.Eurotherm3216i( port, self.ADDRESS_
        ↪POLICEMAN)

    def getTemperatureCenter(self):
        """Return the temperature (in deg C)."""
        return self.heatercontroller.get_pv_loop1()

    def getTemperatureEdge(self):
        """Return the temperature (in deg C) for the edge heater zone."""
        return self.heatercontroller.get_pv_loop2()

    def getTemperaturePolice(self):
        """Return the temperature (in deg C) for the overtemperature protection
        ↪sensor."""
        return self.policeman.get_pv()

    def getOutputCenter(self):
        """Return the output (in %) for the heater center zone."""
        return self.heatercontroller.get_op_loop1()

```

9.3 Using this module as part of a measurement system

It is very useful to make a graphical user interface (GUI) for your control/measurement program.

One library for making GUIs is wxPython, found on <https://www.wxpython.org/>. One good tutorial (it starts from the basics) is: <http://zetcode.com/wxpython/>

I strongly suggest that your measurement program should be possible to run without any GUI, as it then is much easier to actually get the GUI version of it to work. Your program should have some function like `setTemperature(255)`.

The role of the GUI is this: If you have a temperature text box where a user has entered 255 (possibly degrees C), and a button 'Run!' or 'Go!' or something similar, then the GUI program should read 255 from the box when the user presses the button, and call the function `setTemperature(255)`.

This way it is easy to test the measurement program and the GUI separately.

9.4 Handling extra 0xFE byte after some messages

Some users have reported errors due to instruments not fulfilling the Modbus standard. For example can some additional byte be pasted at the end of the response from the instrument. Here is an example how this can be handled by tweaking the `minimalmodbus.py` file.

Add this to `_extract_payload()` function, after the argument validity testing section:

```
# Fix for broken T3-PT10 which outputs extra 0xFE byte after some messages
# Patch by Edwin van den Oetelaar
# check length of message when functioncode in 3,4
# if received buffer length longer than expected, truncate it,
# this makes sure CRC bytes are taken from right place, not the end of the buffer, it_
↳ignores the extra bytes in the buffer
if functioncode in (0x03, 0x04) :
    try:
        modbuslen = ord(response[NUMBER_OF_RESPONSE_STARTBYTES])
        response = response[:modbuslen+5] # the number of bytes used for CRC(2),
↳slaveid(1),functioncode(1),bytecount(1) = 5
    except IndexError:
        pass
```

9.5 Install or uninstalling a distribution

To install a python (downloaded) package, uncompress it and use:

```
sudo python3 setup.py install
```

On a development machine, go to the package top directory before running the command.

9.5.1 Uninstall

Pip-installed packages can be uninstalled with:

```
sudo pip3 uninstall minimalmodbus
```

9.5.2 Show versions of all installed packages

Use:

```
pip3 freeze
```

9.5.3 Installation target

The location of the installed files is seen in the `_get_diagnostic_string()` output:

```
import minimalmodbus
print(minimalmodbus._get_diagnostic_string())
```

On Linux machines, for example:

```
/usr/local/lib/python2.6/dist-packages
```

On OS X it might end up in for example:

```
/Library/Python/2.6/site-packages/minimalmodbus.py
```

Note that `.pyc` is a byte compiled version. Make the changes in the `.py` file, and delete the `.pyc` file (When available, `.pyc` files are used instead of `.py` files). You might need root privileges to edit the file in this location. Otherwise it is better to uninstall it, put it instead in your home folder and add it to `sys.path`

On Windows machines, for example:

```
C:\python27\Lib\site-packages
```

The Windows installer also creates a `.pyo` file (and also the `.pyc` file).

9.5.4 Python location

Python location on Linux machines:

```
/usr/lib/python2.7/
/usr/lib/python2.7/dist-packages
```

To find locations:

```
~$ which python
/usr/bin/python
~$ which python3
/usr/bin/python3
~$ which python2.7
/usr/bin/python2.7
~$ which python3.2
/usr/bin/python3.2
```

To see which python version that is used:

```
python3 --version
```

9.6 Setting the PYTHONPATH

To set the path:

```
echo $PYTHONPATH
export PYTHONPATH='/home/jonas/pythonprogramming/minimalmodbus/trunk'
```

or:

```
export PYTHONPATH=$PYTHONPATH:/home/jonas/pythonprogramming/minimalmodbus/trunk
```

It is better to set the path in the `.basrc` file.

9.7 Including MinimalModbus in a Yocto build

It is easy to include MinimalModbus in a Yocto build, which is using Bitbake. Yocto is a collaboration with the Open Embedded initiative.

In your layer, create the file `recipes-connectivity/minimalmodbus/python-minimalmodbus_0.5.bb`.

It's content should be:

```
SUMMARY = "Easy-to-use Modbus RTU and Modbus ASCII implementation for Python"
SECTION = "devel/python"
LICENSE = "Apache-2.0"
LIC_FILES_CHKSUM = "file://LICENCE.txt;md5=27da4ba4e954f7f4ba8d1e08a2c756c4"

DEPENDS = "python"
RDEPENDS_${PN} = "python-pyserial"

PR = "r0"

SRC_URI = "${SOURCEFORGE_MIRROR}/project/minimalmodbus/${PV}/MinimalModbus-${PV}.tar.
↳gz"

SRC_URI[md5sum] = "1b2ec44e9537e14dcb8a238ea3eda451"
SRC_URI[sha256sum] = "d9acf6457bc26d3c784caa5d7589303afe95e980ceff860ec2a4051038bc261e
↳"

S = "${WORKDIR}/MinimalModbus-${PV}"

inherit distutils
```

You also need to add this to your `local.conf` file:

```
IMAGE_INSTALL_append = " python-minimalmodbus"
```

When using the recipe for another version of MinimalModbus, change the version number in the filename. Bitbake will complain that the `md5sum` and `sha256sum` not are correct, but Bitbake will print out the correct values so you can change the recipe accordingly.

The details printed in debug mode (requests and responses) are very useful for using the included `dummy_serial` port for unit testing purposes. For examples, see the file `test/test_minimalmodbus.py`.

10.1 Design considerations

My take on the design is that it should be as simple as possible, hence the name `MinimalModbus`, but it should implement the smallest number of functions needed for it to be useful. The target audience for this driver simply wants to talk to Modbus clients using a serial interface using some simple driver.

Only a few functions are implemented. It is very easy to implement lots of (seldom used) functions, resulting in buggy code with large fractions of it almost never used. It is instead much better to implement the features when needed/requested. There are many Modbus function codes, but I guess that most are not used.

It is a goal that the same driver should be compatible for both Python2 and Python3 programs. Some suggestions for making this possible are found here: <https://wiki.python.org/moin/PortingPythonToPy3k>

There should be unittests for all functions, and mock communication data.

Errors should be caught as early as possible, and the error messages should be informative. For this reason there is type checking for the parameters in most functions. This is rather un-pythonic, but is intended to give more clear error messages (for easier remote support).

Note that the term 'address' is ambiguous, why it is better to use the terms 'register address' or 'slave address'.

Use only external links in the `README.txt`, otherwise they will not work on Python Package Index (PyPI). No Sphinx-specific constructs are allowed in that file.

Design priorities:

- Easy to use
- Catch errors early
- Informative error messages
- Good unittest coverage

- Same codebase for Python2 and Python3

10.2 General driver structure

The general structure of the program is shown here:

Function	Description
<code>read_register()</code>	One of the facades for <code>_generic_command()</code> .
<code>_generic_command()</code>	Generates payload, then calls <code>_perform_command()</code> .
<code>_perform_command()</code>	Embeds payload into error-checking codes etc, then calls <code>_communicate()</code> .
<code>_communicate()</code>	Handles raw strings for communication via pySerial.

Most of the logic is located in separate (easy to test) functions on module level. For a description of them, see *Internal documentation for MinimalModbus*.

10.3 Number conversion to and from bytestrings

The Python module `struct` is used for conversion. See <https://docs.python.org/2/library/struct.html>

Several wrapper functions are defined for easy use of the conversion. These functions also do argument validity checking.

Data type	To bytestring	From bytestring
(internal usage)	<code>_num_to_onebyte_string()</code>	
Bit	<code>_bit_to_bytestring()</code>	Same as for bits
Several bits	<code>_bits_to_bytestring()</code>	<code>_bytestring_to_bits()</code>
Integer (char, short)	<code>_num_to_twobyte_string()</code>	<code>_twobyte_string_to_num()</code>
Several registers	<code>_valuelist_to_bytestring()</code>	<code>_bytestring_to_valuelist()</code>
Long integer	<code>_long_to_bytestring()</code>	<code>_bytestring_to_long()</code>
Floating point number	<code>_float_to_bytestring()</code>	<code>_bytestring_to_float()</code>
String	<code>_textstring_to_bytestring()</code>	<code>_bytestring_to_textstring()</code>

Note that the `struct` module produces byte buffers for Python3, but bytestrings for Python2. This is compensated for automatically by using the wrapper functions `_pack()` and `_unpack()`.

For a description of them, see *Internal documentation for MinimalModbus*.

10.4 Unit testing

Unit tests are provided in the tests subfolder. To run them:

```
make test
```

The unittests uses previously recorded communication data for the testing.

A dummy/mock/stub for the serial port, `dummy_serial`, is provided for test purposes. See *Documentation for dummy_serial (which is a serial port mock)*.

The test coverage analysis is found at <https://codecov.io/github/pyhys/minimalmodbus?branch=master>.

To automatically run the tests for the different Python versions:

```
make test_all
```

It will use the tox tool to run the tests, and the settings are done in the `tox.ini` file. Note that Travis CI does not use tox (it has its own config file).

Hardware tests are performed using a Delta DTB4824 process controller together with a USB-to-RS485 converter. See *Internal documentation for hardware testing of MinimalModbus using DTB4824* for more information.

Run it with:

```
python3 tests/test_deltaDTB4824.py
```

The baudrate, portname and mode can optionally be set from command line:

```
python3 tests/test_deltaDTB4824.py -b19200 -D/dev/ttyUSB0 -ascii
```

For more details on testing with this hardware, see *Internal documentation for hardware testing of MinimalModbus using DTB4824*.

10.5 Making sure that error messages are informative for the user

To have a look on the error messages raised during unit testing of `minimalmodbus`, monkey-patch `test_minimalmodbus.SHOW_ERROR_MESSAGES_FOR_ASSERTRAISES` as seen here:

```
>>> import unittest
>>> import test_minimalmodbus
>>> test_minimalmodbus.SHOW_ERROR_MESSAGES_FOR_ASSERTRAISES = True
>>> suite = unittest.TestLoader().loadTestsFromModule(test_minimalmodbus)
>>> unittest.TextTestRunner(verbosity=2).run(suite)
```

This is part of the output:

```
testFunctioncodeNotInteger (test_minimalmodbus.TestEmbedPayload) ...
  TypeError('The functioncode must be an integer. Given: 1.0',)

  TypeError("The functioncode must be an integer. Given: '1'",)

  TypeError('The functioncode must be an integer. Given: [1]',)

  TypeError('The functioncode must be an integer. Given: None',)
ok
testKnownValues (test_minimalmodbus.TestEmbedPayload) ... ok
testPayloadNotString (test_minimalmodbus.TestEmbedPayload) ...
  TypeError('The payload should be a string. Given: 1',)

  TypeError('The payload should be a string. Given: 1.0',)

  TypeError("The payload should be a string. Given: ['ABC']",)

  TypeError('The payload should be a string. Given: None',)
ok
testSlaveaddressNotInteger (test_minimalmodbus.TestEmbedPayload) ...
  TypeError('The slaveaddress must be an integer. Given: 1.0',)

  TypeError("The slaveaddress must be an integer. Given: 'DEF'",)
```

(continues on next page)

(continued from previous page)

```

ok
testWrongFunctioncodeValue (test_minimalmodbus.TestEmbedPayload) ...
    ValueError('The functioncode is too large: 222, but maximum value is 127.',)

    ValueError('The functioncode is too small: -1, but minimum value is 1.',)
ok
testWrongSlaveaddressValue (test_minimalmodbus.TestEmbedPayload) ...
    ValueError('The slaveaddress is too large: 248, but maximum value is 247.',)

    ValueError('The slaveaddress is too small: -1, but minimum value is 0.',)
ok

```

See `test_minimalmodbus` for details on how this is implemented.

It is possible to run just a few tests. To load a single class of test cases:

```
suite = unittest.TestLoader().loadTestsFromTestCase(test_minimalmodbus.TestSetBitOn)
```

If necessary:

```
reload(test_minimalmodbus.minimalmodbus)
```

10.6 Recording communication data for unittesting

With the known data output from an instrument, we can finetune the inner details of the driver (code refactoring) without worrying that we change the output from the code. This data will be the ‘golden standard’ to which we test the code. Use as many as possible of the commands, and paste all the output in a text document. From this it is pretty easy to reshuffle it into unittest code.

Here is an example how to record communication data, which then is pasted into the test code (for use with a mock/dummy serial port). See for example *Internal documentation for unit testing of MinimalModbus* (click ‘[source]’ on right side, see RESPONSES at end of the page). Do like this:

```

>>> import minimalmodbus
>>> instrument_1 = minimalmodbus.Instrument('/dev/ttyUSB0',10)
>>> instrument_1.debug = True
>>> instrument_1.read_register(4097,1)
MinimalModbus debug mode. Writing to instrument: '\n\x03\x10\x01\x00\x01\xd0q'
MinimalModbus debug mode. Response from instrument: '\n\x03\x02\x07\xd0\x1e)'
200.0
>>> instrument_1.write_register(4097,325.8,1)
MinimalModbus debug mode. Writing to instrument:
↪ '\n\x10\x10\x01\x00\x01\x02\x0c\xbaA\xc3'
MinimalModbus debug mode. Response from instrument: '\n\x10\x10\x01\x00\x01U\xb2'
>>> instrument_1.read_register(4097,1)
MinimalModbus debug mode. Writing to instrument: '\n\x03\x10\x01\x00\x01\xd0q'
MinimalModbus debug mode. Response from instrument: '\n\x03\x02\x0c\xba\x996'
325.8
>>> instrument_1.read_bit(2068)
MinimalModbus debug mode. Writing to instrument: '\n\x02\x08\x14\x00\x01\xfa\xd5'
MinimalModbus debug mode. Response from instrument: '\n\x02\x01\x00\xa3\xac'
0
>>> instrument_1.write_bit(2068,1)
MinimalModbus debug mode. Writing to instrument: '\n\x05\x08\x14\xff\x00\xcf%'
MinimalModbus debug mode. Response from instrument: '\n\x05\x08\x14\xff\x00\xcf%'

```


This is also very useful for debugging drivers built on top of MinimalModbus.

10.7 Using the dummy serial port

A dummy serial port is included for testing purposes, see `dummy_serial`. Use it like this:

```
>>> import dummy_serial
>>> import test_minimalmodbus
>>> dummy_serial.RESPONSES = test_minimalmodbus.RESPONSES # Load previously recorded_
↳responses
>>> import minimalmodbus
>>> minimalmodbus.serial.Serial = dummy_serial.Serial # Monkey-patch a dummy serial_
↳port
>>> instrument = minimalmodbus.Instrument('DUMMYPORNAME', 1) # port name, slave_
↳address (in decimal)
>>> instrument.read_register(4097, 1)
823.6
```

In the example above there is recorded data available for `read_register(4097, 1)`. If no recorded data is available, an error message is displayed:

```
>>> instrument.read_register(4098, 1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/jonas/pythonprogramming/minimalmodbus/trunk/minimalmodbus.py", line_
↳174, in read_register
    return self._genericCommand(functioncode, registeraddress,
↳numberOfDecimals=numberOfDecimals)
  File "/home/jonas/pythonprogramming/minimalmodbus/trunk/minimalmodbus.py", line_
↳261, in _genericCommand
    payloadFromSlave = self._performCommand(functioncode, payloadToSlave)
  File "/home/jonas/pythonprogramming/minimalmodbus/trunk/minimalmodbus.py", line_
↳317, in _performCommand
    response = self._communicate(message)
  File "/home/jonas/pythonprogramming/minimalmodbus/trunk/minimalmodbus.py", line_
↳395, in _communicate
    raise IOError('No communication with the instrument (no answer)')
IOError: No communication with the instrument (no answer)
```

The dummy serial port can be used also with instrument drivers built on top of MinimalModbus:

```
>>> import dummy_serial
>>> import test_omegacn7500
>>> dummy_serial.RESPONSES = test_omegacn7500.RESPONSES # Load previously recorded_
↳responses
>>> import omeagcn7500
>>> omeagcn7500.minimalmodbus.serial.Serial = dummy_serial.Serial # Monkey-patch a_
↳dummy serial port
>>> instrument = omeagcn7500.OmegaCN7500('DUMMYPORNAME', 1) # port name, slave_
↳address
>>> instrument.get_pv()
24.6
```

To see the generated request data (without bothering about the response):

```

>>> import dummy_serial
>>> import minimalmodbus
>>> minimalmodbus.serial.Serial = dummy_serial.Serial # Monkey-patch a dummy serial_
↳port
>>> instrument = minimalmodbus.Instrument('DUMMYPORNAME', 1)
>>> instrument.debug = True
>>> instrument.read_bit(2068)
MinimalModbus debug mode. Writing to instrument: '\x01\x02\x08\x14\x00\x01\xfb\xae'
MinimalModbus debug mode. Response from instrument: ''

```

(Then an error message appears)

10.8 Data encoding in Python2 and Python3

The **string** type has changed in Python3 compared to Python2. In Python3 the type **bytes** is used when communicating via pySerial.

Dependent on the Python version number, the data sent from MinimalModbus to pySerial has different types.

10.8.1 String constants

This is a **string** constant both in Python2 and Python3:

```
st = 'abc\x69\xe6\x03'
```

This is a **bytes** constant in Python3, but a **string** constant in Python2 (allowed for 2.6 and higher):

```
by = b'abc\x69\xe6\x03'
```

10.8.2 Type conversion in Python3

To convert a **string** to **bytes**, use one of these:

```
bytes(st, 'latin1') # Note that 'ascii' encoding gives error for some values.
st.encode('latin1')
```

To convert **bytes** to **string**, use one of these:

```
str(by, encoding='latin1')
by.decode('latin1')
```

Encoding	Allowed range
ascii	0-127
latin-1	0-255

10.8.3 Corresponding in Python2

Ideally, we would like to use the same source code for Python2 and Python3. In Python 2.6 and higher there is the `bytes()` function for forward compatibility, but it is merely a synonym for `str()`.

To convert from **'bytes'(string)** to **string**:

```
str(by) # not possible to give encoding
by.decode('latin1') # Gives unicode
```

To convert from **string** to **'bytes'**(string):

```
bytes(st) # not possible to give encoding
st.encode('latin1') # Can not be used for values larger than 127
```

It is thus not possible to use exactly the same code for both Python2 and Python3. Where it is unavoidable, use:

```
if sys.version_info[0] > 2:
    whatever
```

10.9 Extending MinimalModbus

It is straight-forward to extend MinimalModbus to handle more Modbus function codes. Use the method `_perform_command()` to send data to the slave, and to receive the response. Note that the API might change, as this is outside the official API.

This is easily tested in interactive mode. For example the method `read_register()` generates payload, which internally is sent to the instrument using `_perform_command()`:

```
>>> instr.debug = True
>>> instr.read_register(5,1)
MinimalModbus debug mode. Writing to instrument: '\x01\x03\x00\x05\x00\x01\x94\x0b'
MinimalModbus debug mode. Response from instrument: '\x01\x03\x02\x00°9÷'
18.6
```

It is possible to use `_perform_command()` directly. You can use any Modbus function code (1-127), but you need to generate the payload yourself. Note that the same data is sent:

```
>>> instr._perform_command(3, '\x00\x05\x00\x01')
MinimalModbus debug mode. Writing to instrument: '\x01\x03\x00\x05\x00\x01\x94\x0b'
MinimalModbus debug mode. Response from instrument: '\x01\x03\x02\x00°9÷'
'\x02\x00°'
```

Use this if you are to implement other Modbus function codes, as it takes care of CRC generation etc.

10.10 Other useful internal functions

There are several useful (module level) helper functions available in the `minimalmodbus` module. The module level helper functions can be used without any hardware connected. See *Internal documentation for MinimalModbus*. These can be handy when developing your own Modbus instrument hardware.

For example:

```
>>> minimalmodbus._calculate_crc_string('\x01\x03\x00\x05\x00\x01')
'\x94\x0b'
```

And to embed the payload `'\x10\x11\x12'` to slave address 1, with functioncode 16:

```
>>> minimalmodbus._embed_payload(1, MODE_RTU, 16, '\x10\x11\x12')
'\x01\x10\x10\x11\x12\x90\x98'
```

Playing with two's complement:

```
>>> minimalmodbus._twos_complement(-1, bits=8)
255
```

Calculating the minimum silent interval (seconds) at a baudrate of 19200 bits/s:

```
>>> minimalmodbus._calculate_minimum_silent_period(19200)
0.0020052083333333332
```

Note that the API might change, as this is outside the official API.

10.11 Generate documentation

Use the top-level Make to generate HTML and PDF documentation:

```
make docs
make pdf
```

Do linkchecking and measureme test coverage:

```
make linkcheck
make coverage
```

10.12 Webpage

The HTML theme used is the Sphinx 'sphinx_rtd_theme' theme.

Note that Sphinx version 1.3 or later is required to build the documentation.

10.13 Travis CI

Each commit to GitHub is tested on the Travis CI server. Log in to <https://travis-ci.org/> using your GitHub account.

The settings are done in the `.travis.yml` file. Note that Travis CI does not use tox.

Enable the webhook from GitHub to Travis CI.

10.14 Codecov.io

Log in to <https://codecov.io/> using your Github account.

Enable the webhook from GitHub to Codecov.io.

10.15 Notes on distribution

10.15.1 Installing the module from local files

In the top directory:

```
make install
```

or during development (so you do not need to constantly re-install):

```
make installdev
```

It will add the current path to the file: `/usr/local/lib/python2.7/dist-packages/easy-install.pth`.

To uninstall it:

```
make uninstall
```

10.15.2 How to generate a source distribution from the present development code

This will create a subfolder `dist` with the source in wheel format and in `.tar.gz` format:

```
make dist
```

10.16 Preparation for release

10.16.1 Change version number etc

- Manually change the `__version__` field in the `minimalmodbus.py` source file.
- Manually change the release date in `HISTORY.rst`

(Note that the version number in the Sphinx configuration file `doc/conf.py` and in the file `setup.py` are changed automatically. Also the copyright year in `doc/conf.py` is changed automatically).

How to number releases are described in [PEP 440](#).

10.16.2 Code style checking etc

Automatically modify the formatting of the code:

```
make black
```

Check the code:

```
make lint
```

10.16.3 Unittesting

Run unit tests for all supported Python versions:

```
make test-all
```

Also make tests using Delta DTB4824 hardware. See *Internal documentation for hardware testing of MinimalModbus using DTB4824*.

Test the source distribution generation (look in the `PKG-INFO` file):

```
make dist
```

Also make sure that these are functional (see sections below):

- Documentation generation
- Test coverage report generation

10.16.4 Git

Make a tag in the git repository. See below.

10.16.5 GitHub

Releases are automatically generated on GitHub from tags in the repo.

10.16.6 Upload to PyPI

Build the source distribution and wheel, and upload to PYPI:

```
make dist
make upload
```

10.16.7 Test the documentation

Test links on the PyPI page. If adjustments are required on the PyPI page, log in and manually adjust the text. This might be for example parsing problems with the ReST text (allows no Sphinx-specific constructs).

10.16.8 Force documentation rebuild on readthedocs

Log in to <https://readthedocs.org> and force rebuild on the master branch.

Enable the “master” and “stable” documentation versions.

10.16.9 Test the installers

Make sure that the installer works, and the dependencies are handled correctly. Try at least Linux and Windows.

On windows you might need to use:

```
py -m pip install minimalmodbus
```

10.16.10 Test on hardware

Test the package on hardware from Linux and Windows. Download the file `test_deltaDTB4824.py`.

To run the hardware test on Windows:

```
C:\Python27>python.exe C:\Users\jonas\Documents\Pythonprogram\testmodbus\test_
↪deltaDTB4824.py -DCOM7 -b2400 -ascii
```

For python3 you might need to use the `py` command.

10.16.11 Begin a new development version

Check in a new version on GitHub master branch. If the previous release was `X.Y.Z`, then use `X.Y.(Z+1)a1`.

10.16.12 Backup

Burn a CD/DVD with these items:

- Source tree
- Source distributions
- Windows installer
- Generated HTML files
- PDF documentation

10.17 Useful development tools

Each of these have some additional information below on this page.

Git Version control software. See <https://git-scm.com/>

Sphinx For generating HTML documentation. See <http://www.sphinx-doc.org/>

Coverage.py Unittest coverage tool. See <https://coverage.readthedocs.io/>

PyChecker This is a tool for finding bugs in python source code. See <http://pychecker.sourceforge.net/>

pycodestyle Code style checker. See <https://github.com/PyCQA/pycodestyle#readme>

10.18 Git usage

Clone the repository from GitHub (it will create a directory):

```
git clone https://github.com/pyhys/minimalmodbus.git
```

Show details:

```
git remote -v
git status
git branch
```

Stage changes:

```
git add testb.txt
```

Commit locally:

```
git commit -m "test1"
```

Commit remotely (will ask for GitHub username and password):

```
git push origin
```

10.18.1 Git branches

Create a new branch:

```
git branch develop
```

List branches:

```
git branch
```

Change branch:

```
git checkout develop
```

Commit other branch remotely:

```
git push origin develop
```

10.18.2 Make a tag in Git

See the section on Git usage.

The release is done in the ‘master’ branch, not the ‘develop’ branch. List tags:

```
git tag
```

Make a tag in Git:

```
git tag -a 0.7 -m 'Release 0.7'
```

Show info about a tag:

```
git show 0.7
```

Commit tags to remote server:

```
git push origin --tags
```

10.19 Sphinx usage

This documentation is generated with the Sphinx tool: <http://www.sphinx-doc.org/>

It is used to automatically generate HTML documentation from docstrings in the source code. See for example *Internal documentation for MinimalModbus*. To see the source code of the Python file, click [source] on the right part of that page. To see the source of the Sphinx page definition file, click ‘View page Source’ (or possibly ‘Edit on Github’) in the upper right corner.

To install, use:


```
sudo pip3 install sphinx sphinx_rtd_theme
```

Check installed version by typing:

```
sphinx-build --version
```

10.19.1 Sphinx formatting conventions

What	Usage	Result
Inline web link	<code>`Link text <http://example.com/>`_</code>	Link text
Internal link	<code>:ref:`testminimalmodbus`</code>	<i>Internal documentation for unit testing of MinimalModbus</i>
Inline code	<code>``code text``</code>	code text
String	<code>'A'</code>	'A'
String w es-cape ch.	(string within inline code)	'ABC\x00'
(less good)	(string within inline code, double backslash)	'ABC\\x00' For use in Python docstrings.
(less good)	(string with double backslash)	'ABC\x00' Avoid
Environment var	<code>:envvar:`PYTHONPATH`</code>	PYTHONPATH
OS-level command	<code>:command:`make`</code>	make
File	<code>:file:`minimalmodbus.py`</code>	minimalmodbus.py
Path	<code>:file:`path/to/myfile.txt`</code>	path/to/myfile.txt
Type	<code>**bytes**</code>	bytes
Module	<code>:mod:`minimalmodbus`</code>	<i>minimalmodbus</i>
Data	<code>:data:`.BAUDRATE`</code>	BAUDRATE
Data (full)	<code>:data:`minimalmodbus.BAUDRATE`</code>	minimalmodbus.BAUDRATE
Constant	<code>:const:`False`</code>	False
Function	<code>:func:`._checkInt`</code>	<code>_checkInt()</code>
Function (full)	<code>:func:`minimalmodbus._checkInt`</code>	<code>minimalmodbus._checkInt()</code>
Argument	<code>*payload*</code>	<i>payload</i>
Class	<code>:class:`.Instrument`</code>	<i>Instrument</i>
Class (full)	<code>:class:`minimalmodbus.Instrument`</code>	<i>minimalmodbus.Instrument</i>
Method	<code>:meth:`.read_bit`</code>	<code>read_bit()</code>
Method (full)	<code>:meth:`minimalmodbus.Instrument.read_bit`</code>	<code>minimalmodbus.Instrument.read_bit()</code>

Note that only the functions and methods that are listed in the index will show as links.

Headings

- Top level heading underlining symbol: = (equals)
- Next lower level: - (minus)
- A third level if necessary (avoid this): ` (backquote)

Internal links

- Add an internal marker .. `_my-reference-label:` before a heading.

- Then make an internal link to it using `:ref:`my-reference-label``.

Strings with backslash

- In Python docstrings, use raw strings (a `r` before the tripplequote), to have the backslashes reach Sphinx.

Informative boxes

- `.. seealso::` Example of a `**seealso**` box.
- `.. note::` Example of a `**note**` box.
- `.. warning::` Example of a `**warning**` box.

See also:

Example of a **seealso** box.

Note: Example of a **note** box.

Warning: Example of a **warning** box.

10.19.2 Useful Sphinx-related links

Online resources for the formatting used (reStructuredText):

Sphinx reStructuredText Primer <http://www.sphinx-doc.org/en/master/usage/restructuredtext/basics.html>

Example usage for API documentation https://pythonhosted.org/an_example_pypi_project/sphinx.html

reStructuredText Markup Specification <http://docutils.sourceforge.net/docs/ref/rst/restructuredtext.html>

10.19.3 Sphinx build commands

To build the documentation, in the top project directory run:

```
make docs
```

That should generate HTML files to the directory `docs/_build/html`.

To generate PDF:

```
make pdf
```

In order to generate PDF documentation, you need to install `pdflatex` (approx 1 GByte!):

```
sudo apt-get install texlive texlive-latex-extra
```

10.20 Unittest coverage measurement using coverage.py

Install the script `coverage.py`:

```
sudo pip install coverage
```

Collect test data:

```
make coverage
```

10.21 Using the flake8 style checker tool

This tool checks the coding style, using pep8 and flake. Install it:

```
sudo apt-get install python-flake8
```

Run it:

```
flake8 minimalmodbus.py
```

Configurations are made in a [flake8] section of the `tox.ini` file.

10.22 TODO

See also GitHub issues: <https://github.com/pyhys/minimalmodbus/issues>

- Troubleshoot local PDF generation
- Improve installation troubleshooting
- Test virtual serial port on Windows using com0com
- Unittests for measuring the sleep time in `_communicate`.
- Logging instead of `_print_out()`

Change to Python3 only, and then:

- Change internal representation to bytearray
- Better printout of the bytearray in error messages
- Tool for interpretation of Modbus messages
- Use Enum for payloadformat
- Add type hinting
- Run mypy checks
- Possibly use pytest instead

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

Note that MinimalModbus is released very infrequently, but your issues are not forgotten. Do not use GitHub issues for problems getting the Modbus communication to your instrument to work. For that Stack Overflow is much better, see *Support*.

You can contribute in many ways:

11.1 Types of Contributions

11.1.1 Help other users

It is greatly appreciated if you can help other users to get their Modbus equipment up and running. Please subscribe (“watch”) the tags “modbus” and “minimalmodbus” on Stack Overflow.

Here are the newest minimalmodbus questions on Stack Overflow: <https://stackoverflow.com/questions/tagged/minimalmodbus?tab=Newest>

11.1.2 Report Bugs

Report bugs at <https://github.com/pyhys/minimalmodbus/issues>.

Try to isolate the bug by running in interactive mode (Python interpreter) with debug mode activated.

Of course it is appreciated if you can spend a few moments trying to locate the problem, as it might possibly be related to your particular instrument (and thus difficult to reproduce without it). The source code is very readable, so it should be straight-forward to work with.

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.
- The output from `_get_diagnostic_string()`.

11.1.3 Fix Bugs

Look through the GitHub issues for bugs. Anything tagged with “bug” is open to whoever wants to implement it.

11.1.4 Implement Features

Look through the GitHub issues for features. Anything tagged with “enhancement” is open to whoever wants to implement it.

11.1.5 Write Documentation

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

11.1.6 Submit Feedback

The best way to send feedback is to file an issue at <https://github.com/pyhys/minimalmodbus/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 :)

11.2 Get Started!

Ready to contribute? Here’s how to set up *minimalmodbus* for local development.

1. Fork the *minimalmodbus* repo on GitHub.
2. Clone your fork locally:

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

3. Install your local copy into a virtualenv. Assuming you have *virtualenvwrapper* installed, this is how you set up your fork for local development:

```
$ mkvirtualenv minimalmodbus
$ cd minimalmodbus/
$ python setup.py develop
```

4. Create a branch for local development:

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

Now you can make your changes locally.

5. When you're done making changes, check that your changes pass flake8 and the tests, including testing other Python versions with tox:

```
$ flake8 minimalmodbus tests
$ python setup.py test
$ tox
```

To get flake8 and tox, just pip install them into your virtualenv.

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

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

7. Submit a pull request through the GitHub website.

11.3 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 2.7, 3.5, 3.6, and 3.7. Check https://travis-ci.org/pyhys/minimalmodbus/pull_requests and make sure that the tests pass for all supported Python versions.

12.1 Development Lead

- Jonas Berg <https://github.com/pyhys>

12.2 Contributors

Significant contributions by:

- Aaron LaLonde
- Andres Ulloa
- Angelo Compagnucci
- Asier Abalos
- Austin Stover
- Dino
- Dominik Socha
- Edwin van den Oetelaar
- GitHub user: draput
- GitHub user: gnbl
- GitHub user: mrrs6
- GitHub user: noafterglow
- Luca Di Gregorio
- Matthias Bolte
- Michael Penza

- Peter
- Russ Garrett
- Simon Funke

Related software

The MinimalModbus package is intended for easy-to-use communication with instruments using the Modbus (RTU and ASCII) protocol. There are a few other Python packages for Modbus protocol implementation. For more advanced use, you should consider using one of these:

pyModbus From <https://github.com/riptideio/pymodbus>: ‘Pymodbus is a full Modbus protocol implementation using twisted for its asynchronous communications core.’

modbus-tk From <https://github.com/ljean/modbus-tk>: ‘Make possible to write modbus TCP and RTU master and slave mainly for testing purpose. It is shipped with slave simulator and a master with a web-based hmi. It is a full-stack implementation and as a consequence could also be used on real-world project.’

umodbus From <https://github.com/AdvancedClimateSystems/umodbus/>: ‘uModbus or (μ Modbus) is a pure Python implementation of the Modbus protocol as described in the MODBUS Application Protocol Specification V1.1b3. uModbus implements both a Modbus client (both TCP and RTU) and a Modbus server (both TCP and RTU).’

EasyModbusTCP/UDP/RTU Python From <https://sourceforge.net/projects/easymodbustcp-udp-rtu-python/>: ‘EasyModbusTCP library is available for .NET / Java / and Python. Same Handling for different implementations and different platforms.’

pyModbusTCP From <https://github.com/sourceperl/pyModbusTCP>: ‘A simple Modbus/TCP library for Python’

pylibmodbus From <https://github.com/stephane/pylibmodbus>: ‘Python Interface for libmodbus written with CFFI (Python 2 and Python 3) ‘

14.1 Release 1.0.2 (2019-08-11)

- Adjusted settings for hardware tests
- Improved developer documentation

14.2 Release 1.0.1 (2019-08-10)

- Corrected version number

14.3 Release 1.0.0 (2019-08-10)

New features:

- Implements reading and writing multiple bits simultaneously.
- Support more byteorders (endianness) for floats and long integers.

Breaking changes:

- Renamed method arguments ‘numberOfDecimals’, ‘numberOfRegisters’ to ‘number_of_decimals’, ‘number_of_registers’
- Removed example drivers for Eurotherm 3500 and Omegacn 7500, as I no longer have access to these instruments for testing. It would great if someone would pick up support for these instruments in a separate project.
- Requires pyserial 3.0 or later.
- Removed module level constants for default values, as they were confusingly named.

Other fixes:

- Allow slave addresses also in the reserved range (up to 255). Reported by GitHub user gnbl.

- Serial port read and write buffers are cleared before each request to the instrument. Pull request from GitHub user mrrs6.
- Check whether the serial port is open before trying to open it. Reported by Matthias Bolte.
- Custom exceptions for Modbus errors, by Russ Garrett.
- Silent period between messages is at least 1.75 ms to fulfill Modbus standard. Reported by GitHub user draput.
- Use time.monotonic if available. Suggested by Matthias Bolte.
- Implemented write timeout, to avoid hanging when writing. Instead it will raise an exception. Reported by Austin Stover.
- Better checking of number of registers when reading and writing.
- Rename internal methods and variables to be PEP8 compliant.
- Improved documentation.

14.4 Release 0.7 (2015-07-30)

- Faster CRC calculation by using a lookup table (thanks to Peter)
- Handling of local echo (thanks to Luca Di Gregorio)
- Improved behavior of dummy_serial (number of bytes to read)
- Improved debug messages (thanks to Dino)
- Using project setup by the cookie-cutter tool.
- Reshuffled source files and documentation.
- Moved source to GitHub from Sourceforge.
- Moved documentation to readthedocs.org
- Using the tox tool for testing on multiple Python versions.
- Using Travis CI test framework
- Using codecov.io code coverage measurement framework
- Added support for Python 3.3 and 3.4.
- Dropped support for Python 2.6.

14.5 Release 0.6 (2014-06-22)

- Support for Modbus ASCII mode.

14.6 Release 0.5 (2014-03-23)

- Precalculating number of bytes to read, in order to increase the speed.
- Better handling of several instruments on the same serial port, especially for Windows.
- Improved timing for better compliance with Modbus timing requirements.

14.7 Release 0.4 (2012-09-08)

- Read and write multiple registers.
- Read and write floating point values.
- Read and write long integers.
- Read and write strings.
- Support for negative numbers.
- Use of the Python struct module instead of own bit-tweaking internally.
- Improved documentation.

14.8 Release 0.3.2 (2012-01-25)

- Fine-tuned setup.py for smoother installation.
- Improved documentation.

14.9 Release 0.3.1 (2012-01-24)

- Improved requirements handling in setup.py
- Adjusted MANIFEST.in not to include doc/_templates
- Adjusted RST text formatting in README.txt

14.10 Release 0.3 (2012-01-23)

This is a major rewrite, but the API is backward compatible.

- Extended functionality to support more Modbus function codes.
- Option to close the serial port after each call (useful for Windows XP etc).
- Diagnostic string output available (for support).
- Debug mode available.
- Improved `__repr__` for Instrument instances.
- Improved Python3 compatibility.
- Improved validity checking for function arguments.
- The error messages are made more informative.
- The new example driver `omegacn7500` is included.
- Unit tests included in the distribution.
- A dummy serial port for unit testing is provided (including recorded communication data).
- Updated documentation.

14.11 Release 0.2 (2011-08-19)

- Changes in how to reference the serial port.
- Updated documentation.

14.12 Release 0.1 (2011-06-16)

- First public release.

15.1 Documentation for `dummy_serial` (which is a serial port mock)

`dummy_serial`: A dummy/mock implementation of a serial port for testing purposes.

`dummy_serial.DEFAULT_TIMEOUT = 0.01`

The default timeout value in seconds. Used if not set by the constructor.

`dummy_serial.DEFAULT_BAUDRATE = 19200`

The default baud rate. Used if not set by the constructor.

`dummy_serial.VERBOSE = False`

Set this to `True` for printing the communication, and also details on the port initialization.

Might be monkey-patched in the calling test module.

`dummy_serial.RESPONSES = {'EXAMPLEREQUEST': 'EXAMPLERESPONSE'}`

A dictionary of responses from the dummy serial port.

The key is the message (string) sent to the dummy serial port, and the item is the response (string) from the dummy serial port.

Intended to be monkey-patched in the calling test module.

`dummy_serial.DEFAULT_RESPONSE = 'NotFoundInResponseDictionary'`

Response when no matching message (key) is found in the look-up dictionary.

Should not be an empty string, as that is interpreted as “no data available on port”.

Might be monkey-patched in the calling test module.

class `dummy_serial.Serial(*args, **kwargs)`

Dummy (mock) serial port for testing purposes.

Mimics the behavior of a serial port as defined by the `pySerial` module.

Args:

- `port`:

- timeout:

Note: As the portname argument not is used properly, only one port on `dummy_serial` can be used simultaneously.

is_open

reset_input_buffer ()

reset_output_buffer ()

open ()

Open a (previously initialized) port on `dummy_serial`.

close ()

Close a port on `dummy_serial`.

write (*inputdata*)

Write to a port on `dummy_serial`.

Args: *inputdata* (string/bytes): data for sending to the port on `dummy_serial`. Will affect the response for subsequent read operations.

Note that for Python2, the *inputdata* should be a **string**. For Python3 it should be of type **bytes**.

read (*numberOfBytes*)

Read from a port on `dummy_serial`.

The response is dependent on what was written last to the port on `dummy_serial`, and what is defined in the `RESPONSES` dictionary.

Args: *numberOfBytes* (int): For compability with the real function.

Returns a **string** for Python2 and **bytes** for Python3.

If the response is shorter than *numberOfBytes*, it will sleep for timeout. If the response is longer than *numberOfBytes*, it will return only *numberOfBytes* bytes.

15.2 Internal documentation for MinimalModbus

MinimalModbus: A Python driver for Modbus RTU/ASCII via serial port (via USB, RS485 or RS232).

```
class minimalmodbus.Instrument (port, slaveaddress, mode='rtu',  
                                close_port_after_each_call=False, debug=False)
```

Instrument class for talking to instruments (slaves).

Uses the Modbus RTU or ASCII protocols (via RS485 or RS232).

Args:

- *port* (str): The serial port name, for example `/dev/ttyUSB0` (Linux), `/dev/tty.usbserial` (OS X) or `COM4` (Windows).
- *slaveaddress* (int): Slave address in the range 1 to 247 (use decimal numbers, not hex). Address 0 is for broadcast, and 248-255 are reserved.
- *mode* (str): Mode selection. Can be `MODE_RTU` or `MODE_ASCII`.
- *close_port_after_each_call* (bool): If the serial port should be closed after each call to the instrument.
- *debug* (bool): Set this to `True` to print the communication details

```
__init__ (port, slaveaddress, mode='rtu', close_port_after_each_call=False, debug=False)
```

Initialize instrument and open corresponding serial port.

address = None

Slave address (int). Most often set by the constructor (see the class documentation).

mode = None

Slave mode (str), can be `MODE_RTU` or `MODE_ASCII`. Most often set by the constructor (see the class documentation).

Changing this will not affect how other instruments use the same serial port.

New in version 0.6.

precalculate_read_size = None

If this is `False`, the serial port reads until timeout instead of just reading a specific number of bytes. Defaults to `True`.

Changing this will not affect how other instruments use the same serial port.

New in version 0.5.

debug = None

Set this to `True` to print the communication details. Defaults to `False`.

Most often set by the constructor (see the class documentation).

Changing this will not affect how other instruments use the same serial port.

clear_buffers_before_each_transaction = None

If this is `True`, the serial port read and write buffers are cleared before each request to the instrument, to avoid cumulative byte sync errors across multiple messages. Defaults to `True`.

Changing this will not affect how other instruments use the same serial port.

New in version 1.0.

close_port_after_each_call = None

If this is `True`, the serial port will be closed after each call. Defaults to `False`.

Changing this will not affect how other instruments use the same serial port.

Most often set by the constructor (see the class documentation).

handle_local_echo = None

Set to `True` if your RS-485 adaptor has local echo enabled. Then the transmitted message will immediately appear at the receive line of the RS-485 adaptor. MinimalModbus will then read and discard this data, before reading the data from the slave. Defaults to `False`.

Changing this will not affect how other instruments use the same serial port.

New in version 0.7.

serial = None

The serial port object as defined by the `pySerial` module. Created by the constructor.

Attributes that could be changed after initialisation:

- **port (str): Serial port name.**
 - Most often set by the constructor (see the class documentation).
- **baudrate (int): Baudrate in Baud.**
 - Defaults to 19200.
- **parity (probably int): Parity. See the `pySerial` module for documentation.**
 - Defaults to `serial.PARITY_NONE`.

- **bytesize (int): Bytesize in bits.**
 - Defaults to 8.
- **stopbits (int): The number of stopbits.**
 - Defaults to 1.
- **timeout (float): Read timeout value in seconds.**
 - Defaults to 0.05 s.
- **write_timeout (float): Write timeout value in seconds.**
 - Defaults to 2.0 s.

`__repr__()`

Give string representation of the *Instrument* object.

`_print_debug(text)`

`read_bit(registeraddress, functioncode=2)`

Read one bit from the slave (instrument).

This is for a bit that has its individual address in the instrument.

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- functioncode (int): Modbus function code. Can be 1 or 2.

Returns: The bit value 0 or 1 (int).

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

`write_bit(registeraddress, value, functioncode=5)`

Write one bit to the slave (instrument).

This is for a bit that has its individual address in the instrument.

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- value (int or bool): 0 or 1, or True or False
- functioncode (int): Modbus function code. Can be 5 or 15.

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

`read_bits(registeraddress, number_of_bits, functioncode=2)`

Read multiple bits from the slave (instrument).

This is for bits that have individual addresses in the instrument.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- number_of_bits (int): Number of bits to read
- functioncode (int): Modbus function code. Can be 1 or 2.

Returns: A list of bit values 0 or 1 (int). The first value in the list is for the bit at the given address.

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

write_bits (*registeraddress, values*)

Write multiple bits to the slave (instrument).

This is for bits that have individual addresses in the instrument.

Uses Modbus functioncode 15.

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- values (list of int or bool): 0 or 1, or True or False. The first value in the list is for the bit at the given address.

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

read_register (*registeraddress, number_of_decimals=0, functioncode=3, signed=False*)

Read an integer from one 16-bit register in the slave, possibly scaling it.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).
- number_of_decimals (int): The number of decimals for content conversion.
- functioncode (int): Modbus function code. Can be 3 or 4.
- signed (bool): Whether the data should be interpreted as unsigned or signed.

Note: The parameter `number_of_decimals` was named `numberOfDecimals` before MinimalModbus 1.0

If a value of 77.0 is stored internally in the slave register as 770, then use `number_of_decimals=1` which will divide the received data by 10 before returning the value.

Similarly `number_of_decimals=2` will divide the received data by 100 before returning the value.

Some manufacturers allow negative values for some registers. Instead of an allowed integer range 0 to 65535, a range -32768 to 32767 is allowed. This is implemented as any received value in the upper range (32768 to 65535) is interpreted as negative value (in the range -32768 to -1).

Use the parameter `signed=True` if reading from a register that can hold negative values. Then upper range data will be automatically converted into negative return values (two’s complement).

signed	Data type in slave	Alternative name	Range
False	Unsigned INT16	Unsigned short	0 to 65535
True	INT16	Short	-32768 to 32767

Returns: The register data in numerical value (int or float).

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

write_register (*registeraddress, value, number_of_decimals=0, functioncode=16, signed=False*)

Write an integer to one 16-bit register in the slave, possibly scaling it.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- registeraddress (int): The slave register address (use decimal numbers, not hex).

- `value` (int or float): The value to store in the slave register (might be scaled before sending).
- `number_of_decimals` (int): The number of decimals for content conversion.
- `functioncode` (int): Modbus function code. Can be 6 or 16.
- `signed` (bool): Whether the data should be interpreted as unsigned or signed.

Note: The parameter `number_of_decimals` was named `numberOfDecimals` before MinimalModbus 1.0

To store for example `value=77.0`, use `number_of_decimals=1` if the slave register will hold it as 770 internally. This will multiply `value` by 10 before sending it to the slave register.

Similarly `number_of_decimals=2` will multiply `value` by 100 before sending it to the slave register.

As the largest number that can be written to a register is `0xFFFF = 65535`, the `value` and `number_of_decimals` should max be 65535 when combined. So when using `number_of_decimals=3` the maximum value is 65.535.

For discussion on negative values, the range and on alternative names, see `read_register()`.

Use the parameter `signed=True` if writing to a register that can hold negative values. Then negative input will be automatically converted into upper range data (two's complement).

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

read_long (*registeraddress*, *functioncode=3*, *signed=False*, *byteorder=0*)

Read a long integer (32 bits) from the slave.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `functioncode` (int): Modbus function code. Can be 3 or 4.
- `signed` (bool): Whether the data should be interpreted as unsigned or signed.
- `byteorder` (int): How multi-register data should be interpreted. Defaults to `BYTEORDER_BIG`.

<code>signed</code>	Data type in slave	Alternative name	Range
<code>False</code>	Unsigned INT32	Unsigned long	0 to 4294967295
<code>True</code>	INT32	Long	-2147483648 to 2147483647

Returns: The numerical value (int).

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

write_long (*registeraddress*, *value*, *signed=False*, *byteorder=0*)

Write a long integer (32 bits) to the slave.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Uses Modbus function code 16.

For discussion on number of bits, number of registers, the range and on alternative names, see `read_long()`.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).

- `value` (int or long): The value to store in the slave.
- `signed` (bool): Whether the data should be interpreted as unsigned or signed.
- `byteorder` (int): How multi-register data should be interpreted. Defaults to `BYTEORDER_BIG`.

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

read_float (*registeraddress*, *functioncode=3*, *number_of_registers=2*, *byteorder=0*)

Read a floating point number from the slave.

Floats are stored in two or more consecutive 16-bit registers in the slave. The encoding is according to the standard IEEE 754.

There are differences in the byte order used by different manufacturers. A floating point value of 1.0 is encoded (in single precision) as 3f800000 (hex). In this implementation the data will be sent as `'\x3f\x80'` and `'\x00\x00'` to two consecutive registers by default. Make sure to test that it makes sense for your instrument. If not, change the `byteorder` argument.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `functioncode` (int): Modbus function code. Can be 3 or 4.
- `number_of_registers` (int): The number of registers allocated for the float. Can be 2 or 4.
- `byteorder` (int): How multi-register data should be interpreted. Defaults to `BYTEORDER_BIG`.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Type of floating point number in slave	Size	Registers	Range
Single precision (binary32)	32 bits (4 bytes)	2 registers	1.4E-45 to 3.4E38
Double precision (binary64)	64 bits (8 bytes)	4 registers	5E-324 to 1.8E308

Returns: The numerical value (float).

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

write_float (*registeraddress*, *value*, *number_of_registers=2*, *byteorder=0*)

Write a floating point number to the slave.

Floats are stored in two or more consecutive 16-bit registers in the slave.

Uses Modbus function code 16.

For discussion on precision, number of registers and on byte order, see `read_float()`.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `value` (float or int): The value to store in the slave
- `number_of_registers` (int): The number of registers allocated for the float. Can be 2 or 4.
- `byteorder` (int): How multi-register data should be interpreted. Defaults to `BYTEORDER_BIG`.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

read_string (*registeraddress*, *number_of_registers=16*, *functioncode=3*)

Read an ASCII string from the slave.

Each 16-bit register in the slave are interpreted as two characters (each 1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

International characters (Unicode/UTF-8) are not supported.

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *number_of_registers* (int): The number of registers allocated for the string.
- *functioncode* (int): Modbus function code. Can be 3 or 4.

Note: The parameter *number_of_registers* was named *numberOfRegisters* before MinimalModbus 1.0

Returns: The string (str).

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

write_string (*registeraddress*, *textstring*, *number_of_registers=16*)

Write an ASCII string to the slave.

Each 16-bit register in the slave are interpreted as two characters (each 1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

Uses Modbus function code 16.

International characters (Unicode/UTF-8) are not supported.

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *textstring* (str): The string to store in the slave, must be ASCII.
- *number_of_registers* (int): The number of registers allocated for the string.

Note: The parameter *number_of_registers* was named *numberOfRegisters* before MinimalModbus 1.0

If the *textstring* is longer than the $2 * \text{number_of_registers}$, an error is raised. Shorter strings are padded with spaces.

Returns: None

Raises: TypeError, ValueError, ModbusException, serial.SerialException (inherited from IOError)

read_registers (*registeraddress*, *number_of_registers*, *functioncode=3*)

Read integers from 16-bit registers in the slave.

The slave registers can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Args:

- *registeraddress* (int): The slave register start address (use decimal numbers, not hex).
- *number_of_registers* (int): The number of registers to read, max 125 registers.

- `functioncode` (int): Modbus function code. Can be 3 or 4.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Any scaling of the register data, or converting it to negative number (two's complement) must be done manually.

Returns: The register data (a list of int). The first value in the list is for the register at the given address.

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

`write_registers` (*registeraddress, values*)

Write integers to 16-bit registers in the slave.

The slave register can hold integer values in the range 0 to 65535 (“Unsigned INT16”).

Uses Modbus function code 16.

The number of registers that will be written is defined by the length of the `values` list.

Args:

- `registeraddress` (int): The slave register start address (use decimal numbers, not hex).
- `values` (list of int): The values to store in the slave registers, max 123 values. The first value in the list is for the register at the given address.

Note: The parameter `number_of_registers` was named `numberOfRegisters` before MinimalModbus 1.0

Any scaling of the register data, or converting it to negative number (two's complement) must be done manually.

Returns: None

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

`_generic_command` (*functioncode, registeraddress, value=None, number_of_decimals=0, number_of_registers=0, number_of_bits=0, signed=False, byteorder=0, payloadformat=None*)

Perform generic command for reading and writing registers and bits.

Args:

- `functioncode` (int): Modbus function code.
- `registeraddress` (int): The register address (use decimal numbers, not hex).
- `value` (numerical or string or None or list of int): The value to store in the register. Depends on `payloadformat`.
- `number_of_decimals` (int): The number of decimals for content conversion. Only for a single register.
- `number_of_registers` (int): The number of registers to read/write. Only certain values allowed, depends on `payloadformat`.
- `number_of_bits` (int): The number of registers to read/write.
- `signed` (bool): Whether the data should be interpreted as unsigned or signed. Only for a single register or for `payloadformat='long'`.
- `byteorder` (int): How multi-register data should be interpreted.

The resolution for Python's `time.time()` is lower on Windows than on Linux. It is about 16 ms on Windows according to <https://stackoverflow.com/questions/157359/accurate-timestamping-in-python-logging>

For Python3, the information sent to and from `pySerial` should be of the type bytes. This is taken care of automatically by `MinimalModbus`.

`_performCommand` (*functioncode*, *payload_to_slave*)

Perform the command having the *functioncode*.

Args:

- `functioncode` (int): The function code for the command to be performed. Can for example be 'Write register' = 16.
- `payload_to_slave` (str): Data to be transmitted to the slave (will be embedded in slaveaddress, CRC etc)

Returns: The extracted data payload from the slave (a string). It has been stripped of CRC etc.

Raises: `TypeError`, `ValueError`, `ModbusException`, `serial.SerialException` (inherited from `IOError`)

Makes use of the `_communicate()` method. The request is generated with the `_embed_payload()` function, and the parsing of the response is done with the `_extract_payload()` function.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.ModbusException`

Base class for Modbus communication exceptions.

Inherits from `IOError`, which is an alias for `OSError` in Python3.

`__module__ = 'minimalmodbus'`

`__weakref__`

list of weak references to the object (if defined)

exception `minimalmodbus.SlaveReportedException`

Base class for exceptions that the slave (instrument) reports.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.SlaveDeviceBusyError`

The slave is busy processing some command.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.NegativeAcknowledgeError`

The slave can not fulfil the programming request.

This typically happens when using function code 13 or 14 decimal.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.IllegalRequestError`

The slave has received an illegal request.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.MasterReportedException`

Base class for exceptions that the master (computer) detects.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.NoResponseError`

No response from the slave.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.LocalEchoError`

There is some problem with the local echo.

`__module__ = 'minimalmodbus'`

exception `minimalmodbus.InvalidResponseError`

The response does not fulfill the Modbus standard, for example wrong checksum.

`__module__ = 'minimalmodbus'`

`minimalmodbus._create_payload` (*functioncode, registeraddress, value, number_of_decimals, number_of_registers, number_of_bits, signed, byteorder, payloadformat*)

Create the payload.

Error checking should have been done before calling this function.

For argument descriptions, see the `_generic_command()` method.

`minimalmodbus._parse_payload` (*payload, functioncode, registeraddress, value, number_of_decimals, number_of_registers, number_of_bits, signed, byteorder, payloadformat*)

`minimalmodbus._embed_payload` (*slaveaddress, mode, functioncode, payloaddata*)

Build a request from the slaveaddress, the function code and the payload data.

Args:

- `slaveaddress` (int): The address of the slave.
- `mode` (str): The modbus protocol mode (MODE_RTU or MODE_ASCII)
- `functioncode` (int): The function code for the command to be performed. Can for example be 16 (Write register).
- `payloaddata` (str): The byte string to be sent to the slave.

Returns: The built (raw) request string for sending to the slave (including CRC etc).

Raises: `ValueError`, `TypeError`.

The resulting request has the format:

- RTU Mode: `slaveaddress` byte + `functioncode` byte + `payloaddata` + CRC (which is two bytes).
- ASCII Mode: header (:) + `slaveaddress` (2 characters) + `functioncode` (2 characters) + `payloaddata` + LRC (which is two characters) + footer (CRLF)

The LRC or CRC is calculated from the byte string made up of `slaveaddress` + `functioncode` + `payloaddata`. The header, LRC/CRC, and footer are excluded from the calculation.

`minimalmodbus._extract_payload` (*response, slaveaddress, mode, functioncode*)

Extract the payload data part from the slave's response.

Args:

- `response` (str): The raw response byte string from the slave. This is different for RTU and ASCII.
- `slaveaddress` (int): The address of the slave. Used here for error checking only.
- `mode` (str): The modbus protocol mode (MODE_RTU or MODE_ASCII)
- `functioncode` (int): Used here for error checking only.

Returns: The payload part of the `response` string. Conversion from Modbus ASCII has been done if applicable.

Raises: `ValueError`, `TypeError`, `ModbusException` (or subclasses).

Raises an exception if there is any problem with the received address, the functioncode or the CRC.

The received response should have the format:

- RTU Mode: slaveaddress byte + functioncode byte + payloaddata + CRC (which is two bytes)
- ASCII Mode: header (:) + slaveaddress byte + functioncode byte + payloaddata + LRC (which is two characters) + footer (CRLF)

For development purposes, this function can also be used to extract the payload from the request sent TO the slave.

`minimalmodbus._predict_response_size(mode, functioncode, payload_to_slave)`

Calculate the number of bytes that should be received from the slave.

Args:

- mode (str): The modbus protocol mode (MODE_RTU or MODE_ASCII)
- functioncode (int): Modbus function code.
- payload_to_slave (str): The raw request that is to be sent to the slave (not hex encoded string)

Returns: The predicted number of bytes (int) in the response.

Raises: ValueError, TypeError.

`minimalmodbus._calculate_minimum_silent_period(baudrate)`

Calculate the silent period length between messages.

It should correspond to the time to send 3.5 characters.

Args: baudrate (numerical): The baudrate for the serial port

Returns: The number of seconds (float) that should pass between each message on the bus.

Raises: ValueError, TypeError.

`minimalmodbus._num_to_onebyte_string(inputvalue)`

Convert a numerical value to a one-byte string.

Args: inputvalue (int): The value to be converted. Should be ≥ 0 and ≤ 255 .

Returns: A one-byte string created by `chr(inputvalue)`.

Raises: TypeError, ValueError

`minimalmodbus._num_to_twobyte_string(value, number_of_decimals=0, lsb_first=False, signed=False)`

Convert a numerical value to a two-byte string, possibly scaling it.

Args:

- value (float or int): The numerical value to be converted.
- number_of_decimals (int): Number of decimals, 0 or more, for scaling.
- lsb_first (bool): Whether the least significant byte should be first in the resulting string.
- signed (bool): Whether negative values should be accepted.

Returns: A two-byte string.

Raises: TypeError, ValueError. Gives DeprecationWarning instead of ValueError for some values in Python 2.6.

Use `number_of_decimals=1` to multiply `value` by 10 before sending it to the slave register. Similarly `number_of_decimals=2` will multiply `value` by 100 before sending it to the slave register.

Use the parameter `signed=True` if making a bytestring that can hold negative values. Then negative input will be automatically converted into upper range data (two's complement).

The byte order is controlled by the `lsb_first` parameter, as seen here:

<code>lsb_first</code> parameter	Endianness	Description
False (default)	Big-endian	Most significant byte is sent first
True	Little-endian	Least significant byte is sent first

For example: To store for example `value=77.0`, use `number_of_decimals = 1` if the register will hold it as 770 internally. The value 770 (dec) is 0302 (hex), where the most significant byte is 03 (hex) and the least significant byte is 02 (hex). With `lsb_first = False`, the most significant byte is given first why the resulting string is `\x03\x02`, which has the length 2.

`minimalmodbus._twobyte_string_to_num` (*bytestring*, *number_of_decimals=0*, *signed=False*)
 Convert a two-byte string to a numerical value, possibly scaling it.

Args:

- `bytestring` (str): A string of length 2.
- `number_of_decimals` (int): The number of decimals. Defaults to 0.
- `signed` (bol): Whether large positive values should be interpreted as negative values.

Returns: The numerical value (int or float) calculated from the `bytestring`.

Raises: `TypeError`, `ValueError`

Use the parameter `signed=True` if converting a bytestring that can hold negative values. Then upper range data will be automatically converted into negative return values (two's complement).

Use `number_of_decimals=1` to divide the received data by 10 before returning the value. Similarly `number_of_decimals=2` will divide the received data by 100 before returning the value.

The byte order is big-endian, meaning that the most significant byte is sent first.

For example: A string `\x03\x02` (which has the length 2) corresponds to 0302 (hex) = 770 (dec). If `number_of_decimals = 1`, then this is converted to 77.0 (float).

`minimalmodbus._long_to_bytestring` (*value*, *signed=False*, *number_of_registers=2*, *byteorder=0*)
 Convert a long integer to a bytestring.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Args:

- `value` (int): The numerical value to be converted.
- `signed` (bool): Whether large positive values should be interpreted as negative values.
- `number_of_registers` (int): Should be 2. For error checking only.
- `byteorder` (int): How multi-register data should be interpreted.

Returns: A bytestring (4 bytes).

Raises: `TypeError`, `ValueError`

`minimalmodbus._bytestring_to_long` (*bytestring*, *signed=False*, *number_of_registers=2*, *byteorder=0*)
 Convert a bytestring to a long integer.

Long integers (32 bits = 4 bytes) are stored in two consecutive 16-bit registers in the slave.

Args:

- `bytestring` (str): A string of length 4.
- `signed` (bool): Whether large positive values should be interpreted as negative values.
- `number_of_registers` (int): Should be 2. For error checking only.
- `byteorder` (int): How multi-register data should be interpreted.

Returns: The numerical value (int).

Raises: ValueError, TypeError

`minimalmodbus._float_to_bytestring` (*value*, *number_of_registers=2*, *byteorder=0*)

Convert a numerical value to a bytestring.

Floats are stored in two or more consecutive 16-bit registers in the slave. The encoding is according to the standard IEEE 754.

Type of floating point number in slave	Size	Registers	Range
Single precision (binary32)	32 bits (4 bytes)	2 registers	1.4E-45 to 3.4E38
Double precision (binary64)	64 bits (8 bytes)	4 registers	5E-324 to 1.8E308

A floating point value of 1.0 is encoded (in single precision) as 3f800000 (hex). This will give a byte string `'\x3f\x80\x00\x00'` (big endian).

Args:

- `value` (float or int): The numerical value to be converted.
- `number_of_registers` (int): Can be 2 or 4.
- `byteorder` (int): How multi-register data should be interpreted.

Returns: A bytestring (4 or 8 bytes).

Raises: TypeError, ValueError

`minimalmodbus._bytestring_to_float` (*bytestring*, *number_of_registers=2*, *byteorder=0*)

Convert a four-byte string to a float.

Floats are stored in two or more consecutive 16-bit registers in the slave.

For discussion on precision, number of bits, number of registers, the range, byte order and on alternative names, see `minimalmodbus._float_to_bytestring()`.

Args:

- `bytestring` (str): A string of length 4 or 8.
- `number_of_registers` (int): Can be 2 or 4.
- `byteorder` (int): How multi-register data should be interpreted.

Returns: A float.

Raises: TypeError, ValueError

`minimalmodbus._textstring_to_bytestring` (*inputstring*, *number_of_registers=16*)

Convert a text string to a bytestring.

Each 16-bit register in the slave are interpreted as two characters (1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

Not much of conversion is done, mostly error checking and string padding. If the inputstring is shorter than the allocated space, it is padded with spaces in the end.

Args:

- inputstring (str): The string to be stored in the slave. Max $2 * \text{number_of_registers}$ characters.
- number_of_registers (int): The number of registers allocated for the string.

Returns: A bytearray (str).

Raises: TypeError, ValueError

`minimalmodbus._bytestring_to_textstring` (*bytestring*, *number_of_registers=16*)

Convert a bytearray to a text string.

Each 16-bit register in the slave are interpreted as two characters (1 byte = 8 bits). For example 16 consecutive registers can hold 32 characters (32 bytes).

Not much of conversion is done, mostly error checking.

Args:

- bytearray (str): The string from the slave. Length = $2 * \text{number_of_registers}$
- number_of_registers (int): The number of registers allocated for the string.

Returns: A the text string (str).

Raises: TypeError, ValueError

`minimalmodbus._valuelist_to_bytestring` (*valuelist*, *number_of_registers*)

Convert a list of numerical values to a bytearray.

Each element is 'unsigned INT16'.

Args:

- valuelist (list of int): The input list. The elements should be in the range 0 to 65535.
- number_of_registers (int): The number of registers. For error checking. Should equal the number of elements in valuelist.

Returns: A bytearray (str). Length = $2 * \text{number_of_registers}$

Raises: TypeError, ValueError

`minimalmodbus._bytestring_to_valuelist` (*bytestring*, *number_of_registers*)

Convert a bytearray to a list of numerical values.

The bytearray is interpreted as 'unsigned INT16'.

Args:

- bytearray (str): The string from the slave. Length = $2 * \text{number_of_registers}$
- number_of_registers (int): The number of registers. For error checking.

Returns: A list of integers.

Raises: TypeError, ValueError

`minimalmodbus._now` ()

Return a timestamp for time duration measurements.

Returns a float, that increases with 1.0 per second. The starting point is undefined.

`minimalmodbus._pack` (*formatstring*, *value*)

Pack a value into a bytestring.

Uses the built-in `struct` Python module.

Args:

- `formatstring` (str): String for the packing. See the `struct` module for details.
- `value` (depends on `formatstring`): The value to be packed

Returns: A bytestring (str).

Raises: `ValueError`

Note that the `struct` module produces byte buffers for Python3, but bytestrings for Python2. This is compensated for automatically.

`minimalmodbus._unpack` (*formatstring*, *packed*)

Unpack a bytestring into a value.

Uses the built-in `struct` Python module.

Args:

- `formatstring` (str): String for the packing. See the `struct` module for details.
- `packed` (str): The bytestring to be unpacked.

Returns: A value. The type depends on the `formatstring`.

Raises: `ValueError`

Note that the `struct` module wants byte buffers for Python3, but bytestrings for Python2. This is compensated for automatically.

`minimalmodbus._swap` (*bytestring*)

Swap characters pairwise in a string.

This corresponds to a “byte swap”.

Args:

- `bytestring` (str): input. The length should be an even number.

Return the string with characters swapped.

`minimalmodbus._hexencode` (*bytestring*, *insert_spaces=False*)

Convert a byte string to a hex encoded string.

For example ‘J’ will return ‘4A’, and ‘\x04’ will return ‘04’.

Args:

- `bytestring` (str): Can be for example ‘A\x01B\x45’.
- `insert_spaces` (bool): Insert space characters between pair of characters to increase readability.

Returns: A string of twice the length, with characters in the range ‘0’ to ‘9’ and ‘A’ to ‘F’. The string will be longer if spaces are inserted.

Raises: `TypeError`, `ValueError`

`minimalmodbus._hexdecode` (*hexstring*)

Convert a hex encoded string to a byte string.

For example ‘4A’ will return ‘J’, and ‘04’ will return ‘\x04’ (which has length 1).

Args:

- hexstring (str): Can be for example 'A3' or 'A3B4'. Must be of even length.
- Allowed characters are '0' to '9', 'a' to 'f' and 'A' to 'F' (not space).

Returns: A string of half the length, with characters corresponding to all 0-255 values for each byte.

Raises: TypeError, ValueError

`minimalmodbus._hexlify` (*bytestring*)

Convert a byte string to a hex encoded string, with spaces for easier reading.

This is just a facade for `_hexencode()` with `insert_spaces = True`.

See `_hexencode()` for details.

`minimalmodbus._calculate_number_of_bytes_for_bits` (*number_of_bits*)

Calculate number of full bytes required to house a number of bits.

Args:

- `number_of_bits` (str): Number of bits

Error checking should have been done before.

Algorithm from MODBUS APPLICATION PROTOCOL SPECIFICATION V1.1b

`minimalmodbus._bit_to_bytestring` (*value*)

Create the bit pattern that is used for writing single bits.

Used for functioncode 5. The same value is sent back in the response from the slave.

This is basically a storage of numerical constants.

Args:

- `value` (int): can be 0 or 1

Returns: The bit pattern (string).

Raises: TypeError, ValueError

`minimalmodbus._bits_to_bytestring` (*valuelist*)

Build a bytestring from a list of bits.

This is used for functioncode 15.

Args:

- `valuelist` (list of int): 0 or 1

Returns a bytestring.

`minimalmodbus._bytestring_to_bits` (*bytestring*, *number_of_bits*)

Parse bits from a bytestring.

This is used for parsing the bits in response messages for functioncode 1 and 2.

The first byte in the bytestring contains info on the addressed bit (in LSB in that byte). Second bit from right contains info on the bit on the next address.

Next byte in the bytestring contains data on next 8 bits. Might be padded with zeros toward MSB.

Args:

- `bytestring` (str): input string
- `number_of_bits` (int): Number of bits to extract

Returns a list of values (0 or 1). The length of the list is equal to `number_of_bits`.

`minimalmodbus._twos_complement(x, bits=16)`

Calculate the two's complement of an integer.

Then also negative values can be represented by an upper range of positive values. See https://en.wikipedia.org/wiki/Two%27s_complement

Args:

- `x` (int): input integer.
- `bits` (int): number of bits, must be > 0.

Returns: An int, that represents the two's complement of the input.

Example for bits=8:

x	returns
0	0
1	1
127	127
-128	128
-127	129
-1	255

`minimalmodbus._from_twos_complement(x, bits=16)`

Calculate the inverse(?) of a two's complement of an integer.

Args:

- `x` (int): input integer.
- `bits` (int): number of bits, must be > 0.

Returns: An int, that represents the inverse(?) of two's complement of the input.

Example for bits=8:

x	returns
0	0
1	1
127	127
128	-128
129	-127
255	-1

`minimalmodbus._set_bit_on(x, bit_num)`

Set bit 'bit_num' to True.

Args:

- `x` (int): The value before.
- `bit_num` (int): The bit number that should be set to True.

Returns: The value after setting the bit. This is an integer.

For example: For `x = 4` (dec) = 0100 (bin), setting bit number 0 results in 0101 (bin) = 5 (dec).

`minimalmodbus._check_bit(x, bit_num)`

Check if bit 'bit_num' is set the input integer.

Args:

- `x` (int): The input value.
- `bit_num` (int): The bit number to be checked

Returns: True or False

For example: For `x = 4` (dec) = 0100 (bin), checking bit number 2 results in True, and checking bit number 3 results in False.

`minimalmodbus._CRC16TABLE = (0, 49345, 49537, 320, 49921, 960, 640, 49729, 50689, 1728, 192, ...)`
CRC-16 lookup table with 256 elements.

Built with this code:

```
poly=0xA001
table = []
for index in range(256):
    data = index << 1
    crc = 0
    for _ in range(8, 0, -1):
        data >>= 1
        if (data ^ crc) & 0x0001:
            crc = (crc >> 1) ^ poly
        else:
            crc >>= 1
        table.append(crc)
output = ''
for i, m in enumerate(table):
    if not i%11:
        output += "\n"
    output += "{:5.0f}, ".format(m)
print output
```

`minimalmodbus._calculate_crc_string(inputstring)`

Calculate CRC-16 for Modbus.

Args: `inputstring` (str): An arbitrary-length message (without the CRC).

Returns: A two-byte CRC string, where the least significant byte is first.

`minimalmodbus._calculate_lrc_string(inputstring)`

Calculate LRC for Modbus.

Args: `inputstring` (str): An arbitrary-length message (without the beginning colon and terminating CRLF). It should already be decoded from hex-string.

Returns: A one-byte LRC bytestring (not encoded to hex-string)

Algorithm from the document 'MODBUS over serial line specification and implementation guide V1.02'.

The LRC is calculated as 8 bits (one byte).

For example a LRC 0110 0001 (bin) = 61 (hex) = 97 (dec) = 'a'. This function will then return 'a'.

In Modbus ASCII mode, this should be transmitted using two characters. This example should be transmitted '61', which is a string of length two. This function does not handle that conversion for transmission.

`minimalmodbus._check_mode(mode)`

Check that the Modbus mode is valid.

Args: `mode` (string): The Modbus mode (MODE_RTU or MODE_ASCII)

Raises: TypeError, ValueError

`minimalmodbus._check_functioncode` (*functioncode*, *list_of_allowed_values=None*)

Check that the given `functioncode` is in the `list_of_allowed_values`.

Also verifies that $1 \leq \text{function code} \leq 127$.

Args:

- `functioncode` (int): The function code
- `list_of_allowed_values` (list of int): Allowed values. Use *None* to bypass this part of the checking.

Raises: `TypeError`, `ValueError`

`minimalmodbus._check_slaveaddress` (*slaveaddress*)

Check that the given `slaveaddress` is valid.

Args: `slaveaddress` (int): The slave address

Raises: `TypeError`, `ValueError`

`minimalmodbus._check_registeraddress` (*registeraddress*)

Check that the given `registeraddress` is valid.

Args: `registeraddress` (int): The register address

Raises: `TypeError`, `ValueError`

`minimalmodbus._check_response_payload` (*payload*, *functioncode*, *registeraddress*, *value*,
number_of_decimals, *number_of_registers*, *number_of_bits*, *signed*, *byteorder*, *payloadformat*)

`minimalmodbus._check_response_slaveerrorcode` (*response*)

Check if the slave indicates an error.

Args:

- `response` (string): Response from the slave

The response is in RTU format, but the checksum might be one or two bytes depending on whether it was sent in RTU or ASCII mode.

Checking of type and length of the response should be done before calling this functions.

Raises: `SlaveReportedException` or subclass

`minimalmodbus._check_response_bytecount` (*payload*)

Check that the number of bytes as given in the response is correct.

The first byte in the payload indicates the length of the payload (first byte not counted).

Args: `payload` (string): The payload

Raises: `TypeError`, `ValueError`, `InvalidResponseError`

`minimalmodbus._check_response_registeraddress` (*payload*, *registeraddress*)

Check that the start address as given in the response is correct.

The first two bytes in the payload holds the address value.

Args:

- `payload` (string): The payload
- `registeraddress` (int): What the register address actually should be (use decimal numbers, not hex).

Raises: `TypeError`, `ValueError`, `InvalidResponseError`

`minimalmodbus._check_response_number_of_registers` (*payload, number_of_registers*)

Check that the number of written registers as given in the response is correct.

The bytes 2 and 3 (zero based counting) in the payload holds the value.

Args:

- `payload` (string): The payload
- `number_of_registers` (int): Number of registers that have been written

Raises: `TypeError`, `ValueError`, `InvalidResponseError`

`minimalmodbus._check_response_writedata` (*payload, writedata*)

Check that the write data as given in the response is correct.

The bytes 2 and 3 (zero based counting) in the payload holds the write data.

Args:

- `payload` (string): The payload
- `writedata` (string): The data that should have been written. Length should be 2 bytes.

Raises: `TypeError`, `ValueError`, `InvalidResponseError`

`minimalmodbus._check_string` (*inputstring, description, minlength=0, maxlength=None, force_ascii=False, exception_type=<type 'exceptions.ValueError'>*)

Check that the given string is valid.

Args:

- `inputstring` (string): The string to be checked
- `description` (string): Used in error messages for the checked `inputstring`
- `minlength` (int): Minimum length of the string
- `maxlength` (int or None): Maximum length of the string
- `force_ascii` (bool): Enforce that the string is ASCII
- `exception_type` (Exception): The type of exception to raise for length errors

The `force_ascii` argument is valid only for Python3, as all strings are ASCII in Python2.

Raises: `TypeError`, `ValueError` or the one given by `exception_type`

Uses the function `_check_int()` internally.

`minimalmodbus._check_int` (*inputvalue, minvalue=None, maxvalue=None, description='inputvalue'*)

Check that the given integer is valid.

Args:

- `inputvalue` (int or long): The integer to be checked
- `minvalue` (int or long, or None): Minimum value of the integer
- `maxvalue` (int or long, or None): Maximum value of the integer
- `description` (string): Used in error messages for the checked `inputvalue`

Raises: `TypeError`, `ValueError`

Note: Can not use the function `_check_string()`, as that function uses this function internally.

`minimalmodbus._check_numerical` (*inputvalue*, *minvalue=None*, *maxvalue=None*, *description='inputvalue'*)

Check that the given numerical value is valid.

Args:

- *inputvalue* (numerical): The value to be checked.
- *minvalue* (numerical): Minimum value Use None to skip this part of the test.
- *maxvalue* (numerical): Maximum value. Use None to skip this part of the test.
- *description* (string): Used in error messages for the checked *inputvalue*

Raises: TypeError, ValueError

Note: Can not use the function `_check_string()`, as it uses this function internally.

`minimalmodbus._check_bool` (*inputvalue*, *description='inputvalue'*)

Check that the given *inputvalue* is a boolean.

Args:

- *inputvalue* (boolean): The value to be checked.
- *description* (string): Used in error messages for the checked *inputvalue*.

Raises: TypeError, ValueError

`minimalmodbus._print_out` (*inputstring*)

Print the *inputstring*. To make it compatible with Python2 and Python3.

Args: *inputstring* (str): The string that should be printed.

Raises: TypeError

`minimalmodbus._get_diagnostic_string` ()

Generate a diagnostic string, showing the module version, the platform etc.

Returns: A descriptive string.

`minimalmodbus._getDiagnosticString` ()

Generate a diagnostic string, showing the module version, the platform etc.

Returns: A descriptive string.

15.3 Internal documentation for unit testing of MinimalModbus

`test_minimalmodbus`: Unittests for the `minimalmodbus` module.

For each function are these tests performed:

- Known results
- Invalid input value
- Invalid input type

This unittest suite uses a mock/dummy serial port from the module `dummy_serial`, so it is possible to test the functionality using previously recorded communication data.

With dummy responses, it is also possible to simulate errors in the communication from the slave. A few different types of communication errors are tested, as seen in this table.

Simulated response error	Tested using function	Tested using Modbus function code
No response	read_bit	2
Wrong CRC in response	write_register	16
Wrong slave address in response	write_register	16
Wrong function code in response	write_register	16
Slave indicates an error	write_register	16
Wrong byte count in response	read_bit	2
Wrong register address in response	write_register	16
Wrong number of registers in response	write_bit	15
Wrong number of registers in response	write_register	16
Wrong write data in response	write_bit	5
Wrong write data in response	write_register	6

`test_minimalmodbus.ALSO_TIME_CONSUMING_TESTS = True`

Set this to `False` to skip the most time consuming tests

`test_minimalmodbus.VERBOSITY = 0`

Verbosity level for the unit testing. Use value 0 or 2. Note that it only has an effect for Python 2.7 and above.

`test_minimalmodbus.SHOW_ERROR_MESSAGES_FOR_ASSERTRAISES = False`

Set this to `True` for printing the error messages caught by `assertRaises()`.

If set to `True`, any unintentional error messages raised during the processing of the command in `assertRaises()` are also caught (not counted). It will be printed in the short form, and will show no traceback. It can also be useful to set `VERBOSITY = 2`.

exception `test_minimalmodbus._NonexistentError`

class `test_minimalmodbus.ExtendedTestCase` (*methodName='runTest'*)

Overriding the `assertRaises()` method to be able to print the error message.

Use `test_minimalmodbus.SHOW_ERROR_MESSAGES_FOR_ASSERTRAISES = True` in order to use this option. It can also be useful to set `test_minimalmodbus.VERBOSITY = 2`.

Based on <https://stackoverflow.com/questions/8672754/how-to-show-the-error-messages-caught-by-assertraises-in-unittest-in-py>

assertRaises (*excClass, callableObj, *args, **kwargs*)

Prints the caught error message (if `SHOW_ERROR_MESSAGES_FOR_ASSERTRAISES` is `True`).

assertAlmostEqualRatio (*first, second, epsilon=1.000001*)

A function to compare floats, with ratio instead of difference.

Args:

- first (float): Input argument for comparison
- second (float): Input argument for comparison
- epsilon (float): Largest allowed ratio of largest to smallest of the two input arguments

class `test_minimalmodbus.TestCreatePayload` (*methodName='runTest'*)

`testKnownValues ()`

`testWrongValues ()`

class `test_minimalmodbus.TestParsePayload` (*methodName='runTest'*)

`testKnownValues ()`


```

    testInvalidPayloads()
class test_minimalmodbus.TestEmbedPayload(methodName='runTest')

    knownValues = [(2, 2, 'rtu', '123', '\x02\x02123X\xc2'), (1, 16, 'rtu', 'ABC', '\x01\x01ABCX\x01')]
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestExtractPayload(methodName='runTest')

    knownValues = [(2, 2, 'rtu', '123', '\x02\x02123X\xc2'), (1, 16, 'rtu', 'ABC', '\x01\x01ABCX\x01')]
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityEmbedExtractPayload(methodName='runTest')

    knownValues = [(2, 2, 'rtu', '123', '\x02\x02123X\xc2'), (1, 16, 'rtu', 'ABC', '\x01\x01ABCX\x01')]
    testKnownValues()
    testRange()
class test_minimalmodbus.TestPredictResponseSize(methodName='runTest')

    knownValues = [('rtu', 1, '\x00>\x00\x01', 6), ('rtu', 1, '\x00>\x00\x07', 6), ('rtu', 1, '\x00>\x00\x0f', 6)]
    testKnownValues()
    testRecordedRtuMessages()
    testRecordedAsciiMessages()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestCalculateMinimumSilentPeriod(methodName='runTest')

    knownValues = [(2400, 0.016), (2400.0, 0.016), (4800, 0.008), (9600, 0.004), (19200, 0.002)]
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestNumToOneByteString(methodName='runTest')

    knownValues = [(0, '\x00'), (7, '\x07'), (255, '\xff')]
    testKnownValues()
    testKnownLoop()
    testWrongInput()

```

```
    testWrongType()
class test_minimalmodbus.TestNumToTwoByteString (methodName='runTest')

    knownValues = [(0.0, 0, False, False, '\x00\x00'), (0, 0, False, False, '\x00\x00'), (
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestTwoByteStringToNum (methodName='runTest')

    knownValues = [(0.0, 0, False, False, '\x00\x00'), (0, 0, False, False, '\x00\x00'), (
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityTwoByteString (methodName='runTest')

    knownValues = [(0.0, 0, False, False, '\x00\x00'), (0, 0, False, False, '\x00\x00'), (
    testSanity()
class test_minimalmodbus.TestBytestringToBits (methodName='runTest')

    knownValues = [('\x00', 1, [0]), ('\x01', 1, [1]), ('\x02', 2, [0, 1]), ('\x04', 3, [0
    testKnownValues()
    testWrongValues()
class test_minimalmodbus.TestBitsToBytestring (methodName='runTest')

    knownValues = [('\x00', 1, [0]), ('\x01', 1, [1]), ('\x02', 2, [0, 1]), ('\x04', 3, [0
    testKnownValues()
    testWrongValues()
class test_minimalmodbus.TestBitToBytestring (methodName='runTest')

    knownValues = [(0, '\x00\x00'), (1, '\xff\x00')]
    testKnownValues()
    testWrongValue()
    testValueNotInteger()
class test_minimalmodbus.TestCalculateNumberOfBytesForBits (methodName='runTest')

    knownValues = [(0, 0), (1, 1), (2, 1), (3, 1), (4, 1), (5, 1), (6, 1), (7, 1), (8, 1),
    testKnownValues()
class test_minimalmodbus.TestLongToBytestring (methodName='runTest')
```

```

    knownValues = [(0, True, 0, '\x00\x00\x00\x00'), (1, False, 0, '\x00\x00\x00\x01'), (1,
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestBytestringToLong (methodName='runTest')

    knownValues = [(0, True, 0, '\x00\x00\x00\x00'), (1, False, 0, '\x00\x00\x00\x01'), (1,
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityLong (methodName='runTest')

    knownValues = [(0, True, 0, '\x00\x00\x00\x00'), (1, False, 0, '\x00\x00\x00\x01'), (1,
    testSanity()
class test_minimalmodbus.TestFloatToBytestring (methodName='runTest')

    knownValues = [(1, 2, 0, '?\x80\x00\x00'), (1.0, 2, 0, '?\x80\x00\x00'), (1.0, 2, 0, '
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestBytestringToFloat (methodName='runTest')

    knownValues = [(1, 2, 0, '?\x80\x00\x00'), (1.0, 2, 0, '?\x80\x00\x00'), (1.0, 2, 0, '
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityFloat (methodName='runTest')

    knownValues = [(1, 2, 0, '?\x80\x00\x00'), (1.0, 2, 0, '?\x80\x00\x00'), (1.0, 2, 0, '
    testSanity()
class test_minimalmodbus.TestValuelistToBytestring (methodName='runTest')

    knownValues = [[[1], 1, '\x00\x01'], ([0, 0], 2, '\x00\x00\x00\x00'), ([1, 2], 2, '\x0
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestBytestringToValuelist (methodName='runTest')

```

```
    knownValues = [[1], 1, '\x00\x01'), ([0, 0], 2, '\x00\x00\x00\x00'), ([1, 2], 2, '\x00\x01\x02')
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityValuelist (methodName='runTest')

    knownValues = [[1], 1, '\x00\x01'), ([0, 0], 2, '\x00\x00\x00\x00'), ([1, 2], 2, '\x00\x01\x02')
    testSanity()
class test_minimalmodbus.TestTextstringToBytestring (methodName='runTest')

    knownValues = [('A', 1, 'A '), ('AB', 1, 'AB'), ('ABC', 2, 'ABC '), ('ABCD', 2, 'ABCD')]
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestBytestringToTextstring (methodName='runTest')

    knownValues = [('A', 1, 'A '), ('AB', 1, 'AB'), ('ABC', 2, 'ABC '), ('ABCD', 2, 'ABCD')]
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityTextstring (methodName='runTest')

    knownValues = [('A', 1, 'A '), ('AB', 1, 'AB'), ('ABC', 2, 'ABC '), ('ABCD', 2, 'ABCD')]
    testSanity()
class test_minimalmodbus.TestTimestamp (methodName='runTest')

    testGood()
class test_minimalmodbus.TestPack (methodName='runTest')

    knownValues = [(-77, '>h', '\xff\xb3'), (-1, '>h', '\xff\xff'), (-770, '>h', '\xfc\xfe')]
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestUnpack (methodName='runTest')

    knownValues = [(-77, '>h', '\xff\xb3'), (-1, '>h', '\xff\xff'), (-770, '>h', '\xfc\xfe')]
    testKnownValues()
    testWrongInputValue()
```

```

    testWrongInputType()
class test_minimalmodbus.TestSwap (methodName='runTest')

    knownValues = [('', ''), ('AB', 'BA'), ('ABCD', 'BADC'), ('ABCDEF', 'BADCFE'), ('ABCDE
    wrongValues = ['A', 'ABC', 'ABCDE', 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
    testKnownValues()
    testWrongValues()
class test_minimalmodbus.TestSanityPackUnpack (methodName='runTest')

    knownValues = [(-77, '>h', '\xff\xb3'), (-1, '>h', '\xff\xff'), (-770, '>h', '\xfc\xfe
    testSanity()
class test_minimalmodbus.TestHexencode (methodName='runTest')

    knownValues = [('', False, ''), ('7', False, '37'), ('J', False, '4A'), (']', False, '
    testKnownValues()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestHexdecode (methodName='runTest')

    knownValues = [('', False, ''), ('7', False, '37'), ('J', False, '4A'), (']', False, '
    testKnownValues()
    testAllowLowercase()
    testWrongInputValue()
    testWrongInputType()
class test_minimalmodbus.TestSanityHexencodeHexdecode (methodName='runTest')

    knownValues = [('', False, ''), ('7', False, '37'), ('J', False, '4A'), (']', False, '
    testKnownValues()
    testKnownValuesLoop()
        Loop through all bytestrings of length two.
class test_minimalmodbus.TestTwosComplement (methodName='runTest')

    knownValues = [(0, 8, 0), (1, 8, 1), (127, 8, 127), (-128, 8, 128), (-127, 8, 129), (-
    testKnownValues()
    testOutOfRange()
    testWrongInputType()
class test_minimalmodbus.TestFromTwosComplement (methodName='runTest')

    knownValues = [(0, 8, 0), (1, 8, 1), (127, 8, 127), (-128, 8, 128), (-127, 8, 129), (-

```

```
    testKnownValues ()
    testOutOfRange ()
    testWrongInputType ()
class test_minimalmodbus.TestSanityTwosComplement (methodName='runTest')

    knownValues = [1, 2, 4, 8, 12, 16]
    testSanity ()
class test_minimalmodbus.TestSetBitOn (methodName='runTest')

    knownValues = [(4, 0, 5), (4, 1, 6), (1, 1, 3)]
    testKnownValues ()
    testWrongInputValue ()
    testWrongInputType ()
class test_minimalmodbus.TestCheckBit (methodName='runTest')

    knownValues = [(0, 0, False), (0, 1, False), (0, 2, False), (0, 3, False), (0, 4, False)]
    testKnownValues ()
    testWrongInputValue ()
    testWrongInputType ()
class test_minimalmodbus.TestCalculateCrcString (methodName='runTest')

    knownValues = [('\x02\x07', 'A\x12'), ('ABCDE', '\x0fP')]
    testKnownValues ()
    testCalculationTime ()
    testNotStringInput ()
class test_minimalmodbus.TestCalculateLrcString (methodName='runTest')

    knownValues = [('ABCDE', '\xb1'), ('\x02001#\x03', 'G')]
    testKnownValues ()
    testNotStringInput ()
class test_minimalmodbus.TestCheckFunctioncode (methodName='runTest')

    testCorrectFunctioncode ()
    testCorrectFunctioncodeNoRange ()
    testWrongFunctioncode ()
    testWrongFunctioncodeNoRange ()
    testWrongFunctioncodeType ()
    testWrongFunctioncodeListValues ()
```

```

    testWrongListType ()
class test_minimalmodbus.TestCheckSlaveaddress (methodName='runTest')

    testKnownValues ()
    testWrongValues ()
    testNotIntegerInput ()
class test_minimalmodbus.TestCheckMode (methodName='runTest')

    testKnownValues ()
    testWrongValues ()
    testNotIntegerInput ()
class test_minimalmodbus.TestCheckRegisteraddress (methodName='runTest')

    testKnownValues ()
    testWrongValues ()
    testWrongType ()
class test_minimalmodbus.TestCheckResponseSlaveErrorCode (methodName='runTest')

    testResponsesWithoutErrors ()
    testResponsesWithErrors ()
    testTooShortResponses ()
class test_minimalmodbus.TestCheckResponseNumberOfBytes (methodName='runTest')

    testCorrectNumberOfBytes ()
    testWrongNumberOfBytes ()
    testNotStringInput ()
class test_minimalmodbus.TestCheckResponseRegisterAddress (methodName='runTest')

    testCorrectResponseRegisterAddress ()
    testTooShortString ()
    testNotString ()
    testWrongResponseRegisterAddress ()
    testInvalidAddress ()
    testAddressNotInteger ()
class test_minimalmodbus.TestCheckResponsenumber_of_registers (methodName='runTest')

    testCorrectResponsenumber_of_registers ()
    testTooShortString ()

```

```
testNotString()
testWrongResponsernumber_of_registers()
testInvalidResponsernumber_of_registersRange()
testnumber_of_registersNotInteger()
class test_minimalmodbus.TestCheckResponseWriteData (methodName='runTest')

testCorrectResponseWritedata()
testWrongResponseWritedata()
testNotString()
testTooShortPayload()
testInvalidReferenceData()
class test_minimalmodbus.TestCheckString (methodName='runTest')

testKnownValues()
testTooShort()
testTooLong()
testNotAscii()
testInconsistentLengthlimits()
testInputNotString()
testNotIntegerInput()
testDescriptionNotString()
testWrongCustomError()
testCustomError()
class test_minimalmodbus.TestCheckInt (methodName='runTest')

testKnownValues()
testTooLargeValue()
testTooSmallValue()
testInconsistentLimits()
testWrongInputType()
class test_minimalmodbus.TestCheckNumerical (methodName='runTest')

testKnownValues()
testTooLargeValue()
testTooSmallValue()
testInconsistentLimits()
testNotNumericInput()
```



```

    testDescriptionNotString ()
class test_minimalmodbus.TestCheckBool (methodName='runTest')

    testKnownValues ()
    testWrongType ()
class test_minimalmodbus.TestGetDiagnosticString (methodName='runTest')

    testReturnsString ()
class test_minimalmodbus.TestPrintOut (methodName='runTest')

    testKnownValues ()
    testInputNotString ()
class test_minimalmodbus.TestDummyCommunication (methodName='runTest')

    setUp ()
        Hook method for setting up the test fixture before exercising it.
    testReadBit ()
    testReadBitWrongValue ()
    testReadBitWrongType ()
    testReadBitWithWrongByteCountResponse ()
    testReadBitWithNoResponse ()
    testWriteBit ()
    testWriteBitWrongValue ()
    testWriteBitWrongType ()
    testWriteBitWithWrongRegisternumbersResponse ()
    testWriteBitWithWrongWritedataResponse ()
    testReadBits ()
    testReadBitsWrongValue ()
    testWriteBits ()
    testWriteBitsWrongValue ()
    testReadRegister ()
    testReadRegisterWrongValue ()
    testReadRegisterWrongType ()
    testWriteRegister ()
    testWriteRegisterWithDecimals ()
    testWriteRegisterWrongValue ()
    testWriteRegisterWrongType ()
    testWriteRegisterWithWrongCrcResponse ()

```

```
testWriteRegisterSuppressErrorMessageAtWrongCRC ()
testWriteRegisterWithWrongSlaveaddressResponse ()
testWriteRegisterWithWrongFunctioncodeResponse ()
testWriteRegisterWithWrongRegisteraddressResponse ()
testWriteRegisterWithWrongRegisternumbersResponse ()
testWriteRegisterWithWrongWritedataResponse ()
testReadLong ()
testReadLongWrongValue ()
testReadLongWrongType ()
testWriteLong ()
testWriteLongWrongValue ()
testWriteLongWrongType ()
testReadFloat ()
testReadFloatWrongValue ()
testReadFloatWrongType ()
testWriteFloat ()
testWriteFloatWrongValue ()
testWriteFloatWrongType ()
testReadString ()
testReadStringWrongValue ()
testReadStringWrongType ()
testWriteString ()
testWriteStringWrongValue ()
testWriteStringWrongType ()
testReadRegisters ()
testReadRegistersWrongValue ()
testReadRegistersWrongType ()
testWriteRegisters ()
testWriteRegistersWrongValue ()
testWriteRegistersWrongType ()
testGenericCommand ()
testGenericCommandWrongValue ()
testGenericCommandWrongType ()
testGenericCommandWrongValueCombinations ()
testPerformcommandKnownResponse ()
testPerformcommandWrongSlaveResponse ()
```

```

testPerformcommandWrongInputValue ()
testPerformcommandWrongInputType ()
testCommunicateKnownResponse ()
testCommunicateWrongType ()
testCommunicateNoMessage ()
testCommunicateNoResponse ()
testCommunicateLocalEcho ()
testCommunicateWrongLocalEcho ()
testPortWillBeOpened ()
testRepresentation ()
testReadPortClosed ()
testPortAlreadyOpen ()
testPortAlreadyClosed ()
tearDown ()

```

Hook method for deconstructing the test fixture after testing it.

```
class test_minimalmodbus.TestDummyCommunicationOmegaSlave1 (methodName='runTest')
```

```

setUp ()
    Hook method for setting up the test fixture before exercising it.
testReadBit ()
testWriteBit ()
testReadRegister ()
testWriteRegister ()
tearDown ()

```

Hook method for deconstructing the test fixture after testing it.

```
class test_minimalmodbus.TestDummyCommunicationOmegaSlave10 (methodName='runTest')
```

```

setUp ()
    Hook method for setting up the test fixture before exercising it.
testReadBit ()
testWriteBit ()
testReadRegister ()
testWriteRegister ()
tearDown ()

```

Hook method for deconstructing the test fixture after testing it.

```
class test_minimalmodbus.TestDummyCommunicationDTB4824_RTU (methodName='runTest')
```

```

setUp ()
    Hook method for setting up the test fixture before exercising it.

```

testReadBit ()

testWriteBit ()

testReadBits ()

testReadRegister ()

testReadRegisters ()

testWriteRegister ()

tearDown ()

Hook method for deconstructing the test fixture after testing it.

class test_minimalmodbus.**TestDummyCommunicationDTB4824_ASCII** (*methodName='runTest'*)

setUp ()

Hook method for setting up the test fixture before exercising it.

testReadBit ()

testWriteBit ()

testReadBits ()

testReadRegister ()

testReadRegisters ()

testWriteRegister ()

tearDown ()

Hook method for deconstructing the test fixture after testing it.

class test_minimalmodbus.**TestDummyCommunicationWithPortClosure** (*methodName='runTest'*)

setUp ()

Hook method for setting up the test fixture before exercising it.

testReadRegisterSeveralTimes ()

testPortAlreadyClosed ()

tearDown ()

Hook method for deconstructing the test fixture after testing it.

class test_minimalmodbus.**TestVerboseDummyCommunicationWithPortClosure** (*methodName='runTest'*)

setUp ()

Hook method for setting up the test fixture before exercising it.

testReadRegister ()

tearDown ()

Hook method for deconstructing the test fixture after testing it.

class test_minimalmodbus.**TestDummyCommunicationThreeInstrumentsPortClosure** (*methodName='runTest'*)

setUp ()

Hook method for setting up the test fixture before exercising it.

testCommunication ()

tearDown()

Hook method for deconstructing the test fixture after testing it.

class test_minimalmodbus.**TestDummyCommunicationHandleLocalEcho** (*methodName='runTest'*)

setUp()

Hook method for setting up the test fixture before exercising it.

testReadRegister()

testReadRegisterWrongEcho()

tearDown()

Hook method for deconstructing the test fixture after testing it.

test_minimalmodbus.**WRONG_ASCII_RESPONSES** = {}

A dictionary of responses from a dummy instrument.

The key is the message (string) sent to the serial port, and the item is the response (string) from the dummy serial port.

15.4 Internal documentation for hardware testing of MinimalModbus using DTB4824

Hardware testing of MinimalModbus using the Delta DTB temperature controller.

For use with Delta DTB4824VR.

15.4.1 Usage

```
python3 scriptname [-rtu] [-ascii] [-b38400] [-D/dev/ttyUSB0]
```

Arguments:

- -b : baud rate
- -D : port name

NOTE: There should be no space between the option switch and its argument.

Defaults to RTU mode.

15.4.2 Recommended test sequence

Make sure that RUN_VERIFY_EXAMPLES and similar flags are all 'True'.

- Run the tests under Linux and Windows
- Use 2400 bps and 38400 bps
- Use Modbus ASCII and Modbus RTU
- Use Python 2.7 and Python 3.x

Sequence (for each use Python 2.7 and 3.x):

- 38400 bps RTU

- 38400 bps ASCII
- 2400 bps ASCII
- 2400 bps RTU

15.4.3 Settings in the temperature controller

To change the settings on the temperature controller panel, hold the SET button for more than 3 seconds. Use the ‘loop arrow’ button for moving to next parameter. Change the value with the up and down arrows, and confirm using the SET button. Press SET again to exit setting mode.

Use these setting values in the temperature controller:

- SP 1 (Decimal point position)
- CoSH on (ON: communication write-in enabled)
- C-SL rtu (use RTU or ASCII)
- C-no 1 (Slave number)
- BPS (see the DEFAULT_BAUDRATE setting below, or the command line argument)
- LEN 8
- PRTY None
- Stop 1

When running, the setpoint is seen on the rightmost part of the display.

15.4.4 USB-to-RS485 converter

BOB-09822 USB to RS-485 Converter:

- <https://www.sparkfun.com/products/9822>
- SP3485 RS-485 transceiver
- FT232RL USB UART IC
- FT232RL pin2: RE^
- FT232RL pin3: DE

DTB4824 terminal	USB-RS485 terminal	Description
DATA+	A	Positive at idle
DATA-	B	Negative at idle

Sometimes after changing the baud rate, there is no communication with the temperature controller. Reset the FTDI chip by unplugging and replugging the USB-to-RS485 converter.

15.4.5 Function codes for DTB4824

From “DTB Series Temperature Controller Instruction Sheet”:

- 02H to read the bits data (Max. 16 bits).
- 03H to read the contents of register (Max. 8 words).

- 05H to write 1 (one) bit into register.
- 06H to write 1 (one) word into register.

15.4.6 Manual testing in interactive mode (at the Python prompt)

Use a setting of 19200 bps, RTU mode and slave address 1 for the DTB4824. Run these commands:

```
import minimalmodbus
instrument = minimalmodbus.Instrument('/dev/ttyUSB0', 1) # Adjust if necessary.
instrument.debug = True
instrument.read_register(4143) # Read firmware version (address in hex is 0x102F)
```

```
test_deltaDTB4824._print_out(text)
test_deltaDTB4824._box(description=None, value=None)
test_deltaDTB4824.show_test_settings(mode, baudrate, portname)
test_deltaDTB4824.show_current_values(instr)
test_deltaDTB4824.show_instrument_settings(instr)
test_deltaDTB4824.verify_value_for_register(instr, value)
test_deltaDTB4824.verify_state_for_bits(instr, state)
test_deltaDTB4824.verify_bits(instr)
test_deltaDTB4824.verify_readonly_register(instr)
test_deltaDTB4824.verify_register(instr)
test_deltaDTB4824.verify_two_instrument_instances(instr, portname, mode, baudrate)
test_deltaDTB4824.measure_roundtrip_time(instr)
test_deltaDTB4824.parse_commandline(argv)
test_deltaDTB4824.main()
```


CHAPTER 16

Indices and tables

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

d

`dummy_serial`, 75

m

`minimalmodbus`, 11

t

`test_deltaDTB4824`, 111

`test_minimalmodbus`, 97

Symbols

`_NonexistantError`, 98
`_box()` (in module `test_deltaDTB4824`), 113
`_print_out()` (in module `test_deltaDTB4824`), 113

A

`address` (*minimalmodbus.Instrument* attribute), 11
`ALSO_TIME_CONSUMING_TESTS` (in module *test_minimalmodbus*), 98
`assertAlmostEqualTo()` (*test_minimalmodbus.ExtendedTestCase* method), 98
`assertRaises()` (*test_minimalmodbus.ExtendedTestCase* method), 98

C

`clear_buffers_before_each_transaction` (*minimalmodbus.Instrument* attribute), 12
`close()` (*dummy_serial.Serial* method), 76
`close_port_after_each_call` (*minimalmodbus.Instrument* attribute), 12

D

`debug` (*minimalmodbus.Instrument* attribute), 11
`DEFAULT_BAUDRATE` (in module *dummy_serial*), 75
`DEFAULT_RESPONSE` (in module *dummy_serial*), 75
`DEFAULT_TIMEOUT` (in module *dummy_serial*), 75
`dummy_serial` (module), 75

E

environment variable
`PYTHONPATH`, 59
`ExtendedTestCase` (class in *test_minimalmodbus*), 98

H

`handle_local_echo` (*minimalmodbus.Instrument* attribute), 12

I

`IllegalRequestError`, 18
`Instrument` (class in *minimalmodbus*), 11
`InvalidResponseError`, 18
`is_open` (*dummy_serial.Serial* attribute), 76

K

`knownValues` (*test_minimalmodbus.TestBitsToBytestring* attribute), 100
`knownValues` (*test_minimalmodbus.TestBitToBytestring* attribute), 100
`knownValues` (*test_minimalmodbus.TestBytestringToBits* attribute), 100
`knownValues` (*test_minimalmodbus.TestBytestringToFloat* attribute), 101
`knownValues` (*test_minimalmodbus.TestBytestringToLong* attribute), 101
`knownValues` (*test_minimalmodbus.TestBytestringToTextstring* attribute), 102
`knownValues` (*test_minimalmodbus.TestBytestringToValuelist* attribute), 101
`knownValues` (*test_minimalmodbus.TestCalculateCrcString* attribute), 104
`knownValues` (*test_minimalmodbus.TestCalculateLrcString* attribute), 104
`knownValues` (*test_minimalmodbus.TestCalculateMinimumSilentPeriod* attribute), 99
`knownValues` (*test_minimalmodbus.TestCalculateNumberOfBytesForBits* attribute), 100
`knownValues` (*test_minimalmodbus.TestCheckBit* attribute), 104
`knownValues` (*test_minimalmodbus.TestEmbedPayload* attribute), 99
`knownValues` (*test_minimalmodbus.TestExtractPayload* attribute), 99
`knownValues` (*test_minimalmodbus.TestFloatToBytestring* attribute), 101
`knownValues` (*test_minimalmodbus.TestFromTwosComplement* attribute), 103

knownValues (*test_minimalmodbus.TestHexdecode attribute*), 103

knownValues (*test_minimalmodbus.TestHexencode attribute*), 103

knownValues (*test_minimalmodbus.TestLongToBytestring attribute*), 100

knownValues (*test_minimalmodbus.TestNumToOneByteString attribute*), 99

knownValues (*test_minimalmodbus.TestNumToTwoByteString attribute*), 100

knownValues (*test_minimalmodbus.TestPack attribute*), 102

knownValues (*test_minimalmodbus.TestPredictResponseSize attribute*), 99

knownValues (*test_minimalmodbus.TestSanityEmbedExtractPayload attribute*), 99

knownValues (*test_minimalmodbus.TestSanityFloat attribute*), 101

knownValues (*test_minimalmodbus.TestSanityHexencodeHexdecode attribute*), 103

knownValues (*test_minimalmodbus.TestSanityLong attribute*), 101

knownValues (*test_minimalmodbus.TestSanityPackUnpack attribute*), 103

knownValues (*test_minimalmodbus.TestSanityTextstring attribute*), 102

knownValues (*test_minimalmodbus.TestSanityTwoByteString attribute*), 100

knownValues (*test_minimalmodbus.TestSanityTwosComplement attribute*), 104

knownValues (*test_minimalmodbus.TestSanityValuelist attribute*), 102

knownValues (*test_minimalmodbus.TestSetBitOn attribute*), 104

knownValues (*test_minimalmodbus.TestSwap attribute*), 103

knownValues (*test_minimalmodbus.TestTextstringToBytestring attribute*), 102

knownValues (*test_minimalmodbus.TestTwoByteStringToNum attribute*), 100

knownValues (*test_minimalmodbus.TestTwosComplement attribute*), 103

knownValues (*test_minimalmodbus.TestUnpack attribute*), 102

knownValues (*test_minimalmodbus.TestValuelistToBytestring attribute*), 101

L

LocalEchoError, 18

M

main() (in module *test_deltaDTB4824*), 113

MasterReportedException, 18

measure_roundtrip_time() (in module *test_deltaDTB4824*), 113

minimalmodbus (module), 11

ModbusException, 18

mode (*minimalmodbus.Instrument attribute*), 11

N

NegativeAcknowledgeError, 18

NoResponseError, 18

O

open() (*dummy_serial.Serial method*), 76

P

parse_commandline() (in module *test_deltaDTB4824*), 113

precalculate_read_size (*minimalmodbus.Instrument attribute*), 11

Python Enhancement Proposals

PEP 440, 55

PYTHONPATH, 59

R

read() (*dummy_serial.Serial method*), 76

read_bit() (*minimalmodbus.Instrument method*), 12

read_bits() (*minimalmodbus.Instrument method*), 13

read_float() (*minimalmodbus.Instrument method*), 15

read_long() (*minimalmodbus.Instrument method*), 15

read_register() (*minimalmodbus.Instrument method*), 13

read_registers() (*minimalmodbus.Instrument method*), 17

read_string() (*minimalmodbus.Instrument method*), 16

reset_input_buffer() (*dummy_serial.Serial method*), 76

reset_output_buffer() (*dummy_serial.Serial method*), 76

RESPONSES (in module *dummy_serial*), 75

S

Serial (class in *dummy_serial*), 75

serial (*minimalmodbus.Instrument attribute*), 12

setUp() (*test_minimalmodbus.TestDummyCommunication method*), 107

setUp() (*test_minimalmodbus.TestDummyCommunicationDTB4824_ASC method*), 110

setUp() (*test_minimalmodbus.TestDummyCommunicationDTB4824_RTU method*), 109

setUp() (*test_minimalmodbus.TestDummyCommunicationHandleLocalE method*), 111

setUp () (*test_minimalmodbus.TestDummyCommunicationOneByteStringToFloat* (class in *test_minimalmodbus*), 109
 setUp () (*test_minimalmodbus.TestDummyCommunicationOneByteStringToLong* (class in *test_minimalmodbus*), 109
 setUp () (*test_minimalmodbus.TestDummyCommunicationFirstInstrumentFromClosureString* (class in *test_minimalmodbus*), 110
 setUp () (*test_minimalmodbus.TestDummyCommunicationWithPortClosureStringToValueList* (class in *test_minimalmodbus*), 110
 setUp () (*test_minimalmodbus.TestVerboseDummyCommunicationWithPortClosureString* (class in *test_minimalmodbus*), 110
 show_current_values () (in module *TestCalculateLrcString* (class in *test_minimalmodbus*), 113
 SHOW_ERROR_MESSAGES_FOR_ASSERTRAISES (in module *test_minimalmodbus*), 98
 show_instrument_settings () (in module *TestCalculateMinimumSilentPeriod* (class in *test_minimalmodbus*), 113
 show_test_settings () (in module *TestCalculateNumberOfBytesForBits* (class in *test_minimalmodbus*), 113
 SlaveDeviceBusyError, 18
 SlaveReportedException, 18
T
 tearDown () (*test_minimalmodbus.TestDummyCommunicationOneByteStringToFloat* (class in *test_minimalmodbus*), 109
 tearDown () (*test_minimalmodbus.TestDummyCommunicationOneByteStringToLong* (class in *test_minimalmodbus*), 110
 tearDown () (*test_minimalmodbus.TestDummyCommunicationFirstInstrumentFromClosureString* (class in *test_minimalmodbus*), 110
 tearDown () (*test_minimalmodbus.TestDummyCommunicationWithPortClosureStringToValueList* (class in *test_minimalmodbus*), 110
 tearDown () (*test_minimalmodbus.TestDummyCommunicationHandshakeLocalEcho* (class in *test_minimalmodbus*), 111
 tearDown () (*test_minimalmodbus.TestDummyCommunicationOmeGasSlave* (class in *test_minimalmodbus*), 109
 tearDown () (*test_minimalmodbus.TestDummyCommunicationOmeGasSlave10* (class in *test_minimalmodbus*), 109
 tearDown () (*test_minimalmodbus.TestDummyCommunicationThreadInterruptible* (class in *test_minimalmodbus*), 110
 tearDown () (*test_minimalmodbus.TestDummyCommunicationWithPortClosureString* (class in *test_minimalmodbus*), 110
 tearDown () (*test_minimalmodbus.TestVerboseDummyCommunicationWithPortClosureString* (class in *test_minimalmodbus*), 110
 test_deltaDTB4824 (module), 111
 test_minimalmodbus (module), 97
 testAddressNotInteger () (*test_minimalmodbus.TestCalculateLrcString* (class in *test_minimalmodbus*), 105
 testAllowLowercase () (*test_minimalmodbus.TestHexdecode* method), 103
 TestBitsToBytestring (class in *test_minimalmodbus*), 100
 TestBitToBytestring (class in *test_minimalmodbus*), 100
 TestBytestringToBits (class in *test_minimalmodbus*), 100
 TestCalculateLrcString (class in *test_minimalmodbus*), 104
 TestCalculateMinimumSilentPeriod (class in *test_minimalmodbus*), 99
 TestCalculateNumberOfBytesForBits (class in *test_minimalmodbus*), 100
 TestCalculationTime () (*test_minimalmodbus.TestCalculateCrcString* method), 104
 TestCheckBit (class in *test_minimalmodbus*), 104
 TestCheckBool (class in *test_minimalmodbus*), 107
 TestCheckFunctioncode (class in *test_minimalmodbus*), 104
 TestCheckInt (class in *test_minimalmodbus*), 106
 TestCheckNumerical (class in *test_minimalmodbus*), 105
 TestCheckRegisteraddress (class in *test_minimalmodbus*), 105
 TestCheckResponseNumber_of_registers (class in *test_minimalmodbus*), 105
 TestCheckResponseNumberOfBytes (class in *test_minimalmodbus*), 105
 TestCheckResponseRegisterAddress (class in *test_minimalmodbus*), 105
 TestCheckResponseSlaveErrorCode (class in *test_minimalmodbus*), 105
 TestCheckResponseWriteData (class in *test_minimalmodbus*), 105
 TestCheckSlaveaddress (class in *test_minimalmodbus*), 105
 TestCheckString (class in *test_minimalmodbus*), 106
 TestCommunicateKnownResponse () (*test_minimalmodbus.TestDummyCommunication* method), 109
 TestCommunicateLocalEcho () (*test_minimalmodbus.TestDummyCommunication* method), 109
 TestCommunicateNoMessage () (*test_minimalmodbus.TestDummyCommunication* method), 109
 TestCommunicateNoResponse () (*test_minimalmodbus.TestDummyCommunication* method), 109

<i>method</i>), 109	<i>(class in test_minimalmodbus)</i> , 110
testCommunicateWrongLocalEcho () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 109	TestEmbedPayload (<i>class in test_minimalmodbus</i>), 99
testCommunicateWrongType () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 109	TestExtractPayload (<i>class in test_minimalmodbus</i>), 99
testCommunication () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 110	TestFloatToBytestring (<i>class in test_minimalmodbus</i>), 101
testCorrectFunctioncode () (<i>test_minimalmodbus.TestCheckFunctioncode method</i>), 104	TestFromTwosComplement (<i>class in test_minimalmodbus</i>), 103
testCorrectFunctioncodeNoRange () (<i>test_minimalmodbus.TestCheckFunctioncode method</i>), 104	testGenericCommand () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 108
testCorrectNumberOfBytes () (<i>test_minimalmodbus.TestCheckResponseNumberOfBytes method</i>), 105	testGenericCommandWrongType () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 108
testCorrectResponseNumber_of_registers () (<i>test_minimalmodbus.TestCheckResponseNumber_of_registers method</i>), 105	testGenericCommandWrongValue () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 108
testCorrectResponseRegisterAddress () (<i>test_minimalmodbus.TestCheckResponseRegisterAddress method</i>), 105	testGenericCommandWrongValueCombinations () (<i>test_minimalmodbus.TestDummyCommunication method</i>), 108
testCorrectResponseWritedata () (<i>test_minimalmodbus.TestCheckResponseWriteData method</i>), 106	TestGetDiagnosticString (<i>class in test_minimalmodbus</i>), 107
TestCreatePayload (<i>class in test_minimalmodbus</i>), 98	TestGood () (<i>test_minimalmodbus.TestTimestamp method</i>), 102
testCustomError () (<i>test_minimalmodbus.TestCheckString method</i>), 106	TestHexdecode (<i>class in test_minimalmodbus</i>), 103
testDescriptionNotString () (<i>test_minimalmodbus.TestCheckNumerical method</i>), 106	TestHexencode (<i>class in test_minimalmodbus</i>), 103
testDescriptionNotString () (<i>test_minimalmodbus.TestCheckString method</i>), 106	testInconsistentLengthlimits () (<i>test_minimalmodbus.TestCheckString method</i>), 106
TestDummyCommunication (<i>class in test_minimalmodbus</i>), 107	testInconsistentLimits () (<i>test_minimalmodbus.TestCheckInt method</i>), 106
TestDummyCommunicationDTB4824_ASCII (<i>class in test_minimalmodbus</i>), 110	testInconsistentLimits () (<i>test_minimalmodbus.TestCheckNumerical method</i>), 106
TestDummyCommunicationDTB4824_RTU (<i>class in test_minimalmodbus</i>), 109	testInputNotString () (<i>test_minimalmodbus.TestCheckString method</i>), 106
TestDummyCommunicationHandleLocalEcho (<i>class in test_minimalmodbus</i>), 111	testInputNotString () (<i>test_minimalmodbus.TestPrintOut method</i>), 107
TestDummyCommunicationOmegaSlave1 (<i>class in test_minimalmodbus</i>), 109	testInvalidAddress () (<i>test_minimalmodbus.TestCheckResponseRegisterAddress method</i>), 105
TestDummyCommunicationOmegaSlave10 (<i>class in test_minimalmodbus</i>), 109	testInvalidPayloads () (<i>test_minimalmodbus.TestParsePayload method</i>), 99
TestDummyCommunicationThreeInstrumentsPortIsUsed (<i>class in test_minimalmodbus</i>), 110	testInvalidReferenceData () (<i>test_minimalmodbus.TestCheckResponseWriteData method</i>), 106
TestDummyCommunicationWithPortClosure	testInvalidResponseNumber_of_registersRange () (<i>test_minimalmodbus.TestCheckResponseNumber_of_registers method</i>), 106

testKnownLoop () (<i>test_minimalmodbus.TestNumToOneByteString</i> method), 99	(<i>test_minimalmodbus.TestCheckSlaveaddress</i> method), 105
testKnownValues () (<i>test_minimalmodbus.TestBitsToBytestring</i> method), 100	testKnownValues () (<i>test_minimalmodbus.TestCheckString</i> method), 106
testKnownValues () (<i>test_minimalmodbus.TestBitToBytestring</i> method), 100	testKnownValues () (<i>test_minimalmodbus.TestCreatePayload</i> method), 98
testKnownValues () (<i>test_minimalmodbus.TestBytestringToBits</i> method), 100	testKnownValues () (<i>test_minimalmodbus.TestEmbedPayload</i> method), 99
testKnownValues () (<i>test_minimalmodbus.TestBytestringToFloat</i> method), 101	testKnownValues () (<i>test_minimalmodbus.TestExtractPayload</i> method), 99
testKnownValues () (<i>test_minimalmodbus.TestBytestringToLong</i> method), 101	testKnownValues () (<i>test_minimalmodbus.TestFloatToBytestring</i> method), 101
testKnownValues () (<i>test_minimalmodbus.TestBytestringToTextstring</i> method), 102	testKnownValues () (<i>test_minimalmodbus.TestFromTwosComplement</i> method), 103
testKnownValues () (<i>test_minimalmodbus.TestBytestringToValuelist</i> method), 102	testKnownValues () (<i>test_minimalmodbus.TestHexdecode</i> method), 103
testKnownValues () (<i>test_minimalmodbus.TestCalculateCrcString</i> method), 104	testKnownValues () (<i>test_minimalmodbus.TestHexencode</i> method), 103
testKnownValues () (<i>test_minimalmodbus.TestCalculateLrcString</i> method), 104	testKnownValues () (<i>test_minimalmodbus.TestLongToBytestring</i> method), 101
testKnownValues () (<i>test_minimalmodbus.TestCalculateMinimumSilentPeriod</i> method), 99	testKnownValues () (<i>test_minimalmodbus.TestNumToOneByteString</i> method), 99
testKnownValues () (<i>test_minimalmodbus.TestCalculateNumberOfBytesForBits</i> method), 100	testKnownValues () (<i>test_minimalmodbus.TestNumToTwoByteString</i> method), 100
testKnownValues () (<i>test_minimalmodbus.TestCheckBit</i> method), 104	testKnownValues () (<i>test_minimalmodbus.TestPack</i> method), 102
testKnownValues () (<i>test_minimalmodbus.TestCheckBool</i> method), 107	testKnownValues () (<i>test_minimalmodbus.TestParsePayload</i> method), 98
testKnownValues () (<i>test_minimalmodbus.TestCheckInt</i> method), 106	testKnownValues () (<i>test_minimalmodbus.TestPredictResponseSize</i> method), 99
testKnownValues () (<i>test_minimalmodbus.TestCheckMode</i> method), 105	testKnownValues () (<i>test_minimalmodbus.TestPrintOut</i> method), 107
testKnownValues () (<i>test_minimalmodbus.TestCheckNumerical</i> method), 106	testKnownValues () (<i>test_minimalmodbus.TestSanityEmbedExtractPayload</i> method), 99
testKnownValues () (<i>test_minimalmodbus.TestCheckRegisteraddress</i> method), 105	testKnownValues () (<i>test_minimalmodbus.TestSanityHexencodeHexdecode</i> method), 103
testKnownValues ()	testKnownValues () (<i>test_minimalmodbus.TestSetBitOn</i> method),

104

testKnownValues () (*test_minimalmodbus.TestSwap method*), 103

testKnownValues () (*test_minimalmodbus.TestTextstringToBytestring method*), 102

testKnownValues () (*test_minimalmodbus.TestTwoByteStringToNum method*), 100

testKnownValues () (*test_minimalmodbus.TestTwosComplement method*), 103

testKnownValues () (*test_minimalmodbus.TestUnpack method*), 102

testKnownValues () (*test_minimalmodbus.TestValuelistToBytestring method*), 101

testKnownValuesLoop () (*test_minimalmodbus.TestSanityHexencodeHexdecode method*), 103

TestLongToBytestring (class in *test_minimalmodbus*), 100

testNotAscii () (*test_minimalmodbus.TestCheckString method*), 106

testNotIntegerInput () (*test_minimalmodbus.TestCheckMode method*), 105

testNotIntegerInput () (*test_minimalmodbus.TestCheckSlaveaddress method*), 105

testNotIntegerInput () (*test_minimalmodbus.TestCheckString method*), 106

testNotNumericInput () (*test_minimalmodbus.TestCheckNumerical method*), 106

testNotString () (*test_minimalmodbus.TestCheckResponseNumberOfRegisters method*), 105

testNotString () (*test_minimalmodbus.TestCheckResponseRegisterAddress method*), 105

testNotString () (*test_minimalmodbus.TestCheckResponseWriteData method*), 106

testNotStringInput () (*test_minimalmodbus.TestCalculateCrcString method*), 104

testNotStringInput () (*test_minimalmodbus.TestCalculateLrcString method*), 104

testNotStringInput () (*test_minimalmodbus.TestCheckResponseNumberOfBytes method*), 105

testnumber_of_registersNotInteger () (*test_minimalmodbus.TestCheckResponseNumber of registers method*), 106

TestNumToOneByteString (class in *test_minimalmodbus*), 99

TestNumToTwoByteString (class in *test_minimalmodbus*), 100

testOutOfRange () (*test_minimalmodbus.TestFromTwosComplement method*), 104

testOutOfRange () (*test_minimalmodbus.TestTwosComplement method*), 103

TestPack (class in *test_minimalmodbus*), 102

TestParsePayload (class in *test_minimalmodbus*), 98

testPerformcommandKnownResponse () (*test_minimalmodbus.TestDummyCommunication method*), 108

testPerformcommandWrongInputType () (*test_minimalmodbus.TestDummyCommunication method*), 109

testPerformcommandWrongInputValue () (*test_minimalmodbus.TestDummyCommunication method*), 108

testPerformcommandWrongSlaveResponse () (*test_minimalmodbus.TestDummyCommunication method*), 108

testPortAlreadyClosed () (*test_minimalmodbus.TestDummyCommunication method*), 109

testPortAlreadyClosed () (*test_minimalmodbus.TestDummyCommunicationWithPortClosure method*), 110

testPortAlreadyOpen () (*test_minimalmodbus.TestDummyCommunication method*), 109

testPortWillBeOpened () (*test_minimalmodbus.TestDummyCommunication method*), 109

TestPredictResponseSize (class in *test_minimalmodbus*), 99

TestPrintOut (class in *test_minimalmodbus*), 107

TestSanityEmbedExtractPayload (class in *test_minimalmodbus*), 99

testReadBit () (*test_minimalmodbus.TestDummyCommunicationDTB4 method*), 110

testReadBit () (*test_minimalmodbus.TestDummyCommunicationDTB4 method*), 109

testReadBit () (*test_minimalmodbus.TestDummyCommunicationOmega method*), 109

testReadBit () (*test_minimalmodbus.TestDummyCommunicationOmega method*), 109

testReadBits () (*test_minimalmodbus.TestDummyCommunicationDTB4 method*), 110

[testReturnsString\(\)](#) ([test_minimalmodbus.TestGetDiagnosticString](#) method), 107
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityFloat](#) method), 101
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityLong](#) method), 101
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityPackUnpack](#) method), 103
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityTextstring](#) method), 102
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityTwoByteString](#) method), 100
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityTwosComplement](#) method), 104
[testSanity\(\)](#) ([test_minimalmodbus.TestSanityValuelist](#) method), 102
[TestSanityEmbedExtractPayload](#) (class in [test_minimalmodbus](#)), 99
[TestSanityFloat](#) (class in [test_minimalmodbus](#)), 101
[TestSanityHexencodeHexdecode](#) (class in [test_minimalmodbus](#)), 103
[TestSanityLong](#) (class in [test_minimalmodbus](#)), 101
[TestSanityPackUnpack](#) (class in [test_minimalmodbus](#)), 103
[TestSanityTextstring](#) (class in [test_minimalmodbus](#)), 102
[TestSanityTwoByteString](#) (class in [test_minimalmodbus](#)), 100
[TestSanityTwosComplement](#) (class in [test_minimalmodbus](#)), 104
[TestSanityValuelist](#) (class in [test_minimalmodbus](#)), 102
[TestSetBitOn](#) (class in [test_minimalmodbus](#)), 104
[TestSwap](#) (class in [test_minimalmodbus](#)), 103
[TestTextstringToBytestring](#) (class in [test_minimalmodbus](#)), 102
[TestTimestamp](#) (class in [test_minimalmodbus](#)), 102
[testTooLargeValue\(\)](#) ([test_minimalmodbus.TestCheckInt](#) method), 106
[testTooLargeValue\(\)](#) ([test_minimalmodbus.TestCheckNumerical](#) method), 106
[testTooLong\(\)](#) ([test_minimalmodbus.TestCheckString](#) method), 106
[testTooShort\(\)](#) ([test_minimalmodbus.TestCheckString](#) method), 106
[testTooShortPayload\(\)](#) ([test_minimalmodbus.TestCheckResponseWriteData](#) method), 106
[testTooShortResponses\(\)](#) ([test_minimalmodbus.TestCheckResponseSlaveErrorCode](#) method), 108
[testTooShortString\(\)](#) ([test_minimalmodbus.TestCheckResponsernumber_of_registers](#) method), 105
[testTooShortString\(\)](#) ([test_minimalmodbus.TestCheckResponseRegisterAddress](#) method), 105
[testTooSmallValue\(\)](#) ([test_minimalmodbus.TestCheckInt](#) method), 106
[testTooSmallValue\(\)](#) ([test_minimalmodbus.TestCheckNumerical](#) method), 106
[TestTwoByteStringToNum](#) (class in [test_minimalmodbus](#)), 100
[TestTwosComplement](#) (class in [test_minimalmodbus](#)), 103
[TestUnpack](#) (class in [test_minimalmodbus](#)), 102
[TestValuelistToBytestring](#) (class in [test_minimalmodbus](#)), 101
[testValueNotInteger\(\)](#) ([test_minimalmodbus.TestBitToBytestring](#) method), 100
[TestVerboseDummyCommunicationWithPortClosure](#) (class in [test_minimalmodbus](#)), 110
[testWriteBit\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteBit\(\)](#) ([test_minimalmodbus.TestDummyCommunicationDTB](#) method), 110
[testWriteBit\(\)](#) ([test_minimalmodbus.TestDummyCommunicationDTB](#) method), 110
[testWriteBit\(\)](#) ([test_minimalmodbus.TestDummyCommunicationOme](#) method), 109
[testWriteBit\(\)](#) ([test_minimalmodbus.TestDummyCommunicationOme](#) method), 109
[testWriteBits\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteBitsWrongValue\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteBitWithWrongRegisternumbersResponse\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteBitWithWrongWritedataResponse\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteBitWrongType\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteBitWrongValue\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 107
[testWriteFloat\(\)](#) ([test_minimalmodbus.TestDummyCommunication](#) method), 108

testWriteFloatWrongType () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteFloatWrongValue () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteLong () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteLongWrongType () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteLongWrongValue () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegister () (test_minimalmodbus.TestDummyCommunication method), 107

testWriteRegister () (test_minimalmodbus.TestDummyCommunication method), 110

testWriteRegister () (test_minimalmodbus.TestDummyCommunication method), 110

testWriteRegister () (test_minimalmodbus.TestDummyCommunication method), 109

testWriteRegister () (test_minimalmodbus.TestDummyCommunication method), 109

testWriteRegisters () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterSuppressErrorMessageAtWrongTime () (test_minimalmodbus.TestDummyCommunication method), 107

testWriteRegistersWrongType () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegistersWrongValue () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterWithDecimals () (test_minimalmodbus.TestDummyCommunication method), 107

testWriteRegisterWithWrongCrcResponse () (test_minimalmodbus.TestDummyCommunication method), 107

testWriteRegisterWithWrongFunctioncodeResponse () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterWithWrongRegisteraddressResponse () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterWithWrongRegisternumbersResponse () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterWithWrongSlaveaddressResponse () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterWithWrongWritedataResponse () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteRegisterWrongType () (test_minimalmodbus.TestDummyCommunication method), 107

testWriteRegisterWrongValue () (test_minimalmodbus.TestDummyCommunication method), 107

testWriteString () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteStringWrongType () (test_minimalmodbus.TestDummyCommunication method), 108

testWriteStringWrongValue () (test_minimalmodbus.TestDummyCommunication method), 108

testWrongCustomError () (test_minimalmodbus.TestCheckString method), 106

testWrongFunctioncode () (test_minimalmodbus.TestCheckFunctioncode method), 104

testWrongFunctioncodeListValues () (test_minimalmodbus.TestCheckFunctioncode method), 104

testWrongFunctioncodeNoRange () (test_minimalmodbus.TestCheckFunctioncode method), 104

testWrongFunctioncodeType () (test_minimalmodbus.TestCheckFunctioncode method), 104

testWrongInput () (test_minimalmodbus.TestNumToOneByteString method), 99

testWrongInputType () (test_minimalmodbus.TestBytestringToFloat method), 101

testWrongInputType () (test_minimalmodbus.TestBytestringToLong method), 101

testWrongInputType () (test_minimalmodbus.TestBytestringToTextstring method), 102

testWrongInputType () (test_minimalmodbus.TestBytestringToValuelist method), 102

testWrongInputType () (test_minimalmodbus.TestCalculateMinimumSilentPeriod method), 102

method), 99
 testWrongInputType ()
 (*test_minimalmodbus.TestCheckBit method*),
 104
 testWrongInputType ()
 (*test_minimalmodbus.TestCheckInt method*),
 106
 testWrongInputType ()
 (*test_minimalmodbus.TestEmbedPayload method*), 99
 testWrongInputType ()
 (*test_minimalmodbus.TestExtractPayload method*), 99
 testWrongInputType ()
 (*test_minimalmodbus.TestFloatToBytestring method*), 101
 testWrongInputType ()
 (*test_minimalmodbus.TestFromTwosComplement method*), 104
 testWrongInputType ()
 (*test_minimalmodbus.TestHexdecode method*),
 103
 testWrongInputType ()
 (*test_minimalmodbus.TestHexencode method*),
 103
 testWrongInputType ()
 (*test_minimalmodbus.TestLongToBytestring method*), 101
 testWrongInputType ()
 (*test_minimalmodbus.TestNumToTwoByteString method*), 100
 testWrongInputType ()
 (*test_minimalmodbus.TestPack method*),
 102
 testWrongInputType ()
 (*test_minimalmodbus.TestPredictResponseSize method*), 99
 testWrongInputType ()
 (*test_minimalmodbus.TestSetBitOn method*),
 104
 testWrongInputType ()
 (*test_minimalmodbus.TestTextstringToBytestring method*), 102
 testWrongInputType ()
 (*test_minimalmodbus.TestTwoByteStringToNum method*), 100
 testWrongInputType ()
 (*test_minimalmodbus.TestTwosComplement method*), 103
 testWrongInputType ()
 (*test_minimalmodbus.TestUnpack method*),
 102
 testWrongInputType ()
 (*test_minimalmodbus.TestValuelistToBytestring method*), 101
 testWrongInputValue ()
 (*test_minimalmodbus.TestBytestringToFloat method*), 101
 testWrongInputValue ()
 (*test_minimalmodbus.TestBytestringToLong method*), 101
 testWrongInputValue ()
 (*test_minimalmodbus.TestBytestringToTextstring method*), 102
 testWrongInputValue ()
 (*test_minimalmodbus.TestBytestringToValuelist method*), 102
 testWrongInputValue ()
 (*test_minimalmodbus.TestCalculateMinimumSilentPeriod method*), 99
 testWrongInputValue ()
 (*test_minimalmodbus.TestCheckBit method*),
 104
 testWrongInputValue ()
 (*test_minimalmodbus.TestEmbedPayload method*), 99
 testWrongInputValue ()
 (*test_minimalmodbus.TestExtractPayload method*), 99
 testWrongInputValue ()
 (*test_minimalmodbus.TestFloatToBytestring method*), 101
 testWrongInputValue ()
 (*test_minimalmodbus.TestHexdecode method*),
 103
 testWrongInputValue ()
 (*test_minimalmodbus.TestHexencode method*),
 103
 testWrongInputValue ()
 (*test_minimalmodbus.TestLongToBytestring method*), 101
 testWrongInputValue ()
 (*test_minimalmodbus.TestNumToTwoByteString method*), 100
 testWrongInputValue ()
 (*test_minimalmodbus.TestPack method*),
 102
 testWrongInputValue ()
 (*test_minimalmodbus.TestPredictResponseSize method*), 99
 testWrongInputValue ()
 (*test_minimalmodbus.TestSetBitOn method*),
 104
 testWrongInputValue ()
 (*test_minimalmodbus.TestTextstringToBytestring method*), 102
 testWrongInputValue ()
 (*test_minimalmodbus.TestTwoByteStringToNum method*), 100

method), 100
 testWrongInputValue ()
 (*test_minimalmodbus.TestUnpack method*),
 102
 testWrongInputValue ()
 (*test_minimalmodbus.TestValuelistToBytestring*
 method), 101
 testWrongListType ()
 (*test_minimalmodbus.TestCheckFunctioncode*
 method), 104
 testWrongNumberOfBytes ()
 (*test_minimalmodbus.TestCheckResponseNumberOfBytes*
 method), 105
 testWrongResponseNumber_of_registers ()
 (*test_minimalmodbus.TestCheckResponseNumber_of_registers*
 method), 106
 testWrongResponseRegisterAddress ()
 (*test_minimalmodbus.TestCheckResponseRegisterAddress*
 method), 105
 testWrongResponseWritedata ()
 (*test_minimalmodbus.TestCheckResponseWriteData*
 method), 106
 testWrongType () (*test_minimalmodbus.TestCheckBool*
 method), 107
 testWrongType () (*test_minimalmodbus.TestCheckRegisteraddress*
 method), 105
 testWrongType () (*test_minimalmodbus.TestNumToOneByteString*
 method), 99
 testWrongValue () (*test_minimalmodbus.TestBitToBytestring*
 method), 100
 testWrongValues ()
 (*test_minimalmodbus.TestBitsToBytestring*
 method), 100
 testWrongValues ()
 (*test_minimalmodbus.TestBytestringToBits*
 method), 100
 testWrongValues ()
 (*test_minimalmodbus.TestCheckMode method*),
 105
 testWrongValues ()
 (*test_minimalmodbus.TestCheckRegisteraddress*
 method), 105
 testWrongValues ()
 (*test_minimalmodbus.TestCheckSlaveaddress*
 method), 105
 testWrongValues ()
 (*test_minimalmodbus.TestCreatePayload*
 method), 98
 testWrongValues () (*test_minimalmodbus.TestSwap*
 method), 103
 verify_bits () (*in module test_deltaDTB4824*), 113
 verify_readonly_register () (*in module*
 test_deltaDTB4824), 113
 verify_register () (*in module*
 test_deltaDTB4824), 113
 verify_state_for_bits () (*in module*
 test_deltaDTB4824), 113
 verify_two_instrument_instances () (*in*
 module test_deltaDTB4824), 113
 verify_value_for_register () (*in module*
 test_deltaDTB4824), 113
 write () (*dummy_serial.Serial method*), 76
 write_register () (*minimalmodbus.Instrument method*),
 13
 write_bits () (*minimalmodbus.Instrument method*),
 13
 write_float () (*minimalmodbus.Instrument*
 method), 16
 write_long () (*minimalmodbus.Instrument*
 method), 15
 write_register () (*minimalmodbus.Instrument*
 method), 14
 write_registers () (*minimalmodbus.Instrument*
 method), 17
 write_string () (*minimalmodbus.Instrument*
 method), 17
 WRONG_ASCII_RESPONSES (*in module*
 test_minimalmodbus), 111
 wrongValues (*test_minimalmodbus.TestSwap at-*
 tribute), 103

V

VERBOSE (*in module dummy_serial*), 75
 VERBOSITY (*in module test_minimalmodbus*), 98