PyModbus

Release 3.6.6

Open Source volunteers

Mar 08, 2024

CONTENTS:

Please select a topic in the left hand column.

CHAPTER

ONE

PYMODBUS - A PYTHON MODBUS STACK

Pymodbus is a full Modbus protocol implementation offering client/server with synchronous/asynchronous API a well as simulators.

Current release is 3.6.6.

Bleeding edge (not released) is dev.

All changes are described in release notes and all API changes are documented

A big thanks to all the volunteers that helps make pymodbus a great project.

Source code on github

1.1 Pymodbus in a nutshell

Pymodbus consist of 5 parts:

- **client**, connect to your favorite device(s)
- **server**, simulate your favorite device(s)
- **repl**, a commandline text based client/server simulator
- **simulator**, an html based server simulator
- **examples**, showing both simple and advances usage

1.1.1 Common features

- Full modbus standard protocol implementation
- Support for custom function codes
- support serial (rs-485), tcp, tls and udp communication
- support all standard frames: socket, rtu, rtu-over-tcp, tcp and ascii
- does not have third party dependencies, apart from pyserial (optional)
- very lightweight project
- requires Python >= 3.8

- thorough test suite, that test all corners of the library
- automatically tested on Windows, Linux and MacOS combined with python 3.8 - 3.12
- strongly typed API (py.typed present)

The modbus protocol specification: Modbus_Application_Protocol_V1_1b3.pdf can be found on modbus org

1.1.2 Client Features

- asynchronous API and synchronous API for applications
- very simple setup and call sequence (just 6 lines of code)
- utilities to convert int/float to/from multiple registers
- payload builder/decoder to help with complex data

Client documentation

1.1.3 Server Features

- asynchronous implementation for high performance
- synchronous API classes for convenience
- simulate real life devices
- full server control context (device information, counters, etc)
- different backend datastores to manage register values
- callback to intercept requests/responses
- work on RS485 in parallel with other devices

Server documentation

1.1.4 REPL Features

- server/client commandline emulator
- easy test of real device (client)
- easy test of client app (server)
- simulation of broken requests/responses
- simulation of error responses (hard to provoke in real devices)

REPL documentation

1.1.5 Simulator Features

- server simulator with WEB interface
- configure the structure of a real device
- monitor traffic online
- allow distributed team members to work on a virtual device using internet
- simulation of broken requests/responses
- simulation of error responses (hard to provoke in real devices)

Simulator documentation

1.2 Use Cases

The client is the most typically used. It is embedded into applications, where it abstract the modbus protocol from the application by providing an easy to use API. The client is integrated into some well known projects like home-assistant.

Although most system administrators will find little need for a Modbus server, the server is handy to verify the functionality of an application.

The simulator and/or server is often used to simulate real life devices testing applications. The server is excellent to perform high volume testing (e.g. houndreds of devices connected to the application). The advantage of the server is that it runs not only a "normal" computers but also on small ones like Raspberry PI.

Since the library is written in python, it allows for easy scripting and/or integration into their existing solutions.

For more information please browse the project documentation:

https://readthedocs.org/docs/pymodbus/en/latest/index.html

1.3 Install

The library is available on pypi.org and github.com to install with

- `pip` for those who just want to use the library
- `git clone` for those who wants to help or just are curious

Be aware that there are a number of project, who have forked pymodbus and

- seems just to provide a version frozen in time
- extended pymodbus with extra functionality

The latter is not because we rejected the extra functionality (we welcome all changes), but because the codeowners made that decision.

In both cases, please understand, we cannot offer support to users of these projects as we do not known what have been changed nor what status the forked code have.

A growing number of Linux distributions include pymodbus in their standard installation.

You need to have python3 installed, preferable 3.11.

1.3.1 Install with pip

You can install using pip by issuing the following commands in a terminal window:

```
pip install pymodbus
```

If you want to use the serial interface:

```
pip install pymodbus[serial]
```

This will install pymodbus with the pyserial dependency.

Pymodbus offers a number of extra options:

- **repl**, needed by pymodbus.repl
- **serial**, needed for serial communication
- **simulator**, needed by pymodbus.simulator
- **documentation**, needed to generate documentation
- **development**, needed for development
- **all**, installs all of the above

which can be installed as:

```
pip install pymodbus[<option>,...]
```

It is possible to install old releases if needed:

```
pip install pymodbus==3.5.4
```

1.3.2 Install with github

On github, fork https://github.com/pymodbus-dev/pymodbus.git

Clone the source, and make a virtual environment:

```
git clone git://github.com/<your account>/pymodbus.git
cd pymodbus
python3 -m venv .venv
```

Activate the virtual environment, this command needs repeated in every new terminal:

```
source .venv/bin/activate
```

To get a specific release:

```
git checkout v3.5.2
```

or the bleeding edge:

```
git checkout dev
```

Some distributions have an old pip, which needs to be upgraded:

pip install –upgrade pip

Install required development tools:

```
pip install ".[development]"
```

Install all (allows creation of documentation etc):

pip install ".[all]"

Install git hooks, that helps control the commit and avoid errors when submitting a Pull Request:

cp githooks/* .git/hooks

This installs dependencies in your virtual environment with pointers directly to the pymodbus directory, so any change you make is immediately available as if installed.

The repository contains a number of important branches and tags.

- **dev** is where all development happens, this branch is not always stable.
- **master** is where are releases are kept.
- **vX.Y.Z** (e.g. v2.5.3) is a specific release

1.4 Example Code

For those of you that just want to get started fast, here you go:

```python
from pymodbus.client import ModbusTcpClient

client = ModbusTcpClient('MyDevice.lan')
client.connect()
client.write_coil(1, True)
result = client.read_coils(1,1)
print(result.bits[0])
client.close()
```

We provide a couple of simple ready to go clients:

- async client
- sync client

For more advanced examples, check out Examples included in the repository. If you have created any utilities that meet a specific need, feel free to submit them so others can benefit.

Also, if you have a question, please create a post in discussions q&a topic, so that others can benefit from the results.

If you think, that something in the code is broken/not running well, please open an issue, read the Template-text first and then post your issue with your setup information.

Example documentation

1.5 Contributing

Just fork the repo and raise your Pull Request against `dev` branch.

We always have more work than time, so feel free to open a discussion / issue on a theme you want to solve.

If your company would like your device tested or have a cloud based device simulation, feel free to contact us. We are happy to help your company solve your modbus challenges.

That said, the current work mainly involves polishing the library and solving issues:

- Fixing bugs/feature requests
- Architecture documentation
- Functional testing against any reference we can find

There are 2 bigger projects ongoing:

- rewriting the internal part of all clients (both sync and async)
- Add features to and simulator, and enhance the web design

1.6 Development instructions

The current code base is compatible with python >= 3.8.

Here are some of the common commands to perform a range of activities:

```
source .venv/bin/activate   <-- Activate the virtual environment
./check_ci.sh               <-- run the same checks as CI runs on a pull request.
```

Make a pull request:

```
git checkout dev            <-- activate development branch
git pull                    <-- update branch with newest changes
git checkout -b feature     <-- make new branch for pull request
... make source changes
git commit                  <-- commit change to git
git push                    <-- push to your account on github

on github open a pull request, check that CI turns green and then wait for review␣
↪comments.
```

Test your changes:

```
cd test
pytest
```

you can also do extended testing:

```
pytest --cov         <-- Coverage html report in build/html
pytest --profile     <-- Call profile report in prof
```

1.6.1 Internals

There are no documentation of the architecture (help is welcome), but most classes and methods are documented:

Pymodbus internals

1.6.2 Generate documentation

Remark Assumes that you have installed documentation tools:;

pip install ".[documentation]"

to build do:

```
cd doc
./build_html
```

The documentation is available in <root>/build/html

Remark: this generates a new zip/tgz file of examples which are uploaded.

1.7 License Information

Released under the BSD License

CHAPTER

TWO

CLIENT

Pymodbus offers both a `synchronous client` and a `asynchronous client`. Both clients offer simple calls for each type of request, as well as a unified response, removing a lot of the complexities in the modbus protocol.

In addition to the "pure" client, pymodbus offers a set of utilities converting to/from registers to/from "normal" python values.

The client is NOT thread safe, meaning the application must ensure that calls are serialized. This is only a problem for synchronous applications that use multiple threads or for asynchronous applications that use multiple `asyncio.create_task`.

It is allowed to have multiple client objects that e.g. each communicate with a TCP based device.

2.1 Client performance

There are currently a big performance gab between the 2 clients (try it on your computer examples/client_performance.py). This is due to a rather old implementation of the synchronous client, we are currently working to update the client code. Our aim is to achieve a similar data rate with both clients and at least double the data rate while keeping the stability. Table below is a test with 1000 calls each reading 10 registers.

client	asynchronous	synchronous
total time	0,33 sec	114,10 sec
ms/call	0,33 ms	114,10 ms
ms/register	0,03 ms	11,41 ms
calls/sec	3.030	8
registers/sec	30.300	87

2.2 Client protocols/framers

Pymodbus offers clients with transport different protocols and different framers

protocol	ASCII	RTU	RTU_OVER_TCP	Socket	TLS
Serial (RS-485)	Yes	Yes	No	No	No
TCP	Yes	No	Yes	Yes	No
TLS	No	No	No	No	Yes
UDP	Yes	No	Yes	Yes	No

2.2.1 Serial (RS-485)

Pymodbus do not connect to the device (server) but connects to a comm port or usb port on the local computer.

RS-485 is a half duplex protocol, meaning the servers do nothing until the client sends a request then the server being addressed responds. The client controls the traffic and as a consequence one RS-485 line can only have 1 client but upto 254 servers (physical devices).

RS-485 is a simple 2 wire cabling with a pullup resistor. It is important to note that many USB converters do not have a builtin resistor, this must be added manually. When experiencing many faulty packets and retries this is often the problem.

2.2.2 TCP

Pymodbus connects directly to the device using a standard socket and have a one-to-one connection with the device. In case of multiple TCP devices the application must instantiate multiple client objects one for each connection.

Tip: a TCP device often represent multiple physical devices (e.g Ethernet-RS485 converter), each of these devices can be addressed normally

2.2.3 TLS

A variant of **TCP** that uses encryption and certificates. **TLS** is mostly used when the devices are connected to the internet.

2.2.4 UDP

A broadcast variant of **TCP**. **UDP** allows addressing of many devices with a single request, however there are no control that a device have received the packet.

2.3 Client usage

Using pymodbus client to set/get information from a device (server) is done in a few simple steps, like the following synchronous example:

```
from pymodbus.client import ModbusTcpClient

client = ModbusTcpClient('MyDevice.lan')   # Create client object
client.connect()                           # connect to device, reconnect automatically
client.write_coil(1, True, slave=1)        # set information in device
result = client.read_coils(2, 3, slave=1)  # get information from device
print(result.bits[0])                      # use information
client.close()                             # Disconnect device
```

and a asynchronous example:

```
from pymodbus.client import ModbusAsyncTcpClient

client = ModbusAsyncTcpClient('MyDevice.lan')    # Create client object
```

(continues on next page)

(continued from previous page)

```
await client.connect()                          # connect to device, reconnect␣
↪automatically
await client.write_coil(1, True, slave=1)       # set information in device
result = await client.read_coils(2, 3, slave=1) # get information from device
print(result.bits[0])                           # use information
client.close()                                  # Disconnect device
```

The line `client = ModbusAsyncTcpClient('MyDevice.lan')` only creates the object it does not activate anything.

The line `await client.connect()` connects to the device (or comm port), if this cannot connect successfully within the timeout it throws an exception. If connected successfully reconnecting later is handled automatically

The line `await client.write_coil(1, True, slave=1)` is an example of a write request, set address 1 to True on device 1 (slave).

The line `result = await client.read_coils(2, 3, slave=1)` is an example of a read request, get the value of address 2, 3 and 4 (count = 3) from device 1 (slave).

The last line `client.close()` closes the connection and render the object inactive.

Large parts of the implementation are shared between the different classes, to ensure high stability and efficient maintenance.

The synchronous clients are not thread safe nor is a single client intended to be used from multiple threads. Due to the nature of the modbus protocol, it makes little sense to have client calls split over different threads, however the application can do it with proper locking implemented.

The asynchronous client only runs in the thread where the asyncio loop is created, it does not provide mechanisms to prevent (semi)parallel calls, that must be prevented at application level.

2.4 Client device addressing

With **TCP**, **TLS** and **UDP**, the tcp/ip address of the physical device is defined when creating the object. The logical devices represented by the device is addressed with the `slave=` parameter.

With **Serial**, the comm port is defined when creating the object. The physical devices are addressed with the `slave=` parameter.

`slave=0` is used as broadcast in order to address all devices. However experience shows that modern devices do not allow broadcast, mostly because it is inheriently dangerous. With `slave=0` the application can get upto 254 responses on a single request!

The simple request calls (mixin) do NOT support broadcast, if an application wants to use broadcast it must call `client.execute` and deal with the responses.

2.5 Client response handling

All simple request calls (mixin) return a unified result independent whether it´s a read, write or diagnostic call.

The application should evaluate the result generically:

```
try:
    rr = await client.read_coils(1, 1, slave=1)
except ModbusException as exc:
    _logger.error(f"ERROR: exception in pymodbus {exc}")
    raise exc
if rr.isError():
    _logger.error("ERROR: pymodbus returned an error!")
    raise ModbusException(txt)
```

`except ModbusException as exc:` happens generally when pymodbus experiences an internal error. There are a few situation where a unexpected response from a device can cause an exception.

`rr.isError()` is set whenever the device reports a problem.

And in case of read retrieve the data depending on type of request

- `rr.bits` is set for coils / input_register requests
- `rr.registers` is set for other requests

2.6 Client interface classes

There are a client class for each type of communication and for asynchronous/synchronous

Serial	`AsyncModbusSerialClient`	`ModbusSerialClient`
TCP	`AsyncModbusTcpClient`	`ModbusTcpClient`
TLS	`AsyncModbusTlsClient`	`ModbusTlsClient`
UDP	`AsyncModbusUdpClient`	`ModbusUdpClient`

2.6.1 Client serial

class `pymodbus.client.`**`AsyncModbusSerialClient`**(*port: str*, *framer: Framer = Framer.RTU*, *baudrate: int = 19200*, *bytesize: int = 8*, *parity: str = 'N'*, *stopbits: int = 1*, ***kwargs: Any*)

Bases: `ModbusBaseClient`, `Protocol`

AsyncModbusSerialClient.

Fixed parameters:

Parameters

`port` – Serial port used for communication.

Optional parameters:

Parameters

- **`baudrate`** – Bits per second.
- **`bytesize`** – Number of bits per byte 7-8.

- **parity** – 'E'ven, 'O'dd or 'N'one
- **stopbits** – Number of stop bits 1, 1.5, 2.
- **handle_local_echo** – Discard local echo from dongle.

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```python
from pymodbus.client import AsyncModbusSerialClient

async def run():
    client = AsyncModbusSerialClient("dev/serial0")

    await client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

`async connect()` → bool

Connect Async client.

`close`(*reconnect: bool = False*) → None

Close connection.

`class pymodbus.client.ModbusSerialClient`(*port: str, framer: Framer = Framer.RTU, baudrate: int = 19200, bytesize: int = 8, parity: str = 'N', stopbits: int = 1, strict: bool = True, **kwargs: Any*)

Bases: `ModbusBaseSyncClient`

ModbusSerialClient.

Fixed parameters:

Parameters

port – Serial port used for communication.

Optional parameters:

Parameters

- **baudrate** – Bits per second.
- **bytesize** – Number of bits per byte 7-8.
- **parity** – 'E'ven, 'O'dd or 'N'one
- **stopbits** – Number of stop bits 0-2.
- **handle_local_echo** – Discard local echo from dongle.

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.
- **strict** – Strict timing, 1.5 character between requests.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```
from pymodbus.client import ModbusSerialClient

def run():
    client = ModbusSerialClient("dev/serial0")

    client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

Remark: There are no automatic reconnect as with AsyncModbusSerialClient

property connected

Connect internal.

connect()

Connect to the modbus serial server.

close()

Close the underlying socket connection.

send(*request*)

Send data on the underlying socket.

If receive buffer still holds some data then flush it.

Sleep if last send finished less than 3.5 character times ago.

recv(*size*)

Read data from the underlying descriptor.

is_socket_open()

Check if socket is open.

2.6.2 Client TCP

class pymodbus.client.**AsyncModbusTcpClient**(*host: str*, *port: int = 502*, *framer: Framer = Framer.SOCKET*, *source_address: tuple[str, int] | None = None*, ***kwargs: Any*)

Bases: `ModbusBaseClient`, `Protocol`

AsyncModbusTcpClient.

Fixed parameters:

Parameters

host – Host IP address or host name

Optional parameters:

Parameters

- **port** – Port used for communication
- **source_address** – source address of client

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```python
from pymodbus.client import AsyncModbusTcpClient

async def run():
    client = AsyncModbusTcpClient("localhost")

    await client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

`async connect()` → bool

Initiate connection to start client.

`close`(*reconnect: bool = False*) → None

Close connection.

`class pymodbus.client.ModbusTcpClient`(*host: str*, *port: int = 502*, *framer:* Framer *= Framer.SOCKET*, *source_address: tuple[str, int] | None = None*, ***kwargs: Any*)

Bases: `ModbusBaseSyncClient`

ModbusTcpClient.

Fixed parameters:

Parameters
host – Host IP address or host name

Optional parameters:

Parameters

- **port** – Port used for communication
- **source_address** – source address of client

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```python
from pymodbus.client import ModbusTcpClient

async def run():
    client = ModbusTcpClient("localhost")

    client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

Remark: There are no automatic reconnect as with AsyncModbusTcpClient

property connected: bool
Connect internal.

connect()
Connect to the modbus tcp server.

close()
Close the underlying socket connection.

send(*request*)
Send data on the underlying socket.

recv(*size*)
Read data from the underlying descriptor.

is_socket_open()
Check if socket is open.

2.6.3 Client TLS

class pymodbus.client.**AsyncModbusTlsClient**(*host: str*, *port: int = 802*, *framer: Framer = Framer.TLS*, *sslctx: SSLContext | None = None*, *certfile: str | None = None*, *keyfile: str | None = None*, *password: str | None = None*, *server_hostname: str | None = None*, ***kwargs: Any*)

Bases: *AsyncModbusTcpClient*

AsyncModbusTlsClient.

Fixed parameters:

Parameters
host – Host IP address or host name

Optional parameters:

Parameters

- **port** – Port used for communication
- **source_address** – Source address of client
- **sslctx** – SSLContext to use for TLS
- **certfile** – Cert file path for TLS server request
- **keyfile** – Key file path for TLS server request
- **password** – Password for for decrypting private key file
- **server_hostname** – Bind certificate to host

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.

- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```python
from pymodbus.client import AsyncModbusTlsClient

async def run():
    client = AsyncModbusTlsClient("localhost")

    await client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

async connect() → bool

Initiate connection to start client.

class pymodbus.client.**ModbusTlsClient**(*host: str*, *port: int = 802*, *framer: Framer = Framer.TLS*, *sslctx: SSLContext | None = None*, *certfile: str | None = None*, *keyfile: str | None = None*, *password: str | None = None*, *server_hostname: str | None = None*, ***kwargs: Any*)

Bases: `ModbusTcpClient`

ModbusTlsClient.

Fixed parameters:

Parameters
host – Host IP address or host name

Optional parameters:

Parameters

- **port** – Port used for communication
- **source_address** – Source address of client
- **sslctx** – SSLContext to use for TLS
- **certfile** – Cert file path for TLS server request
- **keyfile** – Key file path for TLS server request
- **password** – Password for decrypting private key file
- **server_hostname** – Bind certificate to host
- **kwargs** – Experimental parameters

Common optional parameters:

Parameters

- **framer** – Framer enum name

- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```
from pymodbus.client import ModbusTlsClient

async def run():
    client = ModbusTlsClient("localhost")

    client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

Remark: There are no automatic reconnect as with AsyncModbusTlsClient

property connected: bool

Connect internal.

connect()

Connect to the modbus tls server.

2.6.4 Client UDP

class pymodbus.client.**AsyncModbusUdpClient**(*host: str*, *port: int = 502*, *framer: Framer = Framer.SOCKET*, *source_address: tuple[str, int] | None = None*, ***kwargs: Any*)

Bases: ModbusBaseClient, Protocol, DatagramProtocol

AsyncModbusUdpClient.

Fixed parameters:

Parameters

host – Host IP address or host name

Optional parameters:

Parameters

- **port** – Port used for communication.
- **source_address** – source address of client,

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.
- **retry_on_empty** – Retry on empty response.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```
from pymodbus.client import AsyncModbusUdpClient

async def run():
    client = AsyncModbusUdpClient("localhost")

    await client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

property connected

Return true if connected.

class pymodbus.client.**ModbusUdpClient**(*host: str*, *port: int = 502*, *framer: Framer = Framer.SOCKET*, *source_address: tuple[str, int] | None = None*, ***kwargs: Any*)

Bases: ModbusBaseSyncClient

ModbusUdpClient.

Fixed parameters:

Parameters

host – Host IP address or host name

Optional parameters:

Parameters

- **port** – Port used for communication.
- **source_address** – source address of client,

Common optional parameters:

Parameters

- **framer** – Framer enum name
- **timeout** – Timeout for a request, in seconds.
- **retries** – Max number of retries per request.

- **retry_on_empty** – Retry on empty response.
- **broadcast_enable** – True to treat id 0 as broadcast address.
- **reconnect_delay** – Minimum delay in seconds.milliseconds before reconnecting.
- **reconnect_delay_max** – Maximum delay in seconds.milliseconds before reconnecting.
- **on_reconnect_callback** – Function that will be called just before a reconnection attempt.
- **no_resend_on_retry** – Do not resend request when retrying due to missing response.
- **kwargs** – Experimental parameters.

Example:

```
from pymodbus.client import ModbusUdpClient

async def run():
    client = ModbusUdpClient("localhost")

    client.connect()
    ...
    client.close()
```

Please refer to *Pymodbus internals* for advanced usage.

Remark: There are no automatic reconnect as with AsyncModbusUdpClient

property connected: bool

Connect internal.

2.7 Modbus calls

Pymodbus makes all standard modbus requests/responses available as simple calls.

Using Modbus<transport>Client.register() custom messagees can be added to pymodbus, and handled automatically.

class pymodbus.client.mixin.**ModbusClientMixin**

Bases: Generic[T]

ModbusClientMixin.

This is an interface class to facilitate the sending requests/receiving responses like read_coils. execute() allows to make a call with non-standard or user defined function codes (remember to add a PDU in the transport class to interpret the request/response).

Simple modbus message call:

```
response = client.read_coils(1, 10)
# or
response = await client.read_coils(1, 10)
```

Advanced modbus message call:

```
request = ReadCoilsRequest(1,10)
response = client.execute(request)
# or
```

(continues on next page)

(continued from previous page)

```
request = ReadCoilsRequest(1,10)
response = await client.execute(request)
```

Tip: All methods can be used directly (synchronous) or with await <method> (asynchronous) depending on the client used.

`execute`(*_request: ModbusRequest*) → T

Execute request (code ???).

Raises
`ModbusException` –

Call with custom function codes.

Tip: Response is not interpreted.

`read_coils`(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: Any*) → T

Read coils (code 0x01).

Parameters

- **`address`** – Start address to read from
- **`count`** – (optional) Number of coils to read
- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises
`ModbusException` –

`read_discrete_inputs`(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: Any*) → T

Read discrete inputs (code 0x02).

Parameters

- **`address`** – Start address to read from
- **`count`** – (optional) Number of coils to read
- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises
`ModbusException` –

`read_holding_registers`(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: Any*) → T

Read holding registers (code 0x03).

Parameters

- **`address`** – Start address to read from
- **`count`** – (optional) Number of coils to read
- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises
ModbusException –

read_input_registers(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: Any*) → T

Read input registers (code 0x04).

Parameters

- **address** – Start address to read from
- **count** – (optional) Number of coils to read
- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

write_coil(*address: int*, *value: bool*, *slave: int = 0*, ***kwargs: Any*) → T

Write single coil (code 0x05).

Parameters

- **address** – Address to write to
- **value** – Boolean to write
- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

write_register(*address: int*, *value: int*, *slave: int = 0*, ***kwargs: Any*) → T

Write register (code 0x06).

Parameters

- **address** – Address to write to
- **value** – Value to write
- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

read_exception_status(*slave: int = 0*, ***kwargs: Any*) → T

Read Exception Status (code 0x07).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

`diag_query_data`(*msg: bytes*, *slave: int = 0*, ***kwargs: Any*) → T

Diagnose query data (code 0x08 sub 0x00).

Parameters

- **`msg`** – Message to be returned
- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_restart_communication`(*toggle: bool*, *slave: int = 0*, ***kwargs: Any*) → T

Diagnose restart communication (code 0x08 sub 0x01).

Parameters

- **`toggle`** – True if toggled.
- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_read_diagnostic_register`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose read diagnostic register (code 0x08 sub 0x02).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_change_ascii_input_delimeter`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose change ASCII input delimiter (code 0x08 sub 0x03).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_force_listen_only`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose force listen only (code 0x08 sub 0x04).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_clear_counters`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose clear counters (code 0x08 sub 0x0A).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_read_bus_message_count`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose read bus message count (code 0x08 sub 0x0B).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_read_bus_comm_error_count`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose read Bus Communication Error Count (code 0x08 sub 0x0C).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_read_bus_exception_error_count`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose read Bus Exception Error Count (code 0x08 sub 0x0D).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_read_slave_message_count`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose read Slave Message Count (code 0x08 sub 0x0E).

Parameters

- **`slave`** – (optional) Modbus slave ID
- **`kwargs`** – (optional) Experimental parameters.

Raises

`ModbusException` –

`diag_read_slave_no_response_count`(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose read Slave No Response Count (code 0x08 sub 0x0F).

Parameters

- **`slave`** – (optional) Modbus slave ID

- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

diag_read_slave_nak_count(*slave: int = 0*, ***kwargs: Any*) → T
Diagnose read Slave NAK Count (code 0x08 sub 0x10).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

diag_read_slave_busy_count(*slave: int = 0*, ***kwargs: Any*) → T
Diagnose read Slave Busy Count (code 0x08 sub 0x11).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

diag_read_bus_char_overrun_count(*slave: int = 0*, ***kwargs: Any*) → T
Diagnose read Bus Character Overrun Count (code 0x08 sub 0x12).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

diag_read_iop_overrun_count(*slave: int = 0*, ***kwargs: Any*) → T
Diagnose read Iop overrun count (code 0x08 sub 0x13).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

diag_clear_overrun_counter(*slave: int = 0*, ***kwargs: Any*) → T
Diagnose Clear Overrun Counter and Flag (code 0x08 sub 0x14).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises
ModbusException –

diag_getclear_modbus_response(*slave: int = 0*, ***kwargs: Any*) → T

Diagnose Get/Clear modbus plus (code 0x08 sub 0x15).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

diag_get_comm_event_counter(***kwargs: Any*) → T

Diagnose get event counter (code 0x0B).

Parameters

kwargs – (optional) Experimental parameters.

Raises

ModbusException –

diag_get_comm_event_log(***kwargs: Any*) → T

Diagnose get event counter (code 0x0C).

Parameters

kwargs – (optional) Experimental parameters.

Raises

ModbusException –

write_coils(*address: int*, *values: list[bool] | bool*, *slave: int = 0*, ***kwargs: Any*) → T

Write coils (code 0x0F).

Parameters

- **address** – Start address to write to
- **values** – List of booleans to write, or a single boolean to write
- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

write_registers(*address: int*, *values: list[int] | int*, *slave: int = 0*, ***kwargs: Any*) → T

Write registers (code 0x10).

Parameters

- **address** – Start address to write to
- **values** – List of values to write, or a single value to write
- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

report_slave_id(*slave: int = 0*, ***kwargs: Any*) → T

Report slave ID (code 0x11).

Parameters

- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

read_file_record(*records: list[tuple]*, ***kwargs: Any*) → T

Read file record (code 0x14).

Parameters

- **records** – List of (Reference type, File number, Record Number, Record Length)
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

write_file_record(*records: list[tuple]*, ***kwargs: Any*) → T

Write file record (code 0x15).

Parameters

- **records** – List of (Reference type, File number, Record Number, Record Length)
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

mask_write_register(*address: int = 0*, *and_mask: int = 65535*, *or_mask: int = 0*, ***kwargs: Any*) → T

Mask write register (code 0x16).

Parameters

- **address** – The mask pointer address (0x0000 to 0xffff)
- **and_mask** – The and bitmask to apply to the register address
- **or_mask** – The or bitmask to apply to the register address
- **kwargs** – (optional) Experimental parameters.

Raises

ModbusException –

readwrite_registers(*read_address: int = 0*, *read_count: int = 0*, *write_address: int = 0*, *values: list[int] | int = 0*, *slave: int = 0*, ***kwargs*) → T

Read/Write registers (code 0x17).

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **values** – List of values to write, or a single value to write
- **slave** – (optional) Modbus slave ID

- **kwargs** –

Raises
ModbusException –

read_fifo_queue(*address: int = 0*, ***kwargs: Any*) → T

Read FIFO queue (code 0x18).

Parameters

- **address** – The address to start reading from
- **kwargs** –

Raises
ModbusException –

read_device_information(*read_code: int | None = None*, *object_id: int = 0*, ***kwargs: Any*) → T

Read FIFO queue (code 0x2B sub 0x0E).

Parameters

- **read_code** – The device information read code
- **object_id** – The object to read from
- **kwargs** –

Raises
ModbusException –

class **DATATYPE**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `Enum`

Datatype enum (name and number of bytes), used for convert_* calls.

classmethod **convert_from_registers**(*registers: list[int]*, *data_type:* DATATYPE) → int | float | str

Convert registers to int/float/str.

Parameters

- **registers** – list of registers received from e.g. read_holding_registers()
- **data_type** – data type to convert to

Returns
int, float or str depending on "data_type"

Raises
ModbusException – when size of registers is not 1, 2 or 4

classmethod **convert_to_registers**(*value: int | float | str*, *data_type:* DATATYPE) → list[int]

Convert int/float/str to registers (16/32/64 bit).

Parameters

- **value** – value to be converted
- **data_type** – data type to be encoded as registers

Returns
List of registers, can be used directly in e.g. write_registers()

Raises
TypeError – when there is a mismatch between data_type and value

CHAPTER

THREE

SERVER

Pymodbus offers servers with transport protocols for

- *Serial* (RS-485) typically using a dongle
- *TCP*
- *TLS*
- *UDP*
- possibility to add a custom transport protocol

communication in 2 versions:

- `synchronous server`,
- `asynchronous server` using asyncio.

Remark All servers are implemented with asyncio, and the synchronous servers are just an interface layer allowing synchronous applications to use the server as if it was synchronous. Server.

import external classes, to make them easier to use:

`class pymodbus.server.`**`ModbusSerialServer`**(*context*, *framer=Framer.RTU*, *identity=None*, ***kwargs*)

Bases: `ModbusBaseServer`

A modbus threaded serial socket server.

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

`class pymodbus.server.`**`ModbusSimulatorServer`**(*modbus_server: str = 'server'*, *modbus_device: str = 'device'*, *http_host: str = '0.0.0.0'*, *http_port: int = 8080*, *log_file: str = 'server.log'*, *json_file: str = 'setup.json'*, *custom_actions_module: str | None = None*)

Bases: `object`

ModbusSimulatorServer.

Parameters

- **`modbus_server`** – Server name in json file (default: "server")
- **`modbus_device`** – Device name in json file (default: "client")
- **`http_host`** – TCP host for HTTP (default: "localhost")
- **`http_port`** – TCP port for HTTP (default: 8080)
- **`json_file`** – setup file (default: "setup.json")

- `custom_actions_module` – python module with custom actions (default: none)

if either http_port or http_host is none, HTTP will not be started. This class starts a http server, that serves a couple of endpoints:

- **"<addr>/"** static files
- **"<addr>/api/log"** log handling, HTML with GET, REST-API with post
- **"<addr>/api/registers"** register handling, HTML with GET, REST-API with post
- **"<addr>/api/calls"** call (function code / message) handling, HTML with GET, REST-API with post
- **"<addr>/api/server"** server handling, HTML with GET, REST-API with post

Example:

```python
from pymodbus.server import ModbusSimulatorServer

async def run():
    simulator = ModbusSimulatorServer(
        modbus_server="my server",
        modbus_device="my device",
        http_host="localhost",
        http_port=8080)
    await simulator.run_forever(only_start=True)
    ...
    await simulator.stop()
```

`action_add`(*params*, *range_start*, *range_stop*)

Build list of registers matching filter.

`action_clear`(*_params*, *_range_start*, *_range_stop*)

Clear register filter.

`action_monitor`(*params*, *range_start*, *range_stop*)

Start monitoring calls.

`action_reset`(*_params*, *_range_start*, *_range_stop*)

Reset call simulation.

`action_set`(*params*, *_range_start*, *_range_stop*)

Set register value.

`action_simulate`(*params*, *_range_start*, *_range_stop*)

Simulate responses.

`action_stop`(*_params*, *_range_start*, *_range_stop*)

Stop call monitoring.

`build_html_calls`(*params: dict*, *html: str*) → str

Build html calls page.

`build_html_log`(*_params*, *html*)

Build html log page.

`build_html_registers`(*params*, *html*)

Build html registers page.

build_html_server(*_params*, *html*)

Build html server page.

build_json_calls(*params*, *json_dict*)

Build html calls page.

build_json_log(*params*, *json_dict*)

Build json log page.

build_json_registers(*params*, *json_dict*)

Build html registers page.

build_json_server(*params*, *json_dict*)

Build html server page.

async **handle_html**(*request*)

Handle html.

async **handle_html_static**(*request*)

Handle static html.

async **handle_json**(*request*)

Handle api registers.

helper_build_html_submit(*params*)

Build html register submit.

async **run_forever**(*only_start=False*)

Start modbus and http servers.

server_request_tracer(*request*, **_addr*)

Trace requests.

All server requests passes this filter before being handled.

server_response_manipulator(*response*)

Manipulate responses.

All server responses passes this filter before being sent. The filter returns:

- response, either original or modified
- skip_encoding, signals whether or not to encode the response

async **start_modbus_server**(*app*)

Start Modbus server as asyncio task.

async **stop**()

Stop modbus and http servers.

async **stop_modbus_server**(*app*)

Stop modbus server.

class pymodbus.server.**ModbusTcpServer**(*context*, *framer=Framer.SOCKET*, *identity=None*, *address=('', 502)*, *ignore_missing_slaves=False*, *broadcast_enable=False*, *response_manipulator=None*, *request_tracer=None*)

Bases: `ModbusBaseServer`

A modbus threaded tcp socket server.

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

class `pymodbus.server.`**`ModbusTlsServer`**(*context*, *framer=Framer.TLS*, *identity=None*, *address=('', 502)*, *sslctx=None*, *certfile=None*, *keyfile=None*, *password=None*, *ignore_missing_slaves=False*, *broadcast_enable=False*, *response_manipulator=None*, *request_tracer=None*)

Bases: `ModbusTcpServer`

A modbus threaded tls socket server.

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

class `pymodbus.server.`**`ModbusUdpServer`**(*context*, *framer=Framer.SOCKET*, *identity=None*, *address=('', 502)*, *ignore_missing_slaves=False*, *broadcast_enable=False*, *response_manipulator=None*, *request_tracer=None*)

Bases: `ModbusBaseServer`

A modbus threaded udp socket server.

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

async `pymodbus.server.`**`ServerAsyncStop`**()

Terminate server.

`pymodbus.server.`**`ServerStop`**()

Terminate server.

async `pymodbus.server.`**`StartAsyncSerialServer`**(*context=None*, *identity=None*, *custom_functions=[]*, ***kwargs*)

Start and run a serial modbus server.

Parameters

- **`context`** – The ModbusServerContext datastore
- **`identity`** – An optional identify structure
- **`custom_functions`** – An optional list of custom function classes supported by server instance.
- **`kwargs`** – The rest

async `pymodbus.server.`**`StartAsyncTcpServer`**(*context=None*, *identity=None*, *address=None*, *custom_functions=[]*, ***kwargs*)

Start and run a tcp modbus server.

Parameters

- **`context`** – The ModbusServerContext datastore
- **`identity`** – An optional identify structure
- **`address`** – An optional (interface, port) to bind to.
- **`custom_functions`** – An optional list of custom function classes supported by server instance.
- **`kwargs`** – The rest

`async pymodbus.server.StartAsyncTlsServer`(*context=None*, *identity=None*, *address=None*, *sslctx=None*, *certfile=None*, *keyfile=None*, *password=None*, *custom_functions=[]*, ***kwargs*)

Start and run a tls modbus server.

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **sslctx** – The SSLContext to use for TLS (default None and auto create)
- **certfile** – The cert file path for TLS (used if sslctx is None)
- **keyfile** – The key file path for TLS (used if sslctx is None)
- **password** – The password for for decrypting the private key file
- **custom_functions** – An optional list of custom function classes supported by server instance.
- **kwargs** – The rest

`async pymodbus.server.StartAsyncUdpServer`(*context=None*, *identity=None*, *address=None*, *custom_functions=[]*, ***kwargs*)

Start and run a udp modbus server.

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **custom_functions** – An optional list of custom function classes supported by server instance.
- **kwargs** –

`pymodbus.server.StartSerialServer`(***kwargs*)

Start and run a serial modbus server.

`pymodbus.server.StartTcpServer`(***kwargs*)

Start and run a serial modbus server.

`pymodbus.server.StartTlsServer`(***kwargs*)

Start and run a serial modbus server.

`pymodbus.server.StartUdpServer`(***kwargs*)

Start and run a serial modbus server.

`pymodbus.server.get_simulator_commandline`(*extras=None*, *cmdline=None*)

Get command line arguments.

CHAPTER

FOUR

PYMODBUS REPL (READ EVALUATE PRINT LOOP)

4.1 Installation

Project repo pymodbus-repl

4.1.1 Install as pymodbus optional dependency

```
$ pip install ".[repl]"
```

4.1.2 Install directly from the github repo

```
$ pip install "git+https://github.com/pymodbus-dev/repl"
```

4.2 Usage Instructions

RTU and TCP are supported as of now

```
bash-3.2$ pymodbus.console
Usage: pymodbus.console [OPTIONS] COMMAND [ARGS]...

Options:
  --version       Show the version and exit.
  --verbose       Verbose logs
  --support-diag  Support Diagnostic messages
  --help          Show this message and exit.

Commands:
  serial
  tcp
```

TCP Options

```
bash-3.2$ pymodbus.console tcp --help
Usage: pymodbus.console tcp [OPTIONS]

Options:
```

(continues on next page)

(continued from previous page)

```
  --host TEXT     Modbus TCP IP
  --port INTEGER  Modbus TCP port
  --help          Show this message and exit.
```

SERIAL Options

```
bash-3.2$ pymodbus.console serial --help
Usage: pymodbus.console serial [OPTIONS]

Options:
  --method TEXT          Modbus Serial Mode (rtu/ascii)
  --port TEXT            Modbus RTU port
  --baudrate INTEGER     Modbus RTU serial baudrate to use.
  --bytesize [5|6|7|8]   Modbus RTU serial Number of data bits. Possible
                         values: FIVEBITS, SIXBITS, SEVENBITS, EIGHTBITS.
  --parity [N|E|O|M|S]   Modbus RTU serial parity.  Enable parity checking.
                         Possible values: PARITY_NONE, PARITY_EVEN, PARITY_ODD
                         PARITY_MARK, PARITY_SPACE. Default to 'N'
  --stopbits [1|1.5|2]   Modbus RTU serial stop bits. Number of stop bits.
                         Possible values: STOPBITS_ONE,
                         STOPBITS_ONE_POINT_FIVE, STOPBITS_TWO. Default to '1'
  --xonxoff INTEGER      Modbus RTU serial xonxoff.  Enable software flow
                         control.
  --rtscts INTEGER       Modbus RTU serial rtscts. Enable hardware (RTS/CTS)
                         flow control.
  --dsrdtr INTEGER       Modbus RTU serial dsrdtr. Enable hardware (DSR/DTR)
                         flow control.
  --timeout FLOAT        Modbus RTU serial read timeout.
  --write-timeout FLOAT  Modbus RTU serial write timeout.
  --help                 Show this message and exit.
```

To view all available commands type `help`

TCP

```
$ pymodbus.console tcp --host 192.168.128.126 --port 5020

> help
Available commands:
client.change_ascii_input_delimiter          Diagnostic sub command, Change message␣
↪delimiter for future requests.
client.clear_counters                        Diagnostic sub command, Clear all counters␣
↪and diag registers.
client.clear_overrun_count                   Diagnostic sub command, Clear over run␣
↪counter.
client.close                                 Closes the underlying socket connection
client.connect                               Connect to the modbus tcp server
client.debug_enabled                         Returns a boolean indicating if debug is␣
↪enabled.
client.force_listen_only_mode                Diagnostic sub command, Forces the␣
↪addressed remote device to         its Listen Only Mode.
client.get_clear_modbus_plus                 Diagnostic sub command, Get or clear stats␣
↪of remote          modbus plus device.
```

(continues on next page)

(continued from previous page)

```
client.get_com_event_counter                 Read  status word and an event count from␣
↪the remote device's         communication event counter.
client.get_com_event_log                     Read  status word, event count, message␣
↪count, and a field of event bytes from the remote device.
client.host                                  Read Only!
client.idle_time                             Bus Idle Time to initiate next transaction
client.is_socket_open                        Check whether the underlying socket/serial␣
↪is open or not.
client.last_frame_end                        Read Only!
client.mask_write_register                   Mask content of holding register at␣
↪`address`          with `and_mask` and `or_mask`.
client.port                                  Read Only!
client.read_coils                            Reads `count` coils from a given slave␣
↪starting at `address`.
client.read_device_information               Read the identification and additional␣
↪information of remote slave.
client.read_discrete_inputs                  Reads `count` number of discrete inputs␣
↪starting at offset `address`.
client.read_exception_status                 Read the contents of eight Exception Status␣
↪outputs in a remote          device.
client.read_holding_registers                Read `count` number of holding registers␣
↪starting at `address`.
client.read_input_registers                  Read `count` number of input registers␣
↪starting at `address`.
client.readwrite_registers                   Read `read_count` number of holding␣
↪registers starting at         `read_address`  and write `write_registers`        ␣
↪starting at `write_address`.
client.report_slave_id                       Report information about remote slave ID.
client.restart_comm_option                   Diagnostic sub command, initialize and␣
↪restart remote devices serial         interface and clear all of its communications␣
↪event counters .
client.return_bus_com_error_count            Diagnostic sub command, Return count of CRC␣
↪errors         received by remote slave.
client.return_bus_exception_error_count      Diagnostic sub command, Return count of␣
↪Modbus exceptions         returned by remote slave.
client.return_bus_message_count              Diagnostic sub command, Return count of␣
↪message detected on bus          by remote slave.
client.return_diagnostic_register            Diagnostic sub command, Read 16-bit␣
↪diagnostic register.
client.return_iop_overrun_count              Diagnostic sub command, Return count of iop␣
↪overrun errors          by remote slave.
client.return_query_data                     Diagnostic sub command , Loop back data␣
↪sent in response.
client.return_slave_bus_char_overrun_count   Diagnostic sub command, Return count of␣
↪messages not handled          by remote slave due to character overrun condition.
client.return_slave_busy_count               Diagnostic sub command, Return count of␣
↪server busy exceptions sent          by remote slave.
client.return_slave_message_count            Diagnostic sub command, Return count of␣
↪messages addressed to          remote slave.
client.return_slave_no_ack_count             Diagnostic sub command, Return count of NO␣
↪ACK exceptions sent          by remote slave.
client.return_slave_no_response_count        Diagnostic sub command, Return count of No␣
```

(continues on next page)

(continued from previous page)

```
↪responses  by remote slave.
client.silent_interval                        Read Only!
client.state                                  Read Only!
client.timeout                                Read Only!
client.write_coil                             Write `value` to coil at `address`.
client.write_coils                            Write `value` to coil at `address`.
client.write_register                         Write `value` to register at `address`.
client.write_registers                        Write list of `values` to registers␣
↪starting at `address`.
```

SERIAL

```
$ pymodbus.console serial --port /dev/ttyUSB0 --baudrate 19200 --timeout 2
> help
Available commands:
client.baudrate                               Read Only!
client.bytesize                               Read Only!
client.change_ascii_input_delimiter           Diagnostic sub command, Change message␣
↪delimiter for future requests.
client.clear_counters                         Diagnostic sub command, Clear all counters␣
↪and diag registers.
client.clear_overrun_count                    Diagnostic sub command, Clear over run␣
↪counter.
client.close                                  Closes the underlying socket connection
client.connect                                Connect to the modbus serial server
client.debug_enabled                          Returns a boolean indicating if debug is␣
↪enabled.
client.force_listen_only_mode                 Diagnostic sub command, Forces the␣
↪addressed remote device to          its Listen Only Mode.
client.get_baudrate                           Serial Port baudrate.
client.get_bytesize                           Number of data bits.
client.get_clear_modbus_plus                  Diagnostic sub command, Get or clear stats␣
↪of remote          modbus plus device.
client.get_com_event_counter                  Read  status word and an event count from␣
↪the remote device's        communication event counter.
client.get_com_event_log                      Read  status word, event count, message␣
↪count, and a field of event bytes from the remote device.
client.get_parity                             Enable Parity Checking.
client.get_port                               Serial Port.
client.get_serial_settings                    Gets Current Serial port settings.
client.get_stopbits                           Number of stop bits.
client.get_timeout                            Serial Port Read timeout.
client.idle_time                              Bus Idle Time to initiate next transaction
client.inter_byte_timeout                     Read Only!
client.is_socket_open                         c l i e n t . i s   s o c k e t   o p e n
client.mask_write_register                    Mask content of holding register at␣
↪`address`          with `and_mask` and `or_mask`.
client.method                                 Read Only!
client.parity                                 Read Only!
client.port                                   Read Only!
client.read_coils                             Reads `count` coils from a given slave␣
↪starting at `address`.
```

(continues on next page)

(continued from previous page)

```
client.read_device_information             Read the identification and additional␣
↪information of remote slave.
client.read_discrete_inputs                Reads `count` number of discrete inputs␣
↪starting at offset `address`.
client.read_exception_status               Read the contents of eight Exception Status␣
↪outputs in a remote          device.
client.read_holding_registers              Read `count` number of holding registers␣
↪starting at `address`.
client.read_input_registers                Read `count` number of input registers␣
↪starting at `address`.
client.readwrite_registers                 Read `read_count` number of holding␣
↪registers starting at         `read_address`  and write `write_registers`        ␣
↪starting at `write_address`.
client.report_slave_id                     Report information about remote slave ID.
client.restart_comm_option                 Diagnostic sub command, initialize and␣
↪restart remote devices serial        interface and clear all of its communications␣
↪event counters .
client.return_bus_com_error_count          Diagnostic sub command, Return count of CRC␣
↪errors         received by remote slave.
client.return_bus_exception_error_count    Diagnostic sub command, Return count of␣
↪Modbus exceptions         returned by remote slave.
client.return_bus_message_count            Diagnostic sub command, Return count of␣
↪message detected on bus          by remote slave.
client.return_diagnostic_register          Diagnostic sub command, Read 16-bit␣
↪diagnostic register.
client.return_iop_overrun_count            Diagnostic sub command, Return count of iop␣
↪overrun errors         by remote slave.
client.return_query_data                   Diagnostic sub command , Loop back data␣
↪sent in response.
client.return_slave_bus_char_overrun_count   Diagnostic sub command, Return count of␣
↪messages not handled          by remote slave due to character overrun condition.
client.return_slave_busy_count             Diagnostic sub command, Return count of␣
↪server busy exceptions sent          by remote slave.
client.return_slave_message_count          Diagnostic sub command, Return count of␣
↪messages addressed to         remote slave.
client.return_slave_no_ack_count           Diagnostic sub command, Return count of NO␣
↪ACK exceptions sent          by remote slave.
client.return_slave_no_response_count      Diagnostic sub command, Return count of No␣
↪responses  by remote slave.
client.set_baudrate                        Baudrate setter.
client.set_bytesize                        Byte size setter.
client.set_parity                          Parity Setter.
client.set_port                            Serial Port setter.
client.set_stopbits                        Stop bit setter.
client.set_timeout                         Read timeout setter.
client.silent_interval                     Read Only!
client.state                               Read Only!
client.stopbits                            Read Only!
client.timeout                             Read Only!
client.write_coil                          Write `value` to coil at `address`.
client.write_coils                         Write `value` to coil at `address`.
client.write_register                      Write `value` to register at `address`.
```

(continues on next page)

(continued from previous page)

```
client.write_registers                       Write list of `values` to registers␣
↪starting at `address`.
result.decode                                Decode the register response to known␣
↪formatters.
result.raw                                   Return raw result dict.
```

Every command has auto suggestion on the arguments supported, arg and value are to be supplied in `arg=val` format.

```
> client.read_holding_registers count=4 address=9 slave=1
{
    "registers": [
        60497,
        47134,
        34091,
        15424
    ]
}
```

The last result could be accessed with `result.raw` command

```
> result.raw
{
    "registers": [
        15626,
        55203,
        28733,
        18368
    ]
}
```

For Holding and Input register reads, the decoded value could be viewed with `result.decode`

```
> result.decode word_order=little byte_order=little formatters=float64
28.17

>
```

Client settings could be retrieved and altered as well.

```
> # For serial settings

> # Check the serial mode
> client.method
"rtu"

> client.get_serial_settings
{
    "t1.5": 0.00171875,
    "baudrate": 9600,
    "read timeout": 0.5,
    "port": "/dev/ptyp0",
    "t3.5": 0.00401,
    "bytesize": 8,
```

(continues on next page)

(continued from previous page)

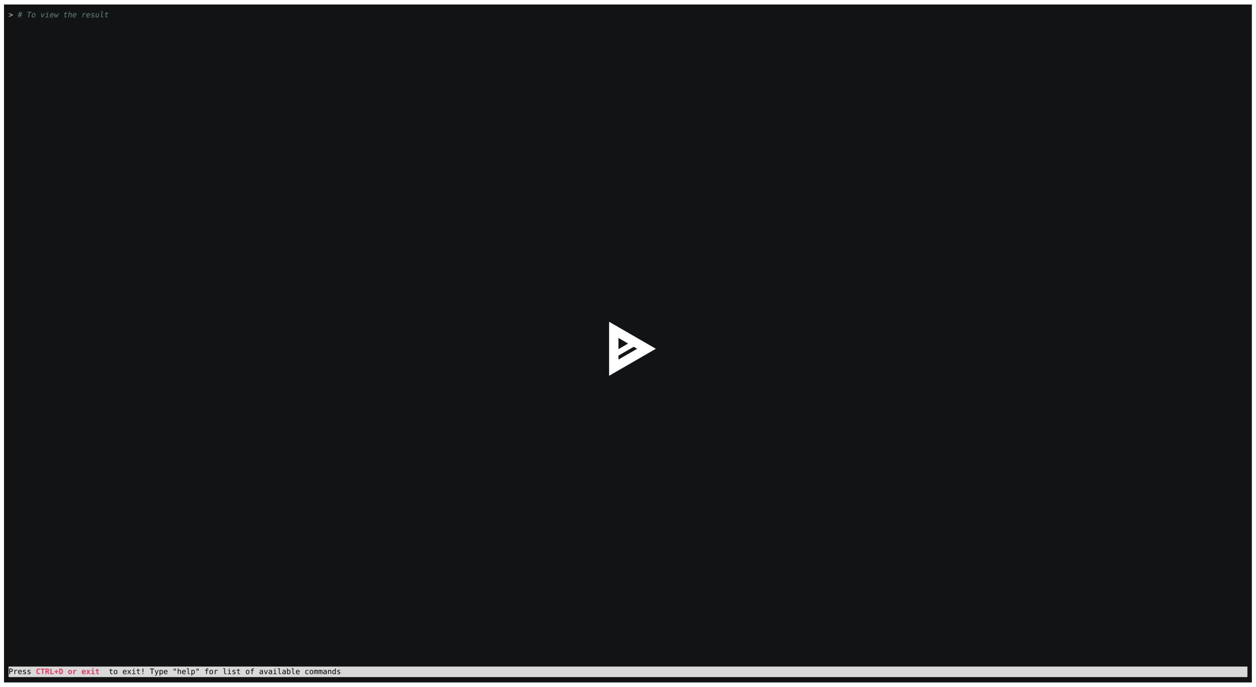
```
    "parity": "N",
    "stopbits": 1.0
}
> client.set_timeout value=1
null

> client.get_timeout
1.0

> client.get_serial_settings
{
    "t1.5": 0.00171875,
    "baudrate": 9600,
    "read timeout": 1.0,
    "port": "/dev/ptyp0",
    "t3.5": 0.00401,
    "bytesize": 8,
    "parity": "N",
    "stopbits": 1.0
}
```

4.3 Demo

4.3.1 Pymodbus REPL Client

Pymodbus REPL comes with many handy features such as payload decoder to directly retrieve the values in desired format and supports all the diagnostic function codes directly .

For more info on REPL Client refer pymodbus repl client

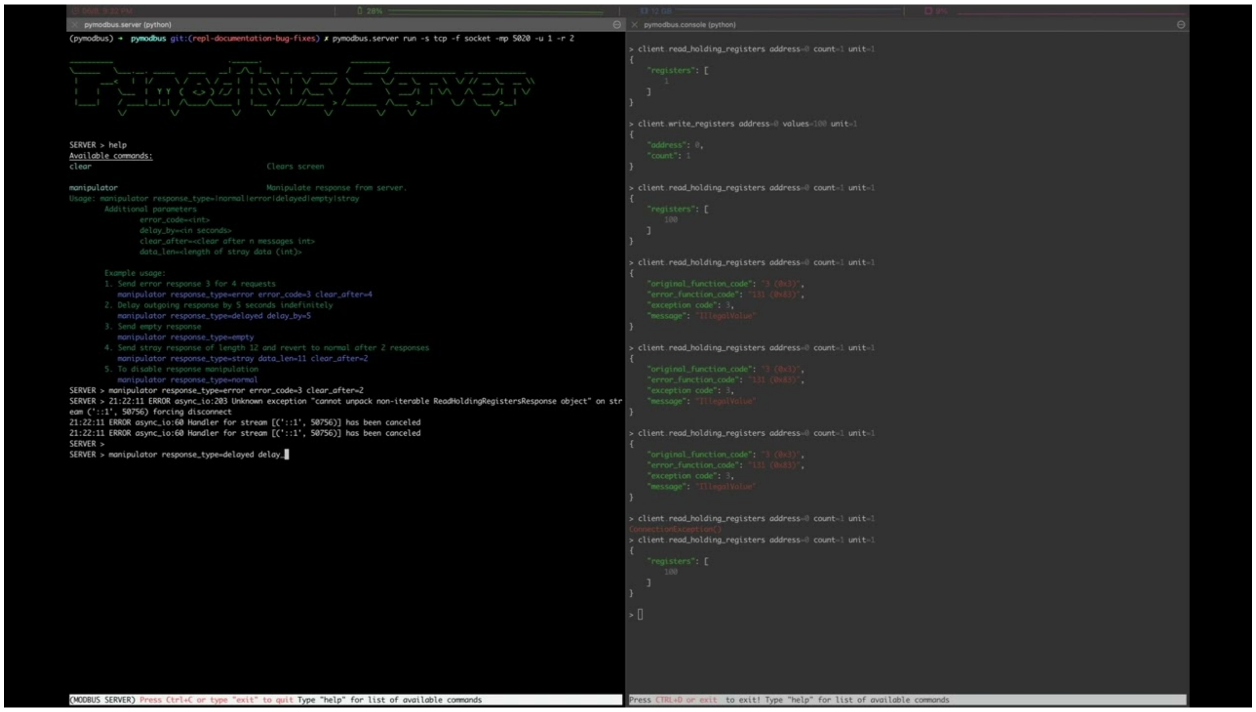


```
> # To view the result

Press CTRL+D or exit  to exit! Type "help" for list of available commands
```

4.3.2 Pymodbus REPL Server

Pymodbus also comes with a REPL server to quickly run an asynchronous server with additional capabilities out of the box like simulating errors, delay, mangled messages etc.

For more info on REPL Server refer pymodbus repl server



```
(pymodbus) → pymodbus git:(repl-documentation-bug-fixes) ✗ pymodbus.server run -s tcp -f socket -mp 5020 -u 1 -r 2

SERVER > help
Available commands:
clear                                   Clears screen

manipulator                             Manipulate response from server.
Usage: manipulator response_type=|normal|error|delayed|empty|stray
        Additional parameters
                error_code=<int>
                delay_by=<in seconds>
                clear_after=<clear after n messages int>
                data_len=<length of stray data (int)>

        Example usage:
        1. Send error response 3 for 4 requests
           manipulator response_type=error error_code=3 clear_after=4
        2. Delay outgoing response by 5 seconds indefinitely
           manipulator response_type=delayed delay_by=5
        3. Send empty response
           manipulator response_type=empty
        4. Send stray response of length 12 and revert to normal after 2 responses
           manipulator response_type=stray data_len=11 clear_after=2
        5. To disable response manipulation
           manipulator response_type=normal
SERVER > manipulator response_type=error error_code=3 clear_after=2
SERVER > 21:22:11 ERROR async_io:203 Unknown exception "cannot unpack non-iterable ReadHoldingRegistersResponse object" on str
eam ('::1', 50756) forcing disconnect
21:22:11 ERROR async_io:60 Handler for stream [('::1', 50756)] has been canceled
21:22:11 ERROR async_io:60 Handler for stream [('::1', 50756)] has been canceled
SERVER >
SERVER > manipulator response_type=delayed delay_

(MODBUS SERVER) Press Ctrl+C or type "exit" to quit Type "help" for list of available commands
```

```
> client.read_holding_registers address=0 count=1 unit=1
{
    "registers": [
        1
    ]
}

> client.write_registers address=0 values=100 unit=1
{
    "address": 0,
    "count": 1
}

> client.read_holding_registers address=0 count=1 unit=1
{
    "registers": [
        100
    ]
}

> client.read_holding_registers address=0 count=1 unit=1
{
    "original_function_code": "3 (0x3)",
    "error_function_code": "131 (0x83)",
    "exception code": 3,
    "message": "IllegalValue"
}

> client.read_holding_registers address=0 count=1 unit=1
{
    "original_function_code": "3 (0x3)",
    "error_function_code": "131 (0x83)",
    "exception code": 3,
    "message": "IllegalValue"
}

> client.read_holding_registers address=0 count=1 unit=1
{
    "original_function_code": "3 (0x3)",
    "error_function_code": "131 (0x83)",
    "exception code": 3,
    "message": "IllegalValue"
}

> client.read_holding_registers address=0 count=1 unit=1
ConnectionException()
> client.read_holding_registers address=0 count=1 unit=1
{
    "registers": [
        100
    ]
}

> 

Press CTRL+D or exit  to exit! Type "help" for list of available commands
```

CHAPTER

FIVE

SIMULATOR

The simulator is a full fledged modbus simulator, which is constantly being evolved with user ideas / amendments.

The purpose of the simulator is to provide support for client application test harnesses with end-to-end testing simulating real life modbus devices.

The datastore simulator allows the user to (all automated)

- simulate a modbus device by adding a simple configuration,
- test how a client handles modbus exceptions,
- test a client apps correct use of the simulated device.

The web interface allows the user to (online / manual)

- test how a client handles modbus errors,
- test how a client handles communication errors like divided messages,
- run your test server in the cloud,
- monitor requests/responses,
- inject modbus errors like malicious a response,
- see/Change values online.

The REST API allow the test process to be automated

- spin up a test server with unix domain sockets in your test harness,
- set expected responses with a simple REST API command,
- check the result with another simple REST API command,
- test your client app in a true end-to-end fashion.

5.1 Configuration

Configuring the pymodbus simulator is done with a json file, or if only using the datastore simulator a python dict (same structure as the device part of the json file).

5.1.1 Json file layout

The json file consist of 2 main entries "server_list" (see *Server entries*) and "device_list" (see *Device entries*) each containing a list of servers/devices

```
{
    "server_list": {
        "<name>": { ... },
        ...
    },
    "device_list": {
        "<name>": { ... },
        ...
    }
}
```

You can define as many server and devices as you like, when starting *pymodbus.simulator* you select one server and one device to simulate.

A entry in "device_list" correspond to the dict you can use as parameter to datastore_simulator is you want to construct your own simulator.

5.1.2 Server entries

The entries for a tcp server with minimal parameters look like:

```
{
     "server_list": {
         "server": {
             "comm": "tcp",
             "host": "0.0.0.0",
             "port": 5020,
             "framer": "socket",
         }
     }
     "device_list": {
         ...
     }
}
```

The example uses **"comm": "tcp"**, so the entries are arguments to *pymodbus.server.ModbusTcpServer*, where detailed information are available.

The entry "comm" allows the following values:

- "serial", to use *pymodbus.server.ModbusSerialServer*,
- "tcp", to use *pymodbus.server.ModbusTcpServer*,
- "tls", to use *pymodbus.server.ModbusTlsServer*,
- "udp"; to use *pymodbus.server.ModbusUdpServer*.

The entry "framer" allows the following values:

- "ascii" to use *pymodbus.framer.ascii_framer.ModbusAsciiFramer*,
- "binary to use `pymodbus.framer.ascii_framer.ModbusBinaryFramer`,

- “rtu” to use `pymodbus.framer.ascii_framer.ModbusRtuFramer`,
- “tls” to use `pymodbus.framer.ascii_framer.ModbusTlsFramer`,
- “socket” to use `pymodbus.framer.ascii_framer.ModbusSocketFramer`.

Warning: not all “framer” types can be used with all “comm” types.

e.g. `"framer": “tls”` only works with `“comm”: “tls”`!

5.1.3 Server configuration examples

```
{
   "server_list": {
        "server": {
            "comm": "tcp",
            "host": "0.0.0.0",
            "port": 5020,
            "ignore_missing_slaves": false,
            "framer": "socket",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        },
        "server_try_serial": {
            "comm": "serial",
            "port": "/dev/tty0",
            "stopbits": 1,
            "bytesize": 8,
            "parity": "N",
            "baudrate": 9600,
            "timeout": 3,
            "reconnect_delay": 2,
            "framer": "rtu",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        },
        "server_try_tls": {
            "comm": "tls",
            "host": "0.0.0.0",
            "port": 5020,
```

(continues on next page)

(continued from previous page)

```
            "certfile": "certificates/pymodbus.crt",
            "keyfile": "certificates/pymodbus.key",
            "ignore_missing_slaves": false,
            "framer": "tls",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        },
        "server_test_try_udp": {
            "comm": "udp",
            "host": "0.0.0.0",
            "port": 5020,
            "ignore_missing_slaves": false,
            "framer": "socket",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        }
    }
}
```

5.1.4 Device entries

Each device is configured in a number of sections, described in detail below

- "setup", defines the overall structure of the device, like e.g. number of registers,
- "invalid", defines invalid registers and causes a modbus exception when reading and/or writing,
- "write", defines registers which allow read/write, other registers causes a modbus exception when writing,
- "bits", defines registers which contain bits (discrete input and coils),
- "uint16", defines registers which contain a 16 bit unsigned integer,
- "uint32", defines sets of registers (2) which contain a 32 bit unsigned integer,
- "float32", defines sets of registers (2) which contain a 32 bit float,
- "string", defines sets of registers which contain a string,
- "repeat", is a special command to copy configuration if a device contains X bay controllers, configure one and use repeat for X-1.

The datastore simulator manages the registers in a big list, which can be manipulated with

- actions (functions that are called with each access)

- manually via the WEB interface
- automated via the REST API interface
- the client (writing values)

It is important to understand that the modbus protocol does not know or care how the physical memory/registers are organized, but it has a huge impact on the client!

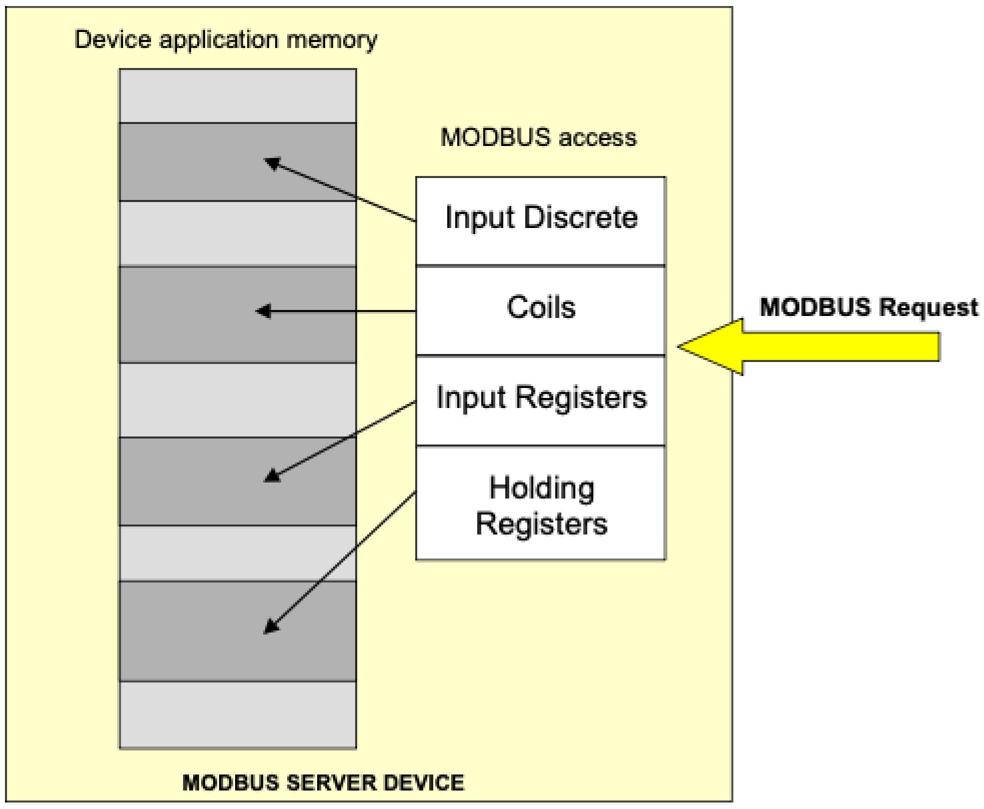
Communication with a modbus device is based on registers which each contain 16 bits (2 bytes). The requests are grouped in 4 groups

- Input Discrete
- Coils
- Input registers
- Holding registers

The 4 blocks are mapped into physical memory, but the modbus protocol makes no assumption or demand on how this is done.

The history of modbus devices have shown 2 forms of mapping.

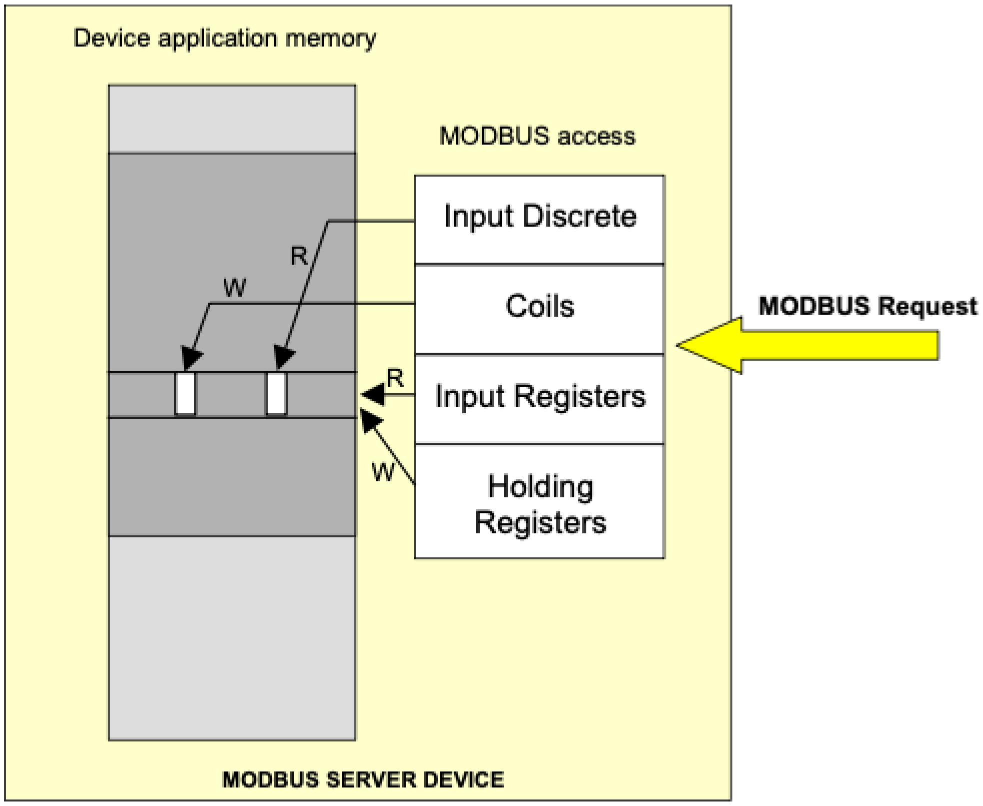
The first form is also the original form. It originates from a time where the devices did not contain memory, but the request was mapped directly to a physical sensor:



When reading holding register 1 (block 4) you get a different register as when reading input register 1 (block 1). Each block references a different physical register memory, in other words the size of the needed memory is the sum of the block sizes.

The second form uses 1 shared block, most modern devices use this form for 2 main reasons:

- the modbus protocol implementation do not connect directly to the sensors but to a shared memory controlled by a small microprocessor.
- designers can group related information independent of type (e.g. a bay controller with register 1 as coil, register 2 as input and register 3 as holding)



When reading holding register 1 the same phyical register is accessed as when reading input register 1. Each block references the same physical register memory, in other words the size of the needed memory is the size of the largest block.

The datastore simulator supports both types.

5.1.4.1 Setup section

Example "setup" configuration:

```
"setup": {
    "co size": 10,
    "di size": 20,
    "hr size": 15,
    "ir size": 25,
    "shared blocks": true,
    "type exception": true,
    "defaults": {
        "value": {
            "bits": 0,
            "uint16": 0,
```

(continues on next page)

(continued from previous page)

```
            "uint32": 0,
            "float32": 0.0,
            "string": " "
        },
        "action": {
            "bits": null,
            "uint16": "register",
            "uint32": "register",
            "float32": "register",
            "string": null
        }
}
```

"co size", **"di size"**, **"hr size"**, **"ir size"**:

Define the size of each block. If using shared block the register list size will be the size of the biggest block (25 registers) If not using shared block the register list size will be the sum of the 4 block sizes (70 registers).

"shared blocks"

Defines if the blocks are independent or shared (true)

Tip: if shared is set to false, please remember to adjust the addresses, depending on in which group they are.

assuming all sizes are set to 10, the addresses for configuration are as follows:

- coils have addresses 0-9,
- discrete_inputs have addresses 10-19,
- input_registers have addresses 20-29,
- holding_registers have addresses 30-39

when configuring the the datatypes (when calling each block start with 0).

This is needed because the datatypes can be in different blocks.

"type exception"

Defines is the server returns a modbus exception if a read/write request violates the specified type. E.g. Read holding register 10 with count 1, but the 10,11 are defined as UINT32 and thus can only be read with multiples of 2.

This feature is designed to control that a client access the device in the manner it was designed.

"defaults"

Defines how to defines registers not configured or or only partial configured.

"value" defines the default value for each type.

"action" defines the default action for each type. Actions are functions that are called whenever the register is accessed and thus allows automatic manipulation.

The datastore simulator have a number of builtin actions, and allows custom actions to be added:

- **"random"**, change the value with every access,

- **"increment"**, increment the value by 1 with every access,
- **"timestamp"**, uses 6 registers and build a timestamp,
- **"reset"**, causes a reboot of the simulator,
- **"uptime"**, sets the number of seconds the server have been running.

The **"random"** and **"increment"** actions may optionally minimum and/or maximum. In case of **"increment"**, the counter is reset to the minimum value, if the maximum is crossed.

```
{"addr": 9, "value": 7, "action": "random", "kwargs": {"minval": 0, "maxval": 12} },
{"addr": 10, "value": 100, "action": "increment", "kwargs": {"minval": 50} }
```

5.1.4.2 Invalid section

Example "invalid" configuration:

```
"invalid": [
    5,
    [10, 15]
],
```

Defines invalid registers which cannot be read or written. When accessed the response in a modbus exception **invalid address**. In the example registers 5, 10, 11, 12, 13, 14, 15 will produce an exception response.

Registers can be singulars (first entry) or arrays (second entry)

5.1.4.3 Write section

Example "write" configuration:

```
"write": [
    4,
    [5, 6]
],
```

Defines registers which can be written to. When writing to registers not defined here the response is a modbus exception **invalid address**.

Registers can be singulars (first entry) or arrays (second entry)

5.1.4.4 Bits section

Example "bits" configuration:

```
"bits": [
    5,
    [6, 7],
    {"addr": 8, "value": 7},
    {"addr": 9, "value": 7, "action": "random"},
    {"addr": [11, 12], "value": 7, "action": "random"}
],
```

defines registers which contain bits (discrete input and coils),

Registers can be singulars (first entry) or arrays (second entry), furthermore a value and/or a action can be defined, the value and/or action is inserted into each register defined in “addr”.

5.1.4.5 Uint16 section

Example “uint16” configuration:

```
"uint16": [
    5,
    [6, 7],
    {"addr": 8, "value": 30123},
    {"addr": 9, "value": 712, "action": "increment"},
    {"addr": [11, 12], "value": 517, "action": "random"}
],
```

defines registers which contain a 16 bit unsigned integer,

Registers can be singulars (first entry) or arrays (second entry), furthermore a value and/or a action can be defined, the value and/or action is inserted into each register defined in “addr”.

5.1.4.6 Uint32 section

Example “uint32” configuration:

```
"uint32": [
    [6, 7],
    {"addr": [8, 9], "value": 300123},
    {"addr": [10, 13], "value": 400712, "action": "increment"},
    {"addr": [14, 15], "value": 500517, "action": "random"}
],
```

defines sets of registers (2) which contain a 32 bit unsigned integer,

Registers can only be arrays in multiples of 2, furthermore a value and/or a action can be defined, the value and/or action is converted (high/low value) and inserted into each register set defined in “addr”.

5.1.4.7 Float32 section

Example “float32” configuration:

```
"float32": [
    [6, 7],
    {"addr": [8, 9], "value": 3123.17},
    {"addr": [10, 13], "value": 712.5, "action": "increment"},
    {"addr": [14, 15], "value": 517.0, "action": "random"}
],
```

defines sets of registers (2) which contain a 32 bit float,

Registers can only be arrays in multiples of 2, furthermore a value and/or a action can be defined, the value and/or action is converted (high/low value) and inserted into each register set defined in “addr”.

Remark remember to set `"value":  <float value>` like 512.0 (float) not 512 (integer).

5.1.4.8 String section

Example "string" configuration:

```
"string": [
    7,
    [8, 9],
    {"addr": [16, 20], "value": "A_B_C_D_E_"}
],
```

defines sets of registers which contain a string,

Registers can be singulars (first entry) or arrays (second entry). Important each string must be defined individually.

- Entry 1 is a string of 2 chars,
- Entry 2 is a string of 4 chars,
- Entry 3 is a string of 10 chars with the value ''A_B_C_D_E_''.

5.1.4.9 Repeat section

Example "repeat" configuration:

```
"repeat": [
    {"addr": [0, 2], "to": [10, 11]},
    {"addr": [0, 2], "to": [10, 15]},
]
```

is a special command to copy configuration if a device contains X bay controllers, configure one and use repeat for X-1.

First entry copies registers 0-2 to 10-11, resulting in 10 == 0, 11 == 1, 12 unchanged.

Second entry copies registers 0-2 to 10-15, resulting in 10 == 0, 11 == 1, 12 == 2, 13 == 0, 14 == 1, 15 == 2, 16 unchanged.

5.1.5 Device configuration examples

```
{
    "server_list": {
        ...
    },
    "device_list": {
        "device": {
            "setup": {
                "co size": 63000,
                "di size": 63000,
                "hr size": 63000,
                "ir size": 63000,
                "shared blocks": true,
                "type exception": true,
                "defaults": {
                    "value": {
                        "bits": 0,
                        "uint16": 0,
```

(continues on next page)

(continued from previous page)

```
                        "uint32": 0,
                        "float32": 0.0,
                        "string": " "
                    },
                    "action": {
                        "bits": null,
                        "uint16": "register",
                        "uint32": "register",
                        "float32": "register",
                        "string": null
                    }
                }
            },
            "invalid": [
                1
            ],
            "write": [
                5
            ],
            "bits": [
                {"addr": 2, "value": 7}
            ],
            "uint16": [
                {"addr": 3, "value": 17001},
                2100
            ],
            "uint32": [
                {"addr": 4, "value": 617001},
                [3037, 3038]
            ],
            "float32": [
                {"addr": 6, "value": 404.17},
                [4100, 4101]
            ],
            "string": [
                5047,
                {"addr": [16, 20], "value": "A_B_C_D_E_"}
            ],
            "repeat": [
            ]
        },
        "device_try": {
            "setup": {
                "co size": 63000,
                "di size": 63000,
                "hr size": 63000,
                "ir size": 63000,
                "shared blocks": true,
                "type exception": true,
                "defaults": {
                    "value": {
                        "bits": 0,
```

(continues on next page)

(continued from previous page)

```
                    "uint16": 0,
                    "uint32": 0,
                    "float32": 0.0,
                    "string": " "
                },
                "action": {
                    "bits": null,
                    "uint16": "register",
                    "uint32": "register",
                    "float32": "register",
                    "string": null
                }
            }
        },
        "invalid": [
            [0, 5],
            77
        ],
        "write": [
            10,
            [61, 76]
        ],
        "bits": [
            10,
            1009,
            [1116, 1119],
            {"addr": 1144, "value": 1},
            {"addr": [1148,1149], "value": 32117},
            {"addr": [1208, 1306], "action": "random"}
        ],
        "uint16": [
            11,
            2027,
            [2126, 2129],
            {"addr": 2164, "value": 1},
            {"addr": [2168,2169], "value": 32117},
            {"addr": [2208, 2306], "action": null}
        ],
        "uint32": [
            12,
            3037,
            [3136, 3139],
            {"addr": 3174, "value": 1},
            {"addr": [3188,3189], "value": 32514},
            {"addr": [3308, 3406], "action": null},
            {"addr": [3688, 3878], "value": 115, "action": "increment"}
        ],
        "float32": [
            14,
            4047,
            [4146, 4149],
            {"addr": 4184, "value": 1},
```

(continues on next page)

(continued from previous page)

```
                {"addr": [4198,4191], "value": 32514.1},
                {"addr": [4308, 4406], "action": null},
                {"addr": [4688, 4878], "value": 115.7, "action": "increment"}
            ],
            "string": [
                {"addr": [16, 20], "value": "A_B_C_D_E_"},
                5047,
                [5146, 5149],
                {"addr": [529, 544], "value": "Brand name, 32 bytes...........X"}
            ],
            "repeat": [
                {"addr": [0, 999], "to": [10000, 10999]},
                {"addr": [10, 1999], "to": [11000, 11999]}
            ]
        }
    },
    "device_minimum": {
            "setup": {
                "co size": 10,
                "di size": 10,
                "hr size": 10,
                "ir size": 10,
                "shared blocks": true,
                "type exception": false,
                "defaults": {
                    "value": {
                        "bits": 0,
                        "uint16": 0,
                        "uint32": 0,
                        "float32": 0.0,
                        "string": " "
                    },
                    "action": {
                        "bits": null,
                        "uint16": null,
                        "uint32": null,
                        "float32": null,
                        "string": null
                    }
                }
            },
            "invalid": [],
            "write": [],
            "bits": [],
            "uint16": [
                [0, 9]
            ],
            "uint32": [],
            "float32": [],
            "string": [],
            "repeat": []
        }
```

(continues on next page)

(continued from previous page)

```
    }
}
```

5.1.6 Configuration used for test

```
{
    "server_list": {
        "server": {
            "comm": "tcp",
            "host": "0.0.0.0",
            "port": 5020,
            "ignore_missing_slaves": false,
            "framer": "socket",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        },
        "server_try_serial": {
            "comm": "serial",
            "port": "/dev/tty0",
            "stopbits": 1,
            "bytesize": 8,
            "parity": "N",
            "baudrate": 9600,
            "timeout": 3,
            "reconnect_delay": 2,
            "framer": "rtu",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        },
        "server_try_tls": {
            "comm": "tls",
            "host": "0.0.0.0",
            "port": 5020,
            "certfile": "certificates/pymodbus.crt",
            "keyfile": "certificates/pymodbus.key",
            "ignore_missing_slaves": false,
            "framer": "tls",
            "identity": {
```

(continues on next page)

(continued from previous page)

```
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        },
        "server_test_try_udp": {
            "comm": "udp",
            "host": "0.0.0.0",
            "port": 5020,
            "ignore_missing_slaves": false,
            "framer": "socket",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
        }
    },
    "device_list": {
        "device": {
            "setup": {
                "co size": 63000,
                "di size": 63000,
                "hr size": 63000,
                "ir size": 63000,
                "shared blocks": true,
                "type exception": true,
                "defaults": {
                    "value": {
                        "bits": 0,
                        "uint16": 0,
                        "uint32": 0,
                        "float32": 0.0,
                        "string": " "
                    },
                    "action": {
                        "bits": null,
                        "uint16": "increment",
                        "uint32": "increment",
                        "float32": "increment",
                        "string": null
                    }
                }
            },
            "invalid": [
                1
```

(continues on next page)

(continued from previous page)

```
            ],
            "write": [
                3
            ],
            "bits": [
                {"addr": 2, "value": 7}
            ],
            "uint16": [
                {"addr": 3, "value": 17001, "action": null},
                2100
            ],
            "uint32": [
                {"addr": [4, 5], "value": 617001, "action": null},
                [3037, 3038]
            ],
            "float32": [
                {"addr": [6, 7], "value": 404.17},
                [4100, 4101]
            ],
            "string": [
                5047,
                {"addr": [16, 20], "value": "A_B_C_D_E_"}
            ],
            "repeat": [
            ]
        },
        "device_try": {
            "setup": {
                "co size": 63000,
                "di size": 63000,
                "hr size": 63000,
                "ir size": 63000,
                "shared blocks": true,
                "type exception": true,
                "defaults": {
                    "value": {
                        "bits": 0,
                        "uint16": 0,
                        "uint32": 0,
                        "float32": 0.0,
                        "string": " "
                    },
                    "action": {
                        "bits": null,
                        "uint16": null,
                        "uint32": null,
                        "float32": null,
                        "string": null
                    }
                }
            },
            "invalid": [
```

(continues on next page)

(continued from previous page)

```
                [0, 5],
                77
            ],
            "write": [
                10
            ],
            "bits": [
                10,
                1009,
                [1116, 1119],
                {"addr": 1144, "value": 1},
                {"addr": [1148,1149], "value": 32117},
                {"addr": [1208, 1306], "action": "random"}
            ],
            "uint16": [
                11,
                2027,
                [2126, 2129],
                {"addr": 2164, "value": 1},
                {"addr": [2168,2169], "value": 32117},
                {"addr": [2208, 2304], "action": "increment"},
                {"addr": 2305,
                    "value": 50,
                    "action": "increment",
                    "kwargs": {"minval": 45, "maxval": 155}
                },
                {"addr": 2306,
                    "value": 50,
                    "action": "random",
                    "kwargs": {"minval": 45, "maxval": 55}
                }
            ],
            "uint32": [
                [12, 13],
                [3037, 3038],
                [3136, 3139],
                {"addr": [3174, 3175], "value": 1},
                {"addr": [3188,3189], "value": 32514},
                {"addr": [3308, 3407], "action": null},
                {"addr": [3688, 3875], "value": 115, "action": "increment"},
                {"addr": [3876, 3877],
                    "value": 50000,
                    "action": "increment",
                    "kwargs": {"minval": 45000, "maxval": 55000}
                },
                {"addr": [3878, 3879],
                    "value": 50000,
                    "action": "random",
                    "kwargs": {"minval": 45000, "maxval": 55000}
                }
            ],
            "float32": [
```

(continues on next page)

(continued from previous page)

```
                [14, 15],
                [4047, 4048],
                [4146, 4149],
                {"addr": [4184, 4185], "value": 1},
                {"addr": [4188, 4191], "value": 32514.2},
                {"addr": [4308, 4407], "action": null},
                {"addr": [4688, 4875], "value": 115.7, "action": "increment"},
                {"addr": [4876, 4877],
                    "value": 50000.0,
                    "action": "increment",
                    "kwargs": {"minval": 45000.0, "maxval": 55000.0}
                },
                {"addr": [4878, 48779],
                    "value": 50000.0,
                    "action": "random",
                    "kwargs": {"minval": 45000.0, "maxval": 55000.0}
                }
            ],
            "string": [
                {"addr": [16, 20], "value": "A_B_C_D_E_"},
                {"addr": [529, 544], "value": "Brand name, 32 bytes..........X"}
            ],
            "repeat": [
            ]
        }
    }
}
```

5.2 Simulator datastore

The simulator datastore is an advanced datastore. The simulator allows to simulate the registers of a real life modbus device by adding a simple dict (definition see *Device entries*).

The simulator datastore allows to add actions (functions) to a register, and thus allows a low level automation.

Documentation `pymodbus.datastore.ModbusSimulatorContext`

5.3 Web frontend

TO BE DOCUMENTED.

5.3.1 pymodbus.simulator

The easiest way to run the simulator with web is to use "pymodbus.simulator" from the commandline.

TO BE DOCUMENTED. HTTP server for modbus simulator.

`class pymodbus.server.simulator.http_server.`**`CallTracer`**(*call: bool = False*, *fc: int = -1*, *address: int = -1*, *count: int = -1*, *data: bytes = b''*)

Bases: `object`

Define call/response traces.

`class pymodbus.server.simulator.http_server.`**`CallTypeMonitor`**(*active: bool = False*, *trace_response: bool = False*, *range_start: int = -1*, *range_stop: int = -1*, *function: int = -1*, *hex: bool = False*, *decode: bool = False*)

Bases: `object`

Define Request/Response monitor.

`class pymodbus.server.simulator.http_server.`**`CallTypeResponse`**(*active: int = -1*, *split: int = 0*, *delay: int = 0*, *junk_len: int = 10*, *error_response: int = 0*, *change_rate: int = 0*, *clear_after: int = 1*)

Bases: `object`

Define Response manipulation.

`class pymodbus.server.simulator.http_server.`**`ModbusSimulatorServer`**(*modbus_server: str = 'server'*, *modbus_device: str = 'device'*, *http_host: str = '0.0.0.0'*, *http_port: int = 8080*, *log_file: str = 'server.log'*, *json_file: str = 'setup.json'*, *custom_actions_module: str | None = None*)

Bases: `object`

ModbusSimulatorServer.

Parameters

- **`modbus_server`** – Server name in json file (default: "server")
- **`modbus_device`** – Device name in json file (default: "client")
- **`http_host`** – TCP host for HTTP (default: "localhost")
- **`http_port`** – TCP port for HTTP (default: 8080)
- **`json_file`** – setup file (default: "setup.json")
- **`custom_actions_module`** – python module with custom actions (default: none)

if either http_port or http_host is none, HTTP will not be started. This class starts a http server, that serves a couple of endpoints:

- **“<addr>/”** static files
- **“<addr>/api/log”** log handling, HTML with GET, REST-API with post
- **“<addr>/api/registers”** register handling, HTML with GET, REST-API with post
- **“<addr>/api/calls”** call (function code / message) handling, HTML with GET, REST-API with post
- **“<addr>/api/server”** server handling, HTML with GET, REST-API with post

Example:

```
from pymodbus.server import ModbusSimulatorServer

async def run():
    simulator = ModbusSimulatorServer(
        modbus_server="my server",
        modbus_device="my device",
        http_host="localhost",
        http_port=8080)
    await simulator.run_forever(only_start=True)
    ...
    await simulator.stop()
```

`async` **`start_modbus_server`**(*app*)

Start Modbus server as asyncio task.

`async` **`stop_modbus_server`**(*app*)

Stop modbus server.

`async` **`run_forever`**(*only_start=False*)

Start modbus and http servers.

`async` **`stop`**()

Stop modbus and http servers.

`async` **`handle_html_static`**(*request*)

Handle static html.

`async` **`handle_html`**(*request*)

Handle html.

`async` **`handle_json`**(*request*)

Handle api registers.

`build_html_registers`(*params*, *html*)

Build html registers page.

`build_html_calls`(*params: dict*, *html: str*) → str

Build html calls page.

`build_html_log`(*_params*, *html*)

Build html log page.

`build_html_server`(*_params*, *html*)

Build html server page.

`build_json_registers`(*params*, *json_dict*)

Build html registers page.

`build_json_calls`(*params*, *json_dict*)

Build html calls page.

`build_json_log`(*params*, *json_dict*)

Build json log page.

`build_json_server`(*params*, *json_dict*)

Build html server page.

`helper_build_html_submit`(*params*)

Build html register submit.

`action_clear`(*_params*, *_range_start*, *_range_stop*)

Clear register filter.

`action_stop`(*_params*, *_range_start*, *_range_stop*)

Stop call monitoring.

`action_reset`(*_params*, *_range_start*, *_range_stop*)

Reset call simulation.

`action_add`(*params*, *range_start*, *range_stop*)

Build list of registers matching filter.

`action_monitor`(*params*, *range_start*, *range_stop*)

Start monitoring calls.

`action_set`(*params*, *_range_start*, *_range_stop*)

Set register value.

`action_simulate`(*params*, *_range_start*, *_range_stop*)

Simulate responses.

`server_response_manipulator`(*response*)

Manipulate responses.

All server responses passes this filter before being sent. The filter returns:

- response, either original or modified
- skip_encoding, signals whether or not to encode the response

`server_request_tracer`(*request*, **_addr*)

Trace requests.

All server requests passes this filter before being handled.

5.4 Pymodbus simulator ReST API

TO BE DOCUMENTED.

CHAPTER SIX

EXAMPLES

Examples are divided in 2 parts:

The first part are some simple client examples which can be copied and run directly. These examples show the basic functionality of the library.

The second part are more advanced examples, but in order to not duplicate code, this requires you to download the examples directory and run the examples in the directory.

6.1 Ready to run examples:

These examples are very basic examples, showing how a client can communicate with a server.

You need to modify the code to adapt it to your situation.

6.1.1 Simple asynchronous client

Source: examples/simple_async_client.py

```
#!/usr/bin/env python3
"""Pymodbus asynchronous client example.

An example of a single threaded synchronous client.

usage: simple_client_async.py

All options must be adapted in the code
The corresponding server must be started before e.g. as:
    python3 server_sync.py
"""
import asyncio

import pymodbus.client as ModbusClient
from pymodbus import (
    ExceptionResponse,
    Framer,
    ModbusException,
    pymodbus_apply_logging_config,
)
```

(continues on next page)

(continued from previous page)

```
async def run_async_simple_client(comm, host, port, framer=Framer.SOCKET):
    """Run async client."""
    # activate debugging
    pymodbus_apply_logging_config("DEBUG")

    print("get client")
    if comm == "tcp":
        client = ModbusClient.AsyncModbusTcpClient(
            host,
            port=port,
            framer=framer,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # source_address=("localhost", 0),
        )
    elif comm == "udp":
        client = ModbusClient.AsyncModbusUdpClient(
            host,
            port=port,
            framer=framer,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # source_address=None,
        )
    elif comm == "serial":
        client = ModbusClient.AsyncModbusSerialClient(
            port,
            framer=framer,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # strict=True,
            baudrate=9600,
            bytesize=8,
            parity="N",
            stopbits=1,
            # handle_local_echo=False,
        )
    elif comm == "tls":
        client = ModbusClient.AsyncModbusTlsClient(
            host,
            port=port,
            framer=Framer.TLS,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # sslctx=sslctx,
            certfile="../examples/certificates/pymodbus.crt",
            keyfile="../examples/certificates/pymodbus.key",
```

(continues on next page)

(continued from previous page)

```
            # password="none",
            server_hostname="localhost",
        )
    else:
        print(f"Unknown client {comm} selected")
        return

    print("connect to server")
    await client.connect()
    # test client is connected
    assert client.connected

    print("get and verify data")
    try:
        # See all calls in client_calls.py
        rr = await client.read_coils(1, 1, slave=1)
    except ModbusException as exc:
        print(f"Received ModbusException({exc}) from library")
        client.close()
        return
    if rr.isError():
        print(f"Received Modbus library error({rr})")
        client.close()
        return
    if isinstance(rr, ExceptionResponse):
        print(f"Received Modbus library exception ({rr})")
        # THIS IS NOT A PYTHON EXCEPTION, but a valid modbus message
        client.close()

    print("close connection")
    client.close()


if __name__ == "__main__":
    asyncio.run(
        run_async_simple_client("tcp", "127.0.0.1", 5020), debug=True
    )
```

6.1.2 Simple synchronous client

Source: examples/simple_sync_client.py

```
#!/usr/bin/env python3
"""Pymodbus synchronous client example.

An example of a single threaded synchronous client.

usage: simple_client_async.py

All options must be adapted in the code
The corresponding server must be started before e.g. as:
```

(continues on next page)

(continued from previous page)

```
    python3 server_sync.py
"""

# --------------------------------------------------------------------------- #
# import the various client implementations
# --------------------------------------------------------------------------- #
import pymodbus.client as ModbusClient
from pymodbus import (
    ExceptionResponse,
    Framer,
    ModbusException,
    pymodbus_apply_logging_config,
)


def run_sync_simple_client(comm, host, port, framer=Framer.SOCKET):
    """Run sync client."""
    # activate debugging
    pymodbus_apply_logging_config("DEBUG")

    print("get client")
    if comm == "tcp":
        client = ModbusClient.ModbusTcpClient(
            host,
            port=port,
            framer=framer,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,y
            # source_address=("localhost", 0),
        )
    elif comm == "udp":
        client = ModbusClient.ModbusUdpClient(
            host,
            port=port,
            framer=framer,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # source_address=None,
        )
    elif comm == "serial":
        client = ModbusClient.ModbusSerialClient(
            port,
            framer=framer,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # strict=True,
            baudrate=9600,
            bytesize=8,
            parity="N",
```

(continues on next page)

(continued from previous page)

```
            stopbits=1,
            # handle_local_echo=False,
        )
    elif comm == "tls":
        client = ModbusClient.ModbusTlsClient(
            host,
            port=port,
            framer=Framer.TLS,
            # timeout=10,
            # retries=3,
            # retry_on_empty=False,
            # sslctx=None,
            certfile="../examples/certificates/pymodbus.crt",
            keyfile="../examples/certificates/pymodbus.key",
            # password=None,
            server_hostname="localhost",
        )
    else:
        print(f"Unknown client {comm} selected")
        return

    print("connect to server")
    client.connect()

    print("get and verify data")
    try:
        rr = client.read_coils(1, 1, slave=1)
    except ModbusException as exc:
        print(f"Received ModbusException({exc}) from library")
        client.close()
        return
    if rr.isError():
        print(f"Received Modbus library error({rr})")
        client.close()
        return
    if isinstance(rr, ExceptionResponse):
        print(f"Received Modbus library exception ({rr})")
        # THIS IS NOT A PYTHON EXCEPTION, but a valid modbus message
        client.close()

    print("close connection")
    client.close()


if __name__ == "__main__":
    run_sync_simple_client("tcp", "127.0.0.1", "5020")
```

6.1.3 Client performance sync vs async

Source: examples/client_performance.py

```
#!/usr/bin/env python3
"""Test performance of client: sync vs. async.

This example show how much faster the async version is.

example run:

(pymodbus) % ./client_performance.py
--- Testing sync client v3.4.1
running 1000 call (each 10 registers), took 114.10 seconds
Averages 114.10 ms pr call and 11.41 ms pr register.
--- Testing async client v3.4.1
running 1000 call (each 10 registers), took 0.33 seconds
Averages 0.33 ms pr call and 0.03 ms pr register.
"""
import asyncio
import time

from pymodbus import Framer
from pymodbus.client import AsyncModbusSerialClient, ModbusSerialClient


LOOP_COUNT = 1000
REGISTER_COUNT = 10


def run_sync_client_test():
    """Run sync client."""
    print("--- Testing sync client v3.4.1")
    client = ModbusSerialClient(
        "/dev/ttys007",
        framer_name=Framer.RTU,
        baudrate=9600,
    )
    client.connect()
    assert client.connected

    start_time = time.time()
    for _i in range(LOOP_COUNT):
        rr = client.read_input_registers(1, REGISTER_COUNT, slave=1)
        if rr.isError():
            print(f"Received Modbus library error({rr})")
            break
    client.close()
    run_time = time.time() - start_time
    avg_call = (run_time / LOOP_COUNT) * 1000
    avg_register = avg_call / REGISTER_COUNT
    print(
        f"running {LOOP_COUNT} call (each {REGISTER_COUNT} registers), took {run_time:.
```

(continues on next page)

(continued from previous page)

```
↪2f} seconds"
    )
    print(f"Averages {avg_call:.2f} ms pr call and {avg_register:.2f} ms pr register.")


async def run_async_client_test():
    """Run async client."""
    print("--- Testing async client v3.4.1")
    client = AsyncModbusSerialClient(
        "/dev/ttys007",
        framer_name=Framer.RTU,
        baudrate=9600,
    )
    await client.connect()
    assert client.connected

    start_time = time.time()
    for _i in range(LOOP_COUNT):
        rr = await client.read_input_registers(1, REGISTER_COUNT, slave=1)
        if rr.isError():
            print(f"Received Modbus library error({rr})")
            break
    client.close()
    run_time = time.time() - start_time
    avg_call = (run_time / LOOP_COUNT) * 1000
    avg_register = avg_call / REGISTER_COUNT
    print(
        f"running {LOOP_COUNT} call (each {REGISTER_COUNT} registers), took {run_time:.
↪2f} seconds"
    )
    print(f"Averages {avg_call:.2f} ms pr call and {avg_register:.2f} ms pr register.")


if __name__ == "__main__":
    run_sync_client_test()
    asyncio.run(run_async_client_test())
```

6.2 Advanced examples

These examples are considered essential usage examples, and are guaranteed to work, because they are tested automatilly with each dev branch commit using CI.

Tip: The examples needs to be run from within the examples directory, unless you modify them. Most examples use helper.py and client_*.py or server_*.py. This is done to avoid maintaining the same code in multiple files.

- `examples.zip`
- `examples.tgz`

6.2.1 Client asynchronous calls

Source: examples/client_async_calls.py

Pymodbus Client modbus async all calls example.

Please see method **async_template_call** for a template on how to make modbus calls and check for different error conditions.

The handle* functions each handle a set of modbus calls with the same register type (e.g. coils).

All available modbus calls are present.

If you are performing a request that is not available in the client mixin, you have to perform the request like this instead:

```
from pymodbus.diag_message import ClearCountersRequest
from pymodbus.diag_message import ClearCountersResponse

request  = ClearCountersRequest()
response = client.execute(request)
if isinstance(response, ClearCountersResponse):
    ... do something with the response
```

This example uses client_async.py and client_sync.py to handle connection, and have the same options.

The corresponding server must be started before e.g. as:

./server_async.py

6.2.2 Client asynchronous

Source: examples/client_async.py

Pymodbus asynchronous client example.

usage:

```
client_async.py [-h] [-c {tcp,udp,serial,tls}]
                [-f {ascii,binary,rtu,socket,tls}]
                [-l {critical,error,warning,info,debug}] [-p PORT]
                [--baudrate BAUDRATE] [--host HOST]

-h, --help
    show this help message and exit
-c, -comm {tcp,udp,serial,tls}
    set communication, default is tcp
-f, --framer {ascii,binary,rtu,socket,tls}
    set framer, default depends on --comm
-l, --log {critical,error,warning,info,debug}
    set log level, default is info
-p, --port PORT
    set port
--baudrate BAUDRATE
    set serial device baud rate
--host HOST
    set host, default is 127.0.0.1
```

The corresponding server must be started before e.g. as:
python3 server_sync.py

6.2.3 Client calls

Source: examples/client_calls.py

Pymodbus Client modbus all calls example.

Please see method **template_call** for a template on how to make modbus calls and check for different error conditions.

The handle* functions each handle a set of modbus calls with the same register type (e.g. coils).

All available modbus calls are present.

If you are performing a request that is not available in the client mixin, you have to perform the request like this instead:

```
from pymodbus.diag_message import ClearCountersRequest
from pymodbus.diag_message import ClearCountersResponse

request  = ClearCountersRequest()
response = client.execute(request)
if isinstance(response, ClearCountersResponse):
    ... do something with the response
```

This example uses client_async.py and client_sync.py to handle connection, and have the same options.

The corresponding server must be started before e.g. as:

./server_async.py

6.2.4 Client custom message

Source: examples/client_custom_msg.py

Pymodbus Synchronous Client Examples.

The following is an example of how to use the synchronous modbus client implementation from pymodbus:

```
with ModbusClient("127.0.0.1") as client:
    result = client.read_coils(1,10)
    print result
```

6.2.5 Client payload

Source: examples/client_payload.py

Pymodbus Client Payload Example.

This example shows how to build a client with a complicated memory layout using builder.

Works out of the box together with payload_server.py

6.2.6 Client synchronous

Source: examples/client_sync.py

Pymodbus Synchronous Client Example.

An example of a single threaded synchronous client.

usage:

```
client_sync.py [-h] [-c {tcp,udp,serial,tls}]
                [-f {ascii,binary,rtu,socket,tls}]
                [-l {critical,error,warning,info,debug}] [-p PORT]
                [--baudrate BAUDRATE] [--host HOST]

-h, --help
    show this help message and exit
-c, --comm {tcp,udp,serial,tls}
    set communication, default is tcp
-f, --framer {ascii,binary,rtu,socket,tls}
    set framer, default depends on --comm
-l, --log {critical,error,warning,info,debug}
    set log level, default is info
-p, --port PORT
    set port
--baudrate BAUDRATE
    set serial device baud rate
--host HOST
    set host, default is 127.0.0.1
```

The corresponding server must be started before e.g. as:

python3 server_sync.py

6.2.7 Server asynchronous

Source: examples/server_async.py

Pymodbus asynchronous Server Example.

An example of a multi threaded asynchronous server.

usage:

```
server_async.py [-h] [--comm {tcp,udp,serial,tls}]
                [--framer {ascii,binary,rtu,socket,tls}]
                [--log {critical,error,warning,info,debug}]
                [--port PORT] [--store {sequential,sparse,factory,none}]
                [--slaves SLAVES]

-h, --help
    show this help message and exit
-c, --comm {tcp,udp,serial,tls}
    set communication, default is tcp
-f, --framer {ascii,binary,rtu,socket,tls}
    set framer, default depends on --comm
```

(continues on next page)

(continued from previous page)

```
-l, --log {critical,error,warning,info,debug}
    set log level, default is info
-p, --port PORT
    set port
    set serial device baud rate
--store {sequential,sparse,factory,none}
    set datastore type
--slaves SLAVES
    set number of slaves to respond to
```

The corresponding client can be started as:

python3 client_sync.py

6.2.8 Server callback

Source: examples/server_callback.py

Pymodbus Server With Callbacks.

This is an example of adding callbacks to a running modbus server when a value is written to it.

6.2.9 Server tracer

Source: examples/server_hook.py

Pymodbus Server With request/response manipulator.

This is an example of using the builtin request/response tracer to manipulate the messages to/from the modbus server

6.2.10 Server payload

Source: examples/server_payload.py

Pymodbus Server Payload Example.

This example shows how to initialize a server with a complicated memory layout using builder.

6.2.11 Server synchronous

Source: examples/server_sync.py

Pymodbus Synchronous Server Example.

An example of a single threaded synchronous server.

usage:

```
server_sync.py [-h] [--comm {tcp,udp,serial,tls}]
               [--framer {ascii,binary,rtu,socket,tls}]
               [--log {critical,error,warning,info,debug}]
               [--port PORT] [--store {sequential,sparse,factory,none}]
               [--slaves SLAVES]
```

(continues on next page)

(continued from previous page)

```
-h, --help
    show this help message and exit
-c, --comm {tcp,udp,serial,tls}
    set communication, default is tcp
-f, --framer {ascii,binary,rtu,socket,tls}
    set framer, default depends on --comm
-l, --log {critical,error,warning,info,debug}
    set log level, default is info
-p, --port PORT
    set port
    set serial device baud rate
--store {sequential,sparse,factory,none}
    set datastore type
--slaves SLAVES
    set number of slaves to respond to
```

The corresponding client can be started as:
python3 client_sync.py

REMARK It is recommended to use the async server! The sync server is just a thin cover on top of the async server and is in some aspects a lot slower.

6.2.12 Server updating

Source: examples/server_updating.py

Pymodbus asynchronous Server with updating task Example.

An example of an asynchronous server and a task that runs continuously alongside the server and updates values.

usage:

```
server_updating.py [-h] [--comm {tcp,udp,serial,tls}]
                   [--framer {ascii,binary,rtu,socket,tls}]
                   [--log {critical,error,warning,info,debug}]
                   [--port PORT] [--store {sequential,sparse,factory,none}]
                   [--slaves SLAVES]

-h, --help
    show this help message and exit
-c, --comm {tcp,udp,serial,tls}
    set communication, default is tcp
-f, --framer {ascii,binary,rtu,socket,tls}
    set framer, default depends on --comm
-l, --log {critical,error,warning,info,debug}
    set log level, default is info
-p, --port PORT
    set port
    set serial device baud rate
--store {sequential,sparse,factory,none}
    set datastore type
--slaves SLAVES
    set number of slaves to respond to
```

The corresponding client can be started as:
python3 client_sync.py

6.2.13 Simulator example

Source: examples/simulator.py

Pymodbus simulator server/client Example.

An example of how to use the simulator (server) with a client.

for usage see documentation of simulator

Tip: pymodbus.simulator starts the server directly from the commandline

6.2.14 Simulator datastore (shared storage) example

Source: examples/datastore_simulator_share.py

Pymodbus datastore simulator Example.

An example of using simulator datastore with json interface.

Detailed description of the device definition can be found at:

https://pymodbus.readthedocs.io/en/latest/source/library/simulator/config.html#device-entries

usage:

```
datastore_simulator_share.py [-h]
                    [--log {critical,error,warning,info,debug}]
                    [--port PORT]
                    [--test_client]

-h, --help
    show this help message and exit
-l, --log {critical,error,warning,info,debug}
    set log level
-p, --port PORT
    set port to use
--test_client
    starts a client to test the configuration
```

The corresponding client can be started as:
python3 client_sync.py

Tip: This is NOT the pymodbus simulator, that is started as pymodbus.simulator.

6.2.15 Message generator

Source: examples/message_generator.py

Modbus Message Generator.

6.2.16 Message Parser

Source: examples/message_parser.py

Modbus Message Parser.

The following is an example of how to parse modbus messages using the supplied framers.

6.2.17 Modbus forwarder

Source: examples/modbus_forwarder.py

Pymodbus synchronous forwarder.

This is a repeater or converter and an example of just how powerful datastore is.

It consist of a server (any comm) and a client (any comm), functionality:

a) server receives a read/write request from external client:

   - client sends a new read/write request to target server
   - client receives response and updates the datastore
   - server sends new response to external client

Both server and client are tcp based, but it can be easily modified to any server/client (see client_sync.py and server_sync.py for other communication types)

WARNING This example is a simple solution, that do only forward read requests.

6.3 Examples contributions

These examples are supplied by users of pymodbus. The pymodbus team thanks for sharing the examples.

6.3.1 Solar

Source: examples/contrib/solar.py

Pymodbus Synchronous Client Example.

Modified to test long term connection.

6.3.2 Redis datastore

Source: examples/contrib/redis_datastore.py

Datastore using redis.

6.3.3 Serial Forwarder

Source: examples/contrib/serial_forwarder.py

Pymodbus SerialRTU2TCP Forwarder

usage : python3 serial_forwarder.py –log DEBUG –port "/dev/ttyUSB0" –baudrate 9600 –server_ip "192.168.1.27" –server_port 5020 –slaves 1 2 3

6.3.4 Sqlalchemy datastore

Source: examples/contrib/sql_datastore.py

Datastore using SQL.

CHAPTER SEVEN

AUTHORS

All these versions would not be possible without volunteers!

This is a complete list for each major version.

A big “thank you” to everybody who helped out.

7.1 Pymodbus version 3 family

Thanks to

- AKJ7
- Alex
- Alex Ruddick
- Alexander Lanin
- Alexandre CUER
- Alois Hockenschlohe
- Arjan
- André Srinivasan
- banana-sun
- Blaise Thompson
- CapraTheBest
- cgernert
- corollaries
- Chandler Riehm
- Chris Hung
- Christian Krause
- dhoomakethu
- doelki
- DominicDataP
- Dries
- duc996

- Farzad Panahi
- Fredo70
- Gao Fang
- Ghostkeeper
- Hangyu Fan
- Hayden Roche
- Iktek
- Jakob Ruhe
- Jakob Schlyter
- James Braza
- James Hilliard
- jan iversen
- Jerome Velociter
- Joe Burmeister
- Jonathan Reichelt Gjertsen
- julian
- Justin Standring
- Kenny Johansson
- Matthias Straka
- laund
- Logan Gunthorpe
- Marko Luther
- Logan Gunthorpe
- Marko Luther
- Matthias Straka
- Mickaël Schoentgen
- Pavel Kostromitinov
- peufeu2
- Philip Couling
- Sebastian Machuca
- Sefa Keleş
- Steffen Beyer
- Thijs W
- Totally a booplicate
- WouterTuinstra
- wriswith

- yyokusa

7.2 Pymodbus version 2 family

Thanks to

- alecjohanson
- Alexey Andreyev
- Andrea Canidio
- Carlos Gomez
- Cougar
- Christian Sandberg
- dhoomakethu
- dices
- Dmitri Zimine
- Emil Vanherp
- er888kh
- Eric Duminil
- Erlend Egeberg Aasland
- hackerboygn
- Jian-Hong Pan
- Jose J Rodriguez
- Justin Searle
- Karl Palsson
- Kim Hansen
- Kristoffer Sjöberg
- Kyle Altendorf
- Lars Kruse
- Malte Kliemann
- Memet Bilgin
- Michael Corcoran
- Mike
- sanjay
- Sekenre
- Siarhei Farbotka
- Steffen Vogel
- tcplomp

- Thor Michael Støre
- Tim Gates
- Ville Skyttä
- Wild Stray
- Yegor Yefremov

7.3 Pymodbus version 1 family

Thanks to

- Antoine Pitrou
- Bart de Waal
- bashwork
- bje-
- Claudio Catterina
- Chintalagiri Shashank
- dhoomakethu
- dragoshenron
- Elvis Stansvik
- Eren Inan Canpolat
- Everley
- Fabio Bonelli
- fleimgruber
- francozappa
- Galen Collins
- Gordon Broom
- Hamilton Kibbe
- Hynek Petrak
- idahogray
- Ingo van Lil
- Jack
- jbiswas
- jon mills
- Josh Kelley
- Karl Palsson
- Matheus Frata
- Patrick Fuller

- Perry Kundert
- Philippe Gauthier
- Rahul Raghunath
- sanjay
- schubduese42
- semyont
- Semyon Teplitsky
- Stuart Longland
- Yegor Yefremov

7.4 Pymodbus version 0 family

Thanks to

- Albert Brandl
- Galen Collins

Import to github was based on code from:

- S.W.A.C. GmbH, Germany.
- S.W.A.C. Bohemia s.r.o., Czech Republic.
- Hynek Petrak
- Galen Collins

CHAPTER

EIGHT

CHANGELOG

All these version would not be possible without a lot of work from volunteers!

We, the maintainers, are greatful for each pull requests small or big, that helps make pymodbus a better product.

Authors: contains a complete list of volunteers have contributed to each major version.

8.1 Version 3.6.6

- Solve transport close() as not inherited method. (#2098)
- enable *mypy –check-untyped-defs* (#2096)
- Add get_expected_response_length to transaction.
- Remove control encode in framersRemove control encode in framers. (#2095)
- Bump codeql in CI to v3. (#2093)
- Improve server types (#2092)
- Remove pointless try/except (#2091)
- Improve transport types (#2090)
- Use explicit ValueError when called with incorrect function code (#2089)
- update message tests (incorporate all old tests). (#2088)
- Improve simulator type hints (#2084)
- Cleanup dead resetFrame code (#2082)
- integrate message.encode() into framer.buildPacket. (#2062)
- Repair client close() (intern= is needed for ModbusProtocol). (#2080)
- Updated Message_Parser example (#2079)
- Fix #2069 use released repl from pypi (#2077)
- Fix field encoding of Read File Record Response (#2075)
- Improve simulator types (#2076)
- Bump actions. (#2071)

8.2 Version 3.6.5

- Update framers to ease message integration (only decode/encode) (#2064)
- Add negtive acknowledge to modbus exceptions (#2065)
- add Message Socket/TLS and amend tests. (#2061)
- Improve factory types (#2060)
- ASCII. (#2054)
- Improve datastore documentation (#2056)
- Improve types for messages (#2058)
- Improve payload types (#2057)
- Reorganize datastore inheritance (#2055)
- Added new message (framer) raw + 100%coverage. (#2053)
- message classes, first step (#1932)
- Use AbstractMethod in transport. (#2051)
- A datastore for each slave. (#2050)
- Only run coverage in ubuntu / python 3.12 (#2049)
- Replace lambda with functools.partial in transport. (#2047)
- Move self.loop in transport to init() (#2046)
- Fix decoder bug (#2045)
- Add support for server testing in package_test_tool. (#2044)
- DictTransactionManager -> ModbusTransactionManager (#2042)
- eliminate redundant server_close() (#2041)
- Remove reactive server (REPL server). (#2038)
- Improve types for client (#2032)
- Improve HTTP server type hints (#2035)
- eliminate asyncio.sleep() and replace time.sleep() with a timeout (#2034)
- Use “new” inter_byte_timeout and is_open for pyserial (#2031)
- Add more type hints to datastore (#2028)
- Add more framer tests, solve a couple of framer problems. (#2024)
- Rework slow tests (use NULL_MODEM) (#1995)
- Allow slave=0 in serial communication. (#2023)
- Client package test tool. (#2022)
- Add REPL documentation back with links to REPL repo (#2017)
- Move repl to a seperate repo (#2009)
- solve more mypy issues with client (#2013)
- solve more mypy issues with datastore (#2010)

- Remove useless. (#2011)
- streamline transport tests. (#2004)
- Improve types for REPL (#2007)
- Specify more types in base framer (#2005)
- Move htmlcov -> build/cov (#2003)
- Avoid pylint complain about lambda. (#1999)
- Improve client types (#1997)
- Fix setblocking call (#1996)
- Actívate warnings in pytest. (#1994)
- Add profile option to pytest. (#1991)
- Simplify message tests (#1990)
- Upgrade pylint and ruff (#1989)
- Add first architecture document. (#1988)
- Update CONTRIBUTING.rst.
- Return None for broadcast. (#1987)
- Make ModbusClientMixin Generic to fix type issues for sync and async (#1980)
- remove strange None default (#1984)
- Fix incorrect bytearray type hint in diagnostics query (#1983)
- Fix URL to CHANGELOG (#1979)
- move server_hostname to be local in tls client. (#1978)
- Parameter "strict" is and was only used for serial server/client. (#1975)
- Removed unused parameter close_comm_on_error. (#1974)

8.3 Version 3.6.4

- Update datastore_simulator example with client (#1967)
- Test and correct receiving more than one packet (#1965)
- Remove unused FifoTransactionManager. (#1966)
- Always set exclusive serial port access. (#1964)
- Add server/client network stub, to allow test of network packets. (#1963)
- Combine conftest to a central file (#1962)
- Call on_reconnect_callback. (#1959)
- Readd ModbusBaseClient to external API.
- Update README.rst
- minor fix for typo and consistency (#1946)
- More coverage. (#1947)

- Client coverage 100%. (#1943)
- Run coverage in CI with % check of coverage. (#1945)
- transport 100% coverage. (#1941)
- contrib example: TCP drainage simulator with two devices (#1936)
- Remove “pragma no cover”. (#1935)
- transport_serial -> serialtransport. (#1933)
- Fix behavior after Exception response (#1931)
- Correct expected length for udp sync client. (#1930)

8.4 Version 3.6.3

- solve Socket_framer problem with Exception response (#1925)
- Allow socket frames to be split in multiple packets (#1923)
- Reset frame for serial connections.
- Source address None not 0.0.0.0 for IPv6
- Missing Copyright in License file
- Correct wrong url to modbus protocol spec.
- Fix serial port in TestComm.

8.5 Version 3.6.2

- Set documentation to v3.6.2.

8.6 Version 3.6.1

- Solve pypi upload error.

8.7 Version 3.6.0

- doc: Fix a code mismatch in client.rst
- Update README.
- truncated duration to milliseconds
- Update examples for current dev.
- Ignore all remaining implicit optional (#1888)
- docstring
- Remove unnecessary abort() call
- Enable RUF013 (implicit optional) (#1882)

- Support aiohttp 3.9.0b1 (#1886)
- Actually perform aiohttp runner teardown
- Pin to working aiohttp (#1884)
- Docstring typo cleanup (#1879)
- Clean client API imports. (#1819)
- Update issue template.
- Eliminiate implicit optional in reconnect_delay* (#1874)
- Split client base in sync/async version (#1878)
- Rework host/port and listener setup (#1866)
- use baudrate directly (#1872)
- Eliminate more implicit optional (#1871)
- Fix serial server args order (#1870)
- Relax test task/thread checker. (#1867)
- Make doc link references version dependent. (#1864)
- Remove pre-commit (#1860)
- Ruff reduce ignores. (#1862)
- Bump ruff to 0.1.3 and remove ruff.toml (#1861)
- More elegant noop. (#1859)
- Cache (#1829)
- Eliminate more implicit optional (#1858)
- Ignore files downloaded by pytest (#1857)
- Avoid malicious user path input (#1855)
- Add more return types to transport (#1852)
- Do not attempt to close an already-closed serial connection (#1853)
- Fix stopbits docstring typo (#1850)
- Convert type hints to PEP585 (#1846)
- Eliminate even more implicit optional (#1845)
- Eliminate more implicit optionals in client (#1844)
- Eliminate implicit optional in transport_serial (#1843)
- Make client type annotations compatible with async client usage (#1842)
- Merge pull request #1838 from pymodbus-dev/ruff
- Eliminate implicit optional in simulator (#1841)
- eliminate implicit optional for callback_disconnected (#1840)
- pre-commit run –all-files
- Update exclude paths
- Replace black with ruff

- Use other dependency groups for 'all' (#1834)
- Cleanup author/maintainer fields (#1833)
- Consistent messages if imports fail (#1831)
- Client/Server framer as enum. (#1822)
- Solve relative path in examples. (#1828)
- Eliminate implicit optional for CommParams types (#1825)
- Add 3.12 classifier (#1826)
- Bump actions/stale to 8.0.0 (#1824)
- Cleanup paths included in mypy/pylint (#1823)
- Client documentation amended and updated. (#1820)
- Import aiohttp in way pleasing mypy. (#1818)
- Update doc, remove md files. (#1814)
- Bump dependencies. (#1816)
- Solve pylint / pytest.
- fix pylint.
- Examples are without parent module.
- Wrong zip of examples.
- Serial delay (#1810)
- Add python 3.12. (#1800)
- Release errors (pyproject.toml changes). (#1811)

8.8 Version 3.5.4

- Release errors (pyproject.toml changes). (#1811)

8.9 Version 3.5.3

- Simplify transport_serial (modbus use) (#1808)
- Reduce transport_serial (#1807)
- Change to pyproject.toml. (#1805)
- fixes access to asyncio loop via loop property of SerialTransport (#1804)
- Bump aiohttp to support python 3.12. (#1802)
- README wrong links. (#1801)
- CI caching. (#1796)
- Solve pylint unhappy. (#1799)
- Clean except last 7 days. (#1798)
- Reconect_delay == 0, do not reconnect. (#1795)

- Update simulator.py method docstring (#1793)
- add type to isError. (#1781)
- Allow repr(ModbusException) to return complete information (#1779)
- Update docs. (#1777)

8.10 Version 3.5.2

- server tracer example. (#1773)
- sync connect missing. (#1772)
- simulator future problem. (#1771)

8.11 Version 3.5.1

- Always close socket on error (reset_sock). (#1767)
- Revert reset_socket change.
- add close_comm_on_error to example.
- Test long term (HomeAsistant problem). (#1765)
- Update ruff to 0.0.287 (#1764)
- Remove references to ModbusSerialServer.start (#1759) (#1762)
- Readd test to get 100% coverage.
- transport: Don't raise a RunTimeError in ModbusProtocol.error_received() (#1758)

8.12 Version 3.5.0

- Async retry (#1752)
- test_client: Fix test_client_protocol_execute() (#1751)
- Use enums for constants (#1743)
- Local Echo Broadcast with Async Clients (#1744)
- Fix #1746 . Return missing result (#1748)
- Document nullmodem. (#1739)
- Add system health check to all tests. (#1736)
- Handle partial message in ReadDeviceInformationResponse (#1738)
- Broadcast with Handle Local Echo (#1737)
- transport_emulator, part II. (#1710)
- Added file AUTHORS, to list all Volunteers. (#1734)
- Fix #1702 and #1728 (#1733)
- Clear retry count when success. (#1732)

- RFC: Reduce parameters for REPL server classes (#1714)
- retries=1, solved. (#1731)
- Impoved the example “server_updating.py” (#1720)
- pylint 3.11 (#1730)
- Correct retry loop. (#1729)
- Fix faulty not check (#1725)
- bugfix local echo handling on sync clients (#1723)
- Updated copyright in LICENSE.
- Correct README pre-commit.
- Fix custom message parsing in RTU framer (#1716)
- Request tracer (#1715)
- pymodbus.server: allow strings for “-p” paramter (#1713)
- New nullmodem and transport. (#1696)
- xdist loadscope (test is not split). (#1708)
- Add client performance example. (#1707)

8.13 Version 3.4.1

- Fix serial startup problems. (#1701)
- pass source_address in tcp client. (#1700)
- serial server use source_address[0]. (#1699)
- Examples coverage nearly 100%. (#1694)
- new async serial (#1681)
- Docker is not supported (lack of maintainer). (#1693)
- Forwarder write_coil –> write_coil. (#1691)
- Change default source_address to (0.0.0.0, 502) (#1690)
- Update ruff to 0.0.277 (#1689)
- Fix dict comprehension (#1687)
- Removed *requests* dependency from *contrib/explain.py* (#1688)
- Fix broken test (#1685)
- Fix readme badges (#1682)
- Bump aiohttp from 3.8.3 to 3.8.5 (#1680)
- pygments from 2.14.0 to 2.15.0 (#1677)

8.14 Version 3.4.0

- Handle partial local echo. (#1675)
- clarify handle_local_echo. (#1674)
- async_client: add retries/reconnect. (#1672)
- Fix 3.11 problem. (#1673)
- Add new example simulator server/client. (#1671)
- *examples/contrib/explain.py* leveraging Rapid SCADA (#1665)
- _logger missed basicConfig. (#1670)
- Bug fix for #1662 (#1663)
- Bug fix for #1661 (#1664)
- Fix typo in config.rst (#1660)
- test action_increment. (#1659)
- test codeql (#1655)
- mypy complaints. (#1656)
- Remove self.params from async client (#1640)
- Drop test of pypy with python 3.8.
- repair server_async.py (#1644)
- move common framer to base. (#1639)
- Restrict Return diag call to bytes. (#1638)
- use slave= in diag requests. (#1636)
- transport listen in server. (#1628)
- CI test.
- Integrate transport in server. (#1617)
- fix getFrameStart for ExceptionResponse (#1627)
- Add min/min to simulator actions.
- Change to "sync client" in forwarder example (#1625)
- Remove docker (lack of maintenance). (#1623)
- Clean defaults (#1618)
- Reduce CI log with no debug. (#1616)
- prepare server to use transport. (#1607)
- Fix RemoteSlaveContext (#1599)
- Combine stale and lock. (#1608)
- update pytest + extensions. (#1610)
- Change version follow PEP 440. (#1609)
- Fix regression with REPL server not listening (#1604)

- Remove handler= for server classes. (#1602)
- Fix write function codes (#1598)
- transport nullmodem (#1591)
- move test of examples to subdirectory. (#1592)
- transport as object, not base class. (#1572)
- Simple examples. (#1590)
- transport_connect as bool. (#1587)
- Prepare dev (#1588)
- Release corrections. (#1586)

8.15 Version 3.3.2

- Fix RemoteSlaveContext (#1599)
- Change version follow PEP 440. (#1609)
- Fix regression with REPL server not listening (#1604)
- Fix write function codes (#1598)
- Release corrections. (#1586)

8.16 Version 3.3.1

- transport fixes and 100% test coverage. (#1580)
- Delay self.loop until connect(). (#1579)
- Added mechanism to determine if server did not start cleanly (#1539)
- Proof transport reconnect works. (#1577)
- Fix non-shared block doc in config.rst. (#1573)

8.17 Version 3.3.0

- Stabilize windows tests. (#1567)
- Bump mypy 1.3.0 (#1568)
- Transport integrated in async clients. (#1541)
- Client async corrections (due to 3.1.2) (#1565)
- Server_async[udp], solve 3.1.1 problem. (#1564)
- Remove ModbusTcpDiagClient. (#1560)
- Remove old method from Python2/3 transition (#1559)
- Switch to ruff's version of bandit (#1557)
- Allow reading/writing address 0 in the simulator (#1552)

- Remove references to “defer_start”. (#1548)
- Client more robust against faulty response. (#1547)
- Fix missing package_data directives for simulator web (#1544)
- Fix installation instructions (#1543)
- Solve pytest timeout problem. (#1540)
- DiagnosticStatus encode missing tuple check. (#1533)
- test SparseDataStore. (#1532)
- BinaryPayloadBuilder.to_string to BinaryPayloadBuilder.encode (#1526)
- Adding flake8-pytest-style` to ruff (#1520)
- Simplify version management. (#1522)
- pylint and pre-commit autoupdate (#1519)
- Add type hint (#1512)
- Add action to lock issues/PR. (#1508)
- New common transport layer. (#1492)
- Solve serial close raise problem.
- Remove old config values (#1503)
- Document pymodbus.simulator. (#1502)
- Refactor REPL server to reduce complexity (#1499)
- Don’t catch KeyboardInterrupt twice for REPL server (#1498)
- Refactor REPL client to reduce complexity (#1489)
- pymodbus.server: listen on ID 1 by default (#1496)
- Clean framer/__init__.py (#1494)
- Duplicate transactions in UDP. (#1486)
- clean ProcessIncommingPacket. (#1491)
- Enable pyupgrade (U) rules in ruff (#1484)
- clean_workflow.yaml solve parameter problem.
- Correct wrong import in test. (#1483)
- Implement pyflakes-simplify (#1480)
- Test case for UDP duplicate msg issue (#1470)
- Test of write_coil. (#1479)
- Test reuse of client object. (#1475)
- Comment about addressing when shared=false (#1474)
- Remove old aliases to OSError (#1473)
- pymodbus.simulator fixes (#1463)
- Fix wrong error message with pymodbus console (#1456)
- update modbusrtuframer (#1435)

- Server multidrop test.: (#1451)
- mypy problem ModbusResponse.

8.18 Version 3.2.2

- Add forgotten await

8.19 Version 3.2.1

- add missing server.start(). (#1443)
- Don't publish univeral (Python2 / Python 3) wheels (#1423)
- Remove unneccesary custom LOG_LEVEL check (#1424)
- Include py.typed in package (#1422)

8.20 Version 3.2.0

- Add value <-> registers converter helpers. (#1413)
- Add pre-commit config (#1406)
- Make baud rate configurable for examples (#1410)
- Clean __init_ and update log module. (#1411)
- Simulator add calls functionality. (#1390)
- Add note about not being thread safe. (#1404)
- Update docker-publish.yml
- Forward retry_on_empty and retries by calling transaction (#1401)
- serial sync recv interval (#1389)
- Add tests for writing multiple writes with a single value (#1402)
- Enable mypy in CI (#1388)
- Limit use of Singleton. (#1397)
- Cleanup interfaces (#1396)
- Add request names. (#1391)
- Simulator, register look and feel. (#1387)
- Fix enum for REPL server (#1384)
- Remove unneeded attribute (#1383)
- Fix mypy errors in reactive server (#1381)
- remove nosec (#1379)
- Fix type hints for http_server (#1369)
- Merge pull request #1380 from pymodbus-dev/requirements

- remove second client instance in async mode. (#1367)
- Pin setuptools to prevent breakage with Version including “X” (#1373)
- Lint and type hints for REPL (#1364)
- Clean mixin execute (#1366)
- Remove unused setup_commands.py. (#1362)
- Run black on top-level files and /doc (#1361)
- repl config path (#1359)
- Fix NoReponse -> NoResponse (#1358)
- Make whole main async. (#1355)
- Fix more typing issues (#1351)
- Test sync task (#1341)
- Fixed text in ModbusClientMixin’s writes (#1352)
- lint /doc (#1345)
- Remove unused linters (#1344)
- Allow log level as string or integer. (#1343)
- Sync serial, clean recv. (#1340)
- Test server task, async completed (#1318)
- main() should be sync (#1339)
- Bug: Fixed caused by passing wrong arg (#1336)

8.21 Version 3.1.3

- Solve log problem in payload.
- Fix register type check for size bigger than 3 registers (6 bytes) (#1323)
- Re-add SQL tests. (#1329)
- Central logging. (#1324)
- Skip sqlAlchemy test. (#1325)
- Solve 1319 (#1320)

8.22 Version 3.1.2

- Update README.rst
- Correct README link. (#1316)
- More direct readme links for REPL (#1314)
- Add classifier for 3.11 (#1312)
- Update README.rst (#1313)
- Delete ModbusCommonBlock.png (#1311)

- Add modbus standard to README. (#1308)
- fix no auto reconnect after close/connect in TCPclient (#1298)
- Update examples.rst (#1307)
- var name clarification (#1304)
- Bump external libraries. (#1302)
- Reorganize documentation to make it easier accessible (#1299)
- Simulator documentation (first version). (#1296)
- Updated datastore Simulator. (#1255)
- Update links to pydmodbus-dev (#1291)
- Change riptideio to pymodbus-dev. (#1292)
- #1258 Avoid showing unit as a seperate command line argument (#1288)
- Solve docker cache problem. (#1287)

8.23 Version 3.1.1

- add missing server.start() (#1282)
- small performance improvement on debug log (#1279)
- Fix Unix sockets parsing (#1281)
- client: Allow unix domain socket. (#1274)
- transfer timeout to protocol object. (#1275)
- Add ModbusUnixServer / StartAsyncUnixServer. (#1273)
- Added return in AsyncModbusSerialClient.connect (#1271)
- add connect() to the very first example (#1270)
- Solve docker problem. (#1268)
- Test stop of server task. (#1256)

8.24 Version 3.1.0

- Add xdist pr default. (#1253)
- Create docker-publish.yml (#1250)
- Parallelize pytest with pytest-xdist (#1247)
- Support Python3.11 (#1246)
- Fix reconnectDelay to be within (100ms, 5min) (#1244)
- Fix typos in comments (#1233)
- WEB simulator, first version. (#1226)
- Clean async serial problem. (#1235)
- terminate when using 'randomize' and 'change_rate' at the same time (#1231)

- Used tooled python and OS (#1232)
- add 'change_rate' randomization option (#1229)
- add check_ci.sh (#1225)
- Simplify CI and use cache. (#1217)
- Solve issue 1210, update simulator (#1211)
- Add missing client calls in mixin.py. (#1206)
- Advanced simulator with cross memory. (#1195)
- AsyncModbusTcp/UdpClient honors delay_ms == 0 (#1203) (#1205)
- Fix #1188 and some pylint issues (#1189)
- Serial receive incomplete bytes.issue #1183 (#1185)
- Handle echo (#1186)
- Add updating server example. (#1176)

8.25 Version 3.0.2

- Add pygments as requirement for repl
- Update datastore remote to handle write requests (#1166)
- Allow multiple servers. (#1164)
- Fix typo. (#1162)
- Transfer parms. to connected client. (#1161)
- Repl enhancements 2 (#1141)
- Server simulator with datastore with json data. (#1157)
- Avoid unwanted reconnects (#1154)
- Do not initialize framer twice. (#1153)
- Allow timeout as float. (#1152)
- Improve Docker Support (#1145)
- Fix unreachable code in AsyncModbusTcpClient (#1151)
- Fix type hints for port and timeout (#1147)
- Start/stop multiple servers. (#1138)
- Server/asyncio.py correct logging when disconnecting the socket (#1135)
- Add Docker and container registry support (#1132)
- Removes undue reported error when forwarding (#1134)
- Obey timeout parameter on connection (#1131)
- Readme typos (#1129)
- Clean noqa directive. (#1125)
- Add isort and activate CI fail for black/isort. (#1124)

- Update examples. (#1117)
- Move logging configuration behind function call (#1120)
- serial2TCP forwarding example (#1116)
- Make serial import dynamic. (#1114)
- Bugfix ModbusSerialServer setup so handler is called correctly. (#1113)
- Clean configurations. (#1111)

8.26 Version 3.0.1

- Faulty release!

8.27 Version 3.0.0

- Solve multiple incomming frames. (#1107)
- Up coverage, tests are 100%. (#1098)
- Prepare for rc1. (#1097)
- Prepare 3.0.0dev5 (#1095)
- Adapt serial tests. (#1094)
- Allow windows. (#1093)
- Remove server sync code and combine with async code. (#1092)
- Solve test of tls by adding certificates and remove bugs (#1080)
- Simplify server implementation. (#1071)
- Do not filter using unit id in the received response (#1076)
- Hex values for repl arguments (#1075)
- All parameters in class parameter. (#1070)
- Add len parameter to decode_bits. (#1062)
- New combined test for all types of clients. (#1061)
- Dev mixin client (#1056)
- Add/update client documentation, including docstrings etc. (#1055)
- Add unit to arguments (#1041)
- Add timeout to all pytest. (#1037)
- Simplify client parent classes. (#1018)
- Clean copyright statements, to ensure we follow FOSS rules. (#1014)
- Rectify sync/async client parameters. (#1013)
- Clean client directory structure for async. (#1010)
- Remove async_io, simplify AsyncModbus<x>Client. (#1009)
- remove init_<something>_client(). (#1008)

- Remove async factory. (#1001)
- Remove loop parameter from client/server (#999)
- add example async client. (#997)
- Change async ModbusSerialClient to framer= from method=. (#994)
- Add forwarder example with multiple slaves. (#992)
- Remove async get_factory. (#990)
- Remove unused ModbusAccessControl. (#989)
- Solve problem with remote datastore. (#988)
- Remove unused schedulers. (#976)
- Remove twisted (#972)
- Remove/Update tornado/twister tests. (#971)
- remove easy_install and ez_setup (#964)
- Fix mask write register (#961)
- Activate pytest-asyncio. (#949)
- Changed default framer for serial to be ModbusRtuFramer. (#948)
- Remove tornado. (#935)
- Pylint, check method parameter documentation. (#909)
- Add get_response_pdu_size to mask read/write. (#922)
- Minimum python version is 3.8. (#921)
- Ensure make doc fails on warnings and/or errors. (#920)
- Remove central makefile. (#916)
- Re-organize examples (#914)
- Documentation cleanup and clarification (#689)
- Update doc for repl. (#910)
- Include package and tests in coverage measurement (#912)
- Use response byte length if available (#880)
- better fix for rtu incomplete frames (#511)
- Remove twisted/tornado from doc. (#904)
- Update classifiers for pypi. (#907)
- Documentation updates
- PEP8 compatibale code
- More tooling and CI updates
- Remove python2 compatibility code (#564)
- Remove Python2 checks and Python2 code snippets
- Misc co-routines related fixes
- Fix CI for python3 and remove PyPI from CI

- Fix mask_write_register call. (#685)
- Add support for byte strings in the device information fields (#693)
- Catch socket going away. (#722)
- Misc typo errors (#718)
- Support python3.10
- Implement asyncio ModbusSerialServer
- ModbusTLS updates (tls handshake, default framer)
- Support broadcast messages with asyncio client
- Fix for lazy loading serial module with asyncio clients.
- Updated examples and tests
- Support python3.7 and above
- Support creating asyncio clients from with in coroutines.

8.28 Version 2.5.3

- Fix retries on tcp client failing randomly.
- Fix Asyncio client timeout arg not being used.
- Treat exception codes as valid responses
- Fix examples (modbus_payload)
- Add missing identity argument to async ModbusSerialServer

8.29 Version 2.5.2

- Add kwarg *reset_socket* to control closing of the socket on read failures (set to *True* by default).
- Add *–reset-socket/–no-reset-socket* to REPL client.

8.30 Version 2.5.1

- Bug fix TCP Repl server.
- Support multiple UID’s with REPL server.
- Support serial for URL (sync serial client)
- Bug fix/enhancements, close socket connections only on empty or invalid response

8.31 Version 2.5.0

- Support response types *stray* and *empty* in repl server.
- Minor updates in asyncio server.
- Update reactive server to send stray response of given length.
- Transaction manager updates on retries for empty and invalid packets.
- Test fixes for asyncio client and transaction manager.
- Fix sync client and processing of incomplete frames with rtu framers
- Support synchronous diagnostic client (TCP)
- Server updates (REPL and async)
- Handle Memory leak in sync servers due to socketserver memory leak
- Minor fix in documentations
- Travis fix for Mac OSX
- Disable unnecessary deprecation warning while using async clients.
- Use Github actions for builds in favor of travis.
- Documentation updates
- Disable *strict* mode by default.
- Fix *ReportSlaveIdRequest* request
- Sparse datablock initialization updates.
- Support REPL for modbus server (only python3 and asyncio)
- Fix REPL client for write requests
- Fix examples
- Asyncio server
- Asynchronous server (with custom datablock)
- Fix version info for servers
- Fix and enhancements to Tornado clients (seril and tcp)
- Fix and enhancements to Asyncio client and server
- Update Install instructions
- Synchronous client retry on empty and error enhancments
- Add new modbus state *RETRYING*
- Support runtime response manipulations for Servers
- Bug fixes with logging module in servers
- Asyncio modbus serial server support

8.32 Version 2.4.0

- Support async moduls tls server/client
- Add local echo option
- Add exponential backoffs on retries.
- REPL - Support broadcasts.
- Fix framers using wrong unit address.
- Update documentation for serial_forwarder example
- Fix error with rtu client for *local_echo*
- Fix asyncio client not working with already running loop
- Fix passing serial arguments to async clients
- Support timeouts to break out of responspe await when server goes offline
- Misc updates and bugfixes.

8.33 Version 2.3.0

- Support Modbus TLS (client / server)
- Distribute license with source
- BinaryPayloadDecoder/Encoder now supports float16 on python3.6 and above
- Fix asyncio UDP client/server
- Minor cosmetic updates
- Asyncio Server implementation (Python 3.7 and above only)
- Bug fix for DiagnosticStatusResponse when odd sized response is received
- Remove Pycrypto from dependencies and include cryptodome instead
- Remove *SIX* requirement pinned to exact version.
- Minor bug-fixes in documentations.

8.34 Version 2.2.0

- Support Python 3.7
- Fix to task cancellations and CRC errors for async serial clients.
- Fix passing serial settings to asynchronous serial server.
- Fix *AttributeError* when setting *interCharTimeout* for serial clients.
- Provide an option to disable inter char timeouts with Modbus RTU.
- Add support to register custom requests in clients and server instances.
- Fix read timeout calculation in ModbusTCP.
- Fix SQLDbcontext always returning InvalidAddress error.

- Fix SQLDbcontext update failure
- Fix Binary payload example for endianess.
- Fix BinaryPayloadDecoder.to_coils and BinaryPayloadBuilder.fromCoils methods.
- Fix tornado async serial client *TypeError* while processing incoming packet.
- Fix erroneous CRC handling in Modbus RTU framer.
- Support broadcasting in Modbus Client and Servers (sync).
- Fix asyncio examples.
- Improved logging in Modbus Server .
- ReportSlaveIdRequest would fetch information from Device identity instead of hardcoded *Pymodbus*.
- Fix regression introduced in 2.2.0rc2 (Modbus sync client transaction failing)
- Minor update in factory.py, now server logs prints received request instead of only function code

8.35 Version 2.1.0

- Fix Issues with Serial client where in partial data was read when the response size is unknown.
- Fix Infinite sleep loop in RTU Framer.
- Add pygments as extra requirement for repl.
- Add support to modify modbus client attributes via repl.
- Update modbus repl documentation.
- More verbose logs for repl.

8.36 Version 2.0.1

- Fix unicode decoder error with BinaryPayloadDecoder in some platforms
- Avoid unnecessary import of deprecated modules with dependencies on twisted

8.37 Version 2.0.0

- Async client implementation based on Tornado, Twisted and asyncio with backward compatibility support for twisted client.
- Allow reusing existing[running] asyncio loop when creating async client based on asyncio.
- Allow reusing address for Modbus TCP sync server.
- Add support to install tornado as extra requirement while installing pymodbus.
- Support Pymodbus REPL
- Add support to python 3.7.
- Bug fix and enhancements in examples.
- Async client implementation based on Tornado, Twisted and asyncio

8.38 Version 1.5.2

- Fix serial client *is_socket_open* method

8.39 Version 1.5.1

- Fix device information selectors
- Fixed behaviour of the MEI device information command as a server when an invalid object_id is provided by an external client.
- Add support for repeated MEI device information Object IDs (client/server)
- Added support for encoding device information when it requires more than one PDU to pack.
- Added REPR statements for all syncchronous clients
- Added *isError* method to exceptions, Any response received can be tested for success before proceeding.
- Add examples for MEI read device information request

8.40 Version 1.5.0

- Improve transaction speeds for sync clients (RTU/ASCII), now retry on empty happens only when retry_on_empty kwarg is passed to client during intialization
- Fix tcp servers (sync/async) not processing requests with transaction id > 255
- Introduce new api to check if the received response is an error or not (response.isError())
- Move timing logic to framers so that irrespective of client, correct timing logics are followed.
- Move framers from transaction.py to respective modules
- Fix modbus payload builder and decoder
- Async servers can now have an option to defer *reactor.run()* when using *Start<Tcp/Serial/Udo>Server(…,defer_reactor_run=True)*
- Fix UDP client issue while handling MEI messages (ReadDeviceInformationRequest)
- Add expected response lengths for WriteMultipleCoilRequest and WriteMultipleRegisterRequest
- Fix _rtu_byte_count_pos for GetCommEventLogResponse
- Add support for repeated MEI device information Object IDs
- Fix struct errors while decoding stray response
- Modbus read retries works only when empty/no message is received
- Change test runner from nosetest to pytest
- Fix Misc examples

8.41 Version 1.4.0

- Bug fix Modbus TCP client reading incomplete data
- Check for slave unit id before processing the request for serial clients
- Bug fix serial servers with Modbus Binary Framer
- Bug fix header size for ModbusBinaryFramer
- Bug fix payload decoder with endian Little
- Payload builder and decoder can now deal with the wordorder as well of 32/64 bit data.
- Support Database slave contexts (SqlStore and RedisStore)
- Custom handlers could be passed to Modbus TCP servers
- Asynchronous Server could now be stopped when running on a seperate thread (StopServer)
- Signal handlers on Asynchronous servers are now handled based on current thread
- Registers in Database datastore could now be read from remote clients
- Fix examples in contrib (message_parser.py/message_generator.py/remote_server_context)
- Add new example for SqlStore and RedisStore (db store slave context)
- Fix minor comaptibility issues with utilities.
- Update test requirements
- Update/Add new unit tests
- Move twisted requirements to extra so that it is not installed by default on pymodbus installtion

8.42 Version 1.3.2

- ModbusSerialServer could now be stopped when running on a seperate thread.
- Fix issue with server and client where in the frame buffer had values from previous unsuccesful transaction
- Fix response length calculation for ModbusASCII protocol
- Fix response length calculation ReportSlaveIdResponse, DiagnosticStatusResponse
- Fix never ending transaction case when response is received without header and CRC
- Fix tests

8.43 Version 1.3.1

- Recall socket recv until get a complete response
- Register_write_message.py: Observe skip_encode option when encoding a single register request
- Fix wrong expected response length for coils and discrete inputs
- Fix decode errors with ReadDeviceInformationRequest and ReportSlaveIdRequest on Python3
- Move MaskWriteRegisterRequest/MaskWriteRegisterResponse to register_write_message.py from file_message.py

- Python3 compatible examples [WIP]
- Misc updates with examples
- Fix encoding problem for ReadDeviceInformationRequest method on python3
- Fix problem with the usage of ord in python3 while cleaning up receive buffer
- Fix struct unpack errors with BinaryPayloadDecoder on python3 - string vs bytestring error
- Calculate expected response size for ReadWriteMultipleRegistersRequest
- Enhancement for ModbusTcpClient, ModbusTcpClient can now accept connection timeout as one of the parameter
- Misc updates
- Timing improvements over MODBUS Serial interface
- Modbus RTU use 3.5 char silence before and after transactions
- Bug fix on FifoTransactionManager , flush stray data before transaction
- Update repository information
- Added ability to ignore missing slaves
- Added ability to revert to ZeroMode
- Passed a number of extra options through the stack
- Fixed documenation and added a number of examples

8.44 Version 1.2.0

- Reworking the transaction managers to be more explicit and to handle modbus RTU over TCP.
- Adding examples for a number of unique requested use cases
- Allow RTU framers to fail fast instead of staying at fault
- Working on datastore saving and loading

8.45 Version 1.1.0

- Fixing memory leak in clients and servers (removed __del__)
- Adding the ability to override the client framers
- Working on web page api and GUI
- Moving examples and extra code to contrib sections
- Adding more documentation

8.46 Version 1.0.0

- Adding support for payload builders to form complex encoding and decoding of messages.
- Adding BCD and binary payload builders
- Adding support for pydev
- Cleaning up the build tools
- Adding a message encoding generator for testing.
- Now passing kwargs to base of PDU so arguments can be used correctly at all levels of the protocol.
- A number of bug fixes (see bug tracker and commit messages)

CHAPTER

NINE

API CHANGES

Versions (X.Y.Z) where Z > 0 e.g. 3.0.1 do NOT have API changes!

9.1 API changes 3.6.0

- framer= is an enum: pymodbus.Framer, but still accept a framer class

9.2 API changes 3.5.0

- Remove handler parameter from ModbusUdpServer
- Remove loop parameter from ModbusSerialServer
- Remove handler and allow_reuse_port from repl default config
- Static classes from the `constants` module are now inheriting from `enum.Enum` and using *UPPER_CASE* naming scheme, this affects: `- MoreData - DeviceInformation - ModbusPlusOperation - Endian - ModbusStatus`
- Async clients now accepts *no_resend_on_retry=True*, to not resend the request when retrying.
- ModbusSerialServer now accepts request_tracer=.

9.3 API changes 3.4.0

- Modbus<x>Client .connect() returns True/False (connected or not)
- Modbue<x>Server handler=, allow_reuse_addr=, backlog= are no longer accepted
- ModbusTcpClient / AsyncModbusTcpClient no longer support unix path
- StartAsyncUnixServer / ModbusUnixServer removed (never worked on Windows)
- ModbusTlsServer reqclicert= is no longer accepted
- ModbusSerialServer auto_connect= is no longer accepted
- ModbusSimulatorServer.serve_forever(only_start=False) added to allow return

9.4 API changes 3.3.0

- ModbusTcpDiagClient is removed due to lack of support
- Clients have an optional parameter: on_reconnect_callback, Function that will be called just before a reconnection attempt.
- general parameter unit= -> slave=
- move SqlSlaveContext, RedisSlaveContext to examples/contrib (due to lack of maintenance)
- `BinaryPayloadBuilder.to_string` was renamed to `BinaryPayloadBuilder.encode`
- on_reconnect_callback for async clients works slightly different
- utilities/unpack_bitstring now expects an argument named *data* not *string*

9.5 API changes 3.2.0

- helper to convert values in mixin: convert_from_registers, convert_to_registers
- import pymodbus.version -> from pymodbus import __version__, __version_full__
- pymodbus.pymodbus_apply_logging_config(log_file_name="pymodbus.log") to enable file pymodbus_apply_logging_config
- pymodbus.pymodbus_apply_logging_config have default DEBUG, it not called root settings will be used.
- pymodbus/interfaces/IModbusDecoder removed.
- pymodbus/interfaces/IModbusFramer removed.
- pymodbus/interfaces/IModbusSlaveContext -> pymodbus/datastore/ModbusBaseSlaveContext.
- StartAsync<type>Server, removed defer_start argument, return is None. instead of using defer_start instantiate the Modbus<type>Server directly.
- *ReturnSlaveNoReponseCountResponse* has been corrected to *ReturnSlaveNoResponseCountResponse*
- Option *–modbus-config* for REPL server renamed to *–modbus-config-path*
- client.protocol.<something> –> client.<something>
- client.factory.<something> –> client.<something>

9.6 API changes 3.1.0

- Added –host to client_* examples, to allow easier use.
- unit= in client calls are no longer converted to slave=, but raises a runtime exception.
- Added missing client calls (all standard request are not available as methods).
- client.mask_write_register() changed parameters.
- server classes no longer accept reuse_port= (the socket do not accept it)

9.7 API changes 3.0.0

Base for recording changes.

CHAPTER

TEN

PYMODBUS INTERNALS

10.1 NullModem

Pymodbus offers a special NullModem transport to help end-to-end test without network.

The NullModem is activated by setting host= (port= for serial) to NULLMODEM_HOST (import pymodbus.transport)

The NullModem works with the normal transport types, and simply substitutes the physical connection: - *Serial* (RS-485) typically using a dongle - *TCP* - *TLS* - *UDP*

The NullModem is currently integrated in - `Modbus<x>Client` - `AsyncModbus<x>Client` - `Modbus<x>Server` - `AsyncModbus<x>Server`

Of course the NullModem requires that server and client(s) run in the same python instance.

10.2 Datastore

Datastore is responsible for managing registers for a server.

10.2.1 Datastore classes

class `pymodbus.datastore.`**`ModbusSparseDataBlock`**(*values=None*, *mutable=True*)

A sparse modbus datastore.

E.g Usage. sparse = ModbusSparseDataBlock({10: [3, 5, 6, 8], 30: 1, 40: [0]*20})

This would create a datablock with 3 blocks One starts at offset 10 with length 4, one at 30 with length 1, and one at 40 with length 20

sparse = ModbusSparseDataBlock([10]*100) Creates a sparse datablock of length 100 starting at offset 0 and default value of 10

sparse = ModbusSparseDataBlock() –> Create empty datablock sparse.setValues(0, [10]*10) –> Add block 1 at offset 0 with length 10 (default value 10) sparse.setValues(30, [20]*5) –> Add block 2 at offset 30 with length 5 (default value 20)

Unless 'mutable' is set to True during initialization, the datablock cannot be altered with setValues (new datablocks cannot be added)

classmethod create(*values=None*)

Create sparse datastore.

Use setValues to initialize registers.

Parameters
`values` – Either a list or a dictionary of values

Returns
An initialized datastore

`reset()`

Reset the store to the initially provided defaults.

`validate`(*address*, *count=1*)

Check to see if the request is in range.

Parameters

- `address` – The starting address
- `count` – The number of values to test for

Returns
True if the request in within range, False otherwise

`getValues`(*address*, *count=1*)

Return the requested values of the datastore.

Parameters

- `address` – The starting address
- `count` – The number of values to retrieve

Returns
The requested values from a:a+c

`setValues`(*address*, *values*, *use_as_default=False*)

Set the requested values of the datastore.

Parameters

- `address` – The starting address
- `values` – The new values to be set
- `use_as_default` – Use the values as default

Raises
ParameterException –

`class pymodbus.datastore.ModbusSlaveContext`(**_args*, ***kwargs*)

This creates a modbus data model with each data access stored in a block.

`reset()`

Reset all the datastores to their default values.

`validate`(*fc_as_hex*, *address*, *count=1*)

Validate the request to make sure it is in range.

Parameters

- `fc_as_hex` – The function we are working with
- `address` – The starting address
- `count` – The number of values to test

Returns
True if the request in within range, False otherwise

`getValues`(*fc_as_hex*, *address*, *count=1*)

Get *count* values from datastore.

Parameters

- `fc_as_hex` – The function we are working with
- `address` – The starting address
- `count` – The number of values to retrieve

Returns
The requested values from a:a+c

`setValues`(*fc_as_hex*, *address*, *values*)

Set the datastore with the supplied values.

Parameters

- `fc_as_hex` – The function we are working with
- `address` – The starting address
- `values` – The new values to be set

`register`(*function_code*, *fc_as_hex*, *datablock=None*)

Register a datablock with the slave context.

Parameters

- `function_code` – function code (int)
- `fc_as_hex` – string representation of function code (e.g "cf")
- `datablock` – datablock to associate with this function code

`class pymodbus.datastore.ModbusServerContext`(*slaves=None*, *single=True*)

This represents a master collection of slave contexts.

If single is set to true, it will be treated as a single context so every slave_id returns the same context. If single is set to false, it will be interpreted as a collection of slave contexts.

`slaves()`

Define slaves.

`class pymodbus.datastore.ModbusSimulatorContext`(*config: dict[str, Any]*, *custom_actions: dict[str, Callable] | None*)

Modbus simulator.

Parameters

- `config` – A dict with structure as shown below.
- `actions` – A dict with "<name>": <function> structure.

Raises
`RuntimeError` – if json contains errors (msg explains what)

It builds and maintains a virtual copy of a device, with simulation of device specific functions.

The device is described in a dict, user supplied actions will be added to the builtin actions.

It is used in conjunction with a pymodbus server.

Example:

```
store = ModbusSimulatorContext(<config dict>, <actions dict>)
StartAsyncTcpServer(<host>, context=store)

Now the server will simulate the defined device with features like:

- invalid addresses
- write protected addresses
- optional control of access for string, uint32, bit/bits
- builtin actions for e.g. reset/datetime, value increment by read
- custom actions
```

Description of the json file or dict to be supplied:

```
{
    "setup": {
        "di size": 0,  --> Size of discrete input block (8 bit)
        "co size": 0,  --> Size of coils block (8 bit)
        "ir size": 0,  --> Size of input registers block (16 bit)
        "hr size": 0,  --> Size of holding registers block (16 bit)
        "shared blocks": True,  --> share memory for all blocks (largest size wins)
        "defaults": {
            "value": {  --> Initial values (can be overwritten)
                "bits": 0x01,
                "uint16": 122,
                "uint32": 67000,
                "float32": 127.4,
                "string": " ",
            },
            "action": {  --> default action (can be overwritten)
                "bits": None,
                "uint16": None,
                "uint32": None,
                "float32": None,
                "string": None,
            },
        },
        "type exception": False,  --> return IO exception if read/write on non␣
↪boundary
    },
    "invalid": [  --> List of invalid addresses, IO exception returned
        51,                --> single register
        [78, 99],         --> start, end registers, repeated as needed
    ],
    "write": [   --> allow write, efault is ReadOnly
        [5, 5]  --> start, end bytes, repeated as needed
    ],
    "bits": [  --> Define bits (1 register == 1 byte)
        [30, 31],  --> start, end registers, repeated as needed
        {"addr": [32, 34], "value": 0xF1},  --> with value
        {"addr": [35, 36], "action": "increment"},  --> with action
        {"addr": [37, 38], "action": "increment", "value": 0xF1}  --> with action␣
↪and value
```

(continues on next page)

(continued from previous page)

```
        {"addr": [37, 38], "action": "increment", "kwargs": {"min": 0, "max": 100}}␣
↪ --> with action with arguments
    ],
    "uint16": [  --> Define uint16 (1 register == 2 bytes)
        --> same as type_bits
    ],
    "uint32": [  --> Define 32 bit integers (2 registers == 4 bytes)
        --> same as type_bits
    ],
    "float32": [  --> Define 32 bit floats (2 registers == 4 bytes)
        --> same as type_bits
    ],
    "string": [  --> Define strings (variable number of registers (each 2 bytes))
        [21, 22],  --> start, end registers, define 1 string
        {"addr": 23, 25], "value": "ups"},  --> with value
        {"addr": 26, 27], "action": "user"},  --> with action
        {"addr": 28, 29], "action": "", "value": "user"}  --> with action and value
    ],
    "repeat": [ --> allows to repeat section e.g. for n devices
        {"addr": [100, 200], "to": [50, 275]}   --> Repeat registers 100-200 to 50+␣
↪until 275
    ]
}
```

get_text_register(*register*)

Get raw register.

classmethod build_registers_from_value(*value*, *is_int*)

Build registers from int32 or float32.

classmethod build_value_from_registers(*registers*, *is_int*)

Build int32 or float32 value from registers.

10.3 Framer

10.3.1 pymodbus.framer.ascii_framer module

Ascii_framer.

class pymodbus.framer.ascii_framer.**ModbusAsciiFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus ASCII Frame Controller.

[Start][Address][Function][Data][LRC][End]
1c 2c 2c Nc 2c 2c

- data can be 0 - 2x252 chars
- end is "\r\n" (Carriage return line feed), however the line feed character can be changed via a special command
- start is ":"

This framer is used for serial transmission. Unlike the RTU protocol, the data in this framer is transferred in plain text ascii.

`buildPacket`(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The request/response to send

Returns
The encoded packet

`decode_data`(*data*)

Decode data.

`frameProcessIncomingPacket`(*single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

`method = 'ascii'`

10.3.2 pymodbus.framer.binary_framer module

Binary framer.

`class pymodbus.framer.binary_framer.ModbusBinaryFramer`(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus Binary Frame Controller.

[Start][Address][Function][Data][CRC][End]
1b 1b 1b Nb 2b 1b

- data can be 0 - 2x252 chars
- end is "}"
- start is "{"

The idea here is that we implement the RTU protocol, however, instead of using timing for message delimiting, we use start and end of message characters (in this case { and }). Basically, this is a binary framer.

The only case we have to watch out for is when a message contains the { or } characters. If we encounter these characters, we simply duplicate them. Hopefully we will not encounter those characters that often and will save a little bit of bandwitch without a real-time system.

Protocol defined by jamod.sourceforge.net.

`buildPacket`(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The request/response to send

Returns
The encoded packet

`decode_data`(*data*)

Decode data.

frameProcessIncomingPacket(*single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

method = 'binary'

10.3.3 pymodbus.framer.rtu_framer module

RTU framer.

class pymodbus.framer.rtu_framer.**ModbusRtuFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus RTU Frame controller.

[Start Wait] [Address][Function Code] [Data][CRC][End Wait]
3.5 chars 1b 1b Nb 2b 3.5 chars

Wait refers to the amount of time required to transmit at least x many characters. In this case it is 3.5 characters. Also, if we receive a wait of 1.5 characters at any point, we must trigger an error message. Also, it appears as though this message is little endian. The logic is simplified as the following:

```
block-on-read:
    read until 3.5 delay
    check for errors
    decode
```

The following table is a listing of the baud wait times for the specified baud rates:

```
------------------------------------------------------------------
 Baud  1.5c (18 bits)   3.5c (38 bits)
------------------------------------------------------------------
 1200   13333.3 us       31666.7 us
 4800    3333.3 us        7916.7 us
 9600    1666.7 us        3958.3 us
19200     833.3 us        1979.2 us
38400     416.7 us         989.6 us
------------------------------------------------------------------
1 Byte = start + 8 bits + parity + stop = 11 bits
(1/Baud)(bits) = delay seconds
```

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The populated request/response to send

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*_single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

method = 'rtu'

recvPacket(*size*)

Receive packet from the bus with specified len.

Parameters
size – Number of bytes to read

Returns

sendPacket(*message*)

Send packets on the bus with 3.5char delay between frames.

Parameters
message – Message to be sent over the bus

Returns

10.3.4 pymodbus.framer.socket_framer module

Socket framer.

class pymodbus.framer.socket_framer.**ModbusSocketFramer**(*decoder*, *client=None*)

Bases: ModbusFramer

Modbus Socket Frame controller.

Before each modbus TCP message is an MBAP header which is used as a message frame. It allows us to easily separate messages as follows:

```
[         MBAP Header         ] [ Function Code] [ Data ]         [ tid ][ pid ][␣
↪length ][ uid ]
  2b     2b     2b        1b           1b           Nb

while len(message) > 0:
    tid, pid, length`, uid = struct.unpack(">HHHB", message)
    request = message[0:7 + length - 1`]
    message = [7 + length - 1:]

* length = uid + function code + data
* The -1 is to account for the uid byte
```

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The populated request/response to send

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*single*, *callback*, *slave*, *tid=None*, ***kwargs*)

Process new packet pattern.

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 // N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

method = 'socket'

10.4 Constants

Constants For Modbus Server/Client.

This is the single location for storing default values for the servers and clients.

`class pymodbus.constants.`**`DeviceInformation`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `int`, `Enum`

Represents what type of device information to read.

`BASIC`

This is the basic (required) device information to be returned. This includes VendorName, ProductCode, and MajorMinorRevision code.

`REGULAR`

In addition to basic data objects, the device provides additional and optional identification and description data objects. All of the objects of this category are defined in the standard but their implementation is optional.

`EXTENDED`

In addition to regular data objects, the device provides additional and optional identification and description private data about the physical device itself. All of these data are device dependent.

`SPECIFIC`

Request to return a single data object.

`BASIC = 1`

`EXTENDED = 3`

`REGULAR = 2`

`SPECIFIC = 4`

`class pymodbus.constants.`**`Endian`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `str`, `Enum`

An enumeration representing the various byte endianness.

`AUTO`

This indicates that the byte order is chosen by the current native environment.

`BIG`

This indicates that the bytes are in big endian format

`LITTLE`

This indicates that the bytes are in little endian format

Note: I am simply borrowing the format strings from the python struct module for my convenience.

`AUTO = '@'`

`BIG = '>'`

`LITTLE = '<'`

`class pymodbus.constants.`**`ModbusPlusOperation`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `int`, `Enum`

Represents the type of modbus plus request.

`GET_STATISTICS`

Operation requesting that the current modbus plus statistics be returned in the response.

`CLEAR_STATISTICS`

Operation requesting that the current modbus plus statistics be cleared and not returned in the response.

`CLEAR_STATISTICS = 4`

`GET_STATISTICS = 3`

`class pymodbus.constants.`**`ModbusStatus`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `int`, `Enum`

These represent various status codes in the modbus protocol.

`WAITING`

This indicates that a modbus device is currently waiting for a given request to finish some running task.

`READY`

This indicates that a modbus device is currently free to perform the next request task.

`ON`

This indicates that the given modbus entity is on

`OFF`

This indicates that the given modbus entity is off

`SLAVE_ON`

This indicates that the given modbus slave is running

`SLAVE_OFF`

This indicates that the given modbus slave is not running

`OFF = 0`

`ON = 65280`

`READY = 0`

`SLAVE_OFF = 0`

`SLAVE_ON = 255`

`WAITING = 65535`

`class pymodbus.constants.`**`MoreData`**`(`*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `int`, `Enum`

Represents the more follows condition.

NOTHING
  This indicates that no more objects are going to be returned.

KEEP_READING
  This indicates that there are more objects to be returned.

KEEP_READING = 255

NOTHING = 0

10.5 Extra functions

Pymodbus: Modbus Protocol Implementation.

Released under the BSD license

`class pymodbus.`**`ExceptionResponse`**(*function_code*, *exception_code=None*, ***kwargs*)

Bases: `ModbusResponse`

Base class for a modbus exception PDU.

ExceptionOffset = 128

decode(*data*)

Decode a modbus exception response.

Parameters
  data – The packet data to decode

encode()

Encode a modbus exception response.

Returns
  The encoded exception packet

`class pymodbus.`**`Framer`**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: `str`, `Enum`

These represent the different framers.

ASCII = 'ascii'

BINARY = 'binary'

RTU = 'rtu'

SOCKET = 'socket'

TLS = 'tls'

`exception pymodbus.`**`ModbusException`**(*string*)

Bases: `Exception`

Base modbus exception.

isError()

Error

`pymodbus.`**`pymodbus_apply_logging_config`**`(`*level: str | int = 10*, *log_file_name: str | None = None*`)`

Apply basic logging configuration used by default by Pymodbus maintainers.

Parameters

- **`level`** – (optional) set log level, if not set it is inherited.
- **`log_file_name`** – (optional) log additional to file

Please call this function to format logging appropriately when opening issues.

Bit Reading Request/Response messages.

`class pymodbus.bit_read_message.`**`ReadBitsResponseBase`**`(`*values*, *slave=0*, ***kwargs*`)`

Bases: `ModbusResponse`

Base class for Messages responding to bit-reading values.

The requested bits can be found in the .bits list.

`bits`

A list of booleans representing bit values

`decode``(`*data*`)`

Decode response pdu.

Parameters
`data` – The packet data to decode

`encode``()`

Encode response pdu.

Returns
The encoded packet message

`getBit``(`*address*`)`

Get the specified bit's value.

Parameters
`address` – The bit to query

Returns
The value of the requested bit

`resetBit``(`*address*`)`

Set the specified bit to 0.

Parameters
`address` – The bit to reset

`setBit``(`*address*, *value=1*`)`

Set the specified bit.

Parameters

- **`address`** – The bit to set
- **`value`** – The value to set the bit to

`class pymodbus.bit_read_message.`**`ReadCoilsRequest`**`(`*address=None*, *count=None*, *slave=0*, ***kwargs*`)`

Bases: `ReadBitsRequestBase`

This function code is used to read from 1 to 2000(0x7d0) contiguous status of coils in a remote device.

The Request PDU specifies the starting address, ie 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.

execute(*context*)

Run a read coils request against a datastore.

Before running the request, we make sure that the request is in the max valid range (0x001-0x7d0). Next we make sure that the request is valid against the current datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized `ReadCoilsResponse`, or an *ExceptionResponse* if an error occurred

function_code = 1

function_code_name = 'read_coils'

class pymodbus.bit_read_message.**ReadCoilsResponse**(*values=None*, *slave=0*, ***kwargs*)

Bases: *ReadBitsResponseBase*

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.

The requested coils can be found in boolean form in the .bits list.

function_code = 1

class pymodbus.bit_read_message.**ReadDiscreteInputsRequest**(*address=None*, *count=None*, *slave=0*, ***kwargs*)

Bases: `ReadBitsRequestBase`

This function code is used to read from 1 to 2000(0x7d0).

Contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, ie 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.

execute(*context*)

Run a read discrete input request against a datastore.

Before running the request, we make sure that the request is in the max valid range (0x001-0x7d0). Next we make sure that the request is valid against the current datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized `ReadDiscreteInputsResponse`, or an *ExceptionResponse* if an error occurred

function_code = 2

function_code_name = 'read_discrete_input'

class pymodbus.bit_read_message.**ReadDiscreteInputsResponse**(*values=None*, *slave=0*, ***kwargs*)

Bases: `ReadBitsResponseBase`

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 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.

The requested coils can be found in boolean form in the .bits list.

function_code = 2

Bit Writing Request/Response.

TODO write mask request/response

class pymodbus.bit_write_message.**WriteMultipleCoilsRequest**(*address=None*, *values=None*, *slave=None*, ***kwargs*)

Bases: `ModbusRequest`

This function code is used to forcea 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."

decode(*data*)

Decode a write coils request.

Parameters
`data` – The packet data to decode

encode()

Encode write coils request.

Returns
The byte encoded message

execute(*context*)

Run a write coils request against a datastore.

Parameters
`context` – The datastore to request from

Returns
The populated response or exception message

function_code = 15

function_code_name = 'write_coils'

get_response_pdu_size()

Get response pdu size.

Func_code (1 byte) + Output Address (2 byte) + Quantity of Outputs (2 Bytes) :return:

`class pymodbus.bit_write_message.`**`WriteMultipleCoilsResponse`**`(`*address=None*, *count=None*, ***kwargs*`)`

Bases: `ModbusResponse`

The normal response returns the function code.

Starting address, and quantity of coils forced.

`decode``(`*data*`)`

Decode a write coils response.

Parameters
`data` – The packet data to decode

`encode``()`

Encode write coils response.

Returns
The byte encoded message

`function_code = 15`

`class pymodbus.bit_write_message.`**`WriteSingleCoilRequest`**`(`*address=None*, *value=None*, *slave=None*, ***kwargs*`)`

Bases: `ModbusRequest`

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.

`decode``(`*data*`)`

Decode a write coil request.

Parameters
`data` – The packet data to decode

`encode``()`

Encode write coil request.

Returns
The byte encoded message

`execute``(`*context*`)`

Run a write coil request against a datastore.

Parameters
`context` – The datastore to request from

Returns
The populated response or exception message

`function_code = 5`

`function_code_name = 'write_coil'`

`get_response_pdu_size()`

Get response pdu size.

Func_code (1 byte) + Output Address (2 byte) + Output Value (2 Bytes) :return:

class `pymodbus.bit_write_message.`**`WriteSingleCoilResponse`**(*address=None*, *value=None*, ***kwargs*)

Bases: `ModbusResponse`

The normal response is an echo of the request.

Returned after the coil state has been written.

`decode`(*data*)

Decode a write coil response.

Parameters
`data` – The packet data to decode

`encode`()

Encode write coil response.

Returns
The byte encoded message

`function_code = 5`

Modbus Device Controller.

These are the device management handlers. They should be maintained in the server context and the various methods should be inserted in the correct locations.

class `pymodbus.device.`**`DeviceInformationFactory`**

Bases: `object`

This is a helper factory.

That really just hides some of the complexity of processing the device information requests (function code 0x2b 0x0e).

classmethod **`get`**(*control*, *read_code=DeviceInformation.BASIC*, *object_id=0*)

Get the requested device data from the system.

Parameters

- **`control`** – The control block to pull data from
- **`read_code`** – The read code to process
- **`object_id`** – The specific object_id to read

Returns
The requested data (id, length, value)

class `pymodbus.device.`**`ModbusDeviceIdentification`**(*info=None*, *info_name=None*)

Bases: `object`

This is used to supply the device identification.

For the readDeviceIdentification function

For more information read section 6.21 of the modbus application protocol.

property MajorMinorRevision

property **ModelName**

property **ProductCode**

property **ProductName**

property **UserApplicationName**

property **VendorName**

property **VendorUrl**

summary()

Return a summary of the main items.

Returns
An dictionary of the main items

update(*value*)

Update the values of this identity.

using another identify as the value

Parameters
value – The value to copy values from

class pymodbus.device.**ModbusPlusStatistics**

Bases: `object`

This is used to maintain the current modbus plus statistics count.

As of right now this is simply a stub to complete the modbus implementation. For more information, see the modbus implementation guide page 87.

encode()

Return a summary of the modbus plus statistics.

Returns
54 16-bit words representing the status

reset()

Clear all of the modbus plus statistics.

summary()

Return a summary of the modbus plus statistics.

Returns
54 16-bit words representing the status

Diagnostic Record Read/Write.

These need to be tied into a the current server context or linked to the appropriate data

class pymodbus.diag_message.**ChangeAsciiInputDelimiterRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Change ascii input delimiter.

The character "CHAR" passed in the request data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases of a Line Feed is not required at the end of ASCII messages.

`execute(*args)`

Execute the diagnostic request on the given device.

Returns
The initialized response message

`sub_function_code = 3`

`class pymodbus.diag_message.ChangeAsciiInputDelimiterResponse(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleResponse`

Change ascii input delimiter.

The character "CHAR" passed in the request data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases of a Line Feed is not required at the end of ASCII messages.

`sub_function_code = 3`

`class pymodbus.diag_message.ClearCountersRequest(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleRequest`

Clear ll counters and the diagnostic register.

Also, counters are cleared upon power-up

`execute(*args)`

Execute the diagnostic request on the given device.

Returns
The initialized response message

`sub_function_code = 10`

`class pymodbus.diag_message.ClearCountersResponse(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleResponse`

Clear ll counters and the diagnostic register.

Also, counters are cleared upon power-up

`sub_function_code = 10`

`class pymodbus.diag_message.ClearOverrunCountRequest(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleRequest`

Clear the overrun error counter and reset the error flag.

An error flag should be cleared, but nothing else in the specification mentions is, so it is ignored.

`execute(*args)`

Execute the diagnostic request on the given device.

Returns
The initialized response message

`sub_function_code = 20`

`class pymodbus.diag_message.ClearOverrunCountResponse(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleResponse`

Clear the overrun error counter and reset the error flag.

`sub_function_code = 20`

`class pymodbus.diag_message.DiagnosticStatusRequest(**kwargs)`

Bases: `ModbusRequest`

This is a base class for all of the diagnostic request functions.

`decode(data)`

Decode a diagnostic request.

Parameters
`data` – The data to decode into the function code

`encode()`

Encode a diagnostic response.

we encode the data set in self.message

Returns
The encoded packet

`function_code = 8`

`function_code_name = 'diagnostic_status'`

`get_response_pdu_size()`

Get response pdu size.

Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

`class pymodbus.diag_message.DiagnosticStatusResponse(**kwargs)`

Bases: `ModbusResponse`

Diagnostic status.

This is a base class for all of the diagnostic response functions

It works by performing all of the encoding and decoding of variable data and lets the higher classes define what extra data to append and how to execute a request

`decode(data)`

Decode diagnostic response.

Parameters
`data` – The data to decode into the function code

`encode()`

Encode diagnostic response.

we encode the data set in self.message

Returns
The encoded packet

`function_code = 8`

`class pymodbus.diag_message.ForceListenOnlyModeRequest(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleRequest`

Forces the addressed remote device to its Listen Only Mode for MODBUS communications.

This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 4

class pymodbus.diag_message.**ForceListenOnlyModeResponse**(***kwargs*)

Bases: *DiagnosticStatusResponse*

Forces the addressed remote device to its Listen Only Mode for MODBUS communications.

This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

This does not send a response

should_respond = False

sub_function_code = 4

class pymodbus.diag_message.**GetClearModbusPlusRequest**(*slave=None*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Get/Clear modbus plus request.

In addition to the Function code (08) and Subfunction code (00 15 hex) in the query, a two-byte Operation field is used to specify either a “Get Statistics” or a “Clear Statistics” operation. The two operations are exclusive - the “Get” operation cannot clear the statistics, and the “Clear” operation does not return statistics prior to clearing them. Statistics are also cleared on power-up of the slave device.

encode()

Encode a diagnostic response.

we encode the data set in self.message

Returns
The encoded packet

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

get_response_pdu_size()

Return a series of 54 16-bit words (108 bytes) in the data field of the response.

This function differs from the usual two-byte length of the data field. The data contains the statistics for the Modbus Plus peer processor in the slave device. Func_code (1 byte) + Sub function code (2 byte) + Operation (2 byte) + Data (108 bytes) :return:

sub_function_code = 21

class pymodbus.diag_message.**GetClearModbusPlusResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return a series of 54 16-bit words (108 bytes) in the data field of the response.

This function differs from the usual two-byte length of the data field. The data contains the statistics for the Modbus Plus peer processor in the slave device.

sub_function_code = 21

class pymodbus.diag_message.**RestartCommunicationsOptionRequest**(*toggle=False*, *slave=None*, ***kwargs*)

Bases: *DiagnosticStatusRequest*

Restart communication.

The remote device serial line port must be initialized and restarted, and all of its communications event counters are cleared. If the port is currently in Listen Only Mode, no response is returned. This function is the only one that brings the port out of Listen Only Mode. If the port is not currently in Listen Only Mode, a normal response is returned. This occurs before the restart is executed.

execute(**_args*)

Clear event log and restart.

Returns
The initialized response message

sub_function_code = 1

class pymodbus.diag_message.**RestartCommunicationsOptionResponse**(*toggle=False*, ***kwargs*)

Bases: *DiagnosticStatusResponse*

Restart Communication.

The remote device serial line port must be initialized and restarted, and all of its communications event counters are cleared. If the port is currently in Listen Only Mode, no response is returned. This function is the only one that brings the port out of Listen Only Mode. If the port is not currently in Listen Only Mode, a normal response is returned. This occurs before the restart is executed.

sub_function_code = 1

class pymodbus.diag_message.**ReturnBusCommunicationErrorCountRequest**(*data=0*, ***kwargs*)

Bases: DiagnosticStatusSimpleRequest

Return bus comm. count.

The response data field returns the quantity of CRC errors encountered by the remote device since its last restart, clear counter operation, or power-up

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 12

class pymodbus.diag_message.**ReturnBusCommunicationErrorCountResponse**(*data=0*, ***kwargs*)

Bases: DiagnosticStatusSimpleResponse

Return bus comm. error.

The response data field returns the quantity of CRC errors encountered by the remote device since its last restart, clear counter operation, or power-up

sub_function_code = 12

class pymodbus.diag_message.**ReturnBusExceptionErrorCountRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Return bus exception.

The response data field returns the quantity of modbus exception responses returned by the remote device since its last restart, clear counters operation, or power-up

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 13

class pymodbus.diag_message.**ReturnBusExceptionErrorCountResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return bus exception.

The response data field returns the quantity of modbus exception responses returned by the remote device since its last restart, clear counters operation, or power-up

sub_function_code = 13

class pymodbus.diag_message.**ReturnBusMessageCountRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Return bus message count.

The response data field returns the quantity of messages that the remote device has detected on the communications systems since its last restart, clear counters operation, or power-up

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 11

class pymodbus.diag_message.**ReturnBusMessageCountResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return bus message count.

The response data field returns the quantity of messages that the remote device has detected on the communications systems since its last restart, clear counters operation, or power-up

sub_function_code = 11

class pymodbus.diag_message.**ReturnDiagnosticRegisterRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

The contents of the remote device's 16-bit diagnostic register are returned in the response.

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 2

class pymodbus.diag_message.**ReturnDiagnosticRegisterResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return diagnostic register.

The contents of the remote device's 16-bit diagnostic register are returned in the response

sub_function_code = 2

class pymodbus.diag_message.**ReturnIopOverrunCountRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Return IopOverrun.

An IOP overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction. This function is specific to the 884.

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 19

class pymodbus.diag_message.**ReturnIopOverrunCountResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return Iop overrun count.

The response data field returns the quantity of messages addressed to the slave that it could not handle due to an 884 IOP overrun condition, since its last restart, clear counters operation, or power-up.

sub_function_code = 19

class pymodbus.diag_message.**ReturnQueryDataRequest**(*message=b'\x00\x00'*, *slave=None*, ***kwargs*)

Bases: `DiagnosticStatusRequest`

Return query data.

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

execute(**_args*)

Execute the loopback request (builds the response).

Returns
The populated loopback response message

sub_function_code = 0

class pymodbus.diag_message.**ReturnQueryDataResponse**(*message=b'\x00\x00'*, ***kwargs*)

Bases: `DiagnosticStatusResponse`

Return query data.

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

sub_function_code = 0

class pymodbus.diag_message.**ReturnSlaveBusCharacterOverrunCountRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Return slave character overrun.

The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 18

class pymodbus.diag_message.**ReturnSlaveBusCharacterOverrunCountResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return the quantity of messages addressed to the remote device unhandled due to a character overrun.

Since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

sub_function_code = 18

class pymodbus.diag_message.**ReturnSlaveBusyCountRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Return slave busy count.

The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

execute(**args*)

Execute the diagnostic request on the given device.

Returns
The initialized response message

sub_function_code = 17

class pymodbus.diag_message.**ReturnSlaveBusyCountResponse**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleResponse`

Return slave busy count.

The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

sub_function_code = 17

class pymodbus.diag_message.**ReturnSlaveMessageCountRequest**(*data=0*, ***kwargs*)

Bases: `DiagnosticStatusSimpleRequest`

Return slave message count.

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up `execute(*args)`

Execute the diagnostic request on the given device.

Returns
The initialized response message

`sub_function_code = 14`

`class pymodbus.diag_message.ReturnSlaveMessageCountResponse(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleResponse`

Return slave message count.

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

`sub_function_code = 14`

`class pymodbus.diag_message.ReturnSlaveNAKCountRequest(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleRequest`

Return slave NAK count.

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7 .

`execute(*args)`

Execute the diagnostic request on the given device.

Returns
The initialized response message

`sub_function_code = 16`

`class pymodbus.diag_message.ReturnSlaveNAKCountResponse(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleResponse`

Return slave NAK.

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7.

`sub_function_code = 16`

`class pymodbus.diag_message.ReturnSlaveNoResponseCountRequest(data=0, **kwargs)`

Bases: `DiagnosticStatusSimpleRequest`

Return slave no response.

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

`execute(*args)`

Execute the diagnostic request on the given device.

Returns
The initialized response message

`sub_function_code = 15`

class pymodbus.diag_message.**ReturnSlaveNoResponseCountResponse**(*data=0*, ***kwargs*)

Bases: DiagnosticStatusSimpleResponse

Return slave no response.

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

sub_function_code = 15

Modbus Remote Events.

An event byte returned by the Get Communications Event Log function can be any one of four types. The type is defined by bit 7 (the high-order bit) in each byte. It may be further defined by bit 6.

class pymodbus.events.**CommunicationRestartEvent**

Bases: *ModbusEvent*

Restart remote device Initiated Communication.

The remote device stores this type of event byte when its communications port is restarted. The remote device can be restarted by the Diagnostics function (code 08), with sub-function Restart Communications Option (code 00 01).

That function also places the remote device into a "Continue on Error" or "Stop on Error" mode. If the remote device is placed into "Continue on Error" mode, the event byte is added to the existing event log. If the remote device is placed into "Stop on Error" mode, the byte is added to the log and the rest of the log is cleared to zeros.

The event is defined by a content of zero.

decode(*event*)

Decode the event message to its status bits.

Parameters
event – The event to decode

Raises
ParameterException –

encode()

Encode the status bits to an event message.

Returns
The encoded event message

value = 0

class pymodbus.events.**EnteredListenModeEvent**

Bases: *ModbusEvent*

Enter Remote device Listen Only Mode.

The remote device stores this type of event byte when it enters the Listen Only Mode. The event is defined by a content of 04 hex.

decode(*event*)

Decode the event message to its status bits.

Parameters
event – The event to decode

Raises
ParameterException –

encode()

Encode the status bits to an event message.

Returns
The encoded event message

value = 4

class pymodbus.events.**ModbusEvent**

Bases: `object`

Define modbus events.

decode(*event*)

Decode the event message to its status bits.

Parameters
event – The event to decode

Raises
NotImplementedException –

encode()

Encode the status bits to an event message.

Raises
NotImplementedException –

class pymodbus.events.**RemoteReceiveEvent**(***kwargs*)

Bases: *ModbusEvent*

Remote device MODBUS Receive Event.

The remote device stores this type of event byte when a query message is received. It is stored before the remote device processes the message. This event is defined by bit 7 set to logic "1". The other bits will be set to a logic "1" if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
----------------------------------
0   Not Used
2   Not Used
3   Not Used
4   Character Overrun
5   Currently in Listen Only Mode
6   Broadcast Receive
7   1
```

decode(*event: bytes*) → None

Decode the event message to its status bits.

Parameters
event – The event to decode

encode() → bytes

Encode the status bits to an event message.

Returns
The encoded event message

class pymodbus.events.**RemoteSendEvent**(***kwargs*)

Bases: *ModbusEvent*

Remote device MODBUS Send Event.

The remote device stores this type of event byte when it finishes processing a request message. It is stored if the remote device returned a normal or exception response, or no response.

This event is defined by bit 7 set to a logic "0", with bit 6 set to a "1". The other bits will be set to a logic "1" if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
-----------------------------------------------------------
0   Read Exception Sent (Exception Codes 1-3)
1   Slave Abort Exception Sent (Exception Code 4)
2   Slave Busy Exception Sent (Exception Codes 5-6)
3   Slave Program NAK Exception Sent (Exception Code 7)
4   Write Timeout Error Occurred
5   Currently in Listen Only Mode
6   1
7   0
```

decode(*event*)

Decode the event message to its status bits.

Parameters
event – The event to decode

encode()

Encode the status bits to an event message.

Returns
The encoded event message

Pymodbus Exceptions.

Custom exceptions to be used in the Modbus code.

exception pymodbus.exceptions.**ConnectionException**(*string=''*)

Bases: *ModbusException*

Error resulting from a bad connection.

exception pymodbus.exceptions.**InvalidMessageReceivedException**(*string=''*)

Bases: *ModbusException*

Error resulting from invalid response received or decoded.

exception pymodbus.exceptions.**MessageRegisterException**(*string=''*)

Bases: *ModbusException*

Error resulting from failing to register a custom message request/response.

exception pymodbus.exceptions.**ModbusIOException**(*string='', function_code=None*)

Bases: *ModbusException*

Error resulting from data i/o.

exception `pymodbus.exceptions.`**`NoSuchSlaveException`**`(`*string=''*`)`

Bases: *ModbusException*

Error resulting from making a request to a slave that does not exist.

exception `pymodbus.exceptions.`**`NotImplementedException`**`(`*string=''*`)`

Bases: *ModbusException*

Error resulting from not implemented function.

exception `pymodbus.exceptions.`**`ParameterException`**`(`*string=''*`)`

Bases: *ModbusException*

Error resulting from invalid parameter.

Modbus Request/Response Decoder Factories.

The following factories make it easy to decode request/response messages. To add a new request/response pair to be decodeable by the library, simply add them to the respective function lookup table (order doesn't matter, but it does help keep things organized).

Regardless of how many functions are added to the lookup, O(1) behavior is kept as a result of a pre-computed lookup dictionary.

class `pymodbus.factory.`**`ClientDecoder`**

Bases: `object`

Response Message Factory (Client).

To add more implemented functions, simply add them to the list

`decode``(`*message*`)`

Decode a response packet.

Parameters
`message` – The raw packet to decode

Returns
The decoded modbus message or None if error

`function_table = [<class 'pymodbus.register_read_message.ReadHoldingRegistersResponse'>, <class 'pymodbus.bit_read_message.ReadDiscreteInputsResponse'>, <class 'pymodbus.register_read_message.ReadInputRegistersResponse'>, <class 'pymodbus.bit_read_message.ReadCoilsResponse'>, <class 'pymodbus.bit_write_message.WriteMultipleCoilsResponse'>, <class 'pymodbus.register_write_message.WriteMultipleRegistersResponse'>, <class 'pymodbus.register_write_message.WriteSingleRegisterResponse'>, <class 'pymodbus.bit_write_message.WriteSingleCoilResponse'>, <class 'pymodbus.register_read_message.ReadWriteMultipleRegistersResponse'>, <class 'pymodbus.diag_message.DiagnosticStatusResponse'>, <class 'pymodbus.other_message.ReadExceptionStatusResponse'>, <class 'pymodbus.other_message.GetCommEventCounterResponse'>, <class 'pymodbus.other_message.GetCommEventLogResponse'>, <class 'pymodbus.other_message.ReportSlaveIdResponse'>, <class 'pymodbus.file_message.ReadFileRecordResponse'>, <class 'pymodbus.file_message.WriteFileRecordResponse'>, <class 'pymodbus.register_write_message.MaskWriteRegisterResponse'>, <class 'pymodbus.file_message.ReadFifoQueueResponse'>, <class 'pymodbus.mei_message.ReadDeviceInformationResponse'>]`

`lookupPduClass`(*function_code*)

Use *function_code* to determine the class of the PDU.

Parameters
`function_code` – The function code specified in a frame.

Returns
The class of the PDU that has a matching *function_code*.

`register`(*function*)

Register a function and sub function class with the decoder.

class pymodbus.factory.**ServerDecoder**

Bases: `object`

Request Message Factory (Server).

To add more implemented functions, simply add them to the list

`decode`(*message*)

Decode a request packet.

Parameters
`message` – The raw modbus request packet

Returns
The decoded modbus message or None if error

classmethod `getFCdict`() → Dict[int, Callable]

Build function code - class list.

`lookupPduClass`(*function_code*)

Use *function_code* to determine the class of the PDU.

Parameters
`function_code` – The function code specified in a frame.

Returns
The class of the PDU that has a matching *function_code*.

`register`(*function*)

Register a function and sub function class with the decoder.

Parameters
`function` – Custom function class to register

Raises
MessageRegisterException –

File Record Read/Write Messages.

Currently none of these messages are implemented

class pymodbus.file_message.**FileRecord**(***kwargs*)

Bases: `object`

Represents a file record and its relevant data.

class pymodbus.file_message.**ReadFifoQueueRequest**(*address=0*, ***kwargs*)

Bases: `ModbusRequest`

Read fifo queue request.

This function code allows to read the contents of a First-In-First-Out (FIFO) queue of register in a remote device. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers.

The queue count register is returned first, followed by the queued data registers. The function reads the queue contents, but does not clear them.

`decode`(*data*)

Decode the incoming request.

Parameters
`data` – The data to decode into the address

`encode`()

Encode the request packet.

Returns
The byte encoded packet

`execute`(*_context*)

Run a read exception status request against the store.

Returns
The populated response

`function_code = 24`

`function_code_name = 'read_fifo_queue'`

`class pymodbus.file_message.ReadFifoQueueResponse`(*values=None*, ***kwargs*)

Bases: `ModbusResponse`

Read Fifo queue response.

In a normal response, the byte count shows the quantity of bytes to follow, including the queue count bytes and value register bytes (but not including the error check field). The queue count is the quantity of data registers in the queue (not including the count register).

If the queue count exceeds 31, an exception response is returned with an error code of 03 (Illegal Data Value).

`classmethod calculateRtuFrameSize`(*buffer*)

Calculate the size of the message.

Parameters
`buffer` – A buffer containing the data that have been received.

Returns
The number of bytes in the response.

`decode`(*data*)

Decode a the response.

Parameters
`data` – The packet data to decode

`encode`()

Encode the response.

Returns
The byte encoded message

function_code = 24

class pymodbus.file_message.**ReadFileRecordRequest**(*records=None*, ***kwargs*)

Bases: `ModbusRequest`

Read file record request.

This function code is used to perform a file record read. All request data lengths are provided in terms of number of bytes and all record lengths are provided in terms of registers.

A file is an organization of records. Each file contains 10000 records, addressed 0000 to 9999 decimal or 0x0000 to 0x270f. For example, record 12 is addressed as 12. The function can read multiple groups of references. The groups can be separating (non-contiguous), but the references within each group must be sequential. Each group is defined in a separate "sub-request" field that contains seven bytes:

```
The reference type: 1 byte (must be 0x06)
The file number: 2 bytes
The starting record number within the file: 2 bytes
The length of the record to be read: 2 bytes
```

The quantity of registers to be read, combined with all other fields in the expected response, must not exceed the allowable length of the MODBUS PDU: 235 bytes.

decode(*data*)

Decode the incoming request.

Parameters
data – The data to decode into the address

encode()

Encode the request packet.

Returns
The byte encoded packet

execute(*_context*)

Run a read exception status request against the store.

Returns
The populated response

function_code = 20

function_code_name = 'read_file_record'

class pymodbus.file_message.**ReadFileRecordResponse**(*records=None*, ***kwargs*)

Bases: `ModbusResponse`

Read file record response.

The normal response is a series of "sub-responses," one for each "sub-request." The byte count field is the total combined count of bytes in all "sub-responses." In addition, each "sub-response" contains a field that shows its own byte count.

decode(*data*)

Decode the response.

Parameters
data – The packet data to decode

`encode()`

Encode the response.

Returns
The byte encoded message

`function_code = 20`

`class pymodbus.file_message.WriteFileRecordRequest(`*records=None*, ***kwargs*`)`

Bases: `ModbusRequest`

Write file record request.

This function code is used to perform a file record write. All request data lengths are provided in terms of number of bytes and all record lengths are provided in terms of the number of 16 bit words.

`decode(`*data*`)`

Decode the incoming request.

Parameters
`data` – The data to decode into the address

`encode()`

Encode the request packet.

Returns
The byte encoded packet

`execute(`*_context*`)`

Run the write file record request against the context.

Returns
The populated response

`function_code = 21`

`function_code_name = 'write_file_record'`

`class pymodbus.file_message.WriteFileRecordResponse(`*records=None*, ***kwargs*`)`

Bases: `ModbusResponse`

The normal response is an echo of the request.

`decode(`*data*`)`

Decode the incoming request.

Parameters
`data` – The data to decode into the address

`encode()`

Encode the response.

Returns
The byte encoded message

`function_code = 21`

Encapsulated Interface (MEI) Transport Messages.

`class pymodbus.mei_message.`**`ReadDeviceInformationRequest`**`(`*read_code=None*, *object_id=0*, ***kwargs*`)`

Bases: `ModbusRequest`

Read device information.

This function code allows reading the identification and additional information relative to the physical and functional description of a remote device, only.

The Read Device Identification interface is modeled as an address space composed of a set of addressable data elements. The data elements are called objects and an object Id identifies them.

`decode``(`*data*`)`

Decode data part of the message.

Parameters
`data` – The incoming data

`encode``()`

Encode the request packet.

Returns
The byte encoded packet

`execute``(`*_context*`)`

Run a read exception status request against the store.

Returns
The populated response

`function_code = 43`

`function_code_name = 'read_device_information'`

`sub_function_code = 14`

`class pymodbus.mei_message.`**`ReadDeviceInformationResponse`**`(`*read_code=None*, *information=None*, ***kwargs*`)`

Bases: `ModbusResponse`

Read device information response.

`classmethod` **`calculateRtuFrameSize`**`(`*buffer*`)`

Calculate the size of the message.

Parameters
`buffer` – A buffer containing the data that have been received.

Returns
The number of bytes in the response.

`decode``(`*data*`)`

Decode a the response.

Parameters
`data` – The packet data to decode

`encode``()`

Encode the response.

Returns
The byte encoded message

`function_code = 43`

`sub_function_code = 14`

Diagnostic record read/write.

Currently not all implemented

`class pymodbus.other_message.GetCommEventCounterRequest(**kwargs)`

Bases: `ModbusRequest`

This function code is used to get a status word.

And an event count from the remote device's communication event counter.

By fetching the current count before and after a series of messages, a client can determine whether the messages were handled normally by the remote device.

The device's event counter is incremented once for each successful message completion. It is not incremented for exception responses, poll commands, or fetch event counter commands.

The event counter can be reset by means of the Diagnostics function (code 08), with a subfunction of Restart Communications Option (code 00 01) or Clear Counters and Diagnostic Register (code 00 0A).

`decode(data)`

Decode data part of the message.

Parameters
`data` – The incoming data

`encode()`

Encode the message.

`execute(_context=None)`

Run a read exception status request against the store.

Returns
The populated response

`function_code = 11`

`function_code_name = 'get_event_counter'`

`class pymodbus.other_message.GetCommEventCounterResponse(count=0, **kwargs)`

Bases: `ModbusResponse`

Get comm event counter response.

The normal response contains a two-byte status word, and a two-byte event count. The status word will be all ones (FF FF hex) if a previously-issued program command is still being processed by the remote device (a busy condition exists). Otherwise, the status word will be all zeros.

`decode(data)`

Decode a the response.

Parameters
`data` – The packet data to decode

`encode()`

Encode the response.

Returns
The byte encoded message

`function_code = 11`

`class pymodbus.other_message.GetCommEventLogRequest(**kwargs)`

Bases: `ModbusRequest`

This function code is used to get a status word.

Event count, message count, and a field of event bytes from the remote device.

The status word and event counts are identical to that returned by the Get Communications Event Counter function (11, 0B hex).

The message counter contains the quantity of messages processed by the remote device since its last restart, clear counters operation, or power-up. This count is identical to that returned by the Diagnostic function (code 08), sub-function Return Bus Message Count (code 11, 0B hex).

The event bytes field contains 0-64 bytes, with each byte corresponding to the status of one MODBUS send or receive operation for the remote device. The remote device enters the events into the field in chronological order. Byte 0 is the most recent event. Each new byte flushes the oldest byte from the field.

`decode(data)`

Decode data part of the message.

Parameters
`data` – The incoming data

`encode()`

Encode the message.

`execute(_context=None)`

Run a read exception status request against the store.

Returns
The populated response

`function_code = 12`

`function_code_name = 'get_event_log'`

`class pymodbus.other_message.GetCommEventLogResponse(**kwargs)`

Bases: `ModbusResponse`

Get Comm event log response.

The normal response contains a two-byte status word field, a two-byte event count field, a two-byte message count field, and a field containing 0-64 bytes of events. A byte count field defines the total length of the data in these four field

`decode(data)`

Decode a the response.

Parameters
`data` – The packet data to decode

`encode()`

Encode the response.

Returns
The byte encoded message

`function_code = 12`

class pymodbus.other_message.**ReadExceptionStatusRequest**(*slave=None*, ***kwargs*)

Bases: `ModbusRequest`

This function code is used to read the contents of eight Exception Status outputs in a remote device.

The function provides a simple method for accessing this information, because the Exception Output references are known (no output reference is needed in the function).

decode(*data*)

Decode data part of the message.

Parameters
`data` – The incoming data

encode()

Encode the message.

execute(*_context=None*)

Run a read exception status request against the store.

Returns
The populated response

function_code = 7

function_code_name = 'read_exception_status'

class pymodbus.other_message.**ReadExceptionStatusResponse**(*status=0*, ***kwargs*)

Bases: `ModbusResponse`

The normal response contains the status of the eight Exception Status outputs.

The outputs are packed into one data byte, with one bit per output. The status of the lowest output reference is contained in the least significant bit of the byte. The contents of the eight Exception Status outputs are device specific.

decode(*data*)

Decode a the response.

Parameters
`data` – The packet data to decode

encode()

Encode the response.

Returns
The byte encoded message

function_code = 7

class pymodbus.other_message.**ReportSlaveIdRequest**(*slave=0*, ***kwargs*)

Bases: `ModbusRequest`

This function code is used to read the description of the type.

The current status, and other information specific to a remote device.

decode(*data*)

Decode data part of the message.

Parameters
`data` – The incoming data

encode()

Encode the message.

execute(*context=None*)

Run a report slave id request against the store.

Returns
The populated response

function_code = 17

function_code_name = 'report_slave_id'

class pymodbus.other_message.**ReportSlaveIdResponse**(*identifier=b'\x00'*, *status=True*, ***kwargs*)

Bases: `ModbusResponse`

Show response.

The data contents are specific to each type of device.

decode(*data*)

Decode a the response.

Since the identifier is device dependent, we just return the raw value that a user can decode to whatever it should be.

Parameters
data – The packet data to decode

encode()

Encode the response.

Returns
The byte encoded message

function_code = 17

Modbus Payload Builders.

A collection of utilities for building and decoding modbus messages payloads.

class pymodbus.payload.**BinaryPayloadBuilder**(*payload=None*, *byteorder=Endian.LITTLE*, *wordorder=Endian.BIG*, *repack=False*)

Bases: `object`

A utility that helps build payload messages to be written with the various modbus messages.

It really is just a simple wrapper around the struct module, however it saves time looking up the format strings. What follows is a simple example:

```
builder = BinaryPayloadBuilder(byteorder=Endian.Little)
builder.add_8bit_uint(1)
builder.add_16bit_uint(2)
payload = builder.build()
```

add_16bit_float(*value: float*) → None

Add a 16 bit float to the buffer.

Parameters
value – The value to add to the buffer

add_16bit_int(*value: int*) → None

Add a 16 bit signed int to the buffer.

Parameters
value – The value to add to the buffer

add_16bit_uint(*value: int*) → None

Add a 16 bit unsigned int to the buffer.

Parameters
value – The value to add to the buffer

add_32bit_float(*value: float*) → None

Add a 32 bit float to the buffer.

Parameters
value – The value to add to the buffer

add_32bit_int(*value: int*) → None

Add a 32 bit signed int to the buffer.

Parameters
value – The value to add to the buffer

add_32bit_uint(*value: int*) → None

Add a 32 bit unsigned int to the buffer.

Parameters
value – The value to add to the buffer

add_64bit_float(*value: float*) → None

Add a 64 bit float(double) to the buffer.

Parameters
value – The value to add to the buffer

add_64bit_int(*value: int*) → None

Add a 64 bit signed int to the buffer.

Parameters
value – The value to add to the buffer

add_64bit_uint(*value: int*) → None

Add a 64 bit unsigned int to the buffer.

Parameters
value – The value to add to the buffer

add_8bit_int(*value: int*) → None

Add a 8 bit signed int to the buffer.

Parameters
value – The value to add to the buffer

add_8bit_uint(*value: int*) → None

Add a 8 bit unsigned int to the buffer.

Parameters
value – The value to add to the buffer

add_bits(*values: list[bool]*) → None

Add a collection of bits to be encoded.

If these are less than a multiple of eight, they will be left padded with 0 bits to make it so.

Parameters
`values` – The value to add to the buffer

add_string(*value: str*) → None

Add a string to the buffer.

Parameters
`value` – The value to add to the buffer

build() → list[bytes]

Return the payload buffer as a list.

This list is two bytes per element and can thus be treated as a list of registers.

Returns
The payload buffer as a list

encode() → bytes

Get the payload buffer encoded in bytes.

reset() → None

Reset the payload buffer.

to_coils() → list[bool]

Convert the payload buffer into a coil layout that can be used as a context block.

Returns
The coil layout to use as a block

to_registers()

Convert the payload buffer to register layout that can be used as a context block.

Returns
The register layout to use as a block

class pymodbus.payload.**BinaryPayloadDecoder**(*payload*, *byteorder=Endian.LITTLE*, *wordorder=Endian.BIG*)

Bases: `object`

A utility that helps decode payload messages from a modbus response message.

It really is just a simple wrapper around the struct module, however it saves time looking up the format strings. What follows is a simple example:

```
decoder = BinaryPayloadDecoder(payload)
first   = decoder.decode_8bit_uint()
second  = decoder.decode_16bit_uint()
```

classmethod bit_chunks(*coils*, *size=8*)

Return bit chunks.

decode_16bit_float()

Decode a 16 bit float from the buffer.

`decode_16bit_int()`

Decode a 16 bit signed int from the buffer.

`decode_16bit_uint()`

Decode a 16 bit unsigned int from the buffer.

`decode_32bit_float()`

Decode a 32 bit float from the buffer.

`decode_32bit_int()`

Decode a 32 bit signed int from the buffer.

`decode_32bit_uint()`

Decode a 32 bit unsigned int from the buffer.

`decode_64bit_float()`

Decode a 64 bit float(double) from the buffer.

`decode_64bit_int()`

Decode a 64 bit signed int from the buffer.

`decode_64bit_uint()`

Decode a 64 bit unsigned int from the buffer.

`decode_8bit_int()`

Decode a 8 bit signed int from the buffer.

`decode_8bit_uint()`

Decode a 8 bit unsigned int from the buffer.

`decode_bits(`*package_len=1*`)`

Decode a byte worth of bits from the buffer.

`decode_string(`*size=1*`)`

Decode a string from the buffer.

Parameters

`size` – The size of the string to decode

`classmethod fromCoils(`*coils*, *byteorder=Endian.LITTLE*, *_wordorder=Endian.BIG*`)`

Initialize a payload decoder with the result of reading of coils.

`classmethod fromRegisters(`*registers*, *byteorder=Endian.LITTLE*, *wordorder=Endian.BIG*`)`

Initialize a payload decoder.

With the result of reading a collection of registers from a modbus device.

The registers are treated as a list of 2 byte values. We have to do this because of how the data has already been decoded by the rest of the library.

Parameters

- `registers` – The register results to initialize with
- `byteorder` – The Byte order of each word
- `wordorder` – The endianness of the word (when wordcount is >= 2)

Returns

An initialized PayloadDecoder

Raises
ParameterException –

reset()

Reset the decoder pointer back to the start.

skip_bytes(*nbytes*)

Skip n bytes in the buffer.

Parameters
nbytes – The number of bytes to skip

Contains base classes for modbus request/response/error packets.

class pymodbus.pdu.**ExceptionResponse**(*function_code*, *exception_code=None*, ***kwargs*)

Bases: ModbusResponse

Base class for a modbus exception PDU.

ExceptionOffset = 128

decode(*data*)

Decode a modbus exception response.

Parameters
data – The packet data to decode

encode()

Encode a modbus exception response.

Returns
The encoded exception packet

class pymodbus.pdu.**IllegalFunctionRequest**(*function_code*, ***kwargs*)

Bases: ModbusRequest

Define the Modbus slave exception type "Illegal Function".

This exception code is returned if the slave:

```
- does not implement the function code **or**
- is not in a state that allows it to process the function
```

ErrorCode = 1

decode(*_data*)

Decode so this failure will run correctly.

execute(*_context*)

Build an illegal function request error response.

Returns
The error response packet

class pymodbus.pdu.**ModbusExceptions**

Bases: object

An enumeration of the valid modbus exceptions.

Acknowledge = 5

`GatewayNoResponse = 11`

`GatewayPathUnavailable = 10`

`IllegalAddress = 2`

`IllegalFunction = 1`

`IllegalValue = 3`

`MemoryParityError = 8`

`NegativeAcknowledge = 7`

`SlaveBusy = 6`

`SlaveFailure = 4`

`classmethod decode`(*code*)

Give an error code, translate it to a string error name.

Parameters
`code` – The code number to translate

`class pymodbus.pdu.ModbusRequest`(*slave=0*, ***kwargs*)

Bases: `ModbusPDU`

Base class for a modbus request PDU.

`doException`(*exception*)

Build an error response based on the function.

Parameters
`exception` – The exception to return

Raises
An exception response

`function_code = -1`

`class pymodbus.pdu.ModbusResponse`(*slave=0*, ***kwargs*)

Bases: `ModbusPDU`

Base class for a modbus response PDU.

`should_respond`

A flag that indicates if this response returns a result back to the client issuing the request

`_rtu_frame_size`

Indicates the size of the modbus rtu response used for calculating how much to read.

`function_code = 0`

`isError()` → bool

Check if the error is a success or failure.

`should_respond = True`

Register Reading Request/Response.

`class pymodbus.register_read_message.ReadHoldingRegistersRequest(address=None, count=None, slave=0, **kwargs)`

Bases: `ReadRegistersRequestBase`

Read holding registers.

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.

execute(*context*)

Run a read holding request against a datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized *ReadHoldingRegistersResponse*, or an *ExceptionResponse* if an error occurred

function_code = 3

function_code_name = 'read_holding_registers'

`class pymodbus.register_read_message.ReadHoldingRegistersResponse(values=None, **kwargs)`

Bases: *ReadRegistersResponseBase*

Read holding registers.

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 requested registers can be found in the .registers list.

function_code = 3

`class pymodbus.register_read_message.ReadInputRegistersRequest(address=None, count=None, slave=0, **kwargs)`

Bases: `ReadRegistersRequestBase`

Read input registers.

This function code is used to read from 1 to approx. 125 contiguous input 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 input registers numbered 1-16 are addressed as 0-15.

execute(*context*)

Run a read input request against a datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized *ReadInputRegistersResponse*, or an *ExceptionResponse* if an error occurred

function_code = 4

function_code_name = 'read_input_registers'

`class pymodbus.register_read_message.`**`ReadInputRegistersResponse`**`(`*values=None*, ***kwargs*`)`

Bases: *ReadRegistersResponseBase*

Read/write input registers.

This function code is used to read from 1 to approx. 125 contiguous input 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 input registers numbered 1-16 are addressed as 0-15.

The requested registers can be found in the .registers list.

`function_code = 4`

`class pymodbus.register_read_message.`**`ReadRegistersResponseBase`**`(`*values*, *slave=0*, ***kwargs*`)`

Bases: `ModbusResponse`

Base class for responding to a modbus register read.

The requested registers can be found in the .registers list.

`decode``(`*data*`)`

Decode a register response packet.

Parameters
`data` – The request to decode

`encode``()`

Encode the response packet.

Returns
The encoded packet

`getRegister``(`*index*`)`

Get the requested register.

Parameters
`index` – The indexed register to retrieve

Returns
The request register

`registers`

A list of register values

`class pymodbus.register_read_message.`**`ReadWriteMultipleRegistersRequest`**`(`***kwargs*`)`

Bases: `ModbusRequest`

Read/write multiple registers.

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction. The write operation is performed before the read.

Holding registers are addressed starting at zero. Therefore holding registers 1-16 are addressed in the PDU as 0-15.

The request specifies the starting address and number of holding registers to be read as well as the starting address, number of holding registers, and the data to be written. The byte count specifies the number of bytes to follow in the write data field."

`decode``(`*data*`)`

Decode the register request packet.

Parameters
`data` – The request to decode

`encode()`

Encode the request packet.

Returns
The encoded packet

`execute(`*context*`)`

Run a write single register request against a datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized *ReadWriteMultipleRegistersResponse*, or an *ExceptionResponse* if an error occurred

`function_code = 23`

`function_code_name = 'read_write_multiple_registers'`

`get_response_pdu_size()`

Get response pdu size.

Func_code (1 byte) + Byte Count(1 byte) + 2 * Quantity of Coils (n Bytes) :return:

`class pymodbus.register_read_message.ReadWriteMultipleRegistersResponse(`*values=None*, ***kwargs*`)`

Bases: `ModbusResponse`

Read/write multiple registers.

The normal response contains the data from the group of registers that were read. The byte count field specifies the quantity of bytes to follow in the read data field.

The requested registers can be found in the .registers list.

`decode(`*data*`)`

Decode the register response packet.

Parameters
`data` – The response to decode

`encode()`

Encode the response packet.

Returns
The encoded packet

`function_code = 23`

Register Writing Request/Response Messages.

`class pymodbus.register_write_message.MaskWriteRegisterRequest(`*address=0*, *and_mask=65535*, *or_mask=0*, ***kwargs*`)`

Bases: `ModbusRequest`

This function code is used to modify the contents.

Of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function can be used to set or clear individual bits in the register.

`decode(data)`

Decode the incoming request.

Parameters
`data` – The data to decode into the address

`encode()`

Encode the request packet.

Returns
The byte encoded packet

`execute(context)`

Run a mask write register request against the store.

Parameters
`context` – The datastore to request from

Returns
The populated response

`function_code = 22`

`function_code_name = 'mask_write_register'`

`class pymodbus.register_write_message.MaskWriteRegisterResponse(address=0, and_mask=65535, or_mask=0, **kwargs)`

Bases: `ModbusResponse`

The normal response is an echo of the request.

The response is returned after the register has been written.

`decode(data)`

Decode a the response.

Parameters
`data` – The packet data to decode

`encode()`

Encode the response.

Returns
The byte encoded message

`function_code = 22`

`class pymodbus.register_write_message.WriteMultipleRegistersRequest(address=None, values=None, slave=None, **kwargs)`

Bases: `ModbusRequest`

This function code is used to write a block.

Of contiguous registers (1 to approx. 120 registers) in a remote device.

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

`decode(data)`

Decode a write single register packet packet request.

Parameters
`data` – The request to decode

`encode()`

Encode a write single register packet packet request.

Returns
The encoded packet

`execute(`*context*`)`

Run a write single register request against a datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized response, exception message otherwise

`function_code = 16`

`function_code_name = 'write_registers'`

`get_response_pdu_size()`

Get response pdu size.

Func_code (1 byte) + Starting Address (2 byte) + Quantity of Registers (2 Bytes) :return:

`class pymodbus.register_write_message.WriteMultipleRegistersResponse(`*address=None*, *count=None*, ***kwargs*`)`

Bases: `ModbusResponse`

The normal response returns the function code.

Starting address, and quantity of registers written.

`decode(`*data*`)`

Decode a write single register packet packet request.

Parameters
`data` – The request to decode

`encode()`

Encode a write single register packet packet request.

Returns
The encoded packet

`function_code = 16`

`class pymodbus.register_write_message.WriteSingleRegisterRequest(`*address=None*, *value=None*, *slave=None*, ***kwargs*`)`

Bases: `ModbusRequest`

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.

`decode(`*data*`)`

Decode a write single register packet packet request.

Parameters
`data` – The request to decode

`encode()`

Encode a write single register packet packet request.

Returns
The encoded packet

`execute(`*context*`)`

Run a write single register request against a datastore.

Parameters
`context` – The datastore to request from

Returns
An initialized response, exception message otherwise

`function_code = 6`

`function_code_name = 'write_register'`

`get_response_pdu_size()`

Get response pdu size.

Func_code (1 byte) + Register Address(2 byte) + Register Value (2 bytes) :return:

`class pymodbus.register_write_message.WriteSingleRegisterResponse(`*address=None*, *value=None*, ***kwargs*`)`

Bases: `ModbusResponse`

The normal response is an echo of the request.

Returned after the register contents have been written.

`decode(`*data*`)`

Decode a write single register packet packet request.

Parameters
`data` – The request to decode

`encode()`

Encode a write single register packet packet request.

Returns
The encoded packet

`function_code = 6`

`get_response_pdu_size()`

Get response pdu size.

Func_code (1 byte) + Starting Address (2 byte) + And_mask (2 Bytes) + OrMask (2 Bytes) :return:

Collection of transaction based abstractions.

`class pymodbus.transaction.DictTransactionManager(`*client*, ***kwargs*`)`

Bases: *`ModbusTransactionManager`*

Old alias for ModbusTransactionManager.

class pymodbus.transaction.**ModbusAsciiFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus ASCII Frame Controller.

[Start][Address][Function][Data][LRC][End]
1c 2c 2c Nc 2c 2c

- data can be 0 - 2x252 chars
- end is "\r\n" (Carriage return line feed), however the line feed character can be changed via a special command
- start is ":"

This framer is used for serial transmission. Unlike the RTU protocol, the data in this framer is transferred in plain text ascii.

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The request/response to send

Returns
The encoded packet

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

method = 'ascii'

class pymodbus.transaction.**ModbusBinaryFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus Binary Frame Controller.

[Start][Address][Function][Data][CRC][End]
1b 1b 1b Nb 2b 1b

- data can be 0 - 2x252 chars
- end is "}"
- start is "{"

The idea here is that we implement the RTU protocol, however, instead of using timing for message delimiting, we use start and end of message characters (in this case { and }). Basically, this is a binary framer.

The only case we have to watch out for is when a message contains the { or } characters. If we encounter these characters, we simply duplicate them. Hopefully we will not encounter those characters that often and will save a little bit of bandwitch without a real-time system.

Protocol defined by jamod.sourceforge.net.

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The request/response to send

Returns
The encoded packet

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

method = 'binary'

class pymodbus.transaction.**ModbusRtuFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus RTU Frame controller.

[Start Wait] [Address][Function Code] [Data][CRC][End Wait]
3.5 chars 1b 1b Nb 2b 3.5 chars

Wait refers to the amount of time required to transmit at least x many characters. In this case it is 3.5 characters. Also, if we receive a wait of 1.5 characters at any point, we must trigger an error message. Also, it appears as though this message is little endian. The logic is simplified as the following:

```
block-on-read:
    read until 3.5 delay
    check for errors
    decode
```

The following table is a listing of the baud wait times for the specified baud rates:

```
------------------------------------------------------------------
 Baud  1.5c (18 bits)   3.5c (38 bits)
------------------------------------------------------------------
 1200   13333.3 us       31666.7 us
 4800    3333.3 us        7916.7 us
 9600    1666.7 us        3958.3 us
19200     833.3 us        1979.2 us
38400     416.7 us         989.6 us
------------------------------------------------------------------
1 Byte = start + 8 bits + parity + stop = 11 bits
(1/Baud)(bits) = delay seconds
```

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The populated request/response to send

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*_single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

method = 'rtu'

recvPacket(*size*)

Receive packet from the bus with specified len.

Parameters
`size` – Number of bytes to read

Returns

sendPacket(*message*)

Send packets on the bus with 3.5char delay between frames.

Parameters
`message` – Message to be sent over the bus

Returns

class pymodbus.transaction.**ModbusSocketFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus Socket Frame controller.

Before each modbus TCP message is an MBAP header which is used as a message frame. It allows us to easily separate messages as follows:

```
[         MBAP Header         ] [ Function Code] [ Data ]         [ tid ][ pid ][␣
↪length ][ uid ]
  2b     2b     2b        1b           1b           Nb

while len(message) > 0:
    tid, pid, length`, uid = struct.unpack(">HHHB", message)
    request = message[0:7 + length - 1`]
    message = [7 + length - 1:]

* length = uid + function code + data
* The -1 is to account for the uid byte
```

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The populated request/response to send

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*single*, *callback*, *slave*, *tid=None*, ***kwargs*)

Process new packet pattern.

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 // N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

`method = 'socket'`

class pymodbus.transaction.**ModbusTlsFramer**(*decoder*, *client=None*)

Bases: `ModbusFramer`

Modbus TLS Frame controller.

No prefix MBAP header before decrypted PDU is used as a message frame for Modbus Security Application Protocol. It allows us to easily separate decrypted messages which is PDU as follows:

[Function Code] [Data]
1b Nb

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The populated request/response to send

decode_data(*data*)

Decode data.

frameProcessIncomingPacket(*single*, *callback*, *slave*, *_tid=None*, ***kwargs*)

Process new packet pattern.

`method = 'tls'`

class pymodbus.transaction.**ModbusTransactionManager**(*client*, ***kwargs*)

Bases: `object`

Implement a transaction for a manager.

The transaction protocol can be represented by the following pseudo code:

```
count = 0
do
  result = send(message)
  if (timeout or result == bad)
     count++
  else break
while (count < 3)
```

This module helps to abstract this away from the framer and protocol.

Results are keyed based on the supplied transaction id.

addTransaction(*request*, *tid=None*)

Add a transaction to the handler.

This holds the request in case it needs to be resent. After being sent, the request is removed.

Parameters

- **request** – The request to hold on to
- **tid** – The overloaded transaction id to use

delTransaction(*tid*)

Remove a transaction matching the referenced tid.

Parameters
tid – The transaction to remove

execute(*request*)

Start the producer to send the next request to consumer.write(Frame(request)).

getNextTID()

Retrieve the next unique transaction identifier.

This handles incrementing the identifier after retrieval

Returns
The next unique transaction identifier

getTransaction(*tid*)

Return a transaction matching the referenced tid.

If the transaction does not exist, None is returned

Parameters
tid – The transaction to retrieve

reset()

Reset the transaction identifier.

Modbus Utilities.

A collection of utilities for packing data, unpacking data computing checksums, and decode checksums.

pymodbus.utilities.**default**(*value*)

Return the default value of object.

Parameters
value – The value to get the default of

Returns
The default value

pymodbus.utilities.**pack_bitstring**(*bits: list[bool]*) → bytes

Create a bytestring out of a list of bits.

Parameters
bits – A list of bits

example:

```
bits   = [False, True, False, True]
result = pack_bitstring(bits)
```

pymodbus.utilities.**rtuFrameSize**(*data*, *byte_count_pos*)

Calculate the size of the frame based on the byte count.

Parameters

- **data** – The buffer containing the frame.
- **byte_count_pos** – The index of the byte count in the buffer.

Returns
The size of the frame.

The structure of frames with a byte count field is always the same:

- first, there are some header fields
- then the byte count field

- then as many data bytes as indicated by the byte count,
- finally the CRC (two bytes).

To calculate the frame size, it is therefore sufficient to extract the contents of the byte count field, add the position of this field, and finally increment the sum by three (one byte for the byte count field, two for the CRC).

`pymodbus.utilities.unpack_bitstring(data: bytes) → list[bool]`

Create bit list out of a bytestring.

Parameters

`data` – The modbus data packet to decode

example:

```
bytes  = "bytes to decode"
result = unpack_bitstring(bytes)
```

10.6 Architecture

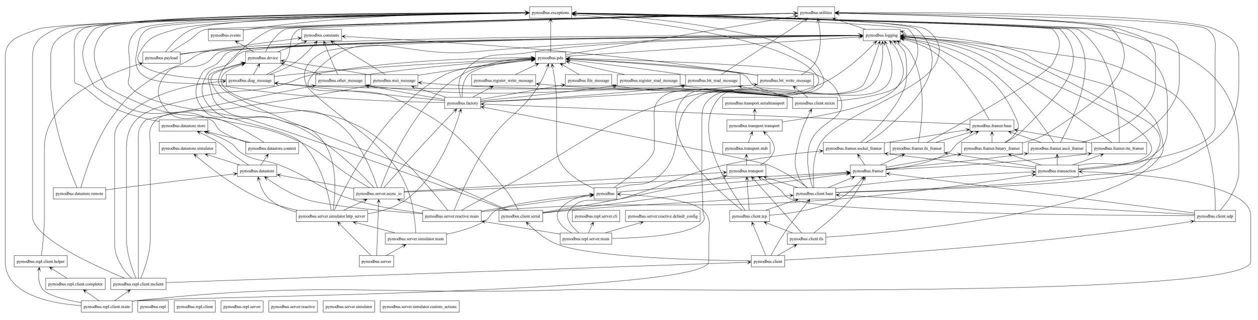
The internal structure of pymodbus is a bit complicated, mostly due to the mixture of sync and async.

The overall architecture can be viewed as:

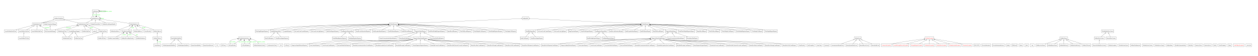
Client classes (interface to applications) mixin (interface with all requests defined as methods) transaction (handles transactions and allow concurrent calls) framers (add pre/post headers to make a valid package) transport (handles actual transportation)

Server classes (interface to applications) datastores (handles registers/values to be returned) transaction (handles transactions and allow concurrent calls) framers (add pre/post headers to make a valid package) transport (handles actual transportation)

In detail the packages can viewed as:



In detail the classes can be viewed as:



PYTHON MODULE INDEX

INDEX