MinimalModbus Documentation

Release 0.7

Jonas Berg

Aug 05, 2017

Contents

Documentation built using Sphinx Aug 05, 2017 for MinimalModbus version 0.7.

Contents:

CHAPTER 1

MinimalModbus

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

Web resources

- **Documentation**: https://minimalmodbus.readthedocs.org.
- Source code on **GitHub**: https://github.com/pyhys/minimalmodbus
- Python package index (PyPI) with download: https://pypi.python.org/pypi/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: http://sourceforge.net/projects/minimalmodbus
- Old Sourceforge repository: http://sourceforge.net/p/minimalmodbus/code/

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. Example code includes drivers for Eurotherm and Omega process controllers. The only dependence is the pySerial module (also pure Python).

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, 3.2, 3.3 and 3.4.

CHAPTER 2

Installation

At the command line:

```
$ pip install minimalmodbus
```

Or, if you have virtualenvwrapper installed:

```
$ mkvirtualenv minimalmodbus
$ pip install minimalmodbus
```

Dependencies

Python versions 2.7 and higher are supported (including 3.x). Tested with Python 2.7, 3.2, 3.3 and 3.4. This module is pure Python.

This module relies on pySerial (also pure Python) to do the heavy lifting, and it is the only dependency. You can find it at the Python package index: https://pypi.python.org/pypi/pyserial

Alternate installation on Linux

From command line (if you have the *pip installer*, available at https://pypi.python.org/pypi/pip):

```
pip install -U minimalmodbus
```

or possibly:

```
sudo pip install -U pyserial
sudo pip install -U minimalmodbus
```

You can also manually download the compressed source files from https://pypi.python.org/pypi/MinimalModbus/ (see the end of that page). In that case you first need to manually install pySerial from https://pypi.python.org/pypi/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):

```
python setup.py install
```

or possibly:

```
sudo python setup.py install
```

If using Python 3, then install with:

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

For Python3 there might be problems with *easy_install* and *pip*. In that case, first manually install pySerial and then manually install MinimalModbus.

To make sure it is installed properly, print the _getDiagnosticString() 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
```

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'.

Alternate installation on Windows

Install from Github, using pip:

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

It will be installed in:

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

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

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

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.

CHAPTER 3

Usage

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, consider donating some Modbus TCP equipment.

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.

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.

As an example on the usage of MinimialModbus, the driver I use for an Eurotherm 3504 process controller is included. It uses the MinimalModbus Python module for its communication. Also a driver for Omega CN7500 is included. For hardware details on these process controllers, see Eurotherm 3500 and Omega CN7500.

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/Rs485.

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 supression.

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: http://pyserial.sourceforge.net/

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

Correspondingly for Modbus ASCII mode:

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

Subclassing

It is better to put the details in a driver for the specific instrument. An example driver for Eurotherm3500 is included in this library, and it is recommended to have a look at its source code. To get the process value (PV from loop1):

```
#!/usr/bin/env python
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)
```

Correspondingly, to use the driver for Omega CN7500:

```
#!/usr/bin/env python
import omegacn7500

instrument = omegacn7500.OmegaCN7500('/dev/ttyUSB1', 1) # port name, slave address

print instrument.get_pv() # print temperature
```

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

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
```

These can be overridden:

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

To see which settings you actually are using:

```
print instrument
```

For details on the allowed parity values, see http://pyserial.sourceforge.net/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
```

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

Running several scripts using the same port will give problems.

Handling communication errors

Your top-level code should be able to handle communication errors. This is typically done with try-except.

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.

CHAPTER 4

API for MinimalModbus

MinimalModbus: A Python driver for the Modbus RTU and Modbus ASCII protocols via serial port (via RS485 or RS232).

minimalmodbus.**BAUDRATE = 19200**
Default value for the baudrate in Baud (int).

minimalmodbus.**PARITY = 'N'**
Default value for the parity. See the pySerial module for documentation. Defaults to serial.PARITY_NONE

minimalmodbus.**BYTESIZE = 8**
Default value for the bytesize (int).

minimalmodbus.**STOPBITS = 1**
Default value for the number of stopbits (int).

minimalmodbus.**TIMEOUT = 0.05**
Default value for the timeout value in seconds (float).

minimalmodbus.**CLOSE_PORT_AFTER_EACH_CALL = False**
Default value for port closure setting.

class minimalmodbus.**Instrument** (*port*, *slaveaddress*, *mode='rtu'*)
Instrument class for talking to instruments (slaves) via 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).
- mode (str): Mode selection. Can be MODE_RTU or MODE_ASCII.

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).

New in version 0.6.

debug = None
Set this to `True` to print the communication details. Defaults to `False`.

close_port_after_each_call = None
If this is `True`, the serial port will be closed after each call. Defaults to *CLOSE_PORT_AFTER_EACH_CALL*. To change it, set the value `minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL=True`.

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`.

New in version 0.5.

handle_local_echo = None
Set to 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`.

New in version 0.7.

read_bit(*registeraddress*, *functioncode=2*)
Read one bit from the slave.

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: ValueError, TypeError, IOError

write_bit(*registeraddress*, *value*, *functioncode=5*)
Write one bit to the slave.

Args:

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

Returns: None

Raises: ValueError, TypeError, IOError

read_register(*registeraddress*, *numberOfDecimals=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).
- numberOfDecimals (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.

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

Similarly `numberOfDecimals=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: ValueError, TypeError, IOError

write_register(*registeraddress*, *value*, *numberOfDecimals=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).
- numberOfDecimals (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.

To store for example `value=77.0`, use `numberOfDecimals=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 `numberOfDecimals=2` will multiply `value` by 100 before sending it to the slave register.

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: ValueError, TypeError, IOError

read_long(*registeraddress*, *functioncode=3*, *signed=False*)
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.

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: ValueError, TypeError, IOError

write_long(*registeraddress*, *value*, *signed=False*)
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.

Returns: None

Raises: ValueError, TypeError, IOError

read_float(*registeraddress*, *functioncode=3*, *numberOfRegisters=2*)
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 consecutetive registers . Make sure to test that it makes sense for your instrument. It is pretty straight-forward to change this code if some other byte order is required by anyone (see support section).

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- functioncode (int): Modbus function code. Can be 3 or 4.
- numberOfRegisters (int): The number of registers allocated for the float. Can be 2 or 4.

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: ValueError, TypeError, IOError

write_float(*registeraddress*, *value*, *numberOfRegisters=2*)
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
- numberOfRegisters (int): The number of registers allocated for the float. Can be 2 or 4.

Returns: None

Raises: ValueError, TypeError, IOError

read_string(*registeraddress*, *numberOfRegisters=16*, *functioncode=3*)
Read a string from the slave.

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).

Args:

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

Returns: The string (str).

Raises: ValueError, TypeError, IOError

write_string(*registeraddress*, *textstring*, *numberOfRegisters=16*)
Write a string to the slave.

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).

Uses Modbus function code 16.

Args:

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

If the textstring is longer than the 2*numberOfRegisters, an error is raised. Shorter strings are padded with spaces.

Returns: None

Raises: ValueError, TypeError, IOError

read_registers(*registeraddress*, *numberOfRegisters*, *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).
- numberOfRegisters (int): The number of registers to read.
- functioncode (int): Modbus function code. Can be 3 or 4.

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).

Raises: ValueError, TypeError, 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.

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

Returns: None

Raises: ValueError, TypeError, IOError

CHAPTER 5

Modbus details

Modbus data types

The Modbus standard defines storage in:

- Bits
- Registers (16-bit). Can hold integers in the range 0 to 65535 (dec), which is 0 to ffff (hex). Also called 'unsigned INT16' or 'unsigned short'.

Some deviations from 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.

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 characters (each 1 byte = 8 bits). Often 16 consecutive registers are used, allowing 32 characters in the string.

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).

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	*read_bit()*	2 [or 1]	*write_bit()*	5 [or 15]
Register Integer, possibly scaled	*read_register()*	3 [or 4]	*write_register()*	16 [or 6]
Long (32 bits = 2 registers)	*read_long()*	3 [or 4]	*write_long()*	16
Float (32 or 64 bits)	*read_float()*	3 [or 4]	*write_float()*	16
String	*read_string()*	3 [or 4]	*write_string()*	16
Registers Integers	*read_registers()*	3 [or 4]	*write_registers()*	16

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

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]

TODO Validate

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.

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'`).

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 reponse will be something like:

```
:0103020136C3
```

CHAPTER 6

Serial communication

Timing of the serial communications

The Modbus RTU standard prescribes a silent period corresponding to 3.5 characters between each message, to be able fo 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 http://www.automation.com/library/articles-white-papers/fieldbus-serial-bus-io-networks/introduction-to-modbus.

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	1.0 ms
115200	115200 bit/s	8.7 us	95 us	0.33 ms

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 simultaneusly 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.

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 it self 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 conviced 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 *Handle local echo*.

Controlling a separate GPIO pin from kernelspace software on embedded Linux machines See for example http://blog.savoirfairelinux.com/en/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.

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.

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 intepreter.

CHAPTER 7

Debug mode

Debug mode

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

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

instrument = minimalmodbus.Instrument('/dev/ttyUSB1', 1) # port name, slave address␣
↪(in decimal)
instrument.debug = True
print instrument.read_register(289, 1)  # Remember to use print() for Python3
```

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.

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 request 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`.

Special characters

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

`repr()` shows as	Can be written as	ASCII hex	ASCII dec	Description
`\t`	`\x09`	09	9	Horizontal Tab (TAB)
`\n`	`\x0a`	0a	10	Linefeed (LF)
`\r`	`\x0d`	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.

CHAPTER 8

Trouble shooting

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: http://pyserial.sourceforge.net/

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 terminiation 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.

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 *Handle local echo*.

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".

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.

Known issues

For the data types involving more than one register (float, long etc), 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 consecutetive registers. Make sure to test that it makes sense for your instrument. It is pretty straight-forward to change this code if some other byte order is required by anyone (see support section).

Changing `close_port_after_each_call` after instantiation of `Instrument` might be problematic. Set the value `minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL=True` immediately after `import minimalmodbus` instead.

When running under Python2.6, for some conversion errors no exception is raised. For example when trying to convert a negative value to a bytestring representing an unsigned long.

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.

It should no longer be necessary to use ``minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL = True`` when running on Windows, as this now is handled in a better way internally. This gives a significantly increased communication speed.

If the underlying pySerial complains that the serial port is already open, it is still possible to make MinimalModbus close the serial port after each call. Use it like:

```
#!/usr/bin/env python
import minimalmodbus
minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL = True

instrument = minimalmodbus.Instrument('/dev/ttyUSB1', 1) # port name, slave address␣
↪(in decimal)
print instrument.read_register(289, 1)
```

Support

Send a mail to minimalmodbus-list@lists.sourceforge.net

Describe the problem in detail, and include any error messsages. Please also include the output after running:

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

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

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

CHAPTER 9

Detailed usage documentation

For introductive usage documentation, see *Usage*.

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):

```
>>> 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
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
```

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 is 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."""
        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)
```

See *eurotherm3500* (click [source]) for more details.

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`.

"""

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()
```

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 http://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.

Workaround for floats with wrong byte order

If your instrument responds with floats implemented in the other byte order than MinimalModbus, here is a workaroud that can be used.

For example you are reading two registers (starting with register 3924) from slave number 2, and the result should be a float of approximately 208:

```
MinimalModbus debug mode. Response from instrument: '\x02\x03\x04\x93\x9dCPD\x95'

\x02 Slave address (here 2)
\x03 Function code (here 3 = read registers)
\x04 Byte count (here 4 bytes)
\x93 Payload. Here 93 (hex) = 147 (dec)
\x9d Payload. Here 9d (hex) = 157 (dec)
```

```
C    Payload. Here ASCII letter C = 43 (hex) = 67 (dec).
P    Payload. Here ASCII letter P = 50 (hex) = 80 (dec).
D    CRC LSB
\x95 CRC MSB
```

So the payload is `\x93\x9dCP`, which is 4 bytes (as each register stores 2 bytes). See http://minimalmodbus.sourceforge.net/index.html#example

You should try this in interactive mode in Python, and to manually re-shuffle the bytes:

```
~$ python
Python 2.7.3 (default, Sep 26 2013, 20:08:41)
[GCC 4.6.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import minimalmodbus
>>>
>>> minimalmodbus._bytestringToFloat("\x93\x9dCP")
-3.9698747127906995e-27
>>>
>>> minimalmodbus._bytestringToFloat("CP\x93\x9d")
208.5766143798828
>>>
```

Suggested work-around:

- Read the register values directly using the *read_registers()* function.
- Then reshuffle the bytes
- Convert it to a float using the internal function `_bytestringToFloat()`.

Something like:

```
values = read_registers(3924, numberOfRegisters=2)
registerstring = chr(values[2]) + chr(values[3]) + chr(values[0]) + chr(values[1])
floatvalue = minimalmodbus._bytestringToFloat(registerstring)
```

See *read_registers()* and `_bytestringToFloat()`.

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 `_extractPayload()` 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
```

Handle local echo

Note: This feature has been implemented in version 0.7. See the API.

If you cannot disable the local echo of your RS485 adapter, you will receive your own message before the message from the slave. Luca Di Gregorio has suggested how to solve this issue.

In the method _communicate(), change this:

```
self.serial.write(message)

# Read response
answer = self.serial.read(number_of_bytes_to_read)
```

to:

```
self.serial.write(message)

# Read response
echo_to_be_discarded = self.serial.read(len(message))
answer = self.serial.read(number_of_bytes_to_read)
```

Install or uninstalling a distribution

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

```
sudo python setup.py install
```

or:

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

On a development machine, go to the `trunk` directory before running the command.

Uninstall

Pip-installed packages can be unistalled with:

```
sudo pip uninstall minimalmodbus
```

Show versions of all installed packages

Use:

```
pip freeze
```

Installation target

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

```
import minimalmodbus
print minimalmodbus._getDiagnosticString()
```

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).

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:

```
python --version
```

Setting the PYTHONPATH

To set the path:

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

or:

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

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

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.

CHAPTER 10

Developer documentation

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.

Design considerations

My take on the design is that is 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 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 ambigous, 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

General driver structure

The general structure of the program is shown here:

Function	Description
read_register()	One of the facades for _genericCommand().
_genericCommand()	Generates payload, then calls _performCommand().
_performCommand()	Embeds payload into error-checking codes etc, then calls _communicate().
_communicate()	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*.

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)	_numToOneByteString()	
Bit	_createBitpattern()	_bitResponseToValue()
Integer (char, short)	_numToTwoByteString()	_twoByteStringToNum()
Several registers	_valuelistToBytestring()	_bytestringToValuelist()
Long integer	_longToBytestring()	_bytestringToLong()
Floating point number	_floatToBytestring()	_bytestringToFloat()
String	_textstringToBytestring()	_bytestringToTextstring()

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*.

Unittesting

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

```
python test_minimalmodbus.py
```

Also 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.

Hardware tests are performed using a Delta DTB4824 process controller. See *Internal documentation for hardware testing of MinimalModbus using DTB4824* for more information.

A brief introduction to unittesting is found here: https://docs.python.org/release/2.5.2/lib/minimal-example.html

The `unittest` module is documented here: https://docs.python.org/2/library/unittest.html

The unittests uses previosly recorded communication data for the testing. Inside the unpacked folder go to `test` and run the unit tests with:

```
python test_all_simulated.py
python3 test_all_simulated.py

python3.4 test_all_simulated.py
python3.3 test_all_simulated.py
python3.2 test_all_simulated.py
python2.7 test_all_simulated.py
```

To automatically run the tests for the different Python versions:

```
tox
```

It is also possible to run the individual test files:

```
python test_minimalmodbus.py
python test_eurotherm3500.py
python test_omegacn7500.py
```

MinimalModbus is also tested with hardware. A Delta temperature controller DTB4824 is used together with a USB-to-RS485 converter.

Run it with:

```
python test_deltaDTB4824.py
```

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

```
python test_deltaDTB4824.py 19200 /dev/ttyUSB0
```

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

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'",)
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)
```

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
>>> minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL = True # Seems mandatory for Windows
>>> 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. See for example the test code for omegacn7500 *Internal documentation for unit testing of omegacn7500* (click '[source]', see RESPONSES at end of the page).

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('DUMMYPORTNAME', 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/pythonprogrammering/minimalmodbus/trunk/minimalmodbus.py", line␣
↪174, in read_register
    return self._genericCommand(functioncode, registeraddress,␣
↪numberOfDecimals=numberOfDecimals)
  File "/home/jonas/pythonprogrammering/minimalmodbus/trunk/minimalmodbus.py", line␣
↪261, in _genericCommand
    payloadFromSlave = self._performCommand(functioncode, payloadToSlave)
  File "/home/jonas/pythonprogrammering/minimalmodbus/trunk/minimalmodbus.py", line␣
↪317, in _performCommand
    response            = self._communicate(message)
  File "/home/jonas/pythonprogrammering/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 omegacn7500
>>> omegacn7500.minimalmodbus.serial.Serial = dummy_serial.Serial # Monkey-patch a␣
↪dummy serial port
>>> instrument = omegacn7500.OmegaCN7500('DUMMYPORTNAME', 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('DUMMYPORTNAME', 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)

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.

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'
```

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

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
```

Extending MinimalModbus

It is straight-forward to extend MinimalModbus to handle more Modbus function codes. Use the method `_performCommand()` 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 `_performCommand()`:

```
>>> 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 `_performCommand()` 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._performCommand(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.

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._calculateCrcString('\x01\x03\x00\x05\x00\x01')
'\x94\x0b'
```

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

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

Playing with two's complement:

```
>>> minimalmodbus._twosComplement(-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.

Found a bug?

Try to isolate the bug by running in interactive mode (Python interpreter) with debug mode activated. Send a mail to the mailing list with the output, and also the output from `_getDiagnosticString()`.

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 is should be straight-forward to work with. Then please send your findings to the mailing list.

Generate documentation

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

```
make docs
make pdf
```

Do linkchecking and test coverage measurements:

make linkcheck make coverage

Alternatively, build the HTML and PDF documentation (in directory `doc` after making sure that `PYTHONPATH` is correct):

```
make html
make latexpdf
```

Verify all external links:

```
make linkcheck
```

Webpage

The HTML theme on http://minimalmodbus.sourceforge.net/ is the Sphinx 'sphinx_rtd_theme' theme.

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

Notes on distribution

Installing the module from local svn files

In the trunk directory:

```
sudo python setup.py install
```

If there are conditional `__name__ == '__main__'` clauses in the module, these can be tested using (adapt path to your system):

```
python /usr/local/lib/python2.6/dist-packages/eurotherm3500.py
python /usr/local/lib/python2.6/dist-packages/minimalmodbus.py
```

How to generate a source distribution from the present development code

This will create a subfolder `dist` with zipped or gztared source folders:

```
python setup.py sdist
python setup.py sdist --formats=gztar,zip
```

Notes on generating binary distributions

This will create the subfolders `build` and `dist`:

```
python setup.py bdist
```

This will create a subfolder `dist` with a Windows installer:

```
python setup.py bdist --formats=wininst
```

Build a distribution before installing it

This will create a subfolder `build`:

```
python setup.py build
```

Development installation

This will create a link to the project, instead of properly installing it:

```
sudo python setup.py develop
```

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

To uninstall it:

```
sudo python setup.py develop --uninstall
```

Preparation for release

Change version number etc

- Manually change the `__version__` field in the `minimalmodbus.py` source file.
- Manually change the release date in `CHANGES.txt`

(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**.

Code style checking etc

Check the code:

```
pychecker eurotherm3500.py
pychecker minimalmodbus.py
pychecker omegacn7500.py
```

(The 2to3 tool is not necessary, as we run the unittests under both Python2 and Python3).

Unittesting

Run unit tests for all supported Python versions:

```
make test-all
```

Alternatively, run unit tests (in the `trunk/test` directory):

```
python test_all_simulated.py
python3 test_all_simulated.py

python2.7 test_all_simulated.py
python3.2 test_all_simulated.py
```

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):

```
python setup.py sdist
```

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

- Documentation generation
- Test coverage report generation

(Prepare subversion)

Make sure the Subversion is updated:

```
svn update
svn status -v --no-ignore
```

Make a tag in Subversion (adapt to version number):

```
svn copy https://svn.code.sf.net/p/minimalmodbus/code/trunk/ https://svn.code.sf.net/
↪p/minimalmodbus/code/tags/0.5 -m "Release 0.5"
```

Upload to PyPI

Build the source distribution (as `.gzip.tar` and `.zip`) , and upload it to PYPI (will use the `README.txt` etc):

```
python setup.py register
python setup.py sdist --formats=gztar,zip upload
```

(Upload to Sourceforge)

Upload the `.gzip.tar` and `.zip` files to Sourceforge by logging in and manually using the web form.

Upload the generated documentation to Sourceforge. In directory `trunk/doc/build/html`:

```
scp -r * pyhys,minimalmodbus@web.sourceforge.net:htdocs
```

Upload the documentation PDF. In directory `trunk/doc/build/latex`:

```
scp minimalmodbus.pdf pyhys,minimalmodbus@web.sourceforge.net:htdocs
```

Upload the test coverage report. In directory `trunk`:

```
scp -r htmlcov pyhys,minimalmodbus@web.sourceforge.net:htdocs
```

Test documentation

Test links on the Sourceforge and PyPI pages. 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).

(Generate Windows installer)

On a Windows machine, build the windows installer:

```
python setup.py bdist_wininst
```

Upload the Windows installer to PYPI by logging in, and uploading it manually.

Upload the Windows installer to Sourceforge by manually using the web form.

Test installer

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

Backup

Burn a CD/DVD with these items:

- Source tree
- Source distributions
- Windows installer
- Generated HTML files
- PDF documentation
- svn repository in archive format

Marketing

- Mailing list
- Sourceforge project news

(Applying patches)

Apply the patch like:

```
/minimalmodbus$ patch -Np0 -d trunk < concurrency_latency_tests_09-21.diff
```

(Downloading backups from the Sourceforge server)

To download the svn repository in archive format, type this in the destination directory on your computer:

```
rsync -av minimalmodbus.svn.sourceforge.net::svn/minimalmodbus/* .
```

Useful development tools

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

SVN Version control software. See http://subversion.apache.org/

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

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

Coverage.py Unittest coverage tool. See http://nedbatchelder.com/code/coverage/

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

pep8.py Code style checker. See https://github.com/PyCQA/pep8#readme

Subversion (svn) usage

Subversion provides an easy way to share code with each other. You can find all MinimalModbus files on the subversion repository on http://sourceforge.net/p/minimalmodbus/code/ Look in the trunk subfolder.

Install SVN on some Linux machines

Install it with:

```
sudo apt-get install subversion
```

Download the files

The usage is:

```
svn checkout URL NewSubfolder
```

where *NewSubfolder* is the name of a subfolder that will be created in present directory. You can also write `svn co` instead of `svn checkout`.

In a proper directory on your computer, download the files (not only the `trunk` subfolder) using:

```
svn checkout svn://svn.code.sf.net/p/minimalmodbus/code/ minimalmodbus
```

Submit contributions

First run the `svn update` command to download the latest changes from the repository. Then make the changes in the files. Use the `svn status` command to see which files you have changed. Then upload your changes with the `svn commit -m 'comment'` command. Note that it easy to revert any changes in SVN, so feel free to test.

Shortlist of frequently used SVN commands

These are the most used commands:

```
svn update
svn status
svn status -v
svn status -v --no-ignore
svn diff
svn log
svn add FILENAME or DIRECTORYNAME
svn remove FILENAME or DIRECTORYNAME
svn commit -m 'Write your log message here'
```

In the 'trunk' directory:

```
svn propset svn:ignore html .
svn proplist
svn propget svn:ignore
```

or if ignoring multiple items, edit the list using:

```
svn propedit svn:ignore .
```

Automatic keyword substitution:

```
svn propset svn:keywords "Date Revision" minimalmodbus.py
svn propset svn:keywords "Date Revision" eurotherm3500.py
svn propset svn:keywords "Date Revision" README.txt
svn propget svn:keywords minimalmodbus.py
```

SVN settings

SVN uses the computer `locale` settings for selecting the language (including keyword substitution).

Language settings:

```
locale      # Shows present locale settings
locale -a   # Shows available locales
export LC_ALL="en_US.utf8"
```

(Installing MinimalModbus from SVN repository)

Update your local copy by:

```
svn update
```

Go to the minimalmodbus/trunk directory:

```
sudo python setup.py install
```

Test it using (adapt path to your system):

```
python /usr/local/lib/python2.6/dist-packages/minimalmodbus.py
```

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
```

Sphinx usage

This documentation is generated with the Sphinx tool: http://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:

```
easy_install sphinx
```

or possibly:

```
sudo easy_install sphinx
```

Check installed version by typing:

```
sphinx-build
```

Spinx formatting conventions

What	Usage	Result
Inline web link	`` `Link text <http://example.com/>`_ ``	Link text
Internal link	`:ref:`testminimalmodbus``	*Internal documentation for unit testing of MinimalModbus*
Inline code	``` ``code text`` ```	`code text`
String	'A'	'A'
String w escape 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	`:envvar:`PYTHONPATH``	PYTHONPATH
OS-level command	`:command:`make``	**make**
File	`:file:`minimalmodbus.py``	minimalmodbus.py
Path	`:file:`path/to/myfile.txt``	path/to/myfile.txt
Type	`**bytes**`	**bytes**
Module	`:mod:`minimalmodbus``	*minimalmodbus*
Data	`:data:`.BAUDRATE``	*BAUDRATE*
Data (full)	`:data:`minimalmodbus.BAUDRATE``	*minimalmodbus.BAUDRATE*
Constant	`:const:`False``	False
Function	`:func:`._checkInt``	_checkInt()
Function (full)	`:func:`minimalmodbus._checkInt``	minimalmodbus._checkInt()
Argument	`*payload*`	*payload*
Class	`:class:`.Instrument``	*Instrument*
Class (full)	`:class:`minimalmodbus.Instrument``	*minimalmodbus.Instrument*
Method	`:meth:`.read_bit``	*read_bit()*
Method (full)	`:meth:`minimalmodbus.Instrument.read_bit``	*minimalmodbus.Instrument.read_bit()*

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.

Useful Sphinx-related links

Online resources for the formatting used (reStructuredText):

Sphinx reStructuredText Primer http://sphinx-doc.org/rest.html

Spinx autodoc features http://sphinx-doc.org/ext/autodoc.html

Sphinx cross-referencing Python objects http://sphinx-doc.org/domains.html#python-roles

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

Sphinx build commands

To build the documentation, go to the directory `trunk/doc` and then run:

```
make html
```

That should generate HTML files to the directory `trunk/doc/build/html`.

To generate PDF:

```
make latexpdf
```

Note that the `PYTHONPATH` must be set properly, so that Sphinx can import the modules to document. See below.

It is also possible to run without the **`make`** command. In the `trunk/doc` directory:

```
sphinx-build -b html -d build/doctrees   -a . build/html
```

If the python source files not are updated in the HTML output, then remove the contents of `trunk/doc/build/doctrees` and rebuild the documentation. (This has now been included in the `Makefile`).

Remember that the `Makefile` uses tabs for indentation, not spaces.

Sometimes there are warnings and errors when generating the HTML pages. They can appear different, but are most often related to problems importing files. In that case start the Python interpreter and try to import the module, for example:

```
>>> import test_minimalmodbus
```

From there you can most often solve the problem.

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

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

Unittest coverage measurement using coverage.py

Install the script `coverage.py`:

```
sudo pip install coverage
```

Collect test data:

```
coverage run test_minimalmodbus.py
```

or:

```
coverage run test_all.py
```

Generate html report (ends up in `trunk/test/htmlcov`):

```
coverage html
```

Or to exclude some third party modules (adapt to your file structure):

```
coverage html --omit=/usr/*
```

Alternatively, adjust the settings in the `.coverage` file.

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.

Using the pep8 style checker tool

This tool checks the coding style. See https://pypi.python.org/pypi/pep8/

Install the pep8 checker tool:

```
sudo pip install pep8
```

Run it:

```
pep8 minimalmodbus.py
```

or:

```
pep8 --statistics minimalmodbus.py

pep8 -r --select=E261 --show-source minimalmodbus.py
```

TODO

Maybe:

- Improved documentation (especially the sections with TODO).
- Tool for interpretation of Modbus messages
- Increase test coverage for minimalmodbus.py
- PEP8 fine tuning of source.
- Improve the dummy_serial behavior, to better mimic Windows behavior.
- Unittests for measuring the sleep time in _communicate.
- Serial port flushing
- Floats with other byte order
- Logging instead of _print_out()
- Timing based on time.clock() for Windows
- string templating compatible with python2.6 (use {2} in format).

CHAPTER 11

Contributing

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

You can contribute in many ways:

Types of Contributions

Report Bugs

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

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting.
- Detailed steps to reproduce the bug.

Fix Bugs

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

Implement Features

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

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.

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 :)

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.

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, 3.3, and 3.4, and for PyPy. Check https://travis-ci.org/pyhys/minimalmodbus/pull_requests and make sure that the tests pass for all supported Python versions.

Tips

To run a subset of tests:

```
$ python -m unittest tests.test_minimalmodbus
```

CHAPTER 12

Credits

Development Lead

- Jonas Berg <pyhys@users.sourceforge.net>

Contributors

Significant contributions by Angelo Compagnucci, Aaron LaLonde, Asier Abalos, Simon Funke, Edwin van den Oetelaar, Dominik Socha, Luca Di Gregorio, Dino, Peter and Michael Penza.

CHAPTER 13

Related software

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

pyModbus From https://github.com/bashwork/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.'

CHAPTER 14

History

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.

Release 0.6 (2014-06-22)

- Support for Modbus ASCII mode.

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.

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.

Release 0.3.2 (2012-01-25)

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

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

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.

Release 0.2 (2011-08-19)

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

Release 0.1 (2011-06-16)

- First public release.

CHAPTER 15

Internal documentation

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 = 5**
The default timeot 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 respones 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 = 'NONE'**
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.

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.

Internal documentation for MinimalModbus

MinimalModbus: A Python driver for the Modbus RTU and Modbus ASCII protocols via serial port (via RS485 or RS232).

minimalmodbus.**BAUDRATE = 19200**
Default value for the baudrate in Baud (int).

minimalmodbus.**PARITY = 'N'**
Default value for the parity. See the pySerial module for documentation. Defaults to serial.PARITY_NONE

minimalmodbus.**BYTESIZE = 8**
Default value for the bytesize (int).

minimalmodbus.**STOPBITS = 1**
Default value for the number of stopbits (int).

minimalmodbus.**TIMEOUT = 0.05**
Default value for the timeout value in seconds (float).

minimalmodbus.**CLOSE_PORT_AFTER_EACH_CALL = False**
Default value for port closure setting.

class minimalmodbus.**Instrument**(*port*, *slaveaddress*, *mode='rtu'*)
Instrument class for talking to instruments (slaves) via 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).
- mode (str): Mode selection. Can be MODE_RTU or MODE_ASCII.

__init__(*port*, *slaveaddress*, *mode='rtu'*)

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).

New in version 0.6.

debug = None
Set this to `True` to print the communication details. Defaults to `False`.

close_port_after_each_call = None
If this is `True`, the serial port will be closed after each call. Defaults to *CLOSE_PORT_AFTER_EACH_CALL*. To change it, set the value `minimalmodbus.CLOSE_PORT_AFTER_EACH_CALL=True`.

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`.

New in version 0.5.

handle_local_echo = None
Set to to `True` if your RS-485 adaptor has local echo enabled. Then the transmitted message will immeadiately 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`.

New in version 0.7.

__repr__()
String representation of the *Instrument* object.

read_bit(*registeraddress*, *functioncode=2*)
Read one bit from the slave.

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: ValueError, TypeError, IOError

write_bit(*registeraddress*, *value*, *functioncode=5*)
Write one bit to the slave.

Args:

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

Returns: None

Raises: ValueError, TypeError, IOError

read_register(*registeraddress*, *numberOfDecimals=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).
- numberOfDecimals (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.

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

Similarly `numberOfDecimals=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: ValueError, TypeError, IOError

write_register(*registeraddress*, *value*, *numberOfDecimals=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).
- numberOfDecimals (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.

To store for example `value=77.0`, use `numberOfDecimals=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 `numberOfDecimals=2` will multiply `value` by 100 before sending it to the slave register.

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: ValueError, TypeError, IOError

read_long(*registeraddress*, *functioncode=3*, *signed=False*)
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.

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: ValueError, TypeError, IOError

write_long(*registeraddress*, *value*, *signed=False*)
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.

Returns: None

Raises: ValueError, TypeError, IOError

read_float(*registeraddress*, *functioncode=3*, *numberOfRegisters=2*)
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 consecutetive registers . Make sure to test that it makes sense for your instrument. It is pretty straight-forward to change this code if some other byte order is required by anyone (see support section).

Args:

- registeraddress (int): The slave register start address (use decimal numbers, not hex).
- functioncode (int): Modbus function code. Can be 3 or 4.
- numberOfRegisters (int): The number of registers allocated for the float. Can be 2 or 4.

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: ValueError, TypeError, IOError

write_float(*registeraddress*, *value*, *numberOfRegisters=2*)
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
- numberOfRegisters (int): The number of registers allocated for the float. Can be 2 or 4.

Returns: None

Raises: ValueError, TypeError, IOError

read_string(*registeraddress*, *numberOfRegisters=16*, *functioncode=3*)
Read a string from the slave.

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).

Args:

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

Returns: The string (str).

Raises: ValueError, TypeError, IOError

write_string(*registeraddress*, *textstring*, *numberOfRegisters=16*)
Write a string to the slave.

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).

Uses Modbus function code 16.

Args:

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

If the textstring is longer than the 2*numberOfRegisters, an error is raised. Shorter strings are padded with spaces.

Returns: None

Raises: ValueError, TypeError, IOError

read_registers(*registeraddress*, *numberOfRegisters*, *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).
- numberOfRegisters (int): The number of registers to read.
- functioncode (int): Modbus function code. Can be 3 or 4.

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).

Raises: ValueError, TypeError, 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.

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

Returns: None

Raises: ValueError, TypeError, IOError

_genericCommand(*functioncode*, *registeraddress*, *value=None*, *numberOfDecimals=0*, *numberOfRegisters=1*, *signed=False*, *payloadformat=None*)
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.
- numberOfDecimals (int): The number of decimals for content conversion. Only for a single register.
- numberOfRegisters (int): The number of registers to read/write. Only certain values allowed, depends on payloadformat.
- signed (bool): Whether the data should be interpreted as unsigned or signed. Only for a single register or for payloadformat='long'.
- payloadformat (None or string): None, 'long', 'float', 'string', 'register', 'registers'. Not necessary for single registers or bits.

If a value of 77.0 is stored internally in the slave register as 770, then use `numberOfDecimals=1` which will divide the received data from the slave by 10 before returning the value. Similarly `numberOfDecimals=2` will divide the received data by 100 before returning the value. Same functionality is also used when writing data to the slave.

Returns: The register data in numerical value (int or float), or the bit value 0 or 1 (int), or `None`.

Raises: ValueError, TypeError, IOError

_performCommand(*functioncode*, *payloadToSlave*)
Performs the command having the *functioncode*.

Args:

- functioncode (int): The function code for the command to be performed. Can for example be 'Write register' = 16.
- payloadToSlave (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: ValueError, TypeError.

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

_communicate(*request*, *number_of_bytes_to_read*)
Talk to the slave via a serial port.

Args: request (str): The raw request that is to be sent to the slave. number_of_bytes_to_read (int): number of bytes to read

Returns: The raw data (string) returned from the slave.

Raises: TypeError, ValueError, IOError

Note that the answer might have strange ASCII control signs, which makes it difficult to print it in the promt (messes up a bit). Use repr() to make the string printable (shows ASCII values for control signs.)

Will block until reaching *number_of_bytes_to_read* or timeout.

If the attribute *Instrument.debug* is `True`, the communication details are printed.

If the attribute *Instrument.close_port_after_each_call* is `True` the serial port is closed after each call.

Timing:

```
                                        Request from master (Master is writing)
                                        |
                                        |       Response from slave (Master is␣
↪reading)
                                        |       |
----W----R-----------------------------W-------R--------------------------------
↪---------
         |                              |       |
         |<----- Silent period ------>|       |
                                        |       |
                 Roundtrip time  ---->|-------|<--
```

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

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

__module__ = 'minimalmodbus'

minimalmodbus.**_embedPayload**(*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 protcol 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.**_extractPayload**(*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.
- slaveaddress (int): The adress of the slave. Used here for error checking only.
- mode (str): The modbus protcol mode (MODE_RTU or MODE_ASCII)
- functioncode (int): Used here for error checking only.

Returns: The payload part of the *response* string.

Raises: ValueError, TypeError. 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.**_predictResponseSize**(*mode*, *functioncode*, *payloadToSlave*)

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

Args:

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

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

Raises: ValueError, TypeError.

minimalmodbus.**_calculate_minimum_silent_period**(*baudrate*)
Calculate the silent period length to comply with the 3.5 character silence between messages.

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.**_numToOneByteString**(*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.**_numToTwoByteString**(*value*, *numberOfDecimals=0*, *LsbFirst=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.
- numberOfDecimals (int): Number of decimals, 0 or more, for scaling.
- LsbFirst (bol): Whether the least significant byte should be first in the resulting string.
- signed (bol): 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 `numberOfDecimals=1` to multiply `value` by 10 before sending it to the slave register. Similarly `numberOfDecimals=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 `LsbFirst` parameter, as seen here:

`LsbFirst` 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 `numberOfDecimals = 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 `LsbFirst = False`, the most significant byte is given first why the resulting string is `\x03\x02`, which has the length 2.

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

Args:

- bytestring (str): A string of length 2.
- numberOfDecimals (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 `numberOfDecimals=1` to divide the received data by 10 before returning the value. Similarly `numberOfDecimals=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 `numberOfDecimals = 1`, then this is converted to 77.0 (float).

`minimalmodbus.`**`_longToBytestring`**(*value*, *signed=False*, *numberOfRegisters=2*)
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.
- numberOfRegisters (int): Should be 2. For error checking only.

Returns: A bytestring (4 bytes).

Raises: TypeError, ValueError

`minimalmodbus.`**`_bytestringToLong`**(*bytestring*, *signed=False*, *numberOfRegisters=2*)
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 (bol): Whether large positive values should be interpreted as negative values.
- numberOfRegisters (int): Should be 2. For error checking only.

Returns: The numerical value (int).

Raises: ValueError, TypeError

`minimalmodbus.`**`_floatToBytestring`**(*value*, *numberOfRegisters=2*)
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.

- numberOfRegisters (int): Can be 2 or 4.

Returns: A bytestring (4 or 8 bytes).

Raises: TypeError, ValueError

minimalmodbus.**_bytestringToFloat**(*bytestring*, *numberOfRegisters=2*)
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._floatToBytestring()`.

Args:

- bytestring (str): A string of length 4 or 8.
- numberOfRegisters (int): Can be 2 or 4.

Returns: A float.

Raises: TypeError, ValueError

minimalmodbus.**_textstringToBytestring**(*inputstring*, *numberOfRegisters=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 that the allocated space, it is padded with spaces in the end.

Args:

- inputstring (str): The string to be stored in the slave. Max 2*numberOfRegisters characters.
- numberOfRegisters (int): The number of registers allocated for the string.

Returns: A bytestring (str).

Raises: TypeError, ValueError

minimalmodbus.**_bytestringToTextstring**(*bytestring*, *numberOfRegisters=16*)
Convert a bytestring 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:

- bytestring (str): The string from the slave. Length = 2*numberOfRegisters
- numberOfRegisters (int): The number of registers allocated for the string.

Returns: A the text string (str).

Raises: TypeError, ValueError

minimalmodbus.**_valuelistToBytestring**(*valuelist*, *numberOfRegisters*)
Convert a list of numerical values to a bytestring.

Each element is 'unsigned INT16'.

Args:

- valuelist (list of int): The input list. The elements should be in the range 0 to 65535.

- numberOfRegisters (int): The number of registers. For error checking.

Returns: A bytestring (str). Length = 2*numberOfRegisters

Raises: TypeError, ValueError

minimalmodbus.**_bytestringToValuelist**(*bytestring*, *numberOfRegisters*)
Convert a bytestring to a list of numerical values.

The bytestring is interpreted as 'unsigned INT16'.

Args:

- bytestring (str): The string from the slave. Length = 2*numberOfRegisters
- numberOfRegisters (int): The number of registers. For error checking.

Returns: A list of integers.

Raises: TypeError, ValueError

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.**_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.**_bitResponseToValue**(*bytestring*)
Convert a response string to a numerical value.

Args: bytestring (str): A string of length 1. Can be for example `\x01`.

Returns: The converted value (int).

Raises: TypeError, ValueError

minimalmodbus.**_createBitpattern**(*functioncode*, *value*)
Create the bit pattern that is used for writing single bits.

This is basically a storage of numerical constants.

Args:

- functioncode (int): can be 5 or 15
- value (int): can be 0 or 1

Returns: The bit pattern (string).

Raises: TypeError, ValueError

minimalmodbus.**_twosComplement**(*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.**_fromTwosComplement**(*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.**_setBitOn**(*x*, *bitNum*)
Set bit 'bitNum' to True.

Args:

- x (int): The value before.
- bitNum (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.**_calculateCrcString**(*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.**_calculateLrcString**(*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.**_checkMode**(*mode*)
Check that the Modbus mode is valie.

Args: mode (string): The Modbus mode (MODE_RTU or MODE_ASCII)

Raises: TypeError, ValueError

minimalmodbus.**_checkFunctioncode**(*functioncode*, *listOfAllowedValues=[]*)
Check that the given functioncode is in the listOfAllowedValues.

Also verifies that 1 <= function code <= 127.

Args:

- functioncode (int): The function code
- listOfAllowedValues (list of int): Allowed values. Use *None* to bypass this part of the checking.

Raises: TypeError, ValueError

minimalmodbus.**_checkSlaveaddress**(*slaveaddress*)
Check that the given slaveaddress is valid.

Args: slaveaddress (int): The slave address

Raises: TypeError, ValueError

minimalmodbus.**_checkRegisteraddress**(*registeraddress*)
Check that the given registeraddress is valid.

Args: registeraddress (int): The register address

Raises: TypeError, ValueError

minimalmodbus.**_checkResponseByteCount**(*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

minimalmodbus.**_checkResponseRegisterAddress**(*payload*, *registeraddress*)
Check that the start adress 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): The register address (use decimal numbers, not hex).

Raises: TypeError, ValueError

minimalmodbus.**_checkResponseNumberOfRegisters**(*payload*, *numberOfRegisters*)
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
- numberOfRegisters (int): Number of registers that have been written

Raises: TypeError, ValueError

minimalmodbus.**_checkResponseWriteData**(*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 to write, length should be 2 bytes.

Raises: TypeError, ValueError

minimalmodbus.**_checkString**(*inputstring*, *description*, *minlength=0*, *maxlength=None*)
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

Raises: TypeError, ValueError

Uses the function `_checkInt()` internally.

minimalmodbus.**_checkInt**(*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 `_checkString()`, as that function uses this function internally.

minimalmodbus.**_checkNumerical**(*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 `_checkString()`, as it uses this function internally.

minimalmodbus.**_checkBool**(*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.**_interpretRawMessage**(*inputstr*)
Generate a human readable description of a Modbus bytestring.

Args: inputstr (str): The bytestring that should be interpreted.

Returns: A descriptive string.

For example, the string `'\n\x03\x10\x01\x00\x01\xd0q'` should give something like:

```
TODO: update

Modbus bytestring decoder
Input string (length 8 characters): '\n\x03\x10\x01\x00\x01\xd0q'
Probably modbus RTU mode.
Slave address: 10 (dec). Function code: 3 (dec).
Valid message. Extracted payload: '\x10\x01\x00\x01'

Pos   Character Hex  Dec  Probable interpretation
------------------------------------------------
  0:  '\n'      0A    10  Slave address
  1:  '\x03'    03     3  Function code
  2:  '\x10'    10    16  Payload
  3:  '\x01'    01     1  Payload
  4:  '\x00'    00     0  Payload
  5:  '\x01'    01     1  Payload
  6:  '\xd0'    D0   208  Checksum, CRC LSB
  7:  'q'       71   113  Checksum, CRC MSB
```

minimalmodbus.**_interpretPayload**(*functioncode*, *payload*)
Generate a human readable description of a Modbus payload.

Args:

- functioncode (int): Function code
- payload (str): The payload that should be interpreted. It should be a byte string.

Returns: A descriptive string.

For example, the payload `'\x10\x01\x00\x01'` for functioncode 3 should give something like:

```
TODO: Update
```

minimalmodbus.**_getDiagnosticString**()
Generate a diagnostic string, showing the module version, the platform, current directory etc.

Returns: A descriptive string.

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.**_NonexistantError**

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 http://stackoverflow.com/questions/8672754/how-to-show-the-error-messages-caught-by-assertraises-in-unittest-in-pyt

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.**TestEmbedPayload**(*methodName='runTest'*)

knownValues = [(2, 2, 'rtu', '123', '\x02\x02123X\xc2'), (1, 16, 'rtu', 'ABC', '\x01\x10ABC<E'), (0, 5, 'rtu', 'hjl', '\x00\x0

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestExtractPayload**(*methodName='runTest'*)

knownValues = [(2, 2, 'rtu', '123', '\x02\x02123X\xc2'), (1, 16, 'rtu', 'ABC', '\x01\x10ABC<E'), (0, 5, 'rtu', 'hjl', '\x00\x0

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityEmbedExtractPayload**(*methodName='runTest'*)

knownValues = [(2, 2, 'rtu', '123', '\x02\x02123X\xc2'), (1, 16, 'rtu', 'ABC', '\x01\x10ABC<E'), (0, 5, 'rtu', 'hjl', '\x00\x0

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\x08', 6), ('rtu', 1, '\x00>\v

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), (38400, 0.001), (115200, 0.0003

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'), (0, 0, True, False, '\x00\x00'), (77.0, 1, Fal

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestTwoByteStringToNum**(*methodName='runTest'*)

knownValues = [(0.0, 0, False, False, '\x00\x00'), (0, 0, False, False, '\x00\x00'), (0, 0, True, False, '\x00\x00'), (77.0, 1, Fal

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityTwoByteString**(*methodName='runTest'*)

knownValues = [(0.0, 0, False, False, '\x00\x00'), (0, 0, False, False, '\x00\x00'), (0, 0, True, False, '\x00\x00'), (77.0, 1, Fal

testSanity()

class test_minimalmodbus.**TestLongToBytestring**(*methodName='runTest'*)

knownValues = [(0, False, 2, '\x00\x00\x00\x00'), (0, True, 2, '\x00\x00\x00\x00'), (1, False, 2, '\x00\x00\x00\x01'), (1, Tru

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestBytestringToLong**(*methodName='runTest'*)

knownValues = [(0, False, 2, '\x00\x00\x00\x00'), (0, True, 2, '\x00\x00\x00\x00'), (1, False, 2, '\x00\x00\x00\x01'), (1, Tru

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityLong**(*methodName='runTest'*)

knownValues = [(0, False, 2, '\x00\x00\x00\x00'), (0, True, 2, '\x00\x00\x00\x00'), (1, False, 2, '\x00\x00\x00\x01'), (1, Tru

testSanity()

class test_minimalmodbus.**TestFloatToBytestring**(*methodName='runTest'*)

knownValues = [(1, 2, '?\x80\x00\x00'), (1.0, 2, '?\x80\x00\x00'), (1.0, 2, '?\x80\x00\x00'), (1.1, 2, '?\x8c\xcc\xcd'), (100, 2,

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestBytestringToFloat**(*methodName='runTest'*)

knownValues = [(1, 2, '?\x80\x00\x00'), (1.0, 2, '?\x80\x00\x00'), (1.0, 2, '?\x80\x00\x00'), (1.1, 2, '?\x8c\xcc\xcd'), (100, 2,

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityFloat**(*methodName='runTest'*)

knownValues = [(1, 2, '?\x80\x00\x00'), (1.0, 2, '?\x80\x00\x00'), (1.0, 2, '?\x80\x00\x00'), (1.1, 2, '?\x8c\xcc\xcd'), (100, 2,

testSanity()

class test_minimalmodbus.**TestValuelistToBytestring**(*methodName='runTest'*)

knownValues = [([1], 1, '\x00\x01'), ([0, 0], 2, '\x00\x00\x00\x00'), ([1, 2], 2, '\x00\x01\x00\x02'), ([1, 256], 2, '\x00\x01\x01

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\x00\x02'), ([1, 256], 2, '\x00\x01\x01

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\x00\x02'), ([1, 256], 2, '\x00\x01\x01

testSanity()

class test_minimalmodbus.**TestTextstringToBytestring**(*methodName='runTest'*)

knownValues = [('A', 1, 'A '), ('AB', 1, 'AB'), ('ABC', 2, 'ABC '), ('ABCD', 2, 'ABCD'), ('A', 16, 'A '), ('A', 32, 'A ')]

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestBytestringToTextstring**(*methodName='runTest'*)

knownValues = [('A', 1, 'A '), ('AB', 1, 'AB'), ('ABC', 2, 'ABC '), ('ABCD', 2, 'ABCD'), ('A', 16, 'A '), ('A', 32, 'A ')]

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityTextstring**(*methodName='runTest'*)

knownValues = [('A', 1, 'A '), ('AB', 1, 'AB'), ('ABC', 2, 'ABC '), ('ABCD', 2, 'ABCD'), ('A', 16, 'A '), ('A', 32, 'A ')]

testSanity()

class test_minimalmodbus.**TestPack**(*methodName='runTest'*)

knownValues = [(-77, '>h', '\xff\xb3'), (-1, '>h', '\xff\xff'), (-770, '>h', '\xfc\xfe'), (-32768, '>h', '\x80\x00'), (32767, '>h',

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestUnpack**(*methodName='runTest'*)

knownValues = [(-77, '>h', '\xff\xb3'), (-1, '>h', '\xff\xff'), (-770, '>h', '\xfc\xfe'), (-32768, '>h', '\x80\x00'), (32767, '>h',

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityPackUnpack**(*methodName='runTest'*)

knownValues = [(-77, '>h', '\xff\xb3'), (-1, '>h', '\xff\xff'), (-770, '>h', '\xfc\xfe'), (-32768, '>h', '\x80\x00'), (32767, '>h',

testSanity()

class test_minimalmodbus.**TestHexencode**(*methodName='runTest'*)

knownValues = [('', False, ''), ('7', False, '37'), ('J', False, '4A'), (']', False, '5D'), ('\x04', False, '04'), ('\x04]', False, '04

testKnownValues()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestHexdecode**(*methodName='runTest'*)

knownValues = [('', False, ''), ('7', False, '37'), ('J', False, '4A'), (']', False, '5D'), ('\x04', False, '04'), ('\x04]', False, '04

testKnownValues()

testAllowLowercase()

testWrongInputValue()

testWrongInputType()

class test_minimalmodbus.**TestSanityHexencodeHexdecode**(*methodName='runTest'*)

knownValues = [('', False, ''), ('7', False, '37'), ('J', False, '4A'), (']', False, '5D'), ('\x04', False, '04'), ('\x04]', False, '04

testKnownValues()

testKnownValuesLoop()

Loop through all bytestrings of length two.

class test_minimalmodbus.**TestBitResponseToValue**(*methodName='runTest'*)

testKnownValues()

testWrongValues()

testWrongType()

class test_minimalmodbus.**TestCreateBitPattern**(*methodName='runTest'*)

knownValues = [(5, 0, '\x00\x00'), (5, 1, '\xff\x00'), (15, 0, '\x00'), (15, 1, '\x01')]

testKnownValues()

testWrongFunctionCode()

testFunctionCodeNotInteger()

testWrongValue()

testValueNotInteger()

class test_minimalmodbus.**TestTwosComplement**(*methodName='runTest'*)

knownValues = [(0, 8, 0), (1, 8, 1), (127, 8, 127), (-128, 8, 128), (-127, 8, 129), (-1, 8, 255), (0, 16, 0), (1, 16, 1), (32767, 16, 3

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), (-1, 8, 255), (0, 16, 0), (1, 16, 1), (32767, 16, 3

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.**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.**TestCheckResponseNumberOfBytes**(*methodName='runTest'*)

testCorrectNumberOfBytes()

testWrongNumberOfBytes()

testNotStringInput()

class test_minimalmodbus.**TestCheckResponseRegisterAddress**(*methodName='runTest'*)

testCorrectResponseRegisterAddress()

testWrongResponseRegisterAddress()

testTooShortString()

testNotString()

testWrongAddress()

testAddressNotInteger()

class test_minimalmodbus.**TestCheckResponseNumberOfRegisters**(*methodName='runTest'*)

testCorrectResponseNumberOfRegisters()

testWrongResponseNumberOfRegisters()

testTooShortString()

testNotString()

testWrongResponseNumberOfRegistersRange()

testNumberOfRegistersNotInteger()

class test_minimalmodbus.**TestCheckResponseWriteData**(*methodName='runTest'*)

testCorrectResponseWritedata()

testWrongResponseWritedata()

testNotString()

testTooShortString()

testTooLongString()

class test_minimalmodbus.**TestCheckString**(*methodName='runTest'*)

testKnownValues()

testTooShort()

testTooLong()

testInconsistentLengthlimits()

testInputNotString()

testNotIntegerInput()

testDescriptionNotString()

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()

testReadBit()

testReadBitWrongValue()

testReadBitWrongType()

testReadBitWithWrongByteCountResponse()

testReadBitWithNoResponse()

testWriteBit()

testWriteBitWrongValue()

testWriteBitWrongType()

testWriteBitWithWrongRegisternumbersResponse()

testWriteBitWithWrongWritedataResponse()

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()

testGenericCommandWrongValueCombinations()

testGenericCommandWrongType()

testPerformcommandKnownResponse()

testPerformcommandWrongSlaveResponse()

testPerformcommandWrongInputValue()

testPerformcommandWrongInputType()

testCommunicateKnownResponse()

testCommunicateWrongType()

testCommunicateNoMessage()

testCommunicateNoResponse()

testCommunicateLocalEcho()

testCommunicateWrongLocalEcho()

testRepresentation()

testReadPortClosed()

testWritePortClosed()

testPortAlreadyOpen()

testPortAlreadyClosed()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationOmegaSlave1**(*methodName='runTest'*)

setUp()

testReadBit()

testWriteBit()

testReadRegister()

testWriteRegister()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationOmegaSlave10**(*methodName='runTest'*)

setUp()

testReadBit()

testWriteBit()

testReadRegister()

testWriteRegister()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationDTB4824_RTU**(*methodName='runTest'*)

setUp()

testReadBit()

testWriteBit()

testReadBits()

testReadRegister()

testReadRegisters()

testWriteRegister()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationDTB4824_ASCII**(*methodName='runTest'*)

setUp()

testReadBit()

testWriteBit()

testReadBits()

testReadRegister()

testReadRegisters()

testWriteRegister()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationWithPortClosure**(*methodName='runTest'*)

setUp()

testReadRegisterSeveralTimes()

testPortAlreadyOpen()

testPortAlreadyClosed()

tearDown()

class test_minimalmodbus.**TestVerboseDummyCommunicationWithPortClosure**(*methodName='runTest'*)

setUp()

testReadRegister()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationDebugmode**(*methodName='runTest'*)

setUp()

testReadRegister()

tearDown()

class test_minimalmodbus.**TestDummyCommunicationHandleLocalEcho**(*methodName='runTest'*)

setUp()

testReadRegister()

testReadRegisterWrongEcho()

tearDown()

test_minimalmodbus.**WRONG_ASCII_RESPONSES** = {}

A dictionary of respones 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.

Internal documentation for hardware testing of MinimalModbus using DTB4824

Hardware testing of MinimalModbus using the Delta DTB temperature controller.

For use with Delta DTB4824VR.

Usage

```
python 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.

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

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.

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.

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.

Manual testing in interactive mode (at the Python prompt)

Use a setting of 19200 bps, RTU mode and slave addess 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.main()
```

CHAPTER 16

Example drivers

API for the Eurotherm3500 example driver

Driver for the Eurotherm3500 process controller, for communication via the Modbus RTU protocol.

class eurotherm3500.**Eurotherm3500**(*portname*, *slaveaddress*)

Bases: *minimalmodbus.Instrument*

Instrument class for Eurotherm 3500 process controller.

Communicates via Modbus RTU protocol (via RS232 or RS485), using the *MinimalModbus* Python module.

Args:

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

Implemented with these function codes (in decimal):

Description	Modbus function code
Read registers	3
Write registers	16

get_pv_loop1()

Return the process value (PV) for loop1.

get_pv_loop2()

Return the process value (PV) for loop2.

get_pv_module3()

Return the process value (PV) for extension module 3 (A).

get_pv_module4()

Return the process value (PV) for extension module 4 (A).

get_pv_module6()

Return the process value (PV) for extension module 6 (A).

is_manual_loop1()
Return True if loop1 is in manual mode.

get_sptarget_loop1()
Return the setpoint (SP) target for loop1.

get_sp_loop1()
Return the (working) setpoint (SP) for loop1.

set_sp_loop1(*value*)
Set the SP1 for loop1.

Note that this is not necessarily the working setpoint.

Args: value (float): Setpoint (most often in degrees)

get_sp_loop2()
Return the (working) setpoint (SP) for loop2.

get_sprate_loop1()
Return the setpoint (SP) change rate for loop1.

set_sprate_loop1(*value*)
Set the setpoint (SP) change rate for loop1.

Args: value (float): Setpoint change rate (most often in degrees/minute)

is_sprate_disabled_loop1()
Return True if Loop1 setpoint (SP) rate is disabled.

disable_sprate_loop1()
Disable the setpoint (SP) change rate for loop1.

enable_sprate_loop1()
Set disable=false for the setpoint (SP) change rate for loop1.

Note that also the SP rate value must be properly set for the SP rate to work.

get_op_loop1()
Return the output value (OP) for loop1 (in %).

is_inhibited_loop1()
Return True if Loop1 is inhibited.

get_op_loop2()
Return the output value (OP) for loop2 (in %).

get_threshold_alarm1()
Return the threshold value for Alarm1.

is_set_alarmsummary()
Return True if some alarm is triggered.

API for the Omega CN7500 example driver

Driver for the Omega CN7500 process controller, for communication using the Modbus RTU protocol.

omegacn7500.**SETPOINT_MAX = 999.9**
Default value for maximum allowed setpoint.

omegacn7500.**TIME_MAX = 900**
Default value for maximum allowed step time.

omegacn7500.**CONTROL_MODES = {0: 'PID', 1: 'ON/OFF', 2: 'Manual Tuning', 3: 'Program'}**
Description of the control mode numerical values.

omegacn7500.**REGISTER_START = {'setpoint': 8192, 'cycles': 4176, 'linkpattern': 4192, 'actualstep': 4160, 'time': 8320}**
Register address start values for pattern related parameters.

omegacn7500.**REGISTER_OFFSET_PER_PATTERN = {'setpoint': 8, 'cycles': 1, 'linkpattern': 1, 'actualstep': 1, 'time': 8}**
Increase in register address value per pattern number (for pattern related parameters).

omegacn7500.**REGISTER_OFFSET_PER_STEP = {'setpoint': 1, 'cycles': 0, 'linkpattern': 0, 'actualstep': 0, 'time': 1}**
Increase in register address value per step number (for pattern related parameters).

class omegacn7500.**OmegaCN7500**(*portname*, *slaveaddress*)
Bases: *minimalmodbus.Instrument*

Instrument class for Omega CN7500 process controller.

Communicates via Modbus RTU protocol (via RS485), using the *minimalmodbus* Python module.

This driver is intended to enable control of the OMEGA CN7500 controller from the command line.

Args:

- portname (str): port name
  - examples:
  - OS X: '/dev/tty.usbserial'
  - Linux: '/dev/ttyUSB0'
  - Windows: '/com3'
- slaveaddress (int): slave address in the range 1 to 247 (in decimal)

The controller can be used to follow predefined temperature programs, called patterns. Eight patterns (numbered 0-7) are available, each having eight temperature steps (numbered 0-7).

Each pattern have these parameters:

•Temperature for each step (8 parameters)

•Time for each step (8 parameters)

•Link to another pattern

•Number of cycles (repetitions of this pattern)

•Actual step (which step to stop at)

Attributes:

- **setpoint_max**
  - Defaults to *SETPOINT_MAX*.
- **time_max**
  - Defaults to *TIME_MAX*.

Implemented with these function codes (in decimal):

Description	Modbus function code
Read registers	3
Write one register	6
Read bits	2
Write one bit	5

get_pv()
Return the process value (PV).

get_output1()
Return the output value for output1 [in %].

run()
Put the process controller in run mode.

stop()
Stop the process controller.

is_running()
Return True if the controller is running.

get_setpoint()
Return the setpoint value (float).

set_setpoint(*setpointvalue*)
Set the setpoint.

Args: setpointvalue (float): Setpoint [most often in degrees]

get_control_mode()
Get the name of the current operation mode.

Returns: A string decribing the controlmode.

The returned string is one of the items in *CONTROL_MODES*.

set_control_mode(*modevalue*)
Set the control method using the corresponding integer value.

Args: modevalue(int): 0-3

The modevalue is one of the keys in *CONTROL_MODES*.

get_start_pattern_no()
Return the starting pattern number (int).

set_start_pattern_no(*patternnumber*)
Set the starting pattern number.

Args: patternnumber (integer): 0-7

get_pattern_step_setpoint(*patternnumber*, *stepnumber*)
Get the setpoint value for a step.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7

Returns: The setpoint value (float).

set_pattern_step_setpoint(*patternnumber*, *stepnumber*, *setpointvalue*)
Set the setpoint value for a step.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7
- setpointvalue (float): Setpoint value

get_pattern_step_time(*patternnumber*, *stepnumber*)
Get the step time.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7

Returns: The step time (int??).

set_pattern_step_time(*patternnumber*, *stepnumber*, *timevalue*)
Set the step time.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7
- timevalue (integer??): 0-900

get_pattern_actual_step(*patternnumber*)
Get the 'actual step' parameter for a given pattern.

Args: patternnumber (integer): 0-7

Returns: The 'actual step' parameter (int).

set_pattern_actual_step(*patternnumber*, *value*)
Set the 'actual step' parameter for a given pattern.

Args:

- patternnumber (integer): 0-7
- value (integer): 0-7

get_pattern_additional_cycles(*patternnumber*)
Get the number of additional cycles for a given pattern.

Args: patternnumber (integer): 0-7

Returns: The number of additional cycles (int).

set_pattern_additional_cycles(*patternnumber*, *value*)
Set the number of additional cycles for a given pattern.

Args:

- patternnumber (integer): 0-7
- value (integer): 0-99

get_pattern_link_topattern(*patternnumber*)
Get the 'linked pattern' value for a given pattern.

Args: patternnumber (integer): From 0-7

Returns: The 'linked pattern' value (int).

set_pattern_link_topattern(*patternnumber*, *value*)
Set the 'linked pattern' value for a given pattern.

Args:

- patternnumber (integer): 0-7
- value (integer): 0-8. A value=8 sets the link parameter to OFF.

get_all_pattern_variables(*patternnumber*)
Get all variables for a given pattern at one time.

Args: patternnumber (integer): 0-7

Returns: A descriptive multiline string.

set_all_pattern_variables(*patternnumber*, *sp0*, *ti0*, *sp1*, *ti1*, *sp2*, *ti2*, *sp3*, *ti3*, *sp4*, *ti4*, *sp5*, *ti5*, *sp6*, *ti6*, *sp7*, *ti7*, *actual_step*, *additional_cycles*, *link_pattern*)
Set all variables for a given pattern at one time.

Args:

- patternnumber (integer): 0-7
- sp[*n*] (float): setpoint value for step *n*
- ti[*n*] (integer??): step time for step *n*, 0-900
- actual_step (int): ?
- additional_cycles(int): ?
- link_pattern(int): ?

Internal documentation for unit testing of eurotherm3500

test_eurotherm3500: Unittests for eurotherm3500

Uses a dummy serial port from the module *dummy_serial*.

class test_eurotherm3500.**TestDummyCommunication**(*methodName='runTest'*)

setUp()

testReadPv1()

testReadPv2()

testReadPv3()

testReadPv4()

testReadPv6()

testReadSp1()

testWriteSp1()

testReadSp1Target()

testReadSp2()

testIsSprate1Disabled()

testReadSprate1()

testWriteSprate1()

testEnableSprate1()

testDisableSprate1()

testReadOp1()

testReadOp2()

testReadAlarm1Threshold()

testReadAlarmSummary()

testLoop1Manual()

testLoop1Inhibited()

test_eurotherm3500.**RESPONSES = {'\x01\x03\x01r\x00\x01%\xed': '\x01\x03\x02\x00\xc0\xb8\x14', '\x01\x10\x00#\x00\x0**
A dictionary of respones from a dummy Eurotherm 3500 instrument.

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

Internal documentation for omegacn7500

Driver for the Omega CN7500 process controller, for communication using the Modbus RTU protocol.

omegacn7500.**SETPOINT_MAX = 999.9**
Default value for maximum allowed setpoint.

omegacn7500.**TIME_MAX = 900**
Default value for maximum allowed step time.

omegacn7500.**CONTROL_MODES = {0: 'PID', 1: 'ON/OFF', 2: 'Manual Tuning', 3: 'Program'}**
Description of the control mode numerical values.

omegacn7500.**REGISTER_START = {'setpoint': 8192, 'cycles': 4176, 'linkpattern': 4192, 'actualstep': 4160, 'time': 8320}**
Register address start values for pattern related parameters.

omegacn7500.**REGISTER_OFFSET_PER_PATTERN = {'setpoint': 8, 'cycles': 1, 'linkpattern': 1, 'actualstep': 1, 'time': 8}**
Increase in register address value per pattern number (for pattern related parameters).

omegacn7500.**REGISTER_OFFSET_PER_STEP = {'setpoint': 1, 'cycles': 0, 'linkpattern': 0, 'actualstep': 0, 'time': 1}**
Increase in register address value per step number (for pattern related parameters).

class omegacn7500.**OmegaCN7500**(*portname*, *slaveaddress*)
Instrument class for Omega CN7500 process controller.

Communicates via Modbus RTU protocol (via RS485), using the *minimalmodbus* Python module.

This driver is intended to enable control of the OMEGA CN7500 controller from the command line.

Args:

- portname (str): port name
  - examples:
  - OS X: '/dev/tty.usbserial'
  - Linux: '/dev/ttyUSB0'
  - Windows: '/com3'
- slaveaddress (int): slave address in the range 1 to 247 (in decimal)

The controller can be used to follow predefined temperature programs, called patterns. Eight patterns (numbered 0-7) are available, each having eight temperature steps (numbered 0-7).

Each pattern have these parameters:

•Temperature for each step (8 parameters)

•Time for each step (8 parameters)

•Link to another pattern

•Number of cycles (repetitions of this pattern)

•Actual step (which step to stop at)

Attributes:

- **setpoint_max**
  - Defaults to *SETPOINT_MAX*.
- **time_max**
  - Defaults to *TIME_MAX*.

Implemented with these function codes (in decimal):

Description	Modbus function code
Read registers	3
Write one register	6
Read bits	2
Write one bit	5

__init__(*portname*, *slaveaddress*)

get_pv()
Return the process value (PV).

get_output1()
Return the output value for output1 [in %].

run()
Put the process controller in run mode.

stop()
Stop the process controller.

is_running()
Return True if the controller is running.

get_setpoint()
Return the setpoint value (float).

set_setpoint(*setpointvalue*)
Set the setpoint.

Args: setpointvalue (float): Setpoint [most often in degrees]

get_control_mode()
Get the name of the current operation mode.

Returns: A string decribing the controlmode.

The returned string is one of the items in *CONTROL_MODES*.

set_control_mode(*modevalue*)
Set the control method using the corresponding integer value.

Args: modevalue(int): 0-3

The modevalue is one of the keys in *CONTROL_MODES*.

get_start_pattern_no()
Return the starting pattern number (int).

set_start_pattern_no (*patternnumber*)
Set the starting pattern number.

Args: patternnumber (integer): 0-7

get_pattern_step_setpoint (*patternnumber*, *stepnumber*)
Get the setpoint value for a step.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7

Returns: The setpoint value (float).

set_pattern_step_setpoint (*patternnumber*, *stepnumber*, *setpointvalue*)
Set the setpoint value for a step.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7
- setpointvalue (float): Setpoint value

get_pattern_step_time (*patternnumber*, *stepnumber*)
Get the step time.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7

Returns: The step time (int??).

set_pattern_step_time (*patternnumber*, *stepnumber*, *timevalue*)
Set the step time.

Args:

- patternnumber (integer): 0-7
- stepnumber (integer): 0-7
- timevalue (integer??): 0-900

get_pattern_actual_step (*patternnumber*)
Get the 'actual step' parameter for a given pattern.

Args: patternnumber (integer): 0-7

Returns: The 'actual step' parameter (int).

set_pattern_actual_step (*patternnumber*, *value*)
Set the 'actual step' parameter for a given pattern.

Args:

- patternnumber (integer): 0-7
- value (integer): 0-7

get_pattern_additional_cycles (*patternnumber*)
Get the number of additional cycles for a given pattern.

Args: patternnumber (integer): 0-7

Returns: The number of additional cycles (int).

set_pattern_additional_cycles(*patternnumber*, *value*)
Set the number of additional cycles for a given pattern.

Args:

- patternnumber (integer): 0-7
- value (integer): 0-99

get_pattern_link_topattern(*patternnumber*)
Get the 'linked pattern' value for a given pattern.

Args: patternnumber (integer): From 0-7

Returns: The 'linked pattern' value (int).

set_pattern_link_topattern(*patternnumber*, *value*)
Set the 'linked pattern' value for a given pattern.

Args:

- patternnumber (integer): 0-7
- value (integer): 0-8. A value=8 sets the link parameter to OFF.

get_all_pattern_variables(*patternnumber*)
Get all variables for a given pattern at one time.

Args: patternnumber (integer): 0-7

Returns: A descriptive multiline string.

set_all_pattern_variables(*patternnumber*, *sp0*, *ti0*, *sp1*, *ti1*, *sp2*, *ti2*, *sp3*, *ti3*, *sp4*, *ti4*, *sp5*, *ti5*, *sp6*, *ti6*, *sp7*, *ti7*, *actual_step*, *additional_cycles*, *link_pattern*)
Set all variables for a given pattern at one time.

Args:

- patternnumber (integer): 0-7
- sp[*n*] (float): setpoint value for step *n*
- ti[*n*] (integer??): step time for step *n*, 0-900
- actual_step (int): ?
- additional_cycles(int): ?
- link_pattern(int): ?

__module__ = 'omegacn7500'

omegacn7500.**_checkPatternNumber**(*patternnumber*)
Check that the given patternnumber is valid.

Args:

- patternnumber (int): The patternnumber to be checked.

Raises: TypeError, ValueError

omegacn7500.**_checkStepNumber**(*stepnumber*)
Check that the given stepumber is valid.

Args:

- stepnumber (int): The stepnumber to be checked.

Raises: TypeError, ValueError

omegacn7500.**_checkSetpointValue**(*setpointvalue*, *maxvalue*)
Check that the given setpointvalue is valid.

Args:

- setpointvalue (numerical): The setpoint value to be checked. Must be positive.
- maxvalue (numerical): Upper limit for setpoint value. Must be positive.

Raises: TypeError, ValueError

omegacn7500.**_checkTimeValue**(*timevalue*, *maxvalue*)
Check that the given timevalue is valid.

Args:

- timevalue (numerical): The time value to be checked. Must be positive.
- maxvalue (numerical): Upper limit for time value. Must be positive.

Raises: TypeError, ValueError

omegacn7500.**_calculateRegisterAddress**(*registertype*, *patternnumber*, *stepnumber=None*)
Calculate the register address for pattern related parameters.

Args:

- registertype (string): The type of parameter, for example 'cycles'. Allowed are the keys from *REGISTER_START*.
- patternnumber (int): The pattern number.
- stepnumber (int): The step number. Use None if it not should affect the calculation.

Returns: The register address (int).

Raises: TypeError, ValueError

Internal documentation for unit testing of omegacn7500

test_omegacn7500: Unittests for omegacn7500

Uses a dummy serial port from the module *dummy_serial*.

class test_omegacn7500.**TestCalculateRegisterAddress**(*methodName='runTest'*)

knownValues = [('setpoint', 0, 0, 8192), ('setpoint', 1, 0, 8200), ('time', 0, 0, 8320), ('time', 0, 1, 8321), ('time', 1, 0, 8328),

testKnownValues()

testWrongValues()

testWrongType()

class test_omegacn7500.**TestCheckPatternNumber**(*methodName='runTest'*)

testKnownResults()

testWrongValue()

testWrongType()

class test_omegacn7500.**TestCheckStepNumber**(*methodName='runTest'*)

testKnownResults()

testWrongValue()

testWrongType()

class test_omegacn7500.**TestCheckSetpointValue**(*methodName='runTest'*)

testKnownResults()

testWrongValue()

testWrongType()

class test_omegacn7500.**TestCheckTimeValue**(*methodName='runTest'*)

testKnownResults()

testWrongValue()

testWrongType()

class test_omegacn7500.**TestDummyCommunication_Slave1**(*methodName='runTest'*)

Testing using dummy communication, with data recorded for slaveaddress = 1

Most of the tests are for making sure that the communication details are OK.

For some examples of testing the methods for argument value errors or argument type errors, see the *testSetControlModeWithWrongValue()* and *testSetControlModeWithWrongValueType()* methods.

setUp()

testReadPv1()

testRun()

testStop()

testIsRunning()

testGetSetpoint()

testSetSetpoint()

testGetControlMode()

testSetControlMode()

testSetControlModeWithWrongValue()

testSetControlModeWithWrongValueType()

testGetStartPatternNo()

testSetStartPatternNo()

testGetPatternStepSetpoint()

testSetPatternStepSetpoint()

testGetPatternStepTime()

testSetPatternStepTime()

testGetPatternActualStep()

testSetPatternActualStep()

testGetPatternAdditionalCycles()

testSetPatternAdditionalCycles()

testGetPatternLinkToPattern()

testSetPatternLinkToPattern()

testGetAllPatternVariables()

testSetAllPatternVariables()

class test_omegacn7500.**TestDummyCommunication_Slave10**(*methodName='runTest'*)
Testing using dummy communication, with data recorded for slaveaddress = 10

setUp()

testReadPv1()

testRun()

testStop()

testIsRunning()

testGetSetpoint()

testSetSetpoint()

testGetControlMode()

testSetControlMode()

testGetStartPatternNo()

testSetStartPatternNo()

testGetPatternStepSetpoint()

testSetPatternStepSetpoint()

testGetPatternStepTime()

testSetPatternStepTime()

testGetPatternActualStep()

testSetPatternActualStep()

testGetPatternAdditionalCycles()

testSetPatternAdditionalCycles()

testGetPatternLinkToPattern()

testSetPatternLinkToPattern()

testGetAllPatternVariables()

testSetAllPatternVariables()

test_omegacn7500.**RESPONSES = {'\x01\x03\x100\x00\x01\x80\xc5': '\x01\x03\x02\x00\x029\x85', '\n\x03 \x03\x00\x01~\xb1**
A dictionary of respones from a dummy Omega CN7500 instrument.

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

test_omegacn7500.**_print_out**(*inputstring*)
Print the inputstring. To make it compatible with Python2 and Python3.

CHAPTER 17

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

G

H

I

K

M

O

P

R

S

T

V

W