

---

# **pyModbusTCP Documentation**

***Release 0.2.1***

**Loïc Lefebvre**

**Nov 28, 2023**



# CONTENTS

<b>1</b>	<b>Quick start guide</b>	<b>1</b>
1.1	Overview of the package . . . . .	1
1.2	Package setup . . . . .	1
1.3	ModbusClient: init . . . . .	2
1.4	ModbusClient: TCP link management . . . . .	2
1.5	ModbusClient: available modbus requests functions . . . . .	3
1.6	ModbusClient: debug mode . . . . .	3
1.7	utils module: Modbus data mangling . . . . .	4
<b>2</b>	<b>pyModbusTCP modules documentation</b>	<b>7</b>
2.1	Module pyModbusTCP.client . . . . .	7
2.2	Module pyModbusTCP.server . . . . .	12
2.3	Module pyModbusTCP.utils . . . . .	18
<b>3</b>	<b>pyModbusTCP examples</b>	<b>23</b>
3.1	Client: minimal code . . . . .	23
3.2	Client: read coils . . . . .	23
3.3	Client: read holding registers . . . . .	24
3.4	Client: write coils . . . . .	24
3.5	Client: add float (inheritance) . . . . .	25
3.6	Client: polling thread . . . . .	26
3.7	Server: basic usage . . . . .	27
3.8	Server: with an allow list . . . . .	28
3.9	Server: with change logger . . . . .	29
3.10	Server: Modbus/TCP serial gateway . . . . .	30
3.11	Server: schedule and alive word . . . . .	34
3.12	Server: virtual data . . . . .	35
	<b>Python Module Index</b>	<b>37</b>
	<b>Index</b>	<b>39</b>



## QUICK START GUIDE

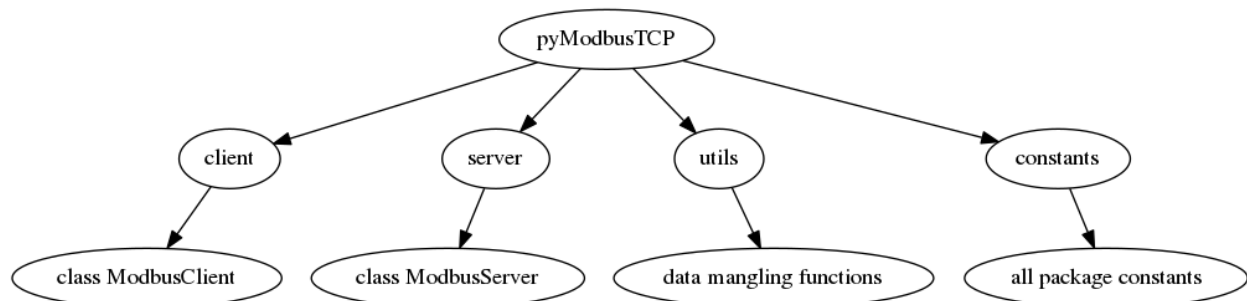
### 1.1 Overview of the package

pyModbusTCP give access to modbus/TCP server through the ModbusClient object. This class is define in the client module.

Since version 0.1.0, a server is available as ModbusServer class. This server is currently in test (API can change at any time).

To deal with frequent need of modbus data mangling (for example convert 32 bits IEEE float to 2x16 bits words) a special module named utils provide some helpful functions.

**Package map:**



### 1.2 Package setup

from PyPi:

```
# install the last available version (stable)
sudo pip3 install pyModbusTCP
# or upgrade from an older version
sudo pip3 install pyModbusTCP --upgrade

# you can also install a specific version (here v0.1.10)
sudo pip3 install pyModbusTCP==v0.1.10
```

from GitHub:

```
git clone https://github.com/sourceperl/pyModbusTCP.git
cd pyModbusTCP
```

(continues on next page)

(continued from previous page)

```
# here change "python" by your python target(s) version(s) (like python3.9)
sudo python setup.py install
```

## 1.3 ModbusClient: init

Init module from constructor (raise ValueError if host/port error):

```
from pyModbusTCP.client import ModbusClient

try:
    c = ModbusClient(host='localhost', port=502)
except ValueError:
    print("Error with host or port params")
```

Or with properties:

```
from pyModbusTCP.client import ModbusClient

c = ModbusClient()
c.host = 'localhost'
c.port = 502
```

## 1.4 ModbusClient: TCP link management

Since version 0.2.0, “auto open” mode is the default behaviour to deal with TCP open/close.

The “auto open” mode keep the TCP connection always open, so the default constructor is:

```
c = ModbusClient(host="localhost", auto_open=True, auto_close=False)
```

It’s also possible to open/close TCP socket before and after each request:

```
c = ModbusClient(host="localhost", auto_open=True, auto_close=True)
```

Another way to deal with connection is to manually set it. Like this:

```
c = ModbusClient(host="localhost", auto_open=False, auto_close=False)

# open the socket for 2 reads then close it.
if c.open():
    regs_list_1 = c.read_holding_registers(0, 10)
    regs_list_2 = c.read_holding_registers(55, 10)
    c.close()
```

## 1.5 ModbusClient: available modbus requests functions

See <http://en.wikipedia.org/wiki/Modbus> for full table.

Domain	Function name	Function code	ModbusClient function
Bit	Read Discrete Inputs	2	<i>read_discrete_inputs()</i>
	Read Coils	1	<i>read_coils()</i>
	Write Single Coil	5	<i>write_single_coil()</i>
	Write Multiple Coils	15	<i>write_multiple_coils()</i>
Register	Read Input Registers	4	<i>read_input_registers()</i>
	Read Holding Registers	3	<i>read_holding_registers()</i>
	Write Single Register	6	<i>write_single_register()</i>
	Write Multiple Registers	16	<i>write_multiple_registers()</i>
	Read/Write Multiple Registers	23	<i>write_read_multiple_registers()</i>
	Mask Write Register	22	n/a
	Read FIFO Queue	24	n/a
File	Read File Record	20	n/a
	Write File Record	21	n/a
	Read Exception Status	7	n/a
Diagnostic	Diagnostic	8	n/a
	Get Com Event Counter	11	n/a
	Get Com Event Log	12	n/a
	Report Slave ID	17	n/a
	Read Device Identification	43	<i>read_device_identification()</i>

## 1.6 ModbusClient: debug mode

If need, you can enable a debug mode for ModbusClient like this:

```
from pyModbusTCP.client import ModbusClient
c = ModbusClient(host="localhost", port=502, debug=True)
```

or:

```
c.debug = True
```

when debug is enable all debug message is print on console and you can see modbus frame:

```
c.read_holding_registers(0, 4)
```

print:

```
Tx
[E7 53 00 00 00 06 01] 03 00 00 00 04
Rx
[E7 53 00 00 00 0B 01] 03 08 00 00 00 6F 00 00 00 00
[0, 111, 0, 0]
```

## 1.7 utils module: Modbus data mangling

When we have to deal with the variety types of registers of PLC device, we often need some data mangling. Utils part of pyModbusTCP can help you in this task. Now, let's see some use cases.

- deal with negative numbers (two's complement):

```
from pyModbusTCP import utils

list_16_bits = [0x0000, 0xFFFF, 0x00FF, 0x8001]

# show "[0, -1, 255, -32767]"
print(utils.get_list_2comp(list_16_bits, 16))

# show "-1"
print(utils.get_2comp(list_16_bits[1], 16))
```

More at [http://en.wikipedia.org/wiki/Two%27s\\_complement](http://en.wikipedia.org/wiki/Two%27s_complement)

- convert integer of val\_size bits (default is 16) to an array of boolean:

```
from pyModbusTCP import utils

# show "[True, False, True, False, False, False, False, False]"
print(utils.get_bits_from_int(0x05, val_size=8))
```

- read of 32 bits registers (also know as long format):

```
from pyModbusTCP import utils

list_16_bits = [0x0123, 0x4567, 0xdead, 0xbeef]

# big endian sample (default)
list_32_bits = utils.word_list_to_long(list_16_bits)
# show "['0x1234567', '0xdeadbeef']"
print([hex(i) for i in list_32_bits])

# little endian sample
list_32_bits = utils.word_list_to_long(list_16_bits, big_endian=False)
# show "['0x45670123', '0xbeefdead']"
print([hex(i) for i in list_32_bits])
```

- IEEE single/double precision floating-point:

```
from pyModbusTCP import utils

# 32 bits IEEE single precision
# encode : python float 0.3 -> int 0x3e99999a
# display "0x3e99999a"
print(hex(utils.encode_ieee(0.3)))
# decode: python int 0x3e99999a -> float 0.3
# show "0.3000000011921" (it's not 0.3, precision leak with float...)
print(utils.decode_ieee(0x3e99999a))
```

(continues on next page)



(continued from previous page)

```
# 64 bits IEEE double precision
# encode: python float 6.62606957e-34 -> int 0x390b860bb596a559
# display "0x390b860bb596a559"
print(hex(utils.encode_ieee(6.62606957e-34, double=True)))
# decode: python int 0x390b860bb596a559 -> float 6.62606957e-34
# display "6.62606957e-34"
print(utils.decode_ieee(0x390b860bb596a559, double=True))
```



## PYMODBUSTCP MODULES DOCUMENTATION

Contents:

### 2.1 Module pyModbusTCP.client

pyModbusTCP Client

*This module provide the ModbusClient class used to deal with modbus server.*

#### 2.1.1 class ModbusClient

```
class pyModbusTCP.client.ModbusClient(host='localhost', port=502, unit_id=1, timeout=30.0,  
                                       debug=False, auto_open=True, auto_close=False)
```

Modbus TCP client.

```
__init__(host='localhost', port=502, unit_id=1, timeout=30.0, debug=False, auto_open=True,  
         auto_close=False)
```

Constructor.

##### Parameters

- **host** (*str*) – hostname or IPv4/IPv6 address server address
- **port** (*int*) – TCP port number
- **unit\_id** (*int*) – unit ID
- **timeout** (*float*) – socket timeout in seconds
- **debug** (*bool*) – debug state
- **auto\_open** (*bool*) – auto TCP connect
- **auto\_close** (*bool*) – auto TCP close)

##### Returns

Object ModbusClient

##### Return type

*ModbusClient*

##### property auto\_close

Get or set automatic TCP close after each request mode (True = turn on).

**property auto\_open**

Get or set automatic TCP connect mode (True = turn on).

**close()**

Close current TCP connection.

**custom\_request(*pdu*)**

Send a custom modbus request.

**Parameters**

**pdu** (*bytes*) – a modbus PDU (protocol data unit)

**Returns**

modbus frame PDU or None if error

**Return type**

bytes or None

**property debug**

Get or set the debug flag (True = turn on).

**property host**

Get or set the server to connect to.

This can be any string with a valid IPv4 / IPv6 address or hostname. Setting host to a new value will close the current socket.

**property is\_open**

Get current status of the TCP connection (True = open).

**property last\_error**

Last error code.

**property last\_error\_as\_txt**

Human-readable text that describe last error.

**property last\_except**

Return the last modbus exception code.

**property last\_except\_as\_full\_txt**

Verbose human-readable text that describe last modbus exception.

**property last\_except\_as\_txt**

Short human-readable text that describe last modbus exception.

**open()**

Connect to modbus server (open TCP connection).

**Returns**

connect status (True on success)

**Return type**

bool

**property port**

Get or set the current TCP port (default is 502).

Setting port to a new value will close the current socket.

**read\_coils**(*bit\_addr*, *bit\_nb=1*)

Modbus function READ\_COILS (0x01).

**Parameters**

- **bit\_addr** (*int*) – bit address (0 to 65535)
- **bit\_nb** (*int*) – number of bits to read (1 to 2000)

**Returns**

bits list or None if error

**Return type**

list of bool or None

**read\_device\_identification**(*read\_code=1*, *object\_id=0*)

Modbus function Read Device Identification (0x2B/0x0E).

**Parameters**

- **read\_code** (*int*) – read device id code, 1 to 3 for respectively: basic, regular and extended stream access, 4 for one specific identification object individual access (default is 1)
- **object\_id** (*int*) – object id of the first object to obtain (default is 0)

**Returns**

a DeviceIdentificationResponse instance with the data or None if the requests fails

**Return type**

*DeviceIdentificationResponse* or None

**read\_discrete\_inputs**(*bit\_addr*, *bit\_nb=1*)

Modbus function READ\_DISCRETE\_INPUTS (0x02).

**Parameters**

- **bit\_addr** (*int*) – bit address (0 to 65535)
- **bit\_nb** (*int*) – number of bits to read (1 to 2000)

**Returns**

bits list or None if error

**Return type**

list of bool or None

**read\_holding\_registers**(*reg\_addr*, *reg\_nb=1*)

Modbus function READ\_HOLDING\_REGISTERS (0x03).

**Parameters**

- **reg\_addr** (*int*) – register address (0 to 65535)
- **reg\_nb** (*int*) – number of registers to read (1 to 125)

**Returns**

registers list or None if fail

**Return type**

list of int or None

**read\_input\_registers**(*reg\_addr*, *reg\_nb=1*)

Modbus function READ\_INPUT\_REGISTERS (0x04).

**Parameters**

- **reg\_addr** (*int*) – register address (0 to 65535)
- **reg\_nb** (*int*) – number of registers to read (1 to 125)

**Returns**

registers list or None if fail

**Return type**

list of int or None

**property timeout**

Get or set requests timeout (default is 30 seconds).

The argument may be a floating point number for sub-second precision. Setting timeout to a new value will close the current socket.

**property unit\_id**

Get or set the modbus unit identifier (default is 1).

Any int from 0 to 255 is valid.

**property version**

Return the current package version as a str.

**write\_multiple\_coils**(*bits\_addr*, *bits\_value*)

Modbus function WRITE\_MULTIPLE\_COILS (0x0F).

**Parameters**

- **bits\_addr** (*int*) – bits address (0 to 65535)
- **bits\_value** (*list*) – bits values to write

**Returns**

True if write ok

**Return type**

bool

**write\_multiple\_registers**(*regs\_addr*, *regs\_value*)

Modbus function WRITE\_MULTIPLE\_REGISTERS (0x10).

**Parameters**

- **regs\_addr** (*int*) – registers address (0 to 65535)
- **regs\_value** (*list*) – registers values to write

**Returns**

True if write ok

**Return type**

bool

**write\_read\_multiple\_registers**(*write\_addr*, *write\_values*, *read\_addr*, *read\_nb=1*)

Modbus function WRITE\_READ\_MULTIPLE\_REGISTERS (0x17).

**Parameters**

- **write\_addr** (*int*) – write registers address (0 to 65535)
- **write\_values** (*list*) – registers values to write
- **read\_addr** (*int*) – read register address (0 to 65535)
- **read\_nb** (*int*) – number of registers to read (1 to 125)

**Returns**

registers list or None if fail

**Return type**

list of int or None

**write\_single\_coil**(*bit\_addr*, *bit\_value*)

Modbus function WRITE\_SINGLE\_COIL (0x05).

**Parameters**

- **bit\_addr** (*int*) – bit address (0 to 65535)
- **bit\_value** (*bool*) – bit value to write

**Returns**

True if write ok

**Return type**

bool

**write\_single\_register**(*reg\_addr*, *reg\_value*)

Modbus function WRITE\_SINGLE\_REGISTER (0x06).

**Parameters**

- **reg\_addr** (*int*) – register address (0 to 65535)
- **reg\_value** (*int*) – register value to write

**Returns**

True if write ok

**Return type**

bool

## 2.1.2 class DeviceIdentificationResponse

```
class pyModbusTCP.client.DeviceIdentificationResponse(conformity_level: int = 0, more_follows: int = 0, next_object_id: int = 0, objects_by_id: ~typing.Dict[int, bytes] = <factory>)
```

Modbus TCP client function read\_device\_identification() response struct.

**Parameters**

- **conformity\_level** (*int*) – this represents supported access and object type
- **more\_follows** (*int*) – for stream request can be set to 0xff if other objects are available (0x00 in other cases)
- **next\_object\_id** (*int*) – the next object id to be asked by following transaction
- **objects\_by\_id** (*dict*) – a dictionary with requested object (dict keys are object id as int)

```
__init__(conformity_level: int = 0, more_follows: int = 0, next_object_id: int = 0, objects_by_id: ~typing.Dict[int, bytes] = <factory>) → None
```

## 2.2 Module pyModbusTCP.server

pyModbusTCP Server

*This module provide the class for the modbus server, it's data handler interface and finally the data bank.*

### 2.2.1 class ModbusServer

```
class pyModbusTCP.server.ModbusServer(host='localhost', port=502, no_block=False, ipv6=False,  
                                       data_bank=None, data_hdl=None, ext_engine=None,  
                                       device_id=None)
```

Modbus TCP server

```
__init__(host='localhost', port=502, no_block=False, ipv6=False, data_bank=None, data_hdl=None,  
         ext_engine=None, device_id=None)
```

Constructor

Modbus server constructor.

#### Parameters

- **host** (*str*) – hostname or IPv4/IPv6 address server address (default is 'localhost')
- **port** (*int*) – TCP port number (default is 502)
- **no\_block** (*bool*) – no block mode, i.e. start() will return (default is False)
- **ipv6** (*bool*) – use ipv6 stack (default is False)
- **data\_bank** (*DataBank*) – instance of custom data bank, if you don't want the default one (optional)
- **data\_hdl** (*DataHandler*) – instance of custom data handler, if you don't want the default one (optional)
- **ext\_engine** (*callable*) – an external engine reference (ref to ext\_engine(session\_data)) (optional)
- **device\_id** (*DeviceIdentification*) – instance of DeviceIdentification class for read device identification request (optional)

```
class ClientInfo(address="", port=0)
```

Container class for client information

```
exception DataFormatError
```

Exception raise by ModbusServer for data format errors.

```
exception Error
```

Base exception for ModbusServer related errors.

```
class MBAP(transaction_id=0, protocol_id=0, length=0, unit_id=0)
```

MBAP (Modbus Application Protocol) container class.

```
class ModbusService(request, client_address, server)
```

```
exception NetworkError
```

Exception raise by ModbusServer on I/O errors.



```
class PDU(raw=b'')
    PDU (Protocol Data Unit) container class.

class ServerInfo
    Container class for server information

class SessionData
    Container class for server session data.

property is_run
    Return True if server running.

start()
    Start the server.

    This function will block (or not if no_block flag is set).

stop()
    Stop the server.
```

### 2.2.2 class DataHandler

```
class pyModbusTCP.server.DataHandler(data_bank=None)
    Default data handler for ModbusServer, map server threads calls to DataBank.
    Custom handler must derive from this class.

    __init__(data_bank=None)
        Constructor
        Modbus server data handler constructor.

        Parameters
        data_bank (DataBank) – a reference to custom DefaultDataBank

    read_coils(address, count, srv_info)
        Call by server for reading in coils space

        Parameters
        • address (int) – start address
        • count (int) – number of coils
        • srv_info (ModbusServer.ServerInfo) – some server info

        Return type
        Return

    read_d_inputs(address, count, srv_info)
        Call by server for reading in the discrete inputs space

        Parameters
        • address (int) – start address
        • count (int) – number of discrete inputs
        • srv_info (ModbusServer.ServerInfo) – some server info

        Return type
        Return
```

**read\_h\_regs**(*address*, *count*, *srv\_info*)

Call by server for reading in the holding registers space

**Parameters**

- **address** (*int*) – start address
- **count** (*int*) – number of holding registers
- **srv\_info** ([ModbusServer.ServerInfo](#)) – some server info

**Return type**

Return

**read\_i\_regs**(*address*, *count*, *srv\_info*)

Call by server for reading in the input registers space

**Parameters**

- **address** (*int*) – start address
- **count** (*int*) – number of input registers
- **srv\_info** ([ModbusServer.ServerInfo](#)) – some server info

**Return type**

Return

**write\_coils**(*address*, *bits\_l*, *srv\_info*)

Call by server for writing in the coils space

**Parameters**

- **address** (*int*) – start address
- **bits\_l** (*list*) – list of boolean to write
- **srv\_info** ([ModbusServer.ServerInfo](#)) – some server info

**Return type**

Return

**write\_h\_regs**(*address*, *words\_l*, *srv\_info*)

Call by server for writing in the holding registers space

**Parameters**

- **address** (*int*) – start address
- **words\_l** (*list*) – list of word value to write
- **srv\_info** ([ModbusServer.ServerInfo](#)) – some server info

**Return type**

Return

### 2.2.3 class DataBank

```
class pyModbusTCP.server.DataBank(coils_size=65536, coils_default_value=False, d_inputs_size=65536,
                                   d_inputs_default_value=False, h_regs_size=65536,
                                   h_regs_default_value=0, i_regs_size=65536, i_regs_default_value=0,
                                   virtual_mode=False)
```

Data space class with thread safe access functions

```
__init__(coils_size=65536, coils_default_value=False, d_inputs_size=65536,
          d_inputs_default_value=False, h_regs_size=65536, h_regs_default_value=0, i_regs_size=65536,
          i_regs_default_value=0, virtual_mode=False)
```

Constructor

Modbus server data bank constructor.

#### Parameters

- **coils\_size** (*int*) – Number of coils to allocate (default is 65536)
- **coils\_default\_value** (*bool*) – Coils default value at startup (default is False)
- **d\_inputs\_size** (*int*) – Number of discrete inputs to allocate (default is 65536)
- **d\_inputs\_default\_value** (*bool*) – Discrete inputs default value at startup (default is False)
- **h\_regs\_size** (*int*) – Number of holding registers to allocate (default is 65536)
- **h\_regs\_default\_value** (*int*) – Holding registers default value at startup (default is 0)
- **i\_regs\_size** (*int*) – Number of input registers to allocate (default is 65536)
- **i\_regs\_default\_value** (*int*) – Input registers default value at startup (default is 0)
- **virtual\_mode** (*bool*) – Disallow all modbus data space to work with virtual values (default is False)

```
get_coils(address, number=1, srv_info=None)
```

Read data on server coils space

#### Parameters

- **address** (*int*) – start address
- **number** (*int*) – number of bits (optional)
- **srv\_info** (*ModbusServer.ServerInfo*) – some server info (must be set by server only)

#### Returns

list of bool or None if error

#### Return type

list or None

```
get_discrete_inputs(address, number=1, srv_info=None)
```

Read data on server discrete inputs space

#### Parameters

- **address** (*int*) – start address
- **number** (*int*) – number of bits (optional)
- **srv\_info** (*ModbusServerInfo*) – some server info (must be set by server only)

**Returns**

list of bool or None if error

**Return type**

list or None

**get\_holding\_registers**(*address*, *number=1*, *srv\_info=None*)

Read data on server holding registers space

**Parameters**

- **address** (*int*) – start address
- **number** (*int*) – number of words (optional)
- **srv\_info** (*ModbusServerInfo*) – some server info (must be set by server only)

**Returns**

list of int or None if error

**Return type**

list or None

**get\_input\_registers**(*address*, *number=1*, *srv\_info=None*)

Read data on server input registers space

**Parameters**

- **address** (*int*) – start address
- **number** (*int*) – number of words (optional)
- **srv\_info** (*ModbusServerInfo*) – some server info (must be set by server only)

**Returns**

list of int or None if error

**Return type**

list or None

**on\_coils\_change**(*address*, *from\_value*, *to\_value*, *srv\_info*)

Call by server when a value change occur in coils space

This method is provided to be overridden with user code to catch changes

**Parameters**

- **address** (*int*) – address of coil
- **from\_value** (*bool*) – coil original value
- **to\_value** (*bool*) – coil next value
- **srv\_info** (*ModbusServerInfo*) – some server info

**on\_holding\_registers\_change**(*address*, *from\_value*, *to\_value*, *srv\_info*)

Call by server when a value change occur in holding registers space

This method is provided to be overridden with user code to catch changes

**Parameters**

- **address** (*int*) – address of register
- **from\_value** (*int*) – register original value
- **to\_value** (*int*) – register next value

- **srv\_info** (*ModbusServerInfo*) – some server info

**set\_coils**(*address*, *bit\_list*, *srv\_info=None*)

Write data to server coils space

**Parameters**

- **address** (*int*) – start address
- **bit\_list** (*list*) – a list of bool to write
- **srv\_info** (*ModbusServerInfo*) – some server info (must be set by server only)

**Returns**

True if success or None if error

**Return type**

bool or None

**Raises**

**ValueError** – if bit\_list members cannot be converted to bool

**set\_discrete\_inputs**(*address*, *bit\_list*)

Write data to server discrete inputs space

**Parameters**

- **address** (*int*) – start address
- **bit\_list** (*list*) – a list of bool to write

**Returns**

True if success or None if error

**Return type**

bool or None

**Raises**

**ValueError** – if bit\_list members cannot be converted to bool

**set\_holding\_registers**(*address*, *word\_list*, *srv\_info=None*)

Write data to server holding registers space

**Parameters**

- **address** (*int*) – start address
- **word\_list** (*list*) – a list of word to write
- **srv\_info** (*ModbusServerInfo*) – some server info (must be set by server only)

**Returns**

True if success or None if error

**Return type**

bool or None

**Raises**

**ValueError** – if word\_list members cannot be converted to int

**set\_input\_registers**(*address*, *word\_list*)

Write data to server input registers space

**Parameters**

- **address** (*int*) – start address

- **word\_list** (*list*) – a list of word to write

**Returns**

True if success or None if error

**Return type**

bool or None

**Raises**

**ValueError** – if word\_list members cannot be converted to int

## 2.2.4 class DeviceIdentification

```
class pyModbusTCP.server.DeviceIdentification(vendor_name=b", product_code=b",  
                                              major_minor_revision=b", vendor_url=b",  
                                              product_name=b", model_name=b",  
                                              user_application_name=b", objects_id=None)
```

Container class for device identification objects (MEI type 0x0E) return by function 0x2B.

```
__init__(vendor_name=b", product_code=b", major_minor_revision=b", vendor_url=b",  
         product_name=b", model_name=b", user_application_name=b", objects_id=None)
```

Constructor

**Parameters**

- **vendor\_name** (*bytes*) – VendorName mandatory object
- **product\_code** (*bytes*) – ProductCode mandatory object
- **major\_minor\_revision** (*bytes*) – MajorMinorRevision mandatory object
- **vendor\_url** (*bytes*) – VendorUrl regular object
- **product\_name** (*bytes*) – ProductName regular object
- **model\_name** (*bytes*) – ModelName regular object
- **user\_application\_name** (*bytes*) – UserApplicationName regular object
- **objects\_id** (*dict*) – Objects values by id as dict example: {42:b'value'} (optional)

## 2.3 Module pyModbusTCP.utils

*This module provide a set of functions for modbus data mangling.*

### 2.3.1 Bit functions

pyModbusTCP utils functions

```
pyModbusTCP.utils.byte_length(bit_length)
```

Return the number of bytes needs to contain a *bit\_length* structure.

**Parameters**

**bit\_length** (*int*) – the number of bits

**Returns**

the number of bytes

**Return type**

int

`pyModbusTCP.utils.get_bits_from_int(val_int, val_size=16)`

Get the list of bits of val\_int integer (default size is 16 bits).

Return bits list, the least significant bit first. Use list.reverse() for msb first.

**Parameters**

- **val\_int** (*int*) – integer value
- **val\_size** (*int*) – bit length of integer (word = 16, long = 32) (optional)

**Returns**

list of boolean “bits” (the least significant first)

**Return type**

list

`pyModbusTCP.utils.reset_bit(value, offset)`

Reset a bit at offset position.

**Parameters**

- **value** (*int*) – value of integer where reset the bit
- **offset** (*int*) – bit offset (0 is lsb)

**Returns**

value of integer with bit reset

**Return type**

int

`pyModbusTCP.utils.set_bit(value, offset)`

Set a bit at offset position.

**Parameters**

- **value** (*int*) – value of integer where set the bit
- **offset** (*int*) – bit offset (0 is lsb)

**Returns**

value of integer with bit set

**Return type**

int

`pyModbusTCP.utils.test_bit(value, offset)`

Test a bit at offset position.

**Parameters**

- **value** (*int*) – value of integer to test
- **offset** (*int*) – bit offset (0 is lsb)

**Returns**

value of bit at offset position

**Return type**

bool

`pyModbusTCP.utils.toggle_bit(value, offset)`

Return an integer with the bit at offset position inverted.

**Parameters**

- **value** (*int*) – value of integer where invert the bit
- **offset** (*int*) – bit offset (0 is lsb)

**Returns**

value of integer with bit inverted

**Return type**

int

## 2.3.2 Word functions

pyModbusTCP utils functions

`pyModbusTCP.utils.long_list_to_word(val_list, big_endian=True, long_long=False)`

Long (32 bits) or long long (64 bits) list to word (16 bits) list.

By default `long_list_to_word()` use big endian order. For use little endian, set `big_endian` param to False. Input format could be long long with `long_long` param to True.

**Parameters**

- **val\_list** (*list*) – list of 32 bits int value
- **big\_endian** (*bool*) – True for big endian/False for little (optional)
- **long\_long** (*bool*) – True for long long 64 bits, default is long 32 bits (optional)

**Returns**

list of 16 bits int value

**Return type**

list

`pyModbusTCP.utils.word_list_to_long(val_list, big_endian=True, long_long=False)`

Word list (16 bits) to long (32 bits) or long long (64 bits) list.

By default, `word_list_to_long()` use big endian order. For use little endian, set `big_endian` param to False. Output format could be long long with `long_long` option set to True.

**Parameters**

- **val\_list** (*list*) – list of 16 bits int value
- **big\_endian** (*bool*) – True for big endian/False for little (optional)
- **long\_long** (*bool*) – True for long long 64 bits, default is long 32 bits (optional)

**Returns**

list of 32 bits int value

**Return type**

list



### 2.3.3 Two's complement functions

pyModbusTCP utils functions

`pyModbusTCP.utils.get_2comp(val_int, val_size=16)`

Get the 2's complement of Python int `val_int`.

**Parameters**

- **val\_int** (*int*) – int value to apply 2's complement
- **val\_size** (*int*) – bit size of int value (word = 16, long = 32) (optional)

**Returns**

2's complement result

**Return type**

int

**Raises**

**ValueError** – if mismatch between `val_int` and `val_size`

`pyModbusTCP.utils.get_list_2comp(val_list, val_size=16)`

Get the 2's complement of Python list `val_list`.

**Parameters**

- **val\_list** (*list*) – list of int value to apply 2's complement
- **val\_size** (*int*) – bit size of int value (word = 16, long = 32) (optional)

**Returns**

2's complement result

**Return type**

list

### 2.3.4 IEEE floating-point functions

pyModbusTCP utils functions

`pyModbusTCP.utils.decode_ieee(val_int, double=False)`

Decode Python int (32 bits integer) as an IEEE single or double precision format.

Support NaN.

**Parameters**

- **val\_int** (*int*) – a 32 or 64 bits integer as an int Python value
- **double** (*bool*) – set to decode as a 64 bits double precision, default is 32 bits single (optional)

**Returns**

float result

**Return type**

float

`pyModbusTCP.utils.encode_ieee(val_float, double=False)`

Encode Python float to int (32 bits integer) as an IEEE single or double precision format.

Support NaN.

**Parameters**

- **val\_float** (*float*) – float value to convert
- **double** (*bool*) – set to encode as a 64 bits double precision, default is 32 bits single (optional)

**Returns**

IEEE 32 bits (single precision) as Python int

**Return type**

int

## 2.3.5 Misc functions

pyModbusTCP utils functions

pyModbusTCP.utils.**crc16**(*frame*)

Compute CRC16.

**Parameters**

**frame** (*bytes*) – frame

**Returns**

CRC16

**Return type**

int

pyModbusTCP.utils.**valid\_host**(*host\_str*)

Validate a host string.

Can be an IPv4/6 address or a valid hostname.

**Parameters**

**host\_str** (*str*) – the host string to test

**Returns**

True if host\_str is valid

**Return type**

bool

## PYMODBUSTCP EXAMPLES

*Here some examples to see pyModbusTCP in some use cases*

### 3.1 Client: minimal code

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

""" Minimal code example. """

from pyModbusTCP.client import ModbusClient

# read 3 coils at @@ on localhost server
print('coils=%s' % ModbusClient().read_coils(0, 3))
```

### 3.2 Client: read coils

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

""" Read 10 coils and print result on stdout. """

import time
from pyModbusTCP.client import ModbusClient

# init modbus client
c = ModbusClient(host='localhost', port=502, auto_open=True, debug=False)

# main read loop
while True:
    # read 10 bits (= coils) at address 0, store result in coils list
    coils_l = c.read_coils(0, 10)

    # if success display registers
    if coils_l:
        print('coil ad #0 to 9: %s' % coils_l)
    else:
        print('unable to read coils')
```

(continues on next page)

(continued from previous page)

```
# sleep 2s before next polling
time.sleep(2)
```

### 3.3 Client: read holding registers

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

""" Read 10 holding registers and print result on stdout. """

import time
from pyModbusTCP.client import ModbusClient

# init modbus client
c = ModbusClient(debug=False, auto_open=True)

# main read loop
while True:
    # read 10 registers at address 0, store result in regs_l
    regs_l = c.read_holding_registers(0, 10)

    # if success display registers
    if regs_l:
        print('reg ad #0 to 9: %s' % regs_l)
    else:
        print('unable to read registers')

    # sleep 2s before next polling
    time.sleep(2)
```

### 3.4 Client: write coils

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

"""Write 4 coils to True, wait 2s, write False and redo it."""

import time
from pyModbusTCP.client import ModbusClient

# init
c = ModbusClient(host='localhost', port=502, auto_open=True, debug=False)
bit = True

# main loop
while True:
    # write 4 bits in modbus address 0 to 3
    print('write bits')
```

(continues on next page)

(continued from previous page)

```

print('-----\n')
for ad in range(4):
    is_ok = c.write_single_coil(ad, bit)
    if is_ok:
        print('coil #s: write to %s' % (ad, bit))
    else:
        print('coil #s: unable to write %s' % (ad, bit))
    time.sleep(0.5)

print('')
time.sleep(1)

# read 4 bits in modbus address 0 to 3
print('read bits')
print('-----\n')
bits = c.read_coils(0, 4)
if bits:
    print('coils #0 to 3: %s' % bits)
else:
    print('coils #0 to 3: unable to read')

# toggle
bit = not bit
# sleep 2s before next polling
print('')
time.sleep(2)

```

### 3.5 Client: add float (inheritance)

```

#!/usr/bin/env python3

""" How-to add float support to ModbusClient. """

from pyModbusTCP.client import ModbusClient
from pyModbusTCP.utils import encode_ieee, decode_ieee, \
    long_list_to_word, word_list_to_long

class FloatModbusClient(ModbusClient):
    """A ModbusClient class with float support."""

    def read_float(self, address, number=1):
        """Read float(s) with read holding registers."""
        reg_l = self.read_holding_registers(address, number * 2)
        if reg_l:
            return [decode_ieee(f) for f in word_list_to_long(reg_l)]
        else:
            return None

    def write_float(self, address, floats_list):

```

(continues on next page)

(continued from previous page)

```

        """Write float(s) with write multiple registers."""
        b32_l = [encode_ieee(f) for f in floats_list]
        b16_l = long_list_to_word(b32_l)
        return self.write_multiple_registers(address, b16_l)

if __name__ == '__main__':
    # init modbus client
    c = FloatModbusClient(host='localhost', port=502, auto_open=True)

    # write 10.0 at @0
    c.write_float(0, [10.0])

    # read @0 to 9
    float_l = c.read_float(0, 10)
    print(float_l)

    c.close()

```

### 3.6 Client: polling thread

```

#!/usr/bin/env python3

"""
modbus polling thread
~~~~~

Start a thread for polling a set of registers, display result on console.
Exit with ctrl+c.
"""

import time
from threading import Thread, Lock
from pyModbusTCP.client import ModbusClient

SERVER_HOST = "localhost"
SERVER_PORT = 502

# set global
regs = []

# init a thread lock
regs_lock = Lock()

def polling_thread():
    """Modbus polling thread."""
    global regs, regs_lock
    c = ModbusClient(host=SERVER_HOST, port=SERVER_PORT, auto_open=True)
    # polling loop

```

(continues on next page)

(continued from previous page)

```

while True:
    # do modbus reading on socket
    reg_list = c.read_holding_registers(0, 10)
    # if read is ok, store result in regs (with thread lock)
    if reg_list:
        with regs_lock:
            regs = list(reg_list)
    # 1s before next polling
    time.sleep(1)

# start polling thread
tp = Thread(target=polling_thread)
# set daemon: polling thread will exit if main thread exit
tp.daemon = True
tp.start()

# display loop (in main thread)
while True:
    # print regs list (with thread lock synchronization)
    with regs_lock:
        print(regs)
    # 1s before next print
    time.sleep(1)

```

### 3.7 Server: basic usage

```

#!/usr/bin/env python3

"""
Modbus/TCP server
~~~~~

Run this as root to listen on TCP privileged ports (<= 1024).

Add "--host 0.0.0.0" to listen on all available IPv4 addresses of the host.
$ sudo ./server.py --host 0.0.0.0
"""

import argparse
import logging
from pyModbusTCP.server import ModbusServer

# init logging
logging.basicConfig()
# parse args
parser = argparse.ArgumentParser()
parser.add_argument('-H', '--host', type=str, default='localhost', help='Host (default: ↵
↵localhost)')

```

(continues on next page)

(continued from previous page)

```

parser.add_argument('-p', '--port', type=int, default=502, help='TCP port (default: 502)
→')
parser.add_argument('-d', '--debug', action='store_true', help='set debug mode')
args = parser.parse_args()
# logging setup
if args.debug:
    logging.getLogger('pyModbusTCP.server').setLevel(logging.DEBUG)
# start modbus server
server = ModbusServer(host=args.host, port=args.port)
server.start()

```

### 3.8 Server: with an allow list

```

#!/usr/bin/env python3

"""
An example of Modbus/TCP server which allow modbus read and/or write only from
specific IPs.

Run this as root to listen on TCP privileged ports (<= 1024).
"""

import argparse
from pyModbusTCP.server import ModbusServer, DataHandler
from pyModbusTCP.constants import EXP_ILLEGAL_FUNCTION

# some const
ALLOW_R_L = ['127.0.0.1', '192.168.0.10']
ALLOW_W_L = ['127.0.0.1']

# a custom data handler with IPs filter
class MyDataHandler(DataHandler):
    def read_coils(self, address, count, srv_info):
        if srv_info.client.address in ALLOW_R_L:
            return super().read_coils(address, count, srv_info)
        else:
            return DataHandler.Return(exp_code=EXP_ILLEGAL_FUNCTION)

    def read_d_inputs(self, address, count, srv_info):
        if srv_info.client.address in ALLOW_R_L:
            return super().read_d_inputs(address, count, srv_info)
        else:
            return DataHandler.Return(exp_code=EXP_ILLEGAL_FUNCTION)

    def read_h_regs(self, address, count, srv_info):
        if srv_info.client.address in ALLOW_R_L:
            return super().read_h_regs(address, count, srv_info)
        else:

```

(continues on next page)



(continued from previous page)

```

        return DataHandler.Return(exp_code=EXP_ILLEGAL_FUNCTION)

    def read_i_regs(self, address, count, srv_info):
        if srv_info.client.address in ALLOW_R_L:
            return super().read_i_regs(address, count, srv_info)
        else:
            return DataHandler.Return(exp_code=EXP_ILLEGAL_FUNCTION)

    def write_coils(self, address, bits_l, srv_info):
        if srv_info.client.address in ALLOW_W_L:
            return super().write_coils(address, bits_l, srv_info)
        else:
            return DataHandler.Return(exp_code=EXP_ILLEGAL_FUNCTION)

    def write_h_regs(self, address, words_l, srv_info):
        if srv_info.client.address in ALLOW_W_L:
            return super().write_h_regs(address, words_l, srv_info)
        else:
            return DataHandler.Return(exp_code=EXP_ILLEGAL_FUNCTION)

if __name__ == '__main__':
    # parse args
    parser = argparse.ArgumentParser()
    parser.add_argument('-H', '--host', type=str, default='localhost', help='Host_
↳(default: localhost)')
    parser.add_argument('-p', '--port', type=int, default=502, help='TCP port (default:_
↳502)')
    args = parser.parse_args()
    # init modbus server and start it
    server = ModbusServer(host=args.host, port=args.port, data_hdl=MyDataHandler())
    server.start()

```

### 3.9 Server: with change logger

```

#!/usr/bin/env python3

"""
An example of Modbus/TCP server with a change logger.

Run this as root to listen on TCP privileged ports (<= 1024).
"""

import argparse
import logging
from pyModbusTCP.server import ModbusServer, DataBank

class MyDataBank(DataBank):
    """A custom ModbusServerDataBank for override on xxx_change methods."""

```

(continues on next page)

(continued from previous page)

```

def on_coils_change(self, address, from_value, to_value, srv_info):
    """Call by server when change occur on coils space."""
    msg = 'change in coil space [{0!r:^5} > {1!r:^5}] at @ 0x{2:04X} from ip: {3:<15}'
    ↪ '

    msg = msg.format(from_value, to_value, address, srv_info.client.address)
    logging.info(msg)

def on_holding_registers_change(self, address, from_value, to_value, srv_info):
    """Call by server when change occur on holding registers space."""
    msg = 'change in hreg space [{0!r:^5} > {1!r:^5}] at @ 0x{2:04X} from ip: {3:<15}'
    ↪ '

    msg = msg.format(from_value, to_value, address, srv_info.client.address)
    logging.info(msg)

if __name__ == '__main__':
    # parse args
    parser = argparse.ArgumentParser()
    parser.add_argument('-H', '--host', type=str, default='localhost', help='Host ↪
    ↪ (default: localhost)')
    parser.add_argument('-p', '--port', type=int, default=502, help='TCP port (default: ↪
    ↪ 502)')
    args = parser.parse_args()
    # logging setup
    logging.basicConfig(format='%(asctime)s %(message)s', level=logging.INFO)
    # init modbus server and start it
    server = ModbusServer(host=args.host, port=args.port, data_bank=MyDataBank())
    server.start()

```

### 3.10 Server: Modbus/TCP serial gateway

```

#!/usr/bin/env python3

"""
Modbus/TCP basic gateway (RTU slave(s) attached)
~~~~~

[pyModbusTCP server] -> [ModbusSerialWorker] -> [serial RTU devices]

Run this as root to listen on TCP privileged ports (<= 1024).

Open /dev/ttyUSB0 at 115200 bauds and relay it RTU messages to slave(s).
$ sudo ./server_serial_gw.py --baudrate 115200 /dev/ttyUSB0
"""

import argparse
import logging
import queue
import struct

```

(continues on next page)

(continued from previous page)

```

from threading import Event
from queue import Queue
from pyModbusTCP.server import ModbusServer
from pyModbusTCP.utils import crc16
from pyModbusTCP.constants import EXP_GATEWAY_PATH_UNAVAILABLE, EXP_GATEWAY_TARGET_
↳ DEVICE_FAILED_TO_RESPOND
# need sudo pip install pyserial==3.4
from serial import Serial, serialutil

# some class
class ModbusRTUFrame:
    """ Modbus RTU frame container class. """

    def __init__(self, raw=b''):
        # public
        self.raw = raw

    @property
    def pdu(self):
        """Return PDU part of frame."""
        return self.raw[1:-2]

    @property
    def slave_address(self):
        """Return slave address part of frame."""
        return self.raw[0]

    @property
    def function_code(self):
        """Return function code part of frame."""
        return self.raw[1]

    @property
    def is_valid(self):
        """Check if frame is valid.

        :return: True if frame is valid
        :rtype: bool
        """
        return len(self.raw) > 4 and crc16(self.raw) == 0

    def build(self, raw_pdu, slave_ad):
        """Build a full modbus RTU message from PDU and slave address.

        :param raw_pdu: modbus as raw value
        :type raw_pdu: bytes
        :param slave_ad: address of the slave
        :type slave_ad: int
        """
        # [address] + PDU
        tmp_raw = struct.pack('B', slave_ad) + raw_pdu

```

(continues on next page)

(continued from previous page)

```

    # [address] + PDU + [CRC 16]
    tmp_raw += struct.pack('<H', crc16(tmp_raw))
    self.raw = tmp_raw

class RtuQuery:
    """ Request container to deal with modbus serial worker. """

    def __init__(self):
        self.completed = Event()
        self.request = ModbusRTUFrame()
        self.response = ModbusRTUFrame()

class ModbusSerialWorker:
    """ A serial worker to manage I/O with RTU devices. """

    def __init__(self, port, timeout=1.0, end_of_frame=0.05):
        # public
        self.serial_port = port
        self.timeout = timeout
        self.end_of_frame = end_of_frame
        # internal request queue
        # accept 5 simultaneous requests before overloaded exception is return
        self.rtu_queries_q = Queue(maxsize=5)

    def loop(self):
        """Serial worker main loop."""
        while True:
            # get next exchange from queue
            rtu_query = self.rtu_queries_q.get()
            # send to serial
            self.serial_port.reset_input_buffer()
            self.serial_port.write(rtu_query.request.raw)
            # receive from serial
            # wait for first byte of data until timeout delay
            self.serial_port.timeout = self.timeout
            rx_raw = self.serial_port.read(1)
            # if ok, wait for the remaining
            if rx_raw:
                self.serial_port.timeout = self.end_of_frame
                # wait for next bytes of data until end of frame delay
                while True:
                    rx_chunk = self.serial_port.read(256)
                    if not rx_chunk:
                        break
                    else:
                        rx_raw += rx_chunk
                rtu_query.response.raw = rx_raw
            # mark all as done
            rtu_query.completed.set()
            self.rtu_queries_q.task_done()

```

(continues on next page)

(continued from previous page)

```

def srv_engine_entry(self, session_data):
    """Server engine entry point (pass request to serial worker queries queue).

    :param session_data: server session data
    :type session_data: ModbusServer.SessionData
    """
    # init a serial exchange from session data
    rtu_query = RtuQuery()
    rtu_query.request.build(raw_pdu=session_data.request.pdu.raw,
                           slave_ad=session_data.request.mbap.unit_id)
    try:
        # add a request in the serial worker queue, can raise queue.Full
        self.rtu_queries_q.put(rtu_query, block=False)
        # wait result
        rtu_query.completed.wait()
        # check receive frame status
        if rtu_query.response.is_valid:
            session_data.response.pdu.raw = rtu_query.response.pdu
            return
        # except status for slave failed to respond
        exp_status = EXP_GATEWAY_TARGET_DEVICE_FAILED_TO_RESPOND
    except queue.Full:
        # except status for overloaded gateway
        exp_status = EXP_GATEWAY_PATH_UNAVAILABLE
    # return modbus exception
    func_code = rtu_query.request.function_code
    session_data.response.pdu.build_except(func_code=func_code, exp_status=exp_
    ↪status)

if __name__ == '__main__':
    # parse args
    parser = argparse.ArgumentParser()
    parser.add_argument('device', type=str, help='serial device (like /dev/ttyUSB0)')
    parser.add_argument('-H', '--host', type=str, default='localhost', help='host_
    ↪(default: localhost)')
    parser.add_argument('-p', '--port', type=int, default=502, help='TCP port (default:_
    ↪502)')
    parser.add_argument('-b', '--baudrate', type=int, default=9600, help='serial rate_
    ↪(default is 9600)')
    parser.add_argument('-t', '--timeout', type=float, default=1.0, help='timeout delay_
    ↪(default is 1.0 s)')
    parser.add_argument('-e', '--eof', type=float, default=0.05, help='end of frame_
    ↪delay (default is 0.05 s)')
    parser.add_argument('-d', '--debug', action='store_true', help='set debug mode')
    args = parser.parse_args()
    # init logging
    logging.basicConfig(level=logging.DEBUG if args.debug else None)
    logger = logging.getLogger(__name__)
    try:
        # init serial port

```

(continues on next page)

(continued from previous page)

```

logger.debug('Open serial port %s at %d bauds', args.device, args.baudrate)
serial_port = Serial(port=args.device, baudrate=args.baudrate)
# init serial worker
serial_worker = ModbusSerialWorker(serial_port, args.timeout, args.eof)
# start modbus server with custom engine
logger.debug('Start modbus server (%s, %d)', args.host, args.port)
srv = ModbusServer(host=args.host, port=args.port,
                   no_block=True, ext_engine=serial_worker.srv_engine_entry)

srv.start()
# start serial worker loop
logger.debug('Start serial worker')
serial_worker.loop()
except serialutil.SerialException as e:
    logger.critical('Serial device error: %r', e)
    exit(1)
except ModbusServer.Error as e:
    logger.critical('Modbus server error: %r', e)
    exit(2)

```

### 3.11 Server: schedule and alive word

```

#!/usr/bin/env python3

"""
Modbus/TCP server with start/stop schedule
~~~~~

Run this as root to listen on TCP privileged ports (<= 1024).

Default Modbus/TCP port is 502, so we prefix call with sudo. With argument
"--host 0.0.0.0", server listen on all IPv4 of the host. Instead of just
open tcp/502 on local interface.
$ sudo ./server_schedule.py --host 0.0.0.0
"""

import argparse
import time
from pyModbusTCP.server import ModbusServer
# need https://github.com/dbader/schedule
import schedule

def alive_word_job():
    """Update holding register @0 with day second (since 00:00).

    Job called every 10s by scheduler.
    """
    server.data_bank.set_holding_registers(0, [int(time.time()) % (24*3600) // 10])

```

(continues on next page)

(continued from previous page)

```

# parse args
parser = argparse.ArgumentParser()
parser.add_argument('-H', '--host', type=str, default='localhost', help='Host (default: ↵
↵localhost)')
parser.add_argument('-p', '--port', type=int, default=502, help='TCP port (default: 502)
↵')
args = parser.parse_args()
# init modbus server and start it
server = ModbusServer(host=args.host, port=args.port, no_block=True)
server.start()
# init scheduler
# schedule a daily downtime (from 18:00 to 06:00)
schedule.every().day.at('18:00').do(server.stop)
schedule.every().day.at('06:00').do(server.start)
# update life word at @0
schedule.every(10).seconds.do(alive_word_job)
# main loop
while True:
    schedule.run_pending()
    time.sleep(1)

```

## 3.12 Server: virtual data

```

#!/usr/bin/env python3

"""
Modbus/TCP server with virtual data
~~~~~

Map the system date and time to @ 0 to 5 on the "holding registers" space.
Only the reading of these registers in this address space is authorized. All
other requests return an illegal data address except.

Run this as root to listen on TCP privileged ports (<= 1024).
"""

import argparse
from pyModbusTCP.server import ModbusServer, DataBank
from datetime import datetime

class MyDataBank(DataBank):
    """A custom ModbusServerDataBank for override get_holding_registers method."""

    def __init__(self):
        # turn off allocation of memory for standard modbus object types
        # only "holding registers" space will be replaced by dynamic build values.
        super().__init__(virtual_mode=True)

    def get_holding_registers(self, address, number=1, srv_info=None):

```

(continues on next page)

(continued from previous page)

```
"""Get virtual holding registers."""
# populate virtual registers dict with current datetime values
now = datetime.now()
v_regs_d = {0: now.day, 1: now.month, 2: now.year,
            3: now.hour, 4: now.minute, 5: now.second}
# build a list of virtual regs to return to server data handler
# return None if any of virtual registers is missing
try:
    return [v_regs_d[a] for a in range(address, address+number)]
except KeyError:
    return

if __name__ == '__main__':
    # parse args
    parser = argparse.ArgumentParser()
    parser.add_argument('-H', '--host', type=str, default='localhost', help='Host_
↪(default: localhost)')
    parser.add_argument('-p', '--port', type=int, default=502, help='TCP port (default:_
↪502)')
    args = parser.parse_args()
    # init modbus server and start it
    server = ModbusServer(host=args.host, port=args.port, data_bank=MyDataBank())
    server.start()
```



## PYTHON MODULE INDEX

### p

`pyModbusTCP.client`, [7](#)  
`pyModbusTCP.server`, [12](#)  
`pyModbusTCP.utils`, [22](#)



## Symbols

`__init__()` (*pyModbusTCP.client.DeviceIdentificationResponse* method), 11  
`__init__()` (*pyModbusTCP.client.ModbusClient* method), 7  
`__init__()` (*pyModbusTCP.server.DataBank* method), 15  
`__init__()` (*pyModbusTCP.server.DataHandler* method), 13  
`__init__()` (*pyModbusTCP.server.DeviceIdentification* method), 18  
`__init__()` (*pyModbusTCP.server.ModbusServer* method), 12

## A

`auto_close` (*pyModbusTCP.client.ModbusClient* property), 7  
`auto_open` (*pyModbusTCP.client.ModbusClient* property), 7

## B

`byte_length()` (in module *pyModbusTCP.utils*), 18

## C

`close()` (*pyModbusTCP.client.ModbusClient* method), 8  
`crc16()` (in module *pyModbusTCP.utils*), 22  
`custom_request()` (*pyModbusTCP.client.ModbusClient* method), 8

## D

`DataBank` (class in *pyModbusTCP.server*), 15  
`DataHandler` (class in *pyModbusTCP.server*), 13  
`debug` (*pyModbusTCP.client.ModbusClient* property), 8  
`decode_ieee()` (in module *pyModbusTCP.utils*), 21  
`DeviceIdentification` (class in *pyModbusTCP.server*), 18  
`DeviceIdentificationResponse` (class in *pyModbusTCP.client*), 11

## E

`encode_ieee()` (in module *pyModbusTCP.utils*), 21

## G

`get_2comp()` (in module *pyModbusTCP.utils*), 21  
`get_bits_from_int()` (in module *pyModbusTCP.utils*), 19  
`get_coils()` (*pyModbusTCP.server.DataBank* method), 15  
`get_discrete_inputs()` (*pyModbusTCP.server.DataBank* method), 15  
`get_holding_registers()` (*pyModbusTCP.server.DataBank* method), 16  
`get_input_registers()` (*pyModbusTCP.server.DataBank* method), 16  
`get_list_2comp()` (in module *pyModbusTCP.utils*), 21

## H

`host` (*pyModbusTCP.client.ModbusClient* property), 8

## I

`is_open` (*pyModbusTCP.client.ModbusClient* property), 8  
`is_run` (*pyModbusTCP.server.ModbusServer* property), 13

## L

`last_error` (*pyModbusTCP.client.ModbusClient* property), 8  
`last_error_as_txt` (*pyModbusTCP.client.ModbusClient* property), 8  
`last_except` (*pyModbusTCP.client.ModbusClient* property), 8  
`last_except_as_full_txt` (*pyModbusTCP.client.ModbusClient* property), 8  
`last_except_as_txt` (*pyModbusTCP.client.ModbusClient* property), 8  
`long_list_to_word()` (in module *pyModbusTCP.utils*), 20

## M

`ModbusClient` (class in *pyModbusTCP.client*), 7  
`ModbusServer` (class in *pyModbusTCP.server*), 12  
`ModbusServer.ClientInfo` (class in *pyModbusTCP.server*), 12

ModbusServer.DataFormatError, 12  
 ModbusServer.Error, 12  
 ModbusServer.MBAP (class in pyModbusTCP.server), 12  
 ModbusServer.ModbusService (class in pyModbusTCP.server), 12  
 ModbusServer.NetworkError, 12  
 ModbusServer.PDU (class in pyModbusTCP.server), 12  
 ModbusServer.ServerInfo (class in pyModbusTCP.server), 13  
 ModbusServer.SessionData (class in pyModbusTCP.server), 13  
 module  
     pyModbusTCP.client, 7  
     pyModbusTCP.server, 12  
     pyModbusTCP.utils, 18, 20–22

## O

on\_coils\_change() (pyModbusTCP.server.DataBank method), 16  
 on\_holding\_registers\_change() (pyModbusTCP.server.DataBank method), 16  
 open() (pyModbusTCP.client.ModbusClient method), 8

## P

port (pyModbusTCP.client.ModbusClient property), 8  
 pyModbusTCP.client  
     module, 7  
 pyModbusTCP.server  
     module, 12  
 pyModbusTCP.utils  
     module, 18, 20–22

## R

read\_coils() (pyModbusTCP.client.ModbusClient method), 8  
 read\_coils() (pyModbusTCP.server.DataHandler method), 13  
 read\_d\_inputs() (pyModbusTCP.server.DataHandler method), 13  
 read\_device\_identification() (pyModbusTCP.client.ModbusClient method), 9  
 read\_discrete\_inputs() (pyModbusTCP.client.ModbusClient method), 9  
 read\_h\_regs() (pyModbusTCP.server.DataHandler method), 13  
 read\_holding\_registers() (pyModbusTCP.client.ModbusClient method), 9  
 read\_i\_regs() (pyModbusTCP.server.DataHandler method), 14  
 read\_input\_registers() (pyModbusTCP.client.ModbusClient method), 9  
 reset\_bit() (in module pyModbusTCP.utils), 19

## S

set\_bit() (in module pyModbusTCP.utils), 19  
 set\_coils() (pyModbusTCP.server.DataBank method), 17  
 set\_discrete\_inputs() (pyModbusTCP.server.DataBank method), 17  
 set\_holding\_registers() (pyModbusTCP.server.DataBank method), 17  
 set\_input\_registers() (pyModbusTCP.server.DataBank method), 17  
 start() (pyModbusTCP.server.ModbusServer method), 13  
 stop() (pyModbusTCP.server.ModbusServer method), 13

## T

test\_bit() (in module pyModbusTCP.utils), 19  
 timeout (pyModbusTCP.client.ModbusClient property), 10  
 toggle\_bit() (in module pyModbusTCP.utils), 19

## U

unit\_id (pyModbusTCP.client.ModbusClient property), 10

## V

valid\_host() (in module pyModbusTCP.utils), 22  
 version (pyModbusTCP.client.ModbusClient property), 10

## W

word\_list\_to\_long() (in module pyModbusTCP.utils), 20  
 write\_coils() (pyModbusTCP.server.DataHandler method), 14  
 write\_h\_regs() (pyModbusTCP.server.DataHandler method), 14  
 write\_multiple\_coils() (pyModbusTCP.client.ModbusClient method), 10  
 write\_multiple\_registers() (pyModbusTCP.client.ModbusClient method), 10  
 write\_read\_multiple\_registers() (pyModbusTCP.client.ModbusClient method), 10  
 write\_single\_coil() (pyModbusTCP.client.ModbusClient method), 11  
 write\_single\_register() (pyModbusTCP.client.ModbusClient method), 11