# Modbus Documentation

***Release 0.6.0***

**Auke Willem Oosterhoff <oosterhoff@baopt.nl>**

May 08, 2016

uModbus or ($\mu$Modbus) is a pure Python implementation of the Modbus protocol as described in the MODBUS Application Protocol Specification V1.1b3. uModbus implements both a Modbus client (both TCP and RTU) and a Modbus TCP server (only TCP). ThereThe “u” or “$\mu$” in the name comes from the the SI prefix “micro-”. uModbus is very small and lightweight. The source can be found on GitHub. Documentation is available at Read the Docs.

CHAPTER 1

# Quickstart

Creating a Modbus TCP server is easy:

```
#!/usr/bin/env python
# scripts/examples/simple_data_store.py
import logging
from socketserver import TCPServer
from collections import defaultdict

from umodbus import get_server, RequestHandler, conf
from umodbus.utils import log_to_stream

# Add stream handler to logger 'uModbus'.
log_to_stream(level=logging.DEBUG)

# A very simple data store which maps addresss against their values.
data_store = defaultdict(int)

# Enable values to be signed (default is False).
conf.SIGNED_VALUES = True

TCPServer.allow_reuse_address = True
app = get_server(TCPServer, ('localhost', 502), RequestHandler)


@app.route(slave_ids=[1], function_codes=[3, 4], addresses=list(range(0, 10)))
def read_data_store(slave_id, function_code, address):
    """" Return value of address. """
    return data_store[address]


@app.route(slave_ids=[1], function_codes=[6, 16], addresses=list(range(0, 10)))
def write_data_store(slave_id, function_code, address, value):
    """" Set value for address. """
    data_store[address] = value

if __name__ == '__main__':
    try:
        app.serve_forever()
    finally:
        app.shutdown()
        app.server_close()
```

Doing a Modbus request requires even less code:

```python
#!/usr/bin/env python
# scripts/examples/simple_tcp_client.py
import socket

from umodbus import conf
from umodbus.client import tcp

# Enable values to be signed (default is False).
conf.SIGNED_VALUES = True

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 502))

# Returns a message or Application Data Unit (ADU) specific for doing
# Modbus TCP/IP.
message = tcp.write_multiple_coils(slave_id=1, starting_address=1, values=[1, 0, 1, 1])

# Response depends on Modbus function code. This particular returns the
# amount of coils written, in this case it is.
response = tcp.send_message(message, sock)

sock.close()
```

CHAPTER 2

# Features

The following Modbus functions have been implemented:

- 01: Read Coils
- 02: Read Discrete Inputs
- 03: Read Holding Registers
- 04: Read Input Registers
- 05: Write Single Coil
- 06: Write Single Register
- 15: Write Multiple Coils
- 16: Write Multiple Registers

Other featues:

- Support for signed and unsigned register values.

# Roadmap

uModbus is far from complete. The next, unordered list shows what is going to be implemented in the future:

- Support for all Modbus functions
- Modbus RTU server
- Use asyncio for handling of requests
- Other Modbus 'flavours', so uModbus is able to handle 32 bit values.

CHAPTER 4

# License

CHAPTER 5

# How uModus works

## 5.1 Installation

uModbus has been tested on Python 2.7, Python 3.3+ and Pypy.

### 5.1.1 As package

uModbus is available on Pypi and can be installed through Pip:

```
$ pip install umodbus
```

Or you can install from source using *setup.py*:

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

### 5.1.2 For development, debugging and testing

uModbus has no runtime dependencies. However to run the the tests or build the documentation some dependencies are required. They are listed in dev_requirements.txt and can be installed through Pip:

```
$ pip install -r dev_requirements.txt
```

Now you can build the docs:

```
$ sphinx-build -b html docs/source docs/build
```

Or run the tests:

```
$ py.test tests
```

## 5.2 Modbus Server

### 5.2.1 Viewpoint

The uModbus server code is build with routing in mind. Routing (groups of) requests to a certain callback must be easy. This is in contrast with with other Modbus implementation which often focus on reading and writing from a data store.

Because of this difference in view point uModbus doesn't know the concepts of Modbus' data models like discrete inputs, coils, input registers and holding registers and their read/write properties.

## 5.2.2 Routing

The route system was inspired by Flask. Like Flask, uModbus requires a global app or server. This server contains a route map. Routes can be added to this route map.

The following code example demonstrates how to implement a very simple data store for 10 addresses.

## 5.2.3 Sample use

```python
#!/usr/bin/env python
# scripts/examples/simple_data_store.py
import logging
from socketserver import TCPServer
from collections import defaultdict

from umodbus import get_server, RequestHandler, conf
from umodbus.utils import log_to_stream

# Add stream handler to logger 'uModbus'.
log_to_stream(level=logging.DEBUG)

# A very simple data store which maps addresses against their values.
data_store = defaultdict(int)

# Enable values to be signed (default is False).
conf.SIGNED_VALUES = True

TCPServer.allow_reuse_address = True
app = get_server(TCPServer, ('localhost', 502), RequestHandler)


@app.route(slave_ids=[1], function_codes=[1, 2], addresses=list(range(0, 10)))
def read_data_store(slave_id, function_code, address):
    """" Return value of address. """
    return data_store[address]


@app.route(slave_ids=[1], function_codes=[5, 15], addresses=list(range(0, 10)))
def write_data_store(slave_id, function_code, address, value):
    """" Set value for address. """
    data_store[address] = value

if __name__ == '__main__':
    try:
        app.serve_forever()
    finally:
        app.shutdown()
        app.server_close()
```

## 5.3 Modbus Client

uModbus contains a client for Modbus TCP and Modbus RTU.

### 5.3.1 Modbus TCP

#### Example

All functions codes for Modbus TCP/IP are supported. U can use the client like this:

```
#!/usr/bin/env python
# scripts/examples/simple_tcp_client.py
import socket

from umodbus import conf
from umodbus.client import tcp

# Enable values to be signed (default is False).
conf.SIGNED_VALUES = True

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
sock.connect(('localhost', 502))

# Returns a message or Application Data Unit (ADU) specific for doing
# Modbus TCP/IP.
message = tcp.write_multiple_coils(slave_id=1, starting_address=1, values=[1, 0, 1, 1])

# Response depends on Modbus function code. This particular returns the
# amount of coils written, in this case it is.
response = tcp.send_message(message, sock)

sock.close()
```

#### API

`umodbus.client.tcp.`**`send_message`**(*adu*, *sock*)
Send ADU over socket to to server and return parsed response.

**Parameters**

- **`adu`** – Request ADU.
- **`sock`** – Socket instance.

**Returns** Parsed response from server.

`umodbus.client.tcp.`**`parse_response_adu`**(*resp_adu*, *req_adu=None*)
Parse response ADU and return response data. Some functions require request ADU to fully understand request ADU.

**Parameters**

- **`resp_adu`** – Resonse ADU.
- **`req_adu`** – Request ADU, default None.

**Returns** Response data.

`umodbus.client.tcp.`**`read_coils`**(*slave_id*, *starting_address*, *quantity*)
Return ADU for Modbus function code 01: Read Coils.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`read_discrete_inputs`**(*slave_id*, *starting_address*, *quantity*)
Return ADU for Modbus function code 02: Read Discrete Inputs.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`read_holding_registers`**(*slave_id*, *starting_address*, *quantity*)
Return ADU for Modbus function code 03: Read Holding Registers.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`read_input_registers`**(*slave_id*, *starting_address*, *quantity*)
Return ADU for Modbus function code 04: Read Input Registers.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`write_single_coil`**(*slave_id*, *address*, *value*)
Return ADU for Modbus function code 05: Write Single Coil.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`write_single_register`**(*slave_id*, *address*, *value*)
Return ADU for Modbus function code 06: Write Single Register.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`write_multiple_coils`**(*slave_id*, *starting_address*, *values*)
Return ADU for Modbus function code 15: Write Multiple Coils.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.tcp.`**`write_multiple_registers`**(*slave_id*, *starting_address*, *values*)
Return ADU for Modbus function code 16: Write Multiple Registers.

**Parameters** **`slave_id`** – Number of slave.

**Returns** Byte array with ADU.

### 5.3.2 Modbus RTU

#### Example

**Note:** uModbus doesn't support all functions defined for Modbus RTU. It currently support the following functions:

- 01: Read Coils

- 02: Read Discrete Inputs
- 03: Read Holding Registers
- 04: Read Input Registers
- 05: Write Single Coil
- 06: Write Single Register
- 15: Write Multiple Coils
- 16: Write Multiple Registers

```
#!/usr/bin/env python
# scripts/example/simple_rtu_client.py
import fcntl
import struct
from serial import Serial, PARITY_NONE

from umodbus.client.serial impor rtu


def get_serial_port():
    """ Return serial.Serial instance, ready to use for RS485."""
    port = Serial(port='/dev/ttyS1', baudrate=9600, parity=PARITY_NONE,
                  stopbits=1, bytesize=8, timeout=1)

    fh = port.fileno()

    # A struct with configuration for serial port.
    serial_rs485 = struct.pack('hhhhhhhh', 1, 0, 0, 0, 0, 0, 0, 0)
    fcntl.ioctl(fh, 0x542F, serial_rs485)

    return port

serial_port = get_serial_port()

# Returns a message or Application Data Unit (ADU) specific for doing
# Modbus RTU.
message = rtu.write_multiple_coils(slave_id=1, address=1, values=[1, 0, 1, 1])

# Response depends on Modbus function code. This particular returns the
# amount of coils written, in this case it is.
response = rtu.send_message(message, serial_port)

serial_port.close()
```

## API

`umodbus.client.serial.rtu.`**`send_message`**(*adu*, *serial_port*)

Send Modbus message over serial port and parse response.

`umodbus.client.serial.rtu.`**`parse_response_adu`**(*resp_adu*, *req_adu=None*)

Parse response ADU and return response data. Some functions require request ADU to fully understand request ADU.

**Parameters**

- **`resp_adu`** – Resonse ADU.

- **req_adu** – Request ADU, default None.

**Returns** Response data.

`umodbus.client.serial.rtu.`**`read_coils`**(*slave_id*, *starting_address*, *quantity*)

Return ADU for Modbus function code 01: Read Coils.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`read_discrete_inputs`**(*slave_id*, *starting_address*, *quantity*)

Return ADU for Modbus function code 02: Read Discrete Inputs.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`read_holding_registers`**(*slave_id*, *starting_address*, *quantity*)

Return ADU for Modbus function code 03: Read Holding Registers.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`read_input_registers`**(*slave_id*, *starting_address*, *quantity*)

Return ADU for Modbus function code 04: Read Input Registers.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`write_single_coil`**(*slave_id*, *address*, *value*)

Return ADU for Modbus function code 05: Write Single Coil.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`write_single_register`**(*slave_id*, *address*, *value*)

Return ADU for Modbus function code 06: Write Single Register.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`write_multiple_coils`**(*slave_id*, *starting_address*, *values*)

Return ADU for Modbus function code 15: Write Multiple Coils.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

`umodbus.client.serial.rtu.`**`write_multiple_registers`**(*slave_id*, *starting_address*, *values*)

Return ADU for Modbus function code 16: Write Multiple Registers.

**Parameters** **slave_id** – Number of slave.

**Returns** Byte array with ADU.

## 5.4 Configuration

`umodbus.conf` is a global configuration object and is an instance of *umodbus.config.Config*. This instance can be used like this:

```
from umodbus import conf

conf.SIGNED_VALUES = True
```

**class** `umodbus.config.`**`Config`**

Class to hold global configuration.

**`SIGNED_VALUES`**

Whether values are signed or not. Default is False.

This value can also be set using the environment variable *UMODBUS_SIGNED_VALUES*.

## 5.5 Changelog

### 5.5.1 0.6.0 (2016-05-08)

**Feature**

- *#24* Add Modbus RTU client.

### 5.5.2 0.5.0 (2016-05-03)

**Bugs**

- #36 Parameter *function_code* is missing in signature of routes.

### 5.5.3 0.4.2 (2016-04-07)

**Bugs**

- #20 uModbus should close connection when client closes it.

### 5.5.4 0.4.1 (2016-01-22)

**Bugs**

- #31 Add subpackages *umodbus.client* and *umodbus._functions* to *setup.py*.

### 5.5.5 0.4.0 (2016-01-22)

**Features**

- #23 Implemenent Modbus client.
- #28 Implemenent signed integers for Modbus client.

### 5.5.6 0.3.1 (2015-12-12)

**Bugs**

- #18 Edit interface of *get_server* so socket options can now be set easily.

### 5.5.7 0.3.0 (2015-12-05)

**Features**

- #17 *RequestHandler.handle()* can be overridden easily.

### 5.5.8 0.2.0 (2015-11-19)

**Features**

- #10 Support for signed values.

**Bugs**

- #13 Fix shutdown of server in *simple_data_store.py*

### 5.5.9 0.1.2 (2015-11-16)

**Bugs**

- #8 *WriteMultipleCoils.create_from_request_pdu* sometimes doesn't unpack PDU correct.

### 5.5.10 0.1.1 (2015-11-12)

**Bugs**

- #7 Fix default stream and log level of *utils.log_to_stream*.

### 5.5.11 0.1.0 (2015-11-10)

- First release.

CHAPTER 6

# The Modbus protocol explained

## 6.1 Decompose requests

A Modbus requests and responses contains a Application Data Unit (ADU) which contains a Protocol Data Unit (PDU). The ADU is the enveloppe containing a message, the PDU is the message itself. Modbus requests can be send communication layers layers, like RTU or TCP/IP. The ADU for these layers differs. But the PDU, the message, has always the same strcuture, independent of the way it's transported.

### 6.1.1 PDU

**Note:** This section is based on MODBUS Application Protocol Specification V1.1b3

The Protocol Data Unit (PDU) is the request or response message and is indepedent of the underlying communication layer. This module only implements requests PDU's.

A request PDU contains two parts: a function code and request data. A response PDU contains the function code from the request and response data. The general structure is listed in table below:

| **Field** | **Size** (bytes) |
|---|---|
| Function code | 1 |
| data | N |

Below you see the request PDU with function code 1, requesting status of 3 coils, starting from coil 100:

```
>>> req_pdu = b'{\d\'
>>> function_code = req_pdu[:1]
>>> function_code
b'{'
>>> starting_address = req_pdu[1:3]
>>> starting_address
b'\d'
>>> quantity = req_pdu[3:]
>>> quantity
b'\'
```

A response PDU could look like this:

```
>>> resp_pdu = b'{{'
>>> function_code = resp_pdu[:1]
>>> function_code
```

```
b'{'
>>> byte_count = resp[1:2]
>>> byte_count
b'{'
>>> coil_status = resp[2:]
'b'
```

### 6.1.2 ADU for TCP/IP requests and responses

**Note:** This section is based on MODBUS Messaging on TCP/IP Implementation Guide V1.0b.

The Application Data Unit (ADU) for Modbus messages carried over a TCP/IP are build out of two components: a MBAP header and a PDU. The Modbus Application Header (MBAP) is what makes Modbus TCP/IP requests and responsen different from their counterparts send over a serial line. Below the components of the Modbus TCP/IP are listed together with their size in bytes:

| **Component** | **Size** (bytes) |
|---|---|
| MBAP Header | 7 |
| PDU | N |

Below you see an hexidecimal presentation of request over TCP/IP with Modbus function code 1. It requests data of slave with 1, starting at coil 100, for the length of 3 coils:

```
>>> # Read coils, starting from coil 100 for the length of 3 coils.
>>> adu = b'\\\\{{\d\'
```

The length of the ADU is 12 bytes:

```
>>> len(adu)
12
```

The MBAP header is 7 bytes long:

```
>>> mbap = adu[:7]
>>> mbap
b'\\\\{'
```

The MBAP header contains the following fields:

| **Field** | **Length** (bytes) | **Description** |
|---|---|---|
| Transaction identifier | 2 | Identification of a Modbus request/response transaction. |
| Protocol identifier | 2 | Protocol ID, is 0 for Modbus. |
| Length | 2 | Number of following bytes |
| Unit identifier | 1 | Identification of a remote slave |

When unpacked, these fields have the following values:

```
>>> transaction_id = mbap[:2]
>>> transaction_id
b'\'
>>> protocol_id = mbap[2:4]
>>> protocol_id
b'\\'
>>> length = mbap[4:6]
>>> length
b'\'
>>> unit_id = mbap[6:]
>>> unit_id
b'\x01'
```

The request in words: a request with Transaction ID 8 for slave 1. The request uses Protocol ID 0, which is the Modbus protocol. The length of the bytes after the Length field is 6 bytes. These 6 bytes are Unit Identifier (1 byte) + PDU (5 bytes).

### 6.1.3 ADU for RTU requests and responses

**Note:** This section is based on MODBUS over Serial Line Specification and Implementation Guide V1.02.

The ADU for Modbus RTU messages differs from Modbus TCP/IP messages. Messages send over RTU don't have a MBAP header, instead they have an Address field. This field contains the slave id. A CRC is appended to the message. Below all parts of a Modbus RTU message are listed together with their byte size:

| **Component** | **Size** (bytes) |
|---|---|
| Address field | 1 |
| PDU | N |
| CRC | 2 |

The CRC is calculated from the Address field and the PDU.

## 6.2 Modbus Functions

These Modbus functions have been implemented.

### 6.2.1 01: Read Coils

**class** `umodbus.functions.`**`ReadCoils`**(*starting_address*, *quantity*)

Implement Modbus function code 01.

> "This function code is used to read from 1 to 2000 contiguous status of coils in a remote device. The Request PDU specifies the starting address, i.e. the address of the first coil specified, and the number of coils. In the PDU Coils are addressed starting at zero. Therefore coils numbered 1-16 are addressed as 0-15.
>
> The coils in the response message are packed as one coil per bit of the data field. Status is indicated as 1= ON and 0= OFF. The LSB of the first data byte contains the output addressed in the query. The other coils follow toward the high order end of this byte, and from low order to high order in subsequent bytes.
>
> If the returned output quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data."
>
> —MODBUS Application Protocol Specification V1.1b3, chapter 6.1

The request PDU with function code 01 must be 5 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting address | 2 |
| Quantity | 2 |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHH', b'{\d\')
(1, 100, 3)
```

The reponse PDU varies in length, depending on the request. Each 8 coils require 1 byte. The amount of bytes needed represent status of the coils to can be calculated with: bytes = round(quantity / 8) + 1. This response contains (3 / 8 + 1) = 1 byte to describe the status of the coils. The structure of a compleet response PDU looks like this:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Byte count | 1 |
| Coil status | n |

Assume the status of 102 is 0, 101 is 1 and 100 is also 1. This is binary 011 which is decimal 3.

The PDU can packed like this:

```
>>> struct.pack('>BBB', function_code, byte_count, 3)
b'{{'
```

### 6.2.2 02: Read Discrete Inputs

**class** umodbus.functions.**ReadDiscreteInputs**(*starting_address*, *quantity*)
Implement Modbus function code 02.

> "This function code is used to read from 1 to 2000 contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, i.e. the address of the first input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore Discrete inputs numbered 1-16 are addressed as 0-15.
>
> The discrete inputs in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The LSB of the first data byte contains the input addressed in the query. The other inputs follow toward the high order end of this byte, and from low order to high order in subsequent bytes.
>
> If the returned input quantity is not a multiple of eight, the remaining bits in the final d ata byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data."
>
> —MODBUS Application Protocol Specification V1.1b3, chapter 6.2

The request PDU with function code 02 must be 5 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting address | 2 |
| Quantity | 2 |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHH', b'}\d\')
(2, 100, 3)
```

The reponse PDU varies in length, depending on the request. 8 inputs require 1 byte. The amount of bytes needed represent status of the inputs to can be calculated with: bytes = round(quantity / 8) + 1. This response contains (3 / 8 + 1) = 1 byte to describe the status of the inputs. The structure of a compleet response PDU looks like this:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Byte count | 1 |
| Coil status | n |

Assume the status of 102 is 0, 101 is 1 and 100 is also 1. This is binary 011 which is decimal 3.

The PDU can packed like this:

```
>>> struct.pack('>BBB', function_code, byte_count, 3)
b'}{'
```

### 6.2.3 03: Read Holding Registers

**class** `umodbus.functions.`**`ReadHoldingRegisters`**`(`*starting_address*, *quantity*`)`

Implement Modbus function code 03.

> "This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.
>
> The register data in the response message are packed as two bytes per register, with the binary contents right justified within each byte. For each register, the first byte contains the high order bits and the second contains the low order bits."
>
> —MODBUS Application Protocol Specification V1.1b3, chapter 6.3

The request PDU with function code 03 must be 5 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting address | 2 |
| Quantity | 2 |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHH', b'\d\')
(3, 100, 3)
```

The reponse PDU varies in length, depending on the request. By default, holding registers are 16 bit (2 bytes) values. So values of 3 holding registers is expressed in 2 * 3 = 6 bytes.

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Byte count | 1 |
| Register values | Quantity * 2 |

Assume the value of 100 is 8, 101 is 0 and 102 is also 15.

The PDU can packed like this:

```
>>> data = [8, 0, 15]
>>> struct.pack('>BBHHH', function_code, len(data) * 2, *data)
'\\\\'
```

### 6.2.4 04: Read Input Registers

**class** `umodbus.functions.`**`ReadInputRegisters`**(*starting_address*, *quantity*)

### 6.2.5 05: Write Single Coil

**class** `umodbus.functions.`**`WriteSingleCoil`**(*address*, *value*)

Implement Modbus function code 05.

> "This function code is used to write a single output to either ON or OFF in a remote device. The requested ON/OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the output to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the output.
>
> The Request PDU specifies the address of the coil to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0. The requested ON/OFF state is specified by a constant in the Coil Value field. A value of 0XFF00 requests the coil to be ON. A value of 0X0000 requests the coil to be off. All other values are illegal and will not affect the coil.
>
> The normal response is an echo of the request, returned after the coil state has been written."
>
> —MODBUS Application Protocol Specification V1.1b3, chapter 6.5

The request PDU with function code 05 must be 5 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Address | 2 |
| Value | 2 |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHH', b'\dÿ\')
(5, 100, 65280)
```

The reponse PDU is a copy of the request PDU.

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Address | 2 |
| Value | 2 |

### 6.2.6 06: Write Single Register

**class** `umodbus.functions.`**`WriteSingleRegister`**(*address*, *value*)

Implement Modbus function code 06.

> "This function code is used to write a single holding register in a remote device. The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0. The normal response is an echo of the request, returned after the register contents have been written."

—MODBUS Application Protocol Specification V1.1b3, chapter 6.6

The request PDU with function code 06 must be 5 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Address | 2 |
| Value | 2 |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHH', b'\d\')
(6, 100, 3)
```

The reponse PDU is a copy of the request PDU.

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Address | 2 |
| Value | 2 |

### 6.2.7 15: Write Multiple Coils

**class** `umodbus.functions.`**`WriteMultipleCoils`**(*starting_address*, *quantity*, *byte_count*, *values*)

Implement Modbus function 15 (0x0F) Write Multiple Coils.

"This function code is used to force each coil in a sequence of coils to either ON or OFF in a remote device. The Request PDU specifies the coil references to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0.

The requested ON/OFF states are specified by contents of the request data field. A logical '1' in a bit position of the field requests the corresponding output to be ON. A logical '0' requests it to be OFF.

The normal response returns the function code, starting address, and quantity of coils forced."

—MODBUS Application Protocol Specification V1.1b3, chapter 6.11

The request PDU with function code 15 must be at least 7 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting Address | 2 |
| Quantity | 2 |
| Byte count | 1 |
| Value | n |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHHBB', b'\d\{')
(16, 100, 3, 1, 5)
```

The reponse PDU is 5 bytes and contains following structure:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting address | 2 |
| Quantity | 2 |

### 6.2.8 16: Write Multiple Registers

**class** `umodbus.functions.`**`WriteMultipleRegisters`**(*starting_address*, *quantity*, *byte_count*, *values*)

Implement Modbus function 16 (0x10) Write Multiple Registers.

"This function code is used to write a block of contiguous registers (1 to 123 registers) in a remote device.

The requested written values are specified in the request data field. Data is packed as two bytes per register.

The normal response returns the function code, starting address, and quantity of registers written."

—MODBUS Application Protocol Specification V1.1b3, chapter 6.12

The request PDU with function code 16 must be at least 8 bytes:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting Address | 2 |
| Quantity | 2 |
| Byte count | 1 |
| Value | Quantity * 2 |

The PDU can unpacked to this:

```
>>> struct.unpack('>BHHBH', b'\d\{}\')
(16, 100, 1, 2, 5)
```

The reponse PDU is 5 bytes and contains following structure:

| Field | Length (bytes) |
|---|---|
| Function code | 1 |
| Starting address | 2 |
| Quantity | 2 |

# Python Module Index

# Index