PyModbus

Release 3.2.x

Open Source volunteers

Mar 11, 2023

CONTENTS:

CHAPTER

ONE

PYMODBUS - A PYTHON MODBUS STACK

We are happy to announce that we have a new home: pymodbus-dev, which is pure 100% FOSS. The move from a company organization to pymodbus-dev was done to allow a 100% openness in the spirit of FOSS.

1.1 Supported versions

Version 2.5.3 is the last 2.x release (Supports python 2.7.x - 3.7).

Version 3.2.0 is the current release (Supports Python >=3.8).

Important: All API changes after 3.0.0 are documented in API_changes.rst

1.2 Summary

Pymodbus is a full Modbus protocol implementation using a synchronous or asynchronous (using asyncio) core.

The modbus protocol documentation can be found here

Supported modbus communication modes: tcp, rtu-over-tcp, udp, serial, tls

Pymodbus can be used without any third party dependencies (aside from pyserial) and is a very lightweight project.

Pymodbus also provides a lot of ready to use examples as well as a server/client simulator which can be controlled via a REST API and can be easily integrated into test suites.

Requires Python >= 3.8

The tests are run against Python 3.8, 3.9, 3.10, 3.11 on Windows, Linux and MacOS.

1.3 Features

1.3.1 Client Features

- Full read/write protocol on discrete and register
- Most of the extended protocol (diagnostic/file/pipe/setting/information)
- TCP, RTU-OVER-TCP, UDP, TLS, Serial ASCII, Serial RTU, and Serial Binary
- asynchronous(powered by asyncio) and synchronous versions
- Payload builder/decoder utilities
- Pymodbus REPL for quick tests
- Customizable framer to allow for custom implementations

1.3.2 Server Features

- Can function as a fully implemented modbus server
- TCP, RTU-OVER-TCP, UDP, TLS, Serial ASCII, Serial RTU, and Serial Binary
- asynchronous and synchronous versions
- Full server control context (device information, counters, etc)
- A number of backend contexts (database, redis, sqlite, a slave device) as datastore

1.3.2.1 Use Cases

Although most system administrators will find little need for a Modbus server on any modern hardware, they may find the need to query devices on their network for status (PDU, PDR, UPS, etc). Since the library is written in python, it allows for easy scripting and/or integration into their existing solutions.

Continuing, most monitoring software needs to be stress tested against hundreds or even thousands of devices (why this was originally written), but getting access to that many is unwieldy at best.

The pymodbus server will allow a user to test as many devices as their base operating system will allow (*allow* in this case means how many Virtual IP addresses are allowed).

For more information please browse the project documentation:

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

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('127.0.0.1')
client.connect()
client.write_coil(1, True)
result = client.read_coils(1,1)
```

(continues on next page)

(continued from previous page)

```
print(result.bits[0])
client.close()
```

For more advanced examples, check out the 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.

1.4.1 Examples Directory structure

```
examples      -> Essential examples guaranteed to work (tested with our CI)
├── v2.5.3    -> Examples not updated to version 3.0.0.
├── contrib   -> Examples contributed by contributors.
```

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.

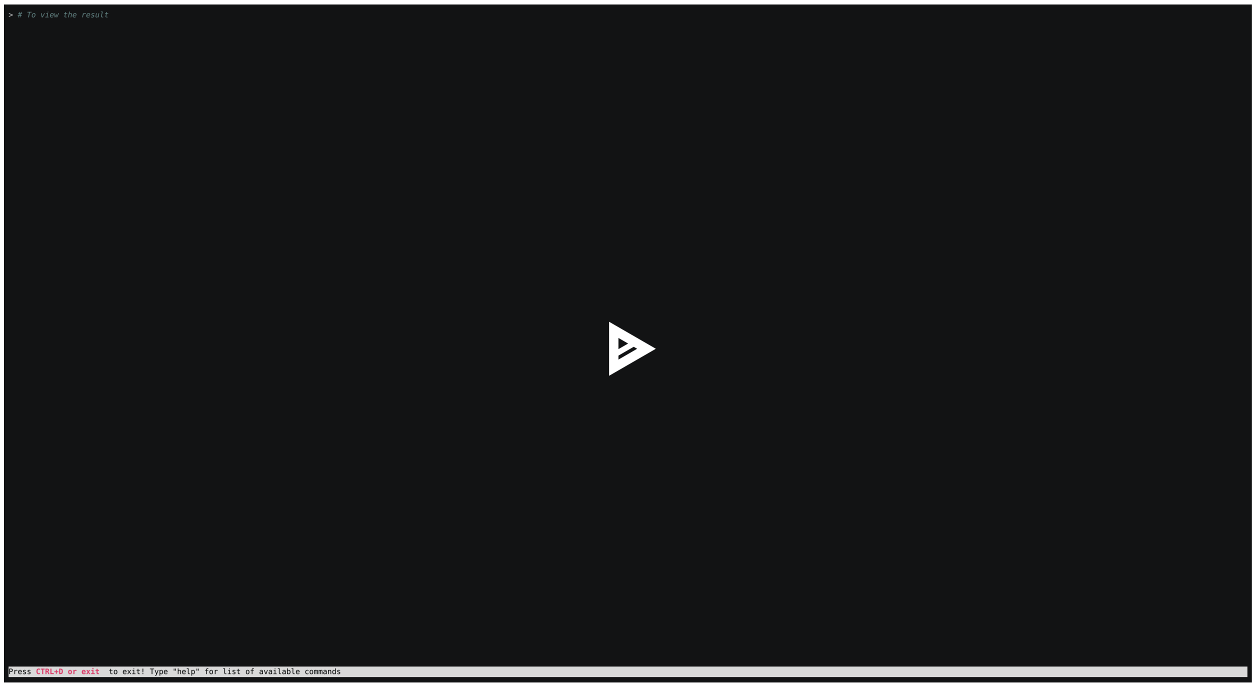
1.5 Pymodbus REPL (Read Evaluate Print Loop)

Warning The Pymodbus REPL documentation is not updated.

1.5.1 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

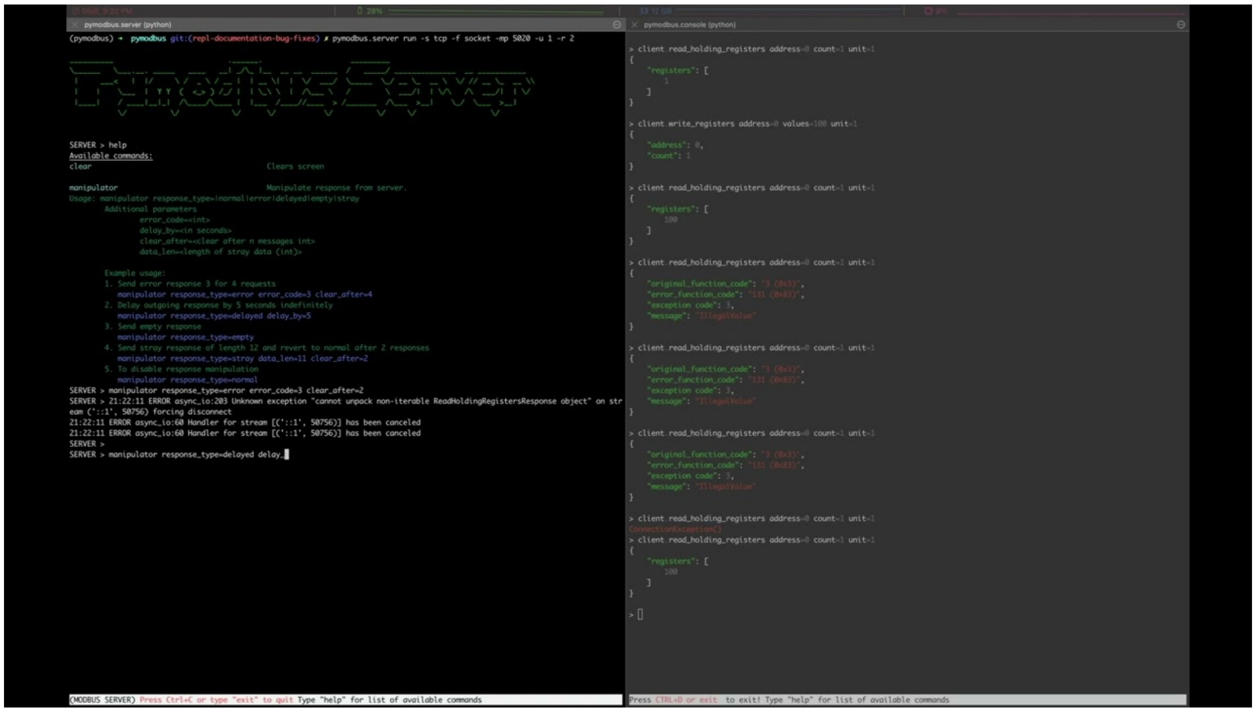


```
> # To view the result
Press CTRL+D or exit  to exit! Type "help" for list of available commands
```

1.5.2 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.server (python)

```
(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
```

pymodbus.console (python)

```
> 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
```

1.6 Installing

You can install using pip or easy install by issuing the following commands in a terminal window (make sure you have correct permissions or a virtualenv currently running):

    pip install -U pymodbus

This will install a base version of pymodbus.

To install pymodbus with options run:

    pip install -U pymodbus[<option>,. . .]

Available options are:

- **repl**, installs pymodbus REPL.
- **serial**, installs serial drivers.
- **datastore**, installs databases (SQLAlchemy and Redis) for datastore.
- **documentation**, installs tools to generate documentation.
- **development**, installs development tools needed to enable test/check of pymodbus changes.

Or to install a specific release:

    pip install -U pymodbus==X.Y.Z

Otherwise you can pull the trunk source and install from there:

```
git clone git://github.com/pymodbus-dev/pymodbus.git
cd pymodbus
pip install -r requirements.txt
```

Before cloning the repo, you need to install python3 (preferable 3.10) and make a virtual environment:

```
python3 -m venv /path/to/new/virtual/environment
```

To activeate the virtual environment please do:

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

To get latest release (for now v3.0.0 with Python 3.8 support):

```
git checkout master
```

To get bleeding edge:

```
git checkout dev
```

To get a specific version:

git checkout tags/vX.Y.Z -b vX.Y.Z

Then:

pip install -r requirements.txt

pip install -e .

pre-commit –install

This installs pymodbus in your virtual environment with pointers directly to the pymodbus directory, so any change you make is immediately available as if installed. It will also install *pre-commit* git hooks.

Either method will install all the required dependencies (at their appropriate versions) for your current python distribution.

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.
- All releases are tagged with **vX.Y.Z** (e.g. v2.5.3)
- All prereleases are tagged with **vX.Y.ZrcQ** (e.g. v3.0.0.0rc1)

If a maintenance release of an old version is needed (e.g. v2.5.4), the release tag is used to create a branch with the same name, and maintenance development is merged here.

1.7 Install with Docker

Pull the latest image on dev branch with `docker pull ghcr.io/pymodbus-dev/pymodbus:dev`:

```
doker pull ghcr.io/pymodbus-dev/pymodbus:dev
dev: Pulling from pymodbus-dev/pymodbus
548fcab5fe88: Pull complete
a4d3f9f008ef: Pull complete
eb83acb05730: Pull complete
71cd28d529fd: Pull complete
66607ad8f4f0: Pull complete
64dff4c66d3b: Pull complete
8b26e5718a7a: Pull complete
dc87d8707532: Pull complete
Digest: sha256:cfeee09a87dde5863574779416490fd47cacbb6f37332a3cdaf995c416e16b69
Status: Downloaded newer image for ghcr.io/pymodbus-dev/pymodbus:dev
ghcr.io/pymodbus-dev/pymodbus:dev
```

The image when run with out any further options supplied will start a repl server in non interactive mode.:

```
 docker run -it --rm -p 8080:8080 -p 5020:5020 ghcr.io/pymodbus-dev/pymodbus:dev

Reactive Modbus Server started.
======== Running on http://127.0.0.1:8080 ========

===========================================================================
Example Usage:
curl -X POST http://127.0.0.1:8080 -d "{"response_type": "error", "error_code": 4}"
===========================================================================
```

The default command can be overridden by passing any valid command at the end.:

```
 docker run -p 8080:8080 -p 5020:5020 -it --rm ghcr.io/pymodbus-dev/pymodbus:dev bash -c
↪"pymodbus.server --help"

 Usage: pymodbus.server [OPTIONS] COMMAND [ARGS]...

 Reactive Modbus server

─ Options␣
↪──────────────────────────────────────────────────────────────────────────────────────
│ --host                                    TEXT     Host address [default: localhost]  ␣
↪                                    │
│ --web-port                                INTEGER  Web app port [default: 8080]       ␣
↪                                    │
│                       -b                           Support broadcast messages         ␣
↪                                    │
│ --repl                  --no-repl                  Enable/Disable repl for server␣
↪[default: repl]                           │
│ --verbose               --no-verbose               Run with debug logs enabled for␣
↪pymodbus [default: no-verbose]          │
│ --install-completion                               Install completion for the current␣
↪shell.                               │
```

(continues on next page)

(continued from previous page)

```
│ --show-completion                                    Show completion for the current␣
↪shell, to copy it or customize the      │
│                                                      installation.                   ␣
↪                                      │
│ --help                                               Show this message and exit.     ␣
↪                                      │
────────────────────────────────────────────────────────────────────────────────────────────
─ Commands␣
↪───────────────────────────────────────────────────────────────────────────────────────────
│ run              Run Reactive Modbus server.                                         ␣
↪                                      │
────────────────────────────────────────────────────────────────────────────────────────────
```

To check the repl console.:

```
 docker run -p 8080:8080 -p 5020:5020 -it --rm ghcr.io/pymodbus-dev/pymodbus:dev bash -c
↪"pymodbus.console --help"
Usage: pymodbus.console [OPTIONS] COMMAND [ARGS]...

  Run Main.

Options:
  --version                        Show the version and exit.
  --verbose                        Verbose logs
  --broadcast-support              Support broadcast messages
  --retry-on-empty                 Retry on empty response
  --retry-on-error                 Retry on error response
  --retries INTEGER                Retry count
  --reset-socket / --no-reset-socket
                                   Reset client socket on error
  --help                           Show this message and exit.

Commands:
  serial  Define serial communication.
  tcp     Define TCP.
```

To run examples (assuming server is running).

```
 docker run -p 8080:8080 -p 5020:5020 -it --rm ghcr.io/pymodbus-dev/pymodbus:dev bash -c
↪"examples/client_async.py"
14:52:13 INFO  client_async:44 ### Create client object
14:52:13 INFO  client_async:120 ### Client starting
```

1.8 Current Work In Progress

The maintenance team is very small with limited capacity and few modbus devices.

However, if you would like your device tested, we accept devices via mail or by IP address.

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
- The remaining edges of the protocol (that we think no one uses)

1.9 Development Instructions

The current code base is compatible python >= 3.8. Here are some of the common commands to perform a range of activities

> pip install -r requirements.txt install all requirements
>
> pip install -e . source directory is "release", useful for testing
>
> ./check_ci run the same checks as CI runs on a pull request.
>
> OBS: tox is no longer supported.

1.10 Generate documentation

> cd doc make clean make html

1.11 Contributing

Just fork the repo and raise your PR against *dev* branch.

Here are some of the items waiting to be done:
> https://github.com/pymodbus-dev/pymodbus/blob/dev/doc/TODO

1.12 License Information

Pymodbus is built on top of code developed from/by:

- Copyright (c) 2001-2005 S.W.A.C. GmbH, Germany.
- Copyright (c) 2001-2005 S.W.A.C. Bohemia s.r.o., Czech Republic.
- Hynek Petrak, https://github.com/HynekPetrak

Released under the BSD License

CHAPTER

TWO

CHANGELOGS

2.1 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)

Thanks to:
AKJ7, Alex, Alex Ruddick, banana-sun, cgernert, Jakob Ruhe, James Braza, jan Iversen

2.2 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)

Thanks to:
duc996, jan iversen

2.3 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)

Thanks to:

Alex, Alexandre CUER, dhoomakethu, jan iversen, peufeu2

2.4 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)

Thanks to:

Alex, Alexandre CUER, Dries, jan iversen, peufeu2

2.5 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)

Thanks to:

Alex, banana-sun, Chris Hung, dhoomakethu, jan iversen, Matthias Straka, Pavel Kostromitinov,

2.6 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)

Thanks to:

Alex, Alexandre CUER, Blaise Thompson, dhoomakethu, Gao Fang, jan Iversen, Joe Burmeister, Sebastian Machuca, Thijs W, WouterTuinstra

2.7 version 3.0.1

- Faulty release!

2.8 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)

2.9 version 3.0.0dev5

- 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)

2.10 version 3.0.0dev4

- Documentation updates
- PEP8 compatibale code
- More tooling and CI updates

2.11 version 3.0.0dev3

- 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

2.12 version 3.0.0dev2

- 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)

2.13 version 3.0.0dev1

- 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

2.14 version 3.0.0dev0

- Support python3.7 and above
- Support creating asyncio clients from with in coroutines.

2.15 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

2.16 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.

2.17 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

2.18 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

2.19 version 2.5.0rc3

- 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.

2.20 version 2.5.0rc2

- Documentation updates
- Disable *strict* mode by default.
- Fix *ReportSlaveIdRequest* request
- Sparse datablock initialization updates.

2.21 version 2.5.0rc1

- 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

2.22 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.

2.23 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

2.24 Version 2.3.0rc1

- 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.

2.25 Version 2.2.0

NOTE: Supports python 3.7, async client is now moved to pymodbus/client/asynchronous

```
from pymodbus.client.asynchronous import ModbusTcpClient
```

- 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

```
# Now
# DEBUG:pymodbus.factory:Factory Request[ReadInputRegistersRequest: 4]
# Before
# DEBUG:pymodbus.factory:Factory Request[4]
```

2.26 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.

2.27 Version 2.0.1

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

2.28 Version 2.0.0

Note This is a Major release and might affect your existing Async client implementation. Refer examples on how to use the latest async clients.

- 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.

2.29 Version 2.0.0rc1

Note This is a Major release and might affect your existing Async client implementation. Refer examples on how to use the latest async clients.

- Async client implementation based on Tornado, Twisted and asyncio

2.30 Version 1.5.2

- Fix serial client *is_socket_open* method

2.31 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.

```
res = client.read_holding_registers(...)
if not res.isError():

    # proceed

else:
    # handle error or raise

"""
```

- Add examples for MEI read device information request

2.32 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

client = Client(. . . , retry_on_empty=True)

- 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

2.33 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

2.34 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

2.35 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

2.36 Version 1.3.0.rc2

- 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

2.37 Version 1.3.0.rc1

- 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

2.38 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

2.39 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

2.40 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)

2.41 Version 0.9.0

Please view the git commit log

CHAPTER THREE

CLIENT

Pymodbus offers clients 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 client`,
- `asynchronous client` using asyncio.

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

```
# create client object
client = ModbusSerial("/dev/tty")

# connect to device
client.connect()

# set/set information
rr = client.read_coils(0x01)
client.write_coil(0x01, values)

# disconnect device
client.close()
```

and a asynchronous example:

```
# create client object
async_client = AsyncModbusSerial("/dev/tty")

# connect to device
await async_client.connect()

# set/set information
rr = await async_client.read_coils(0x01)
await async_client.write_coil(0x01, values)
```

(continues on next page)

(continued from previous page)

```

# disconnect device
await async_client.close()
```

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.

3.1 Transport classes

`class pymodbus.client.ModbusBaseClient`(*framer: Type[ModbusFramer] = None*, *timeout: str | float = 3*, *retries: str | int = 3*, *retry_on_empty: bool = False*, *close_comm_on_error: bool = False*, *strict: bool = True*, *broadcast_enable: bool = False*, *reconnect_delay: int = 100*, *reconnect_delay_max: int = 300000*, *on_reconnect_callback: Optional[Callable[[], None]] = None*, ***kwargs: Any*)

Bases: *ModbusClientMixin*

ModbusBaseClient

Parameters common to all clients:

Parameters

- **framer** – (optional) Modbus Framer class.
- **timeout** – (optional) Timeout for a request, in seconds.
- **retries** – (optional) Max number of retries per request.
- **retry_on_empty** – (optional) Retry on empty response.
- **close_comm_on_error** – (optional) Close connection on error.
- **strict** – (optional) Strict timing, 1.5 character between requests.
- **broadcast_enable** – (optional) True to treat id 0 as broadcast address.
- **reconnect_delay** – (optional) Minimum delay in milliseconds before reconnecting.
- **reconnect_delay_max** – (optional) Maximum delay in milliseconds before reconnecting.
- **on_reconnect_callback** – (optional) Function that will be called just before a reconnection attempt.
- **kwargs** – (optional) Experimental parameters.

Tip: Common parameters and all external methods for all clients are documented here, and not repeated with each client.

Tip: **delay_ms** doubles automatically with each unsuccessful connect, from **reconnect_delay** to **reconnect_delay_max**. Set *reconnect_delay=0* to avoid automatic reconnection.

`ModbusBaseClient` is normally not referenced outside `pymodbus`, unless you want to make a custom client.

Custom client class **must** inherit `ModbusBaseClient`, example:

```
from pymodbus.client import ModbusBaseClient

class myOwnClient(ModbusBaseClient):

    def __init__(self, **kwargs):
        super().__init__(kwargs)

def run():
    client = myOwnClient(...)
    client.connect()
    rr = client.read_coils(0x01)
    client.close()
```

Application methods, common to all clients:

`register`(*custom_response_class: ModbusResponse*) → None

Register a custom response class with the decoder (call **sync**).

Parameters
`custom_response_class` – (optional) Modbus response class.

Raises
`MessageRegisterException` – Check exception text.

Use register() to add non-standard responses (like e.g. a login prompt) and have them interpreted automatically.

`connect()`

Connect to the modbus remote host (call **sync/async**).

Raises
`ModbusException` – Different exceptions, check exception text.

Remark Retries are handled automatically after first successful connect.

`is_socket_open()` → bool

Return whether socket/serial is open or not (call **sync**).

`idle_time()` → float

Time before initiating next transaction (call **sync**).

Applications can call message functions without checking idle_time(), this is done automatically.

`reset_delay()` → None

Reset wait time before next reconnect to minimal period (call **sync**).

`execute`(*request: Optional[ModbusRequest] = None*) → ModbusResponse

Execute request and get response (call **sync/async**).

Parameters
`request` – The request to process

Returns
The result of the request execution

Raises
`ConnectionException` – Check exception text.

`close()` → None
Close the underlying socket connection (call **sync/async**).

`client_made_connection`(*protocol*)
Run transport specific connection.

`client_lost_connection`(*protocol*)
Run transport specific connection lost.

`datagram_received`(*data*, *_addr*)
Receive datagram.

`async async_execute`(*request=None*)
Execute requests asynchronously.

`connection_made`(*transport*)
Call when a connection is made.

The transport argument is the transport representing the connection.

`connection_lost`(*reason*)
Call when the connection is lost or closed.

The argument is either an exception object or None

`data_received`(*data*)
Call when some data is received.

data is a non-empty bytes object containing the incoming data.

`create_future()`
Help function to create asyncio Future object.

`raise_future`(*my_future*, *exc*)
Set exception of a future if not done.

`property async_connected`
Return connection status.

`async async_close()`
Close connection.

`class pymodbus.client.AsyncModbusSerialClient`(*port: str*, *framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.rtu_framer.ModbusRtuFramer'>*, *baudrate: int = 19200*, *bytesize: int = 8*, *parity: str = 'N'*, *stopbits: int = 1*, *handle_local_echo: bool = False*, ***kwargs: ~typing.Any*)

Bases: `ModbusBaseClient`, `Protocol`

AsyncModbusSerialClient.

Parameters

- **port** – Serial port used for communication.
- **framer** – (optional) Framer class.
- **baudrate** – (optional) Bits per second.
- **bytesize** – (optional) Number of bits per byte 7-8.
- **parity** – (optional) 'E'ven, 'O'dd or 'N'one
- **stopbits** – (optional) Number of stop bits 0-2¡.
- **handle_local_echo** – (optional) Discard local echo from dongle.
- **kwargs** – (optional) Experimental parameters

The serial communication is RS-485 based, and usually used with a usb RS485 dongle.

Example:

```
from pymodbus.client import AsyncModbusSerialClient

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

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

async close()

Stop connection.

property connected

Connect internal.

async connect()

Connect Async client.

client_made_connection(*protocol*)

Notify successful connection.

client_lost_connection(*protocol*)

Notify lost connection.

class pymodbus.client.**ModbusSerialClient**(*port: str, framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.rtu_framer.ModbusRtuFramer'>, baudrate: int = 19200, bytesize: int = 8, parity: str = 'N', stopbits: int = 1, handle_local_echo: bool = False, **kwargs: ~typing.Any*)

Bases: *ModbusBaseClient*

ModbusSerialClient.

Parameters

- **port** – Serial port used for communication.
- **framer** – (optional) Framer class.
- **baudrate** – (optional) Bits per second.
- **bytesize** – (optional) Number of bits per byte 7-8.

- **parity** – (optional) 'E'ven, 'O'dd or 'N'one
- **stopbits** – (optional) Number of stop bits 0-2¡.
- **handle_local_echo** – (optional) Discard local echo from dongle.
- **kwargs** – (optional) Experimental parameters

The serial communication is RS-485 based, and usually used with a usb RS485 dongle.

Example:

```
from pymodbus.client import ModbusSerialClient

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

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

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.

class pymodbus.client.**AsyncModbusTcpClient**(*host: str*, *port: int = 502*, *framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.socket_framer.ModbusSocketFramer'>*, *source_address: ~typing.Optional[~typing.Tuple[str, int]] = None*, ***kwargs: ~typing.Any*)

Bases: *ModbusBaseClient*, Protocol

AsyncModbusTcpClient.

Parameters

- **host** – Host IP address or host name
- **port** – (optional) Port used for communication
- **framer** – (optional) Framer class

- **source_address** – (optional) source address of client
- **kwargs** – (optional) Experimental parameters

using unix domain socket can be achieved by setting host="unix:<path>"

Example:

```
from pymodbus.client import AsyncModbusTcpClient

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

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

async connect()

Initiate connection to start client.

async close()

Stop client.

client_made_connection(*protocol*)

Notify successful connection.

client_lost_connection(*protocol*)

Notify lost connection.

class pymodbus.client.ModbusTcpClient(*host: str*, *port: int = 502*, *framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.socket_framer.ModbusSocketFramer'>*, *source_address: ~typing.Optional[~typing.Tuple[str, int]] = None*, ***kwargs: ~typing.Any*)

Bases: *ModbusBaseClient*

ModbusTcpClient.

Parameters

- **host** – Host IP address or host name
- **port** – (optional) Port used for communication
- **framer** – (optional) Framer class
- **source_address** – (optional) source address of client
- **kwargs** – (optional) Experimental parameters

using unix domain socket can be achieved by setting host="unix:<path>"

Example:

```
from pymodbus.client import ModbusTcpClient

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

    client.connect()
```

(continues on next page)

(continued from previous page)

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

Remark: There are no automatic reconnect as with AsyncModbusTcpClient

property connected

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.

class pymodbus.client.**AsyncModbusTlsClient**(*host: str*, *port: int = 802*, *framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.tls_framer.ModbusTlsFramer'>*, *sslctx: ~typing.Optional[str] = None*, *certfile: ~typing.Optional[str] = None*, *keyfile: ~typing.Optional[str] = None*, *password: ~typing.Optional[str] = None*, *server_hostname: ~typing.Optional[str] = None*, ***kwargs: ~typing.Any*)

Bases: *AsyncModbusTcpClient*, Protocol

AsyncModbusTlsClient.

Parameters

- **host** – Host IP address or host name
- **port** – (optional) Port used for communication
- **framer** – (optional) Framer class
- **source_address** – (optional) Source address of client
- **sslctx** – (optional) SSLContext to use for TLS
- **certfile** – (optional) Cert file path for TLS server request
- **keyfile** – (optional) Key file path for TLS server request
- **password** – (optional) Password for for decrypting private key file
- **server_hostname** – (optional) Bind certificate to host
- **kwargs** – (optional) Experimental parameters

Example:

```
from pymodbus.client import AsyncModbusTlsClient

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

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

class pymodbus.client.ModbusTlsClient(*host: str, port: int = 802, framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.tls_framer.ModbusTlsFramer'>, sslctx: ~typing.Optional[str] = None, certfile: ~typing.Optional[str] = None, keyfile: ~typing.Optional[str] = None, password: ~typing.Optional[str] = None, server_hostname: ~typing.Optional[str] = None, **kwargs: ~typing.Any*)

Bases: *ModbusTcpClient*

ModbusTlsClient.

Parameters

- **host** – Host IP address or host name
- **port** – (optional) Port used for communication
- **framer** – (optional) Framer class
- **source_address** – (optional) Source address of client
- **sslctx** – (optional) SSLContext to use for TLS
- **certfile** – (optional) Cert file path for TLS server request
- **keyfile** – (optional) Key file path for TLS server request
- **password** – (optional) Password for decrypting private key file
- **server_hostname** – (optional) Bind certificate to host
- **kwargs** – (optional) Experimental parameters

Example:

```
from pymodbus.client import ModbusTlsClient

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

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

Remark: There are no automatic reconnect as with AsyncModbusTlsClient

property connected

Connect internal.

connect()

Connect to the modbus tls server.

class pymodbus.client.**AsyncModbusUdpClient**(*host: str*, *port: int = 502*, *framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.socket_framer.ModbusSocketFramer'>*, *source_address: ~typing.Optional[~typing.Tuple[str, int]] = None*, ***kwargs: ~typing.Any*)

Bases: *ModbusBaseClient*, Protocol, DatagramProtocol

AsyncModbusUdpClient.

Parameters

- **host** – Host IP address or host name
- **port** – (optional) Port used for communication.
- **framer** – (optional) Framer class.
- **source_address** – (optional) source address of client,
- **kwargs** – (optional) Experimental parameters

Example:

```
from pymodbus.client import AsyncModbusUdpClient

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

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

class pymodbus.client.**ModbusUdpClient**(*host: str*, *port: int = 502*, *framer: ~typing.Type[~pymodbus.framer.ModbusFramer] = <class 'pymodbus.framer.socket_framer.ModbusSocketFramer'>*, *source_address: ~typing.Optional[~typing.Tuple[str, int]] = None*, ***kwargs: ~typing.Any*)

Bases: *ModbusBaseClient*

ModbusUdpClient.

Parameters

- **host** – Host IP address or host name
- **port** – (optional) Port used for communication.
- **framer** – (optional) Framer class.
- **source_address** – (optional) source address of client,
- **kwargs** – (optional) Experimental parameters

Example:

```
from pymodbus.client import ModbusUdpClient

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

    client.connect()
```

(continues on next page)

(continued from previous page)

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

Remark: There are no automatic reconnect as with AsyncModbusUdpClient

3.2 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: `object`

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
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*) → ModbusResponse

Execute request (code ???).

Parameters
`request` – Request to send

Raises
`ModbusException` –

Call with custom function codes.

Tip: Response is not interpreted.

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

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse
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*) → ModbusResponse
Read Exception Status (code 0x07).

Parameters

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

Raises
`ModbusException` –

`diag_query_data`(*msg: bytearray*, *slave: int = 0*, ***kwargs: Any*) → ModbusResponse
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*) → ModbusResponse
Diagnose restart communication (code 0x08 sub 0x01).

Parameters

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

Raises
`ModbusException` –

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

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse
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*) → ModbusResponse
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*) → ModbusResponse
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*) → ModbusResponse
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*) → ModbusResponse
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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

Diagnose get event counter (code 0x0B).

Parameters

kwargs – (optional) Experimental parameters.

Raises

ModbusException –

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

Diagnose get event counter (code 0x0C).

Parameters

kwargs – (optional) Experimental parameters.

Raises

ModbusException –

write_coils(*address: int*, *values: Union[List[bool], bool]*, *slave: int = 0*, ***kwargs: Any*) → ModbusResponse

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: Union[List[int], int]*, *slave: int = 0*, ***kwargs: Any*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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*) → ModbusResponse

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: Union[List[int], int] = 0*, *slave: int = 0*, ***kwargs*) → ModbusResponse

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*) → ModbusResponse

Read FIFO queue (code 0x18).

Parameters

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

Raises
ModbusException –

`read_device_information`(*read_code: Optional[int] = None*, *object_id: int = 0*, ***kwargs: Any*) → ModbusResponse

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*)

Bases: `Enum`

Datatype enum for convert_* calls.

`classmethod convert_from_registers`(*registers: List[int]*, *data_type:* DATATYPE) → Union[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 "to_type"

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

`classmethod convert_to_registers`(*value: Union[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 convert to

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

CHAPTER

FOUR

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=<class 'pymodbus.framer.rtu_framer.ModbusRtuFramer'>*, *identity=None*, ***kwargs*)

Bases: `object`

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.

handler: ModbusSingleRequestHandler = None

on_connection_lost()

Call on lost connection.

async serve_forever()

Start endless loop.

async shutdown()

Terminate server.

async start()

Start connecting.

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: Optional[str] = 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: 8080)
- **http_port** – TCP port for HTTP (default: "localhost")
- **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 StartAsyncSimulatorServer

async def run():
    simulator = StartAsyncSimulatorServer(
        modbus_server="my server",
        modbus_device="my device",
        http_host="localhost",
        http_port=8080)
    await simulator.start()
    ...
    await simulator.close()
```

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*, *html*)
: 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`()
: 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=None, identity=None, address=None, handler=None, allow_reuse_address=False, defer_start=False, backlog=20, **kwargs)`

Bases: `object`

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.

`async serve_forever()`

Start endless loop.

`async server_close()`

Close server.

`async shutdown()`

Shutdown server.

`class pymodbus.server.ModbusTlsServer(context, framer=None, identity=None, address=None, sslctx=None, certfile=None, keyfile=None, password=None, reqclicert=False, handler=None, allow_reuse_address=False, defer_start=False, backlog=20, **kwargs)`

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=None, identity=None, address=None, handler=None, defer_start=False, backlog=20, **kwargs)`

Bases: `object`

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 serve_forever()`

Start endless loop.

`async server_close()`

Close server.

`async shutdown()`

Shutdown server.

`class pymodbus.server.`**`ModbusUnixServer`**(*context*, *path*, *framer=None*, *identity=None*, *handler=None*, ***kwargs*)

Bases: `object`

A modbus threaded Unix 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` **`serve_forever`**()

Start endless loop.

`async` **`server_close`**()

Close server.

`async` **`shutdown`**()

Shutdown server.

`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*, *reqclicert=False*, *allow_reuse_address=False*, *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
- `reqclicert` – Force the sever request client's certificate
- `allow_reuse_address` – Whether the server will allow the reuse of an address.
- `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` –

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

Start and run a tcp modbus server.

Parameters

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

`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.

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.

The simulator replaces *REPL server classes* but not *REPL client classes*

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,
             "allow_reuse_address": true,
             "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`*,
- "unix", to use *`pymodbus.server.ModbusUnixServer`*,
- "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,
            "allow_reuse_address": true,
            "ignore_missing_slaves": false,
            "framer": "socket",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/riptideio/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,
            "auto_reconnect": false,
            "reconnect_delay": 2,
            "framer": "rtu",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/riptideio/pymodbus/",
                "ProductName": "pymodbus Server",
                "ModelName": "pymodbus Server",
                "MajorMinorRevision": "3.1.0"
            }
```

(continues on next page)

(continued from previous page)

```
        },
        "server_try_tls": {
            "comm": "tls",
            "host": "0.0.0.0",
            "port": 5020,
            "certfile": "certificates/pymodbus.crt",
            "keyfile": "certificates/pymodbus.key",
            "allow_reuse_address": true,
            "backlog": 20,
            "ignore_missing_slaves": false,
            "framer": "tls",
            "identity": {
                "VendorName": "pymodbus",
                "ProductCode": "PM",
                "VendorUrl": "https://github.com/riptideio/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/riptideio/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!

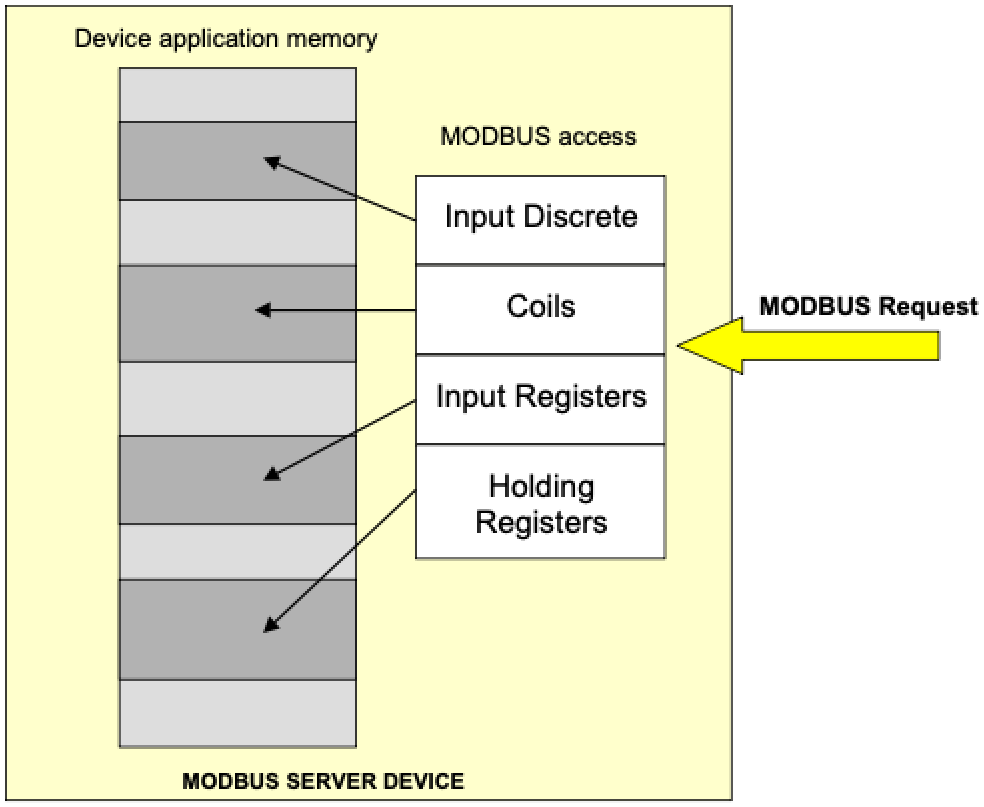
Communication with a modbus device is based on registers which each contain 16 bits (2 bytes). The requests are grouped in 4 groups

- 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.

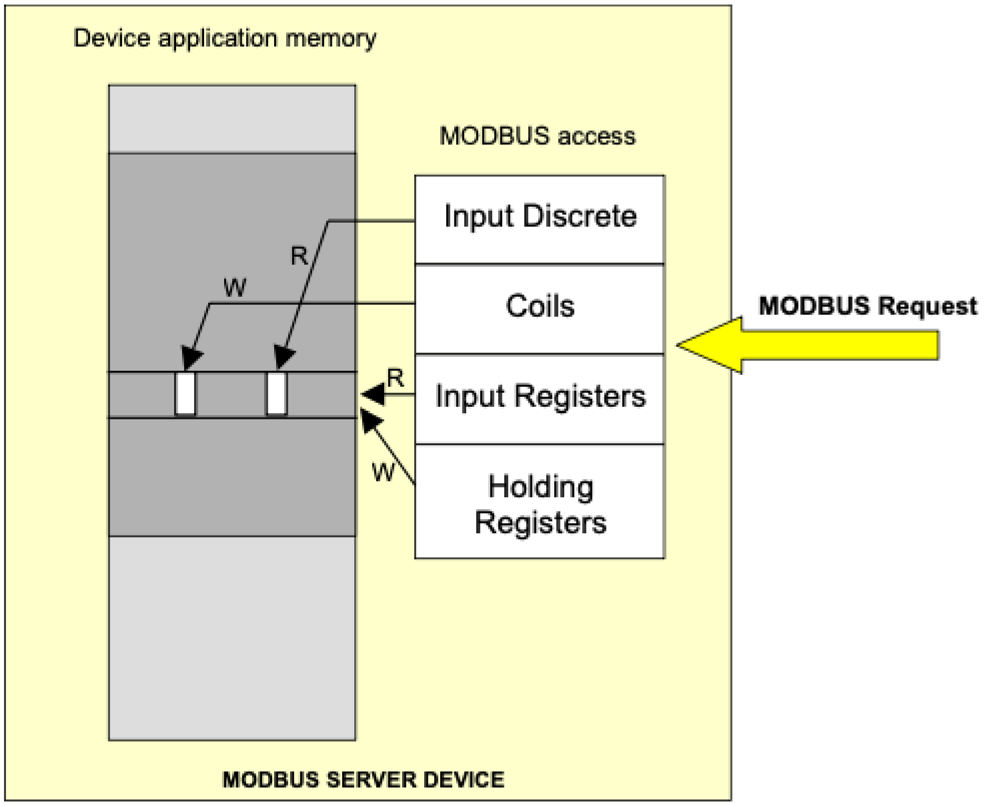
The first form is also the original form. It originates from a time where the devices did not contain memory, but the request was mapped directly to a physical sensor:



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,
            "uint32": 0,
            "float32": 0.0,
            "string": " "
        },
        "action": {
```

(continues on next page)

(continued from previous page)

```
            "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 reegisters) 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)

"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.

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 "float32" 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,
                        "uint32": 0,
                        "float32": 0.0,
                        "string": " "
                    },
                    "action": {
                        "bits": null,
                        "uint16": "register",
                        "uint32": "register",
                        "float32": "register",
                        "string": null
                    }
                }
            },
            "invalid": [
                1
            ],
            "write": [
                5
```

(continues on next page)

(continued from previous page)

```
            ],
            "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,
                        "uint16": 0,
                        "uint32": 0,
                        "float32": 0.0,
                        "string": " "
                    },
                    "action": {
                        "bits": null,
                        "uint16": "register",
                        "uint32": "register",
                        "float32": "register",
                        "string": null
                    }
                }
            },
            "invalid": [
                [0, 5],
                77
            ],
```

(continues on next page)

(continued from previous page)

```
            "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},
                {"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": {
```

(continues on next page)

(continued from previous page)

```
                "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": []
        }
    }
}
```

5.1.6 Configuration used for test

```
{
    "server_list": {
        "server": {
            "comm": "tcp",
            "host": "0.0.0.0",
            "port": 5020,
            "allow_reuse_address": true,
            "ignore_missing_slaves": false,
            "framer": "socket",
            "identity": {
                "VendorName": "pymodbus",
```

(continues on next page)

(continued from previous page)

```
                "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,
            "auto_reconnect": false,
            "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",
            "allow_reuse_address": true,
            "backlog": 20,
            "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",
```

(continues on next page)

(continued from previous page)

```
            "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
            ],
            "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]
```

(continues on next page)

(continued from previous page)

```
            ],
            "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": "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"}
```

(continues on next page)

(continued from previous page)

```
            ],
            "uint16": [
                11,
                2027,
                [2126, 2129],
                {"addr": 2164, "value": 1},
                {"addr": [2168,2169], "value": 32117},
                {"addr": [2208, 2306], "action": null}
            ],
            "uint32": [
                [12, 13],
                [3037, 3038],
                [3136, 3139],
                {"addr": [3174, 3175], "value": 1},
                {"addr": [3188,3189], "value": 32514},
                {"addr": [3308, 3407], "action": null},
                {"addr": [3688, 3879], "value": 115, "action": "increment"}
            ],
            "float32": [
                [14, 15],
                [4047, 4048],
                [4146, 4149],
                {"addr": [4184, 4185], "value": 1},
                {"addr": [4188, 4191], "value": 32514.1},
                {"addr": [4308, 4407], "action": null},
                {"addr": [4688, 4879], "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]}
            ]
        }
    }
}
```

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

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: Optional[str] = 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: 8080)
- **`http_port`** – TCP port for HTTP (default: "localhost")
- **`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 StartAsyncSimulatorServer

async def run():
    simulator = StartAsyncSimulatorServer(
        modbus_server="my server",
        modbus_device="my device",
        http_host="localhost",
        http_port=8080)
    await simulator.start()
    ...
    await simulator.close()
```

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

Start Modbus server as asyncio task.

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

Stop modbus server.

`async` **`run_forever`**()

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*, *html*)

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

REPL

6.1 Dependencies

Depends on prompt_toolkit and click

Install dependencies

```
$ pip install click prompt_toolkit --upgrade
```

Or Install pymodbus with repl support

```
$ pip install pymodbus[repl] --upgrade
```

6.2 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:
  --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. Defaults to 9600
  --bytesize [5|6|7|8]   Modbus RTU serial Number of data bits. Possible
                         values: FIVEBITS, SIXBITS, SEVENBITS, EIGHTBITS.
                         Defaults to 8
  --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.Defaults to 0
  --rtscts INTEGER       Modbus RTU serial rtscts. Enable hardware (RTS/CTS)
                         flow control. Defaults to 0
  --dsrdtr INTEGER       Modbus RTU serial dsrdtr. Enable hardware (DSR/DTR)
                         flow control. Defaults to 0
  --timeout FLOAT        Modbus RTU serial read timeout. Defaults to 0.025 sec
  --write-timeout FLOAT  Modbus RTU serial write timeout. Defaults to 2 sec
  --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.
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.
```

(continues on next page)

(continued from previous page)

```
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␣
↪responses  by remote slave.
client.silent_interval                      Read Only!
client.state                                Read Only!
client.timeout                              Read Only!
```

(continues on next page)

(continued from previous page)

```
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_char_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`.
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`.
```

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

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

(continues on next page)

(continued from previous page)

```
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
}
```

6.3 DEMO

6.4 REPL client classes

Command Completion for pymodbus REPL.

class `pymodbus.repl.client.completer.`**`CmdCompleter`**(*client=None*, *commands=None*, *ignore_case=True*)

Bases: `Completer`

Completer for Pymodbus REPL.

`arg_completions`(*words*, *_word_before_cursor*)

Generate arguments completions based on the input.

property `command_names`

Return command names.

property `commands`

Return commands.

`completing_arg`(*words*, *word_before_cursor*)

Determine if we are currently completing an argument.

Parameters

- **`words`** – The input text broken into word tokens.
- **`word_before_cursor`** – The current word before the cursor, which might be one or more blank spaces.

Returns

Specifies whether we are currently completing an arg.

`completing_command`(*words*, *word_before_cursor*)

Determine if we are dealing with supported command.

Parameters

- `words` – Input text broken in to word tokens.
- `word_before_cursor` – The current word before the cursor, which might be one or more blank spaces.

Returns

`get_completions`(*document*, *complete_event*)

Get completions for the current scope.

Parameters

- `document` – An instance of *prompt_toolkit.Document*.
- `complete_event` – (Unused).

Returns

Yields an instance of *prompt_toolkit.completion.Completion*.

`word_matches`(*word*, *word_before_cursor*)

Match the word and word before cursor.

Parameters

- `word` – The input text broken into word tokens.
- `word_before_cursor` – The current word before the cursor, which might be one or more blank spaces.

Returns

True if matched.

Helper Module for REPL actions.

`class pymodbus.repl.client.helper.Command`(*name*, *signature*, *doc*, *unit=False*)

Bases: `object`

Class representing Commands to be consumed by Completer.

`create_completion()`

Create command completion meta data.

Returns

`get_completion()`

Get a list of completions.

Returns

`get_meta`(*cmd*)

Get Meta info of a given command.

Parameters

`cmd` – Name of command.

Returns

Dict containing meta info.

`class pymodbus.repl.client.helper.Result`(*result*)

Bases: `object`

Represent result command.

`data: Union[Dict[int, Any], Any] = None`

`decode`(*formatters*, *byte_order='big'*, *word_order='big'*)

Decode the register response to known formatters.

Parameters

- `formatters` – int8/16/32/64, uint8/16/32/64, float32/64
- `byte_order` – little/big
- `word_order` – little/big

`function_code: int = None`

`print_result`(*data=None*)

Print result object pretty.

Parameters

`data` – Data to be printed.

`raw`()

Return raw result dict.

`pymodbus.repl.client.helper.get_commands`(*client*)

Retrieve all required methods and attributes.

Of a client object and convert it to commands.

Parameters

`client` – Modbus Client object.

Returns

Pymodbus REPL Entry point.

`class pymodbus.repl.client.main.CaseInsenstiveChoice`(*choices: Sequence[str]*, *case_sensitive: bool = True*)

Bases: `Choice`

Do case Insensitive choice for click commands and options.

`convert`(*value*, *param*, *ctx*)

Convert args to uppercase for evaluation.

`class pymodbus.repl.client.main.NumericChoice`(*choices*, *typ*)

Bases: `Choice`

Do numeric choice for click arguments and options.

`convert`(*value*, *param*, *ctx*)

Convert.

`pymodbus.repl.client.main.bottom_toolbar`()

Do console toolbar.

Returns

`pymodbus.repl.client.main.cli`(*client*)

Run client definition.

Modbus Clients to be used with REPL.

class pymodbus.repl.client.mclient.**ExtendedRequestSupport**

Bases: `object`

Extended request support.

change_ascii_input_delimiter(*data=0*, ***kwargs*)

Change message delimiter for future requests.

Parameters

- **data** – New delimiter character
- **kwargs** –

Returns

clear_counters(*data=0*, ***kwargs*)

Clear all counters and diag registers.

Parameters

- **data** – Data field (0x0000)
- **kwargs** –

Returns

clear_overrun_count(*data=0*, ***kwargs*)

Clear over run counter.

Parameters

- **data** – Data field (0x0000)
- **kwargs** –

Returns

force_listen_only_mode(*data=0*, ***kwargs*)

Force addressed remote device to its Listen Only Mode.

Parameters

- **data** – Data field (0x0000)
- **kwargs** –

Returns

get_clear_modbus_plus(*data=0*, ***kwargs*)

Get/clear stats of remote modbus plus device.

Parameters

- **data** – Data field (0x0000)
- **kwargs** –

Returns

get_com_event_counter(***kwargs*)

Read status word and an event count.

From the remote device's communication event counter.

Parameters

kwargs –

Returns

`get_com_event_log`(***kwargs*)

Read status word.

Event count, message count, and a field of event bytes from the remote device.

Parameters
`kwargs` –

Returns

`mask_write_register`(*address=0*, *and_mask=65535*, *or_mask=0*, *slave=0*, ***kwargs*)

Mask content of holding register at *address* with *and_mask* and *or_mask*.

Parameters

- `address` – Reference address of register
- `and_mask` – And Mask
- `or_mask` – OR Mask
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`read_coils`(*address*, *count=1*, *slave=0*, ***kwargs*)

Read *count* coils from a given slave starting at *address*.

Parameters

- `address` – The starting address to read from
- `count` – The number of coils to read
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns
List of register values

`read_device_information`(*read_code=None*, *object_id=0*, ***kwargs*)

Read the identification and additional information of remote slave.

Parameters

- `read_code` – Read Device ID code (0x01/0x02/0x03/0x04)
- `object_id` – Identification of the first object to obtain.
- `kwargs` –

Returns

`read_discrete_inputs`(*address*, *count=1*, *slave=0*, ***kwargs*)

Read *count* number of discrete inputs starting at offset *address*.

Parameters

- `address` – The starting address to read from
- `count` – The number of coils to read
- `slave` – Modbus slave unit ID

- `kwargs` –

Returns
List of bits

`read_exception_status`(*slave=0*, ***kwargs*)

Read contents of eight Exception Status output in a remote device.

Parameters

- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`read_holding_registers`(*address*, *count=1*, *slave=0*, ***kwargs*)

Read *count* number of holding registers starting at *address*.

Parameters

- `address` – starting register offset to read from
- `count` – Number of registers to read
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`read_input_registers`(*address*, *count=1*, *slave=0*, ***kwargs*)

Read *count* number of input registers starting at *address*.

Parameters

- `address` – starting register offset to read from to
- `count` – Number of registers to read
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`readwrite_registers`(*read_address=0*, *read_count=0*, *write_address=0*, *values=0*, *slave=0*, ***kwargs*)

Read *read_count* number of holding registers.

Starting at *read_address* and write *write_registers* starting at *write_address*.

Parameters

- `read_address` – register offset to read from
- `read_count` – Number of registers to read
- `write_address` – register offset to write to
- `values` – List of register values to write (comma separated)
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`report_slave_id`(*slave=0*, ***kwargs*)

Report information about remote slave ID.

Parameters

- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`restart_comm_option`(*toggle=False*, ***kwargs*)

Initialize and restart remote devices.

Serial interface and clear all of its communications event counters.

Parameters

- `toggle` – Toggle Status [ON(0xff00)/OFF(0x0000]
- `kwargs` –

Returns

`return_bus_com_error_count`(*data=0*, ***kwargs*)

Return count of CRC errors received by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_bus_exception_error_count`(*data=0*, ***kwargs*)

Return count of Modbus exceptions returned by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_bus_message_count`(*data=0*, ***kwargs*)

Return count of message detected on bus by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_diagnostic_register`(*data=0*, ***kwargs*)

Read 16-bit diagnostic register.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_iop_overrun_count`(*data=0*, ***kwargs*)

Return count of iop overrun errors by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_query_data`(*message=0*, ***kwargs*)

Loop back data sent in response.

Parameters

- `message` – Message to be looped back
- `kwargs` –

Returns

`return_slave_bus_char_overrun_count`(*data=0*, ***kwargs*)

Return count of messages not handled.

By remote slave due to character overrun condition.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_slave_busy_count`(*data=0*, ***kwargs*)

Return count of server busy exceptions sent by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_slave_message_count`(*data=0*, ***kwargs*)

Return count of messages addressed to remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_slave_no_ack_count`(*data=0*, ***kwargs*)

Return count of NO ACK exceptions sent by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`return_slave_no_response_count`(*data=0*, ***kwargs*)

Return count of No responses by remote slave.

Parameters

- `data` – Data field (0x0000)
- `kwargs` –

Returns

`write_coil`(*address*, *value*, *slave=0*, ***kwargs*)

Write *value* to coil at *address*.

Parameters

- `address` – coil offset to write to
- `value` – bit value to write
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`write_coils`(*address*, *values*, *slave=0*, ***kwargs*)

Write *value* to coil at *address*.

Parameters

- `address` – coil offset to write to
- `values` – list of bit values to write (comma separated)
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`write_register`(*address*, *value*, *slave=0*, ***kwargs*)

Write *value* to register at *address*.

Parameters

- `address` – register offset to write to
- `value` – register value to write
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

`write_registers`(*address*, *values*, *slave=0*, ***kwargs*)

Write list of *values* to registers starting at *address*.

Parameters

- `address` – register offset to write to
- `values` – list of register value to write (comma separated)
- `slave` – Modbus slave unit ID
- `kwargs` –

Returns

class `pymodbus.repl.client.mclient.`**`ModbusSerialClient`**(*framer*, ***kwargs*)

Bases: `ExtendedRequestSupport`, `ModbusSerialClient`

Modbus serial client.

`get_baudrate`()

Get serial Port baudrate.

Returns
Current baudrate

`get_bytesize`()

Get number of data bits.

Returns
Current bytesize

`get_parity`()

Enable Parity Checking.

Returns
Current parity setting

`get_port`()

Get serial Port.

Returns
Current Serial port

`get_serial_settings`()

Get Current Serial port settings.

Returns
Current Serial settings as dict.

`get_stopbits`()

Get number of stop bits.

Returns
Current Stop bits

`get_timeout`()

Get serial Port Read timeout.

Returns
Current read imeout.

`set_baudrate`(*value*)

Set baudrate setter.

Parameters
`value` – <supported baudrate>

`set_bytesize`(*value*)

Set Byte size.

Parameters
`value` – Possible values (5, 6, 7, 8)

set_parity(*value*)

Set parity Setter.

Parameters
value – Possible values (“N”, “E”, “O”, “M”, “S”)

set_port(*value*)

Set serial Port setter.

Parameters
value – New port

set_stopbits(*value*)

Set stop bit.

Parameters
value – Possible values (1, 1.5, 2)

set_timeout(*value*)

Read timeout setter.

Parameters
value – Read Timeout in seconds

class pymodbus.repl.client.mclient.**ModbusTcpClient**(***kwargs*)

Bases: *ExtendedRequestSupport*, *ModbusTcpClient*

TCP client.

pymodbus.repl.client.mclient.**handle_brodcast**(*func*)

Handle broadcast.

pymodbus.repl.client.mclient.**make_response_dict**(*resp*)

Make response dict.

6.5 REPL server classes

Repl server cli.

pymodbus.repl.server.cli.**error**(*message*)

Show error.

pymodbus.repl.server.cli.**get_terminal_width**()

Get terminal width.

pymodbus.repl.server.cli.**info**(*message*)

Show info.

async pymodbus.repl.server.cli.**interactive_shell**(*server*)

Run CLI interactive shell.

async pymodbus.repl.server.cli.**main**(*server*)

Run main.

pymodbus.repl.server.cli.**print_help**()

Print help.

`async pymodbus.repl.server.cli.`**`run_repl`**`(`*server*`)`

Run repl server.

`pymodbus.repl.server.cli.`**`warning`**`(`*message*`)`

Show warning.

Repl server main.

`class pymodbus.repl.server.main.`**`ModbusFramerTypes`**`(`*value*`)`

Bases: `str`, `Enum`

Framer types.

`ascii = 'ascii'`

`binary = 'binary'`

`rtu = 'rtu'`

`socket = 'socket'`

`tls = 'tls'`

`class pymodbus.repl.server.main.`**`ModbusServerTypes`**`(`*value*`)`

Bases: `str`, `Enum`

Server types.

`serial = 'serial'`

`tcp = 'tcp'`

`tls = 'tls'`

`udp = 'udp'`

`pymodbus.repl.server.main.`**`framers`**`(`*incomplete: str*`)` → List[str]

Return an autocompleted list of supported clouds.

`pymodbus.repl.server.main.`**`process_extra_args`**`(`*extra_args: List[str]*, *modbus_config: dict*`)` → dict

Process extra args passed to server.

`pymodbus.repl.server.main.`**`run`**`(`*ctx: ~typer.models.Context*, *modbus_server: str = <typer.models.OptionInfo object>*, *modbus_framer: str = <typer.models.OptionInfo object>*, *modbus_port: int = <typer.models.OptionInfo object>*, *modbus_slave_id: ~typing.List[int] = <typer.models.OptionInfo object>*, *modbus_config_path: ~pathlib.Path = <typer.models.OptionInfo object>*, *randomize: int = <typer.models.OptionInfo object>*, *change_rate: int = <typer.models.OptionInfo object>*`)`

Run Reactive Modbus server.

Exposing REST endpoint for response manipulation.

`pymodbus.repl.server.main.`**`server`**`(`*ctx: ~typer.models.Context*, *host: str = <typer.models.OptionInfo object>*, *web_port: int = <typer.models.OptionInfo object>*, *broadcast_support: bool = <typer.models.OptionInfo object>*, *repl: bool = <typer.models.OptionInfo object>*, *verbose: bool = <typer.models.OptionInfo object>*`)`

Run server code.

`pymodbus.repl.server.main.`**`servers`**(*incomplete: str*) → List[str]

Return an autocompleted list of supported clouds.

CHAPTER

SEVEN

DATASTORE

Datastore is responsible for managing registers for a server.

7.1 Datastore classes

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

Create 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 starting at offset 10 with length 4 , 30 with length 1 and 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)

if mutable is set to True during initialization, the datablock can not be altered with setValues (new datablocks can not 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 unit-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]*)

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
        {"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
    ],
```

(continues on next page)

(continued from previous page)

```
    "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 registers from int32 or float32

class pymodbus.datastore.**RedisSlaveContext**(***kwargs*)

Bases: `ModbusBaseSlaveContext`

This is a modbus slave context using redis as a backing store.

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

Get *count* values from datastore.

Parameters

- **fc** – 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

reset()

Reset all the datastores to their default values.

setValues(*fc*, *address*, *values*)

Set the datastore with the supplied values.

Parameters

- **fc** – The function we are working with
- **address** – The starting address
- **values** – The new values to be set

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

Validate the request to make sure it is in range.

Parameters

- **fc** – 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

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

Bases: `ModbusBaseSlaveContext`

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

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

Get *count* values from datastore.

Parameters

- **fc** – 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

`reset()`

Reset all the datastores to their default values.

`setValues(fc, address, values, update=True)`

Set the datastore with the supplied values.

Parameters

- **fc** – The function we are working with
- **address** – The starting address
- **values** – The new values to be set
- **update** – Update existing register in the db

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

Validate the request to make sure it is in range.

Parameters

- **fc** – 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

CHAPTER

EIGHT

FRAMER

8.1 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.

addToFrame(*message*)

Add the next message to the frame buffer.

This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters
message – The most recent packet

advanceFrame()

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)

Create a ready to send modbus packet.

Built off of a modbus request/response

Parameters
message – The request/response to send

Returns
The encoded packet

`checkFrame()`

Check and decode the next frame.

Returns
True if we successful, False otherwise

`decode_data(`*data*`)`

Decode data.

`getFrame()`

Get the next frame from the buffer.

Returns
The frame data or ""

`isFrameReady()`

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'ascii'`

`populateResult(`*result*`)`

Populate the modbus result header.

The serial packets do not have any header information that is copied.

Parameters
`result` – The response packet

`processIncomingPacket(`*data*, *callback*, *slave*, ***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.

Parameters

- `data` – The new packet data
- `callback` – The function to send results to
- `slave` – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server))
- `kwargs` –

Raises
`ModbusIOException` –

`resetFrame()`

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

8.2 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.

addToFrame(*message*)

Add the next message to the frame buffer.

This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters
`message` – The most recent packet

advanceFrame()

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The request/response to send

Returns
The encoded packet

checkFrame()

Check and decode the next frame.

Returns
True if we are successful, False otherwise

decode_data(*data*)

Decode data.

`getFrame()`

Get the next frame from the buffer.

Returns
The frame data or “”

`isFrameReady()`

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'binary'`

`populateResult(`*result*`)`

Populate the modbus result header.

The serial packets do not have any header information that is copied.

Parameters
`result` – The response packet

`processIncomingPacket(`*data*, *callback*, *slave*, ***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.

Parameters

- `data` – The new packet data
- `callback` – The function to send results to
- `slave` – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- `kwargs` –

Raises
`ModbusIOException` –

`resetFrame()`

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn’t have the resolution to check for millisecond delays).

8.3 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
```

`addToFrame`(*message*)

Add the received data to the buffer handle.

Parameters
`message` – The most recent packet

`advanceFrame`()

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

`buildPacket`(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The populated request/response to send

`checkFrame`()

Check if the next frame is available.

Return True if we were successful.

1. Populate header

2. Discard frame if UID does not match

`decode_data`(*data*)

Decode data.

`getFrame`()

Get the next frame from the buffer.

Returns
The frame data or “”

`getRawFrame`()

Return the complete buffer.

`get_expected_response_length`(*data*)

Get the expected response length.

Parameters
`data` – Message data read so far

Raises
`IndexError` – If not enough data to read byte count

Returns
Total frame size

`isFrameReady`()

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'rtu'`

`populateHeader`(*data=None*)

Try to set the headers *uid*, *len* and *crc*.

This method examines *self._buffer* and writes meta information into *self._header*.

Beware that this method will raise an IndexError if *self._buffer* is not yet long enough.

`populateResult`(*result*)

Populate the modbus result header.

The serial packets do not have any header information that is copied.

Parameters
`result` – The response packet

`processIncomingPacket`(*data*, *callback*, *slave*, ***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.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to
- **slave** – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- **kwargs** –

recvPacket(*size*)

Receive packet from the bus with specified len.

Parameters
size – Number of bytes to read

Returns

resetFrame()

Reset the entire message frame.

This allows us to skip over errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

sendPacket(*message*)

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

Parameters
message – Message to be sent over the bus

Returns

8.4 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
```

`addToFrame`(*message*)

Add new packet data to the current frame buffer.

Parameters
`message` – The most recent packet

`advanceFrame`()

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

`buildPacket`(*message*)

Create a ready to send modbus packet.

Parameters
`message` – The populated request/response to send

`checkFrame`()

Check and decode the next frame.

Return true if we were successful.

`decode_data`(*data*)

Decode data.

`getFrame`()

Return the next frame from the buffered data.

Returns
The next full frame buffer

`getRawFrame`()

Return the complete buffer.

`isFrameReady`()

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder factory know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'socket'`

`populateResult`(*result*)

Populate the modbus result.

With the transport specific header information (pid, tid, uid, checksum, etc)

Parameters
`result` – The response packet

`processIncomingPacket`(*data*, *callback*, *slave*, ***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.

Parameters

- `data` – The new packet data
- `callback` – The function to send results to
- `slave` – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- `kwargs` –

`resetFrame()`

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

8.5 Module contents

Framer start.

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

Bases: `object`

Base Framer class.

`name = ''`

`recvPacket(`*size*`)`

Receive packet from the bus.

With specified len :param size: Number of bytes to read :return:

`sendPacket(`*message*`)`

Send packets on the bus.

With 3.5char delay between frames :param message: Message to be sent over the bus :return:

CHAPTER NINE

CONSTANTS

Constants For Modbus Server/Client.

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

`class pymodbus.constants.Defaults`

Bases: `object`

A collection of modbus default values.

`Port`
The default modbus tcp server port (502)

`TLSPort`
The default modbus tcp over tls server port (802)

`Backoff`
The default exponential backoff delay (0.3 seconds) for a request

`Retries`
The default number of times a client should retry the given request before failing (3)

`RetryOnEmpty`
A flag indicating if a transaction should be retried in the case that an empty response is received. This is useful for slow clients that may need more time to process a request.

`RetryOnInvalid`
A flag indicating if a transaction should be retried in the case that an invalid response is received.

`Timeout`
The default amount of time a client should wait for a request to be processed (3 seconds)

`Reconnects`
The default number of times a client should attempt to reconnect before deciding the server is down (0)

`TransactionId`
The starting transaction identifier number (0)

`ProtocolId`
The modbus protocol id. Currently, this is set to 0 in all but proprietary implementations.

`Slave`
The modbus slave address. Currently, this is set to 0x00 which means this request should be broadcast to all the slave devices (really means that all the devices should respond).

`Baudrate`
The speed at which the data is transmitted over the serial line. This defaults to 19200.

`Parity`
The type of checksum to use to verify data integrity. This can be on of the following:

```
- (E)ven - 1 0 1 0 | P(0)
- (O)dd  - 1 0 1 0 | P(1)
- (N)one - 1 0 1 0 | no parity
```

This defaults to (N)one.

`Bytesize`
The number of bits in a byte of serial data. This can be one of 5, 6, 7, or 8. This defaults to 8.

`Stopbits`
The number of bits sent after each character in a message to indicate the end of the byte. This defaults to 1.

`ZeroMode`
Indicates if the slave datastore should use indexing at 0 or 1. More about this can be read in section 4.4 of the modbus specification.

`IgnoreMissingSlaves`
In case a request is made to a missing slave, this defines if an error should be returned or simply ignored. This is useful for the case of a serial server emulater where a request to a non-existent slave on a bus will never respond. The client in this case will simply timeout.

`broadcastEnable`
When False slave_id 0 will be treated as any other slave_id. When True and the slave_id is 0 the server will execute all requests on all server contexts and not respond and the client will skip trying to receive a response. Default value False does not conform to Modbus spec but maintains legacy behavior for existing pymodbus users.

`Backoff = 0.3`

`Baudrate = 19200`

`BroadcastEnable = False`

`Bytesize = 8`

`CloseCommOnError = False`

`Count = 1`

`HandleLocalEcho = False`

`IgnoreMissingSlaves = False`

`Parity = 'N'`

`ProtocolId = 0`

`ReadSize = 1024`

`ReconnectDelay = 100`

`ReconnectDelayMax = 300000`

`Reconnects = 0`

`Retries = 3`

`RetryOnEmpty = False`

`RetryOnInvalid = False`

`Slave:  int = 0`

`Stopbits = 1`

`Strict = True`

`TcpPort = 502`

`Timeout = 3`

`TlsPort = 802`

`TransactionId = 0`

`UdpPort = 502`

`ZeroMode = False`

`class pymodbus.constants.DeviceInformation`

Bases: `object`

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`

Bases: `object`

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`

Bases: `object`

Represents the type of modbus plus request.

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

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

`ClearStatistics = 4`

`GetStatistics = 3`

`class pymodbus.constants.ModbusStatus`

Bases: `object`

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

`SlaveOn`
: This indicates that the given modbus slave is running

`SlaveOff`
: This indicates that the given modbus slave is not running

`Off = 0`

`On = 65280`

`Ready = 0`

`SlaveOff = 0`

`SlaveOn = 255`

`Waiting = 65535`

`class pymodbus.constants.MoreData`

Bases: `object`

Represents the more follows condition.

`Nothing`

This indicates that no more objects are going to be returned.

`KeepReading`

This indicates that there are more objects to be returned.

`KeepReading = 255`

`Nothing = 0`

CHAPTER

TEN

EXTRA FUNCTIONS

Pymodbus: Modbus Protocol Implementation.

Released under the the BSD license

`pymodbus.`**`pymodbus_apply_logging_config`**(*level: Union[str, int] = 10*, *log_file_name: Optional[str] = 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*, *unit=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*, *unit=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*, *unit=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*, *unit=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*, *unit=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*, *unit=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*, *unit=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=1*, *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.ModbusControlBlock(`**_args*, ***_kwargs*`)`

Bases: `object`

This is a global singleton that controls all system information.

All activity should be logged here and all diagnostic requests should come from here.

`property Counter`

`property Delimiter`

`property Events`

`property Identity`

`property ListenOnly`

`property Mode`

`property Plus`

`addEvent(`*event:* ModbusEvent`)`

Add a new event to the event log.

Parameters

`event` – A new event to add to the log

`clearEvents()`

Clear the current list of events.

`getDiagnostic(`*bit*`)`

Get the value in the diagnostic register.

Parameters

`bit` – The bit to get

Returns

The current value of the requested bit

`getDiagnosticRegister()`

Get the entire diagnostic register.

Returns

The diagnostic register collection

`getEvents()`

Return an encoded collection of the event log.

Returns
The encoded events packet

`reset()`

Clear all of the system counters and the diagnostic register.

`setDiagnostic(`*mapping*`)`

Set the value in the diagnostic register.

Parameters
`mapping` – Dictionary of key:value pairs to set

`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**(*unit=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*, *unit=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=0*, *unit=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=0*, ***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*)

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

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.`**`ModbusException`**(*string*)

Bases: `Exception`

Base modbus exception.

`isError`()

Error

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()`

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=None*`)`

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`(*unit=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`(*unit=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='<'*, *wordorder='>'*, *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*)

Add a 16 bit float to the buffer.

Parameters
value – The value to add to the buffer

add_16bit_int(*value*)

Add a 16 bit signed int to the buffer.

Parameters
value – The value to add to the buffer

add_16bit_uint(*value*)

Add a 16 bit unsigned int to the buffer.

Parameters
value – The value to add to the buffer

`add_32bit_float`(*value*)

Add a 32 bit float to the buffer.

Parameters
`value` – The value to add to the buffer

`add_32bit_int`(*value*)

Add a 32 bit signed int to the buffer.

Parameters
`value` – The value to add to the buffer

`add_32bit_uint`(*value*)

Add a 32 bit unsigned int to the buffer.

Parameters
`value` – The value to add to the buffer

`add_64bit_float`(*value*)

Add a 64 bit float(double) to the buffer.

Parameters
`value` – The value to add to the buffer

`add_64bit_int`(*value*)

Add a 64 bit signed int to the buffer.

Parameters
`value` – The value to add to the buffer

`add_64bit_uint`(*value*)

Add a 64 bit unsigned int to the buffer.

Parameters
`value` – The value to add to the buffer

`add_8bit_int`(*value*)

Add a 8 bit signed int to the buffer.

Parameters
`value` – The value to add to the buffer

`add_8bit_uint`(*value*)

Add a 8 bit unsigned int to the buffer.

Parameters
`value` – The value to add to the buffer

`add_bits`(*values*)

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*)

Add a string to the buffer.

Parameters
`value` – The value to add to the buffer

`build()`

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

`reset()`

Reset the payload buffer.

`to_coils()`

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

`to_string()`

Return the payload buffer as a string.

Returns
The payload buffer as a string

class pymodbus.payload.**BinaryPayloadDecoder**(*payload*, *byteorder='<'*, *wordorder='>'*)

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='<'*, *_wordorder='>'*`)`
: Initialize a payload decoder with the result of reading of coils.

`classmethod fromRegisters(`*registers*, *byteorder='<'*, *wordorder='>'*`)`
: 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

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(unit=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(unit=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.

`isError()`

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, unit=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*, *unit=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*, *unit=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*, *unit=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*, *unit=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`

Implements a transaction for a manager.

Where the results are keyed based on the supplied transaction id.

`addTransaction`(*request*, *tid=None*)

Add a transaction to the handler.

This holds the requests 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

`getTransaction`(*tid*)

Return a transaction matching the referenced tid.

If the transaction does not exist, None is returned

Parameters
`tid` – The transaction to retrieve

`class pymodbus.transaction.`**`FifoTransactionManager`**(*client*, ***kwargs*)

Bases: `ModbusTransactionManager`

Implements a transaction.

For a manager where the results are returned in a FIFO manner.

`addTransaction`(*request*, *tid=None*)

Add a transaction to the handler.

This holds the requests 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

`getTransaction`(*tid*)

Return a transaction matching the referenced tid.

If the transaction does not exist, None is returned

Parameters
`tid` – The transaction to retrieve

`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.

`addToFrame`(*message*)

Add the next message to the frame buffer.

This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters
`message` – The most recent packet

`advanceFrame()`

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

`buildPacket(`*message*`)`

Create a ready to send modbus packet.

Built off of a modbus request/response

Parameters
`message` – The request/response to send

Returns
The encoded packet

`checkFrame()`

Check and decode the next frame.

Returns
True if we successful, False otherwise

`decode_data(`*data*`)`

Decode data.

`getFrame()`

Get the next frame from the buffer.

Returns
The frame data or “”

`isFrameReady()`

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'ascii'`

`populateResult(`*result*`)`

Populate the modbus result header.

The serial packets do not have any header information that is copied.

Parameters
`result` – The response packet

`processIncomingPacket(`*data*, *callback*, *slave*, ***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.

Parameters

- **`data`** – The new packet data
- **`callback`** – The function to send results to
- **`slave`** – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server))
- **`kwargs`** –

Raises

`ModbusIOException` –

`resetFrame``()`

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

`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.

`addToFrame``(`*message*`)`

Add the next message to the frame buffer.

This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters

`message` – The most recent packet

`advanceFrame``()`

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

`buildPacket``(`*message*`)`

Create a ready to send modbus packet.

Parameters

`message` – The request/response to send

Returns
The encoded packet

`checkFrame()`

Check and decode the next frame.

Returns
True if we are successful, False otherwise

`decode_data(`*data*`)`

Decode data.

`getFrame()`

Get the next frame from the buffer.

Returns
The frame data or ""

`isFrameReady()`

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'binary'`

`populateResult(`*result*`)`

Populate the modbus result header.

The serial packets do not have any header information that is copied.

Parameters
`result` – The response packet

`processIncomingPacket(`*data*, *callback*, *slave*, ***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.

Parameters

- `data` – The new packet data
- `callback` – The function to send results to
- `slave` – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- `kwargs` –

Raises
`ModbusIOException` –

resetFrame()

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

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
```

addToFrame(*message*)

Add the received data to the buffer handle.

Parameters
message – The most recent packet

advanceFrame()

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The populated request/response to send

checkFrame()

Check if the next frame is available.

Return True if we were successful.

1. Populate header
2. Discard frame if UID does not match

`decode_data`(*data*)

Decode data.

`getFrame`()

Get the next frame from the buffer.

Returns
The frame data or “”

`getRawFrame`()

Return the complete buffer.

`get_expected_response_length`(*data*)

Get the expected response length.

Parameters
`data` – Message data read so far

Raises
`IndexError` – If not enough data to read byte count

Returns
Total frame size

`isFrameReady`()

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'rtu'`

`populateHeader`(*data=None*)

Try to set the headers *uid*, *len* and *crc*.

This method examines *self._buffer* and writes meta information into *self._header*.

Beware that this method will raise an IndexError if *self._buffer* is not yet long enough.

`populateResult`(*result*)

Populate the modbus result header.

The serial packets do not have any header information that is copied.

Parameters
`result` – The response packet

`processIncomingPacket`(*data*, *callback*, *slave*, ***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.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to
- **slave** – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- **kwargs** –

recvPacket(*size*)

Receive packet from the bus with specified len.

Parameters
size – Number of bytes to read

Returns

resetFrame()

Reset the entire message frame.

This allows us to skip over errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

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
```

addToFrame(*message*)

Add new packet data to the current frame buffer.

Parameters
message – The most recent packet

`advanceFrame()`

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

`buildPacket(`*message*`)`

Create a ready to send modbus packet.

Parameters
`message` – The populated request/response to send

`checkFrame()`

Check and decode the next frame.

Return true if we were successful.

`decode_data(`*data*`)`

Decode data.

`getFrame()`

Return the next frame from the buffered data.

Returns
The next full frame buffer

`getRawFrame()`

Return the complete buffer.

`isFrameReady()`

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder factory know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'socket'`

`populateResult(`*result*`)`

Populate the modbus result.

With the transport specific header information (pid, tid, uid, checksum, etc)

Parameters
`result` – The response packet

`processIncomingPacket(`*data*, *callback*, *slave*, ***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.

Parameters

- `data` – The new packet data
- `callback` – The function to send results to

- **slave** – Process if slave id matches, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- **kwargs** –

resetFrame()

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

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

addToFrame(*message*)

Add new packet data to the current frame buffer.

Parameters
message – The most recent packet

advanceFrame()

Skip over the current framed message.

This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)

Create a ready to send modbus packet.

Parameters
message – The populated request/response to send

checkFrame()

Check and decode the next frame.

Return true if we were successful.

decode_data(*data*)

Decode data.

getFrame()

Return the next frame from the buffered data.

Returns
The next full frame buffer

getRawFrame()

Return the complete buffer.

`isFrameReady()`

Check if we should continue decode logic.

This is meant to be used in a while loop in the decoding phase to let the decoder factory know that there is still data in the buffer.

Returns
True if ready, False otherwise

`method = 'tls'`

`populateResult(`*_result*`)`

Populate the modbus result.

`processIncomingPacket(`*data*, *callback*, *slave*, ***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.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to
- **slave** – Process if slave id matcheks, ignore otherwise (could be a list of slave ids (server) or single slave id(client/server)
- **kwargs** –

`resetFrame()`

Reset the entire message frame.

This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

Modbus Utilities.

A collection of utilities for packing data, unpacking data computing checksums, and decode checksums.

`pymodbus.utilities.checkCRC(`*data*, *check*`)`

Check if the data matches the passed in CRC.

Parameters

- **data** – The data to create a crc16 of
- **check** – The CRC to validate

Returns
True if matched, False otherwise

`pymodbus.utilities.checkLRC(`*data*, *check*`)`

Check if the passed in data matches the LRC.

Parameters

- **data** – The data to calculate
- **check** – The LRC to validate

Returns
True if matched, False otherwise

`pymodbus.utilities.computeCRC(`*data*`)`

Compute a crc16 on the passed in string.

For modbus, this is only used on the binary serial protocols (in this case RTU).

The difference between modbus's crc16 and a normal crc16 is that modbus starts the crc value out at 0xffff.

Parameters
`data` – The data to create a crc16 of

Returns
The calculated CRC

`pymodbus.utilities.computeLRC(`*data*`)`

Use to compute the longitudinal redundancy check against a string.

This is only used on the serial ASCII modbus protocol. A full description of this implementation can be found in appendix B of the serial line modbus description.

Parameters
`data` – The data to apply a lrc to

Returns
The calculated LRC

`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*`)`

Create a string out of an array of bits.

Parameters
`bits` – A bit array

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`**`(`*string*`)`

Create bit array out of a string.

Parameters
`string` – The modbus data packet to decode

example:

```
bytes  = "bytes to decode"
result = unpack_bitstring(bytes)
```

CHAPTER

ELEVEN

EXAMPLES

The examples can be downloaded from https://github.com/pymodbus-dev/pymodbus/tree/dev/examples

11.1 Examples version 3.x

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

11.1.1 Asynchronous Client Example

```
#!/usr/bin/env python3
"""Pymodbus Aynchronous Client Example.

An example of a single threaded synchronous client.

usage: client_async.py [-h] [--comm {tcp,udp,serial,tls}]
                       [--framer {ascii,binary,rtu,socket,tls}]
                       [--log {critical,error,warning,info,debug}]
                       [--port PORT]
options:
  -h, --help            show this help message and exit
  --comm {tcp,udp,serial,tls}
                        "serial", "tcp", "udp" or "tls"
  --framer {ascii,binary,rtu,socket,tls}
                        "ascii", "binary", "rtu", "socket" or "tls"
  --log {critical,error,warning,info,debug}
                        "critical", "error", "warning", "info" or "debug"
  --port PORT           the port to use
  --baudrate BAUDRATE   the baud rate to use for the serial device

The corresponding server must be started before e.g. as:
    python3 server_sync.py
"""
import asyncio
import logging
import os

# --------------------------------------------------------------------------- #
```

(continues on next page)

(continued from previous page)

```
# import the various client implementations
# --------------------------------------------------------------------------- #
from examples.helper import get_commandline
from pymodbus.client import (
    AsyncModbusSerialClient,
    AsyncModbusTcpClient,
    AsyncModbusTlsClient,
    AsyncModbusUdpClient,
)


_logger = logging.getLogger()


def setup_async_client(args):
    """Run client setup."""
    _logger.info("### Create client object")
    if args.comm == "tcp":
        client = AsyncModbusTcpClient(
            args.host,
            port=args.port,  # on which port
            # Common optional paramers:
            framer=args.framer,
            #    timeout=10,
            #    retries=3,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # TCP setup parameters
            #    source_address=("localhost", 0),
        )
    elif args.comm == "udp":
        client = AsyncModbusUdpClient(
            args.host,
            port=args.port,
            # Common optional paramers:
            framer=args.framer,
            #    timeout=10,
            #    retries=3,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # UDP setup parameters
            #    source_address=None,
        )
    elif args.comm == "serial":
        client = AsyncModbusSerialClient(
            args.port,
            # Common optional paramers:
            #    framer=ModbusRtuFramer,
            #    timeout=10,
            #    retries=3,
```

(continues on next page)

(continued from previous page)

```
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # Serial setup parameters
            baudrate=args.baudrate,
            #    bytesize=8,
            #    parity="N",
            #    stopbits=1,
            #    handle_local_echo=False,
        )
    elif args.comm == "tls":
        cwd = os.getcwd().split("/")[-1]
        if cwd == "examples":
            path = "."
        elif cwd == "test":
            path = "../examples"
        else:
            path = "examples"
        client = AsyncModbusTlsClient(
            args.host,
            port=args.port,
            # Common optional paramers:
            framer=args.framer,
            #    timeout=10,
            #    retries=3,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # TLS setup parameters
            #    sslctx=sslctx,
            certfile=f"{path}/certificates/pymodbus.crt",
            keyfile=f"{path}/certificates/pymodbus.key",
            #    password="none",
            server_hostname="localhost",
        )
    return client


async def run_async_client(client, modbus_calls=None):
    """Run sync client."""
    _logger.info("### Client starting")
    await client.connect()
    assert client.connected
    if modbus_calls:
        await modbus_calls(client)
    await client.close()
    _logger.info("### End of Program")


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=False,
```

(continues on next page)

(continued from previous page)

```
        description="Run asynchronous client.",
    )
    testclient = setup_async_client(cmd_args)
    asyncio.run(run_async_client(testclient), debug=True)
```

11.1.2 Asynchronous Client basic calls example

```
#!/usr/bin/env python3
"""Pymodbus Client modbus call examples.

Please see:

    async_template_call

    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. The difference between async
and sync is a single 'await' so the calls are not repeated.

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
"""
import asyncio
import logging

import pymodbus.diag_message as req_diag
import pymodbus.mei_message as req_mei
import pymodbus.other_message as req_other
from examples.client_async import run_async_client, setup_async_client
```

(continues on next page)

(continued from previous page)

```python
from examples.client_sync import run_sync_client, setup_sync_client
from examples.helper import get_commandline
from pymodbus.exceptions import ModbusException
from pymodbus.pdu import ExceptionResponse


_logger = logging.getLogger()


SLAVE = 0x01


# --------------------------------------------------
# Template on how to make modbus calls (sync/async).
# all calls follow the same schema,
# --------------------------------------------------


async def async_template_call(client):
    """Show complete modbus call, async version."""
    try:
        rr = await client.read_coils(1, 1, slave=SLAVE)
    except ModbusException as exc:
        txt = f"ERROR: exception in pymodbus {exc}"
        _logger.error(txt)
        raise exc
    if rr.isError():
        txt = "ERROR: pymodbus returned an error!"
        _logger.error(txt)
        raise ModbusException(txt)
    if isinstance(rr, ExceptionResponse):
        txt = "ERROR: received exception from device {rr}!"
        _logger.error(txt)
        # THIS IS NOT A PYTHON EXCEPTION, but a valid modbus message
        raise ModbusException(txt)

    # Validate data
    txt = f"### Template coils response: {str(rr.bits)}"
    _logger.debug(txt)


def template_call(client):
    """Show complete modbus call, sync version."""
    try:
        rr = client.read_coils(1, 1, slave=SLAVE)
    except ModbusException as exc:
        txt = f"ERROR: exception in pymodbus {exc}"
        _logger.error(txt)
        raise exc
    if rr.isError():
        txt = "ERROR: pymodbus returned an error!"
        _logger.error(txt)
```

(continues on next page)

(continued from previous page)

```
        raise ModbusException(txt)
    if isinstance(rr, ExceptionResponse):
        txt = "ERROR: received exception from device {rr}!"
        _logger.error(txt)
        # THIS IS NOT A PYTHON EXCEPTION, but a valid modbus message
        raise ModbusException(txt)

    # Validate data
    txt = f"### Template coils response: {str(rr.bits)}"
    _logger.debug(txt)


# -------------------------------------------------
# Generic error handling, to avoid duplicating code
# -------------------------------------------------


def _check_call(rr):
    """Check modbus call worked generically."""
    assert not rr.isError()  # test that call was OK
    assert not isinstance(rr, ExceptionResponse)  # Device rejected request
    return rr


# ------------------------------------------------------
# Call modbus device (all possible calls are presented).
# ------------------------------------------------------
async def _handle_coils(client):
    """Read/Write coils."""
    _logger.info("### Reading Coil different number of bits (return 8 bits multiples)")
    rr = _check_call(await client.read_coils(1, 1, slave=SLAVE))
    assert len(rr.bits) == 8

    rr = _check_call(await client.read_coils(1, 5, slave=SLAVE))
    assert len(rr.bits) == 8

    rr = _check_call(await client.read_coils(1, 12, slave=SLAVE))
    assert len(rr.bits) == 16

    rr = _check_call(await client.read_coils(1, 17, slave=SLAVE))
    assert len(rr.bits) == 24

    _logger.info("### Write false/true to coils and read to verify")
    _check_call(await client.write_coil(0, True, slave=SLAVE))
    rr = _check_call(await client.read_coils(0, 1, slave=SLAVE))
    assert rr.bits[0]  # test the expected value

    _check_call(await client.write_coils(1, [True] * 21, slave=SLAVE))
    rr = _check_call(await client.read_coils(1, 21, slave=SLAVE))
    resp = [True] * 21
    # If the returned output quantity is not a multiple of eight,
    # the remaining bits in the final data byte will be padded with zeros
```

(continues on next page)

(continued from previous page)

```
    # (toward the high order end of the byte).
    resp.extend([False] * 3)
    assert rr.bits == resp  # test the expected value

    _logger.info("### Write False to address 1-8 coils")
    _check_call(await client.write_coils(1, [False] * 8, slave=SLAVE))
    rr = _check_call(await client.read_coils(1, 8, slave=SLAVE))
    assert rr.bits == [False] * 8  # test the expected value


async def _handle_discrete_input(client):
    """Read discrete inputs."""
    _logger.info("### Reading discrete input, Read address:0-7")
    rr = _check_call(await client.read_discrete_inputs(0, 8, slave=SLAVE))
    assert len(rr.bits) == 8


async def _handle_holding_registers(client):
    """Read/write holding registers."""
    _logger.info("### write holding register and read holding registers")
    _check_call(await client.write_register(1, 10, slave=SLAVE))
    rr = _check_call(await client.read_holding_registers(1, 1, slave=SLAVE))
    assert rr.registers[0] == 10

    _check_call(await client.write_registers(1, [10] * 8, slave=SLAVE))
    rr = _check_call(await client.read_holding_registers(1, 8, slave=SLAVE))
    assert rr.registers == [10] * 8

    _logger.info("### write read holding registers, using **kwargs")
    arguments = {
        "read_address": 1,
        "read_count": 8,
        "write_address": 1,
        "write_registers": [256, 128, 100, 50, 25, 10, 5, 1],
    }
    _check_call(await client.readwrite_registers(slave=SLAVE, **arguments))
    rr = _check_call(await client.read_holding_registers(1, 8, slave=SLAVE))
    assert rr.registers == arguments["write_registers"]


async def _handle_input_registers(client):
    """Read input registers."""
    _logger.info("### read input registers")
    rr = _check_call(await client.read_input_registers(1, 8, slave=SLAVE))
    assert len(rr.registers) == 8


async def _execute_information_requests(client):
    """Execute extended information requests."""
    _logger.info("### Running information requests.")
    rr = _check_call(
        await client.execute(req_mei.ReadDeviceInformationRequest(unit=SLAVE))
```

(continues on next page)

(continued from previous page)

```
    )
    assert rr.information[0] == b"Pymodbus"

    rr = _check_call(await client.execute(req_other.ReportSlaveIdRequest(unit=SLAVE)))
    assert rr.status

    rr = _check_call(
        await client.execute(req_other.ReadExceptionStatusRequest(unit=SLAVE))
    )
    assert not rr.status

    rr = _check_call(
        await client.execute(req_other.GetCommEventCounterRequest(unit=SLAVE))
    )
    assert rr.status and not rr.count

    rr = _check_call(await client.execute(req_other.GetCommEventLogRequest(unit=SLAVE)))
    assert rr.status and not (rr.event_count + rr.message_count + len(rr.events))


async def _execute_diagnostic_requests(client):
    """Execute extended diagnostic requests."""
    _logger.info("### Running diagnostic requests.")
    rr = _check_call(await client.execute(req_diag.ReturnQueryDataRequest(unit=SLAVE)))
    assert not rr.message[0]

    _check_call(
        await client.execute(req_diag.RestartCommunicationsOptionRequest(unit=SLAVE))
    )
    _check_call(
        await client.execute(req_diag.ReturnDiagnosticRegisterRequest(unit=SLAVE))
    )
    _check_call(
        await client.execute(req_diag.ChangeAsciiInputDelimiterRequest(unit=SLAVE))
    )

    # NOT WORKING: _check_call(await client.execute(req_diag.
↪ForceListenOnlyModeRequest(unit=SLAVE)))
    # does not send a response

    _check_call(await client.execute(req_diag.ClearCountersRequest()))
    _check_call(
        await client.execute(
            req_diag.ReturnBusCommunicationErrorCountRequest(unit=SLAVE)
        )
    )
    _check_call(
        await client.execute(req_diag.ReturnBusExceptionErrorCountRequest(unit=SLAVE))
    )
    _check_call(
        await client.execute(req_diag.ReturnSlaveMessageCountRequest(unit=SLAVE))
    )
```

(continues on next page)

(continued from previous page)

```
    _check_call(
        await client.execute(req_diag.ReturnSlaveNoResponseCountRequest(unit=SLAVE))
    )
    _check_call(await client.execute(req_diag.ReturnSlaveNAKCountRequest(unit=SLAVE)))
    _check_call(await client.execute(req_diag.ReturnSlaveBusyCountRequest(unit=SLAVE)))
    _check_call(
        await client.execute(
            req_diag.ReturnSlaveBusCharacterOverrunCountRequest(unit=SLAVE)
        )
    )
    _check_call(await client.execute(req_diag.ReturnIopOverrunCountRequest(unit=SLAVE)))
    _check_call(await client.execute(req_diag.ClearOverrunCountRequest(unit=SLAVE)))
    # NOT WORKING _check_call(await client.execute(req_diag.
↪GetClearModbusPlusRequest(unit=SLAVE)))


# ------------------------
# Run the calls in groups.
# -------------------------


async def run_async_calls(client):
    """Demonstrate basic read/write calls."""
    await async_template_call(client)
    await _handle_coils(client)
    await _handle_discrete_input(client)
    await _handle_holding_registers(client)
    await _handle_input_registers(client)
    await _execute_information_requests(client)
    await _execute_diagnostic_requests(client)


def run_sync_calls(client):
    """Demonstrate basic read/write calls."""
    template_call(client)


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=False,
        description="Run modbus calls in asynchronous client.",
    )
    testclient = setup_async_client(cmd_args)
    asyncio.run(run_async_client(testclient, modbus_calls=run_async_calls))
    testclient = setup_sync_client(cmd_args)
    run_sync_client(testclient, modbus_calls=run_sync_calls)
```

11.1.3 Modbus Payload Example

```
#!/usr/bin/env python3
"""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
"""
import asyncio
import logging
from collections import OrderedDict

from examples.client_async import run_async_client, setup_async_client
from examples.helper import get_commandline
from pymodbus.constants import Endian
from pymodbus.payload import BinaryPayloadBuilder, BinaryPayloadDecoder


_logger = logging.getLogger()
ORDER_DICT = {"<": "LITTLE", ">": "BIG"}


async def run_payload_calls(client):
    """Run binary payload.

    If you need to build a complex message to send, you can use the payload
    builder to simplify the packing logic

    Packing/unpacking depends on your CPU's word/byte order. Modbus messages
    are always using big endian. BinaryPayloadBuilder will per default use
    what your CPU uses.
    The wordorder is applicable only for 32 and 64 bit values
    Lets say we need to write a value 0x12345678 to a 32 bit register
    The following combinations could be used to write the register
    ++++++++++++++++++++++++++++++++++++++++++++
    Word Order  | Byte order | Word1  | Word2  |
    ------------+------------+--------+--------+
        Big     |     Big    | 0x1234 | 0x5678 |
        Big     |    Little  | 0x3412 | 0x7856 |
       Little   |     Big    | 0x5678 | 0x1234 |
       Little   |    Little  | 0x7856 | 0x3412 |
    ++++++++++++++++++++++++++++++++++++++++++++
    """
    for word_endian, byte_endian in (
        (Endian.Big, Endian.Big),
        (Endian.Big, Endian.Little),
        (Endian.Little, Endian.Big),
        (Endian.Little, Endian.Little),
    ):
        print("-" * 60)
```

(continues on next page)

(continued from previous page)

```
        print(f"Word Order: {ORDER_DICT[word_endian]}")
        print(f"Byte Order: {ORDER_DICT[byte_endian]}")
        print()
        builder = BinaryPayloadBuilder(
            wordorder=word_endian,
            byteorder=byte_endian,
        )
        # Normally just do:  builder = BinaryPayloadBuilder()
        my_string = "abcdefgh"
        builder.add_string(my_string)
        builder.add_bits([0, 1, 0, 1, 1, 0, 1, 0])
        builder.add_8bit_int(-0x12)
        builder.add_8bit_uint(0x12)
        builder.add_16bit_int(-0x5678)
        builder.add_16bit_uint(0x1234)
        builder.add_32bit_int(-0x1234)
        builder.add_32bit_uint(0x12345678)
        builder.add_16bit_float(12.34)
        builder.add_16bit_float(-12.34)
        builder.add_32bit_float(22.34)
        builder.add_32bit_float(-22.34)
        builder.add_64bit_int(-0xDEADBEEF)
        builder.add_64bit_uint(0x12345678DEADBEEF)
        builder.add_64bit_uint(0x12345678DEADBEEF)
        builder.add_64bit_float(123.45)
        builder.add_64bit_float(-123.45)
        registers = builder.to_registers()
        print("Writing Registers:")
        print(registers)
        print("\n")
        payload = builder.build()
        address = 0
        slave = 1
        # We can write registers
        rr = await client.write_registers(address, registers, slave=slave)
        assert not rr.isError()
        # Or we can write an encoded binary string
        rr = await client.write_registers(address, payload, skip_encode=True, slave=1)
        assert not rr.isError()

        # ----------------------------------------------------------------------- #
        # If you need to decode a collection of registers in a weird layout, the
        # payload decoder can help you as well.
        # ----------------------------------------------------------------------- #
        print("Reading Registers:")
        count = len(payload)
        rr = await client.read_holding_registers(address, count, slave=slave)
        assert not rr.isError()
        print(rr.registers)
        print("\n")
        decoder = BinaryPayloadDecoder.fromRegisters(
            rr.registers, byteorder=byte_endian, wordorder=word_endian
```

(continues on next page)

(continued from previous page)

```
        )
        # Make sure word/byte order is consistent between BinaryPayloadBuilder and␣
↪BinaryPayloadDecoder
        assert (
            decoder._byteorder == builder._byteorder  # pylint: disable=protected-access
        )
        assert (
            decoder._wordorder == builder._wordorder  # pylint: disable=protected-access
        )

        decoded = OrderedDict(
            [
                ("string", decoder.decode_string(len(my_string))),
                ("bits", decoder.decode_bits()),
                ("8int", decoder.decode_8bit_int()),
                ("8uint", decoder.decode_8bit_uint()),
                ("16int", decoder.decode_16bit_int()),
                ("16uint", decoder.decode_16bit_uint()),
                ("32int", decoder.decode_32bit_int()),
                ("32uint", decoder.decode_32bit_uint()),
                ("16float", decoder.decode_16bit_float()),
                ("16float2", decoder.decode_16bit_float()),
                ("32float", decoder.decode_32bit_float()),
                ("32float2", decoder.decode_32bit_float()),
                ("64int", decoder.decode_64bit_int()),
                ("64uint", decoder.decode_64bit_uint()),
                ("ignore", decoder.skip_bytes(8)),
                ("64float", decoder.decode_64bit_float()),
                ("64float2", decoder.decode_64bit_float()),
            ]
        )
        print("Decoded Data")
        for name, value in iter(decoded.items()):
            print(f"{name}\t{hex(value) if isinstance(value, int) else value}")
        print("\n")


if __name__ == "__main__":
    cmd_args = get_commandline(
        description="Run payload client.",
    )
    testclient = setup_async_client(cmd_args)
    asyncio.run(run_async_client(testclient, modbus_calls=run_payload_calls))
```

11.1.4 Synchronous Client Example

```
#!/usr/bin/env python3
"""Pymodbus Synchronous Client Example.

An example of a single threaded synchronous client.

usage: client_sync.py [-h] [--comm {tcp,udp,serial,tls}]
                      [--framer {ascii,binary,rtu,socket,tls}]
                      [--log {critical,error,warning,info,debug}]
                      [--port PORT]
options:
  -h, --help            show this help message and exit
  --comm {tcp,udp,serial,tls}
                        "serial", "tcp", "udp" or "tls"
  --framer {ascii,binary,rtu,socket,tls}
                        "ascii", "binary", "rtu", "socket" or "tls"
  --log {critical,error,warning,info,debug}
                        "critical", "error", "warning", "info" or "debug"
  --port PORT           the port to use
  --baudrate BAUDRATE   the baud rate to use for the serial device

The corresponding server must be started before e.g. as:
    python3 server_sync.py
"""
import logging
import os

# --------------------------------------------------------------------------- #
# import the various client implementations
# --------------------------------------------------------------------------- #
from examples.helper import get_commandline
from pymodbus.client import (
    ModbusSerialClient,
    ModbusTcpClient,
    ModbusTlsClient,
    ModbusUdpClient,
)


_logger = logging.getLogger()


def setup_sync_client(args):
    """Run client setup."""
    _logger.info("### Create client object")
    if args.comm == "tcp":
        client = ModbusTcpClient(
            args.host,
            port=args.port,
            # Common optional paramers:
            framer=args.framer,
            #    timeout=10,
```

(continues on next page)

(continued from previous page)

```
            #    retries=3,
            #    retry_on_empty=False,y
            #    close_comm_on_error=False,
            #    strict=True,
            # TCP setup parameters
            #    source_address=("localhost", 0),
        )
    elif args.comm == "udp":
        client = ModbusUdpClient(
            args.host,
            port=args.port,
            # Common optional paramers:
            framer=args.framer,
            #    timeout=10,
            #    retries=3,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # UDP setup parameters
            #    source_address=None,
        )
    elif args.comm == "serial":
        client = ModbusSerialClient(
            port=args.port,  # serial port
            # Common optional paramers:
            #    framer=ModbusRtuFramer,
            #    timeout=10,
            #    retries=3,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,.
            #    strict=True,
            # Serial setup parameters
            baudrate=args.baudrate,
            #    bytesize=8,
            #    parity="N",
            #    stopbits=1,
            #    handle_local_echo=False,
        )
    elif args.comm == "tls":
        cwd = os.getcwd().split("/")[-1]
        if cwd == "examples":
            path = "."
        elif cwd == "test":
            path = "../examples"
        else:
            path = "examples"
        client = ModbusTlsClient(
            args.host,
            port=args.port,
            # Common optional paramers:
            framer=args.framer,
            #    timeout=10,
```

(continues on next page)

(continued from previous page)

```
            #    retries=3,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # TLS setup parameters
            #    sslctx=None,
            certfile=f"{path}/certificates/pymodbus.crt",
            keyfile=f"{path}/certificates/pymodbus.key",
            #    password=None,
            server_hostname="localhost",
        )
    return client


def run_sync_client(client, modbus_calls=None):
    """Run sync client."""
    _logger.info("### Client starting")
    client.connect()
    if modbus_calls:
        modbus_calls(client)
    client.close()
    _logger.info("### End of Program")


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=False,
        description="Run synchronous client.",
    )
    testclient = setup_sync_client(cmd_args)
    run_sync_client(testclient)
```

11.1.5 Forwarder Example

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

```

(continues on next page)

(continued from previous page)

```
**WARNING** THIS EXAMPLE IS KNOWN TO HAVE PROBLEMS, a wrong solution.
"""
import asyncio
import logging

from examples.helper import get_commandline
from pymodbus.client import AsyncModbusTcpClient
from pymodbus.datastore import ModbusServerContext
from pymodbus.datastore.remote import RemoteSlaveContext
from pymodbus.server import StartAsyncTcpServer


_logger = logging.getLogger()


async def setup_forwarder(args):
    """Do setup forwarder."""

    return args


async def run_forwarder(args):
    """Run forwarder setup."""
    txt = f"### start forwarder, listen {args.port}, connect to {args.client_port}"
    _logger.info(txt)

    args.client = AsyncModbusTcpClient(
        host="localhost",
        port=args.client_port,
    )
    await args.client.connect()
    assert args.client.connected
    # If required to communicate with a specified client use slave=<slave_id>
    # in RemoteSlaveContext
    # For e.g to forward the requests to slave with slave address 1 use
    # store = RemoteSlaveContext(client, slave=1)
    if args.slaves:
        store = {}
        for i in args.slaves:
            store[i.to_bytes(1, "big")] = RemoteSlaveContext(args.client, slave=i)
    else:
        store = RemoteSlaveContext(args.client, slave=1)
    args.context = ModbusServerContext(slaves=store, single=True)

    await StartAsyncTcpServer(context=args.context, address=("localhost", args.port))
    # loop forever


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=True,
        description="Run asynchronous forwarder.",
```

(continues on next page)

(continued from previous page)

```
        extras=[
            (
                "--client_port",
                {
                    "help": "the port to use",
                    "type": int,
                },
            )
        ],
    )
    asyncio.run(run_forwarder(cmd_args))
```

11.1.6 Asynchronous server example

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

Command line options for examples

options:
  -h, --help            show this help message and exit
  --comm {tcp,udp,serial,tls}
                        "serial", "tcp", "udp" or "tls"
  --framer {ascii,binary,rtu,socket,tls}
                        "ascii", "binary", "rtu", "socket" or "tls"
  --log {critical,error,warning,info,debug}
                        "critical", "error", "warning", "info" or "debug"
  --port PORT           the port to use
  --baudrate BAUDRATE   the baud rate to use for the serial device
  --store {sequential,sparse,factory,none}
                        "sequential", "sparse", "factory" or "none"
  --slaves SLAVES       number of slaves to respond to

The corresponding client can be started as:
    python3 client_sync.py
"""
import asyncio
import logging
import os

from examples.helper import get_commandline
from pymodbus import __version__ as pymodbus_version
```

(continues on next page)

(continued from previous page)

```python
from pymodbus.datastore import (
    ModbusSequentialDataBlock,
    ModbusServerContext,
    ModbusSlaveContext,
    ModbusSparseDataBlock,
)
from pymodbus.device import ModbusDeviceIdentification

# ----------------------------------------------------------------------- #
# import the various client implementations
# ----------------------------------------------------------------------- #
from pymodbus.server import (
    StartAsyncSerialServer,
    StartAsyncTcpServer,
    StartAsyncTlsServer,
    StartAsyncUdpServer,
)


_logger = logging.getLogger()


def setup_server(args):
    """Run server setup."""
    # The datastores only respond to the addresses that are initialized
    # If you initialize a DataBlock to addresses of 0x00 to 0xFF, a request to
    # 0x100 will respond with an invalid address exception.
    # This is because many devices exhibit this kind of behavior (but not all)
    if not args.context:
        _logger.info("### Create datastore")
        if args.store == "sequential":
            # Continuing, use a sequential block without gaps.
            datablock = ModbusSequentialDataBlock(0x00, [17] * 100)
        elif args.store == "sparse":
            # Continuing, or use a sparse DataBlock which can have gaps
            datablock = ModbusSparseDataBlock({0x00: 0, 0x05: 1})
        elif args.store == "factory":
            # Alternately, use the factory methods to initialize the DataBlocks
            # or simply do not pass them to have them initialized to 0x00 on the
            # full address range::
            datablock = ModbusSequentialDataBlock.create()

        if args.slaves:
            # The server then makes use of a server context that allows the server
            # to respond with different slave contexts for different slave ids.
            # By default it will return the same context for every slave id supplied
            # (broadcast mode).
            # However, this can be overloaded by setting the single flag to False and
            # then supplying a dictionary of slave id to context mapping::
            #
            # The slave context can also be initialized in zero_mode which means
            # that a request to address(0-7) will map to the address (0-7).
```

(continues on next page)

(continued from previous page)

```
            # The default is False which is based on section 4.4 of the
            # specification, so address(0-7) will map to (1-8)::
            context = {
                0x01: ModbusSlaveContext(
                    di=datablock,
                    co=datablock,
                    hr=datablock,
                    ir=datablock,
                ),
                0x02: ModbusSlaveContext(
                    di=datablock,
                    co=datablock,
                    hr=datablock,
                    ir=datablock,
                ),
                0x03: ModbusSlaveContext(
                    di=datablock,
                    co=datablock,
                    hr=datablock,
                    ir=datablock,
                    zero_mode=True,
                ),
            }
            single = False
        else:
            context = ModbusSlaveContext(
                di=datablock, co=datablock, hr=datablock, ir=datablock, unit=1
            )
            single = True

        # Build data storage
        args.context = ModbusServerContext(slaves=context, single=single)

    # ----------------------------------------------------------------------- #
    # initialize the server information
    # ----------------------------------------------------------------------- #
    # If you don't set this or any fields, they are defaulted to empty strings.
    # ----------------------------------------------------------------------- #
    args.identity = ModbusDeviceIdentification(
        info_name={
            "VendorName": "Pymodbus",
            "ProductCode": "PM",
            "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
            "ProductName": "Pymodbus Server",
            "ModelName": "Pymodbus Server",
            "MajorMinorRevision": pymodbus_version,
        }
    )
    return args


async def run_async_server(args):
```

(continues on next page)

(continued from previous page)

```
    """Run server."""
    txt = f"### start ASYNC server, listening on {args.port} - {args.comm}"
    _logger.info(txt)
    if args.comm == "tcp":
        address = ("", args.port) if args.port else None
        server = await StartAsyncTcpServer(
            context=args.context,  # Data storage
            identity=args.identity,  # server identify
            # TBD host=
            # TBD port=
            address=address,  # listen address
            # custom_functions=[],  # allow custom handling
            framer=args.framer,  # The framer strategy to use
            # handler=None,  # handler for each session
            allow_reuse_address=True,  # allow the reuse of an address
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # timeout=1,  # waiting time for request to complete
            # TBD strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    elif args.comm == "udp":
        address = ("127.0.0.1", args.port) if args.port else None
        server = await StartAsyncUdpServer(
            context=args.context,  # Data storage
            identity=args.identity,  # server identify
            address=address,  # listen address
            # custom_functions=[],  # allow custom handling
            framer=args.framer,  # The framer strategy to use
            # handler=None,  # handler for each session
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # timeout=1,  # waiting time for request to complete
            # TBD strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    elif args.comm == "serial":
        # socat -d -d PTY,link=/tmp/ptyp0,raw,echo=0,ispeed=9600
        #             PTY,link=/tmp/ttyp0,raw,echo=0,ospeed=9600
        server = await StartAsyncSerialServer(
            context=args.context,  # Data storage
            identity=args.identity,  # server identify
            # timeout=1,  # waiting time for request to complete
            port=args.port,  # serial port
            # custom_functions=[],  # allow custom handling
            framer=args.framer,  # The framer strategy to use
            # handler=None,  # handler for each session
            # stopbits=1,  # The number of stop bits to use
            # bytesize=8,  # The bytesize of the serial messages
            # parity="N",  # Which kind of parity to use
            baudrate=args.baudrate,  # The baud rate to use for the serial device
            # handle_local_echo=False,  # Handle local echo of the USB-to-RS485 adaptor
```

(continues on next page)

(continued from previous page)

```
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    elif args.comm == "tls":
        address = ("", args.port) if args.port else None
        cwd = os.getcwd().split("/")[-1]
        if cwd == "examples":
            path = "."
        elif cwd == "test":
            path = "../examples"
        else:
            path = "examples"
        server = await StartAsyncTlsServer(
            context=args.context,  # Data storage
            host="localhost",  # define tcp address where to connect to.
            # port=port,  # on which port
            identity=args.identity,  # server identify
            # custom_functions=[],  # allow custom handling
            address=address,  # listen address
            framer=args.framer,  # The framer strategy to use
            # handler=None,  # handler for each session
            allow_reuse_address=True,  # allow the reuse of an address
            certfile=f"{path}/certificates/pymodbus.crt",  # The cert file path for TLS␣
↪(used if sslctx is None)
            # sslctx=sslctx,  # The SSLContext to use for TLS (default None and auto␣
↪create)
            keyfile=f"{path}/certificates/pymodbus.key",  # The key file path for TLS␣
↪(used if sslctx is None)
            # password="none",  # The password for for decrypting the private key file
            # reqclicert=False,  # Force the sever request client"s certificate
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # timeout=1,  # waiting time for request to complete
            # TBD strict=True,  # use strict timing, t1.5 for Modbus RTU
            defer_start=False,  # Only define server do not activate
        )
    return server


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=True,
        description="Run asynchronous server.",
    )
    run_args = setup_server(cmd_args)
    asyncio.run(run_async_server(run_args), debug=True)
```

11.1.7 Modbus Payload Server example

```python
#!/usr/bin/env python3
"""Pymodbus Server Payload Example.

This example shows how to initialize a server with a
complicated memory layout using builder.
"""
import asyncio
import logging

from examples.helper import get_commandline
from examples.server_async import run_async_server, setup_server
from pymodbus import pymodbus_apply_logging_config
from pymodbus.constants import Endian
from pymodbus.datastore import (
    ModbusSequentialDataBlock,
    ModbusServerContext,
    ModbusSlaveContext,
)
from pymodbus.payload import BinaryPayloadBuilder


_logger = logging.getLogger()


def setup_payload_server(args):
    """Define payload for server and do setup."""

    pymodbus_apply_logging_config()
    _logger.setLevel(logging.DEBUG)

    # ----------------------------------------------------------------------- #
    # build your payload
    # ----------------------------------------------------------------------- #
    builder = BinaryPayloadBuilder(byteorder=Endian.Little, wordorder=Endian.Little)
    builder.add_string("abcdefgh")
    builder.add_bits([0, 1, 0, 1, 1, 0, 1, 0])
    builder.add_8bit_int(-0x12)
    builder.add_8bit_uint(0x12)
    builder.add_16bit_int(-0x5678)
    builder.add_16bit_uint(0x1234)
    builder.add_32bit_int(-0x1234)
    builder.add_32bit_uint(0x12345678)
    builder.add_16bit_float(12.34)
    builder.add_16bit_float(-12.34)
    builder.add_32bit_float(22.34)
    builder.add_32bit_float(-22.34)
    builder.add_64bit_int(-0xDEADBEEF)
    builder.add_64bit_uint(0x12345678DEADBEEF)
    builder.add_64bit_uint(0xDEADBEEFDEADBEED)
    builder.add_64bit_float(123.45)
    builder.add_64bit_float(-123.45)
```

(continues on next page)

(continued from previous page)

```
    # ----------------------------------------------------------------------- #
    # use that payload in the data store
    # Here we use the same reference block for each underlying store.
    # ----------------------------------------------------------------------- #

    block = ModbusSequentialDataBlock(1, builder.to_registers())
    store = ModbusSlaveContext(di=block, co=block, hr=block, ir=block)
    args.context = ModbusServerContext(slaves=store, single=True)
    return setup_server(args)


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=True,
        description="Run payload server.",
    )
    run_args = setup_payload_server(cmd_args)
    asyncio.run(run_async_server(run_args))
```

11.1.8 Synchronous server example

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

Command line options for examples

options:
  -h, --help            show this help message and exit
  --comm {tcp,udp,serial,tls}
                        "serial", "tcp", "udp" or "tls"
  --framer {ascii,binary,rtu,socket,tls}
                        "ascii", "binary", "rtu", "socket" or "tls"
  --log {critical,error,warning,info,debug}
                        "critical", "error", "warning", "info" or "debug"
  --port PORT           the port to use
  --baudrate BAUDRATE   the baud rate to use for the serial device
  --store {sequential,sparse,factory,none}
                        "sequential", "sparse", "factory" or "none"
  --slaves SLAVES       number of slaves to respond to

The corresponding client can be started as:
```

(continues on next page)

(continued from previous page)

```
    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.
"""
import logging
import os

from examples.helper import get_commandline
from examples.server_async import setup_server

# --------------------------------------------------------------------------- #
# import the various client implementations
# --------------------------------------------------------------------------- #
from pymodbus.server import (
    StartSerialServer,
    StartTcpServer,
    StartTlsServer,
    StartUdpServer,
)


_logger = logging.getLogger()


def run_sync_server(args):
    """Run server."""
    txt = f"### start SYNC server, listening on {args.port} - {args.comm}"
    _logger.info(txt)
    if args.comm == "tcp":
        address = ("", args.port) if args.port else None
        server = StartTcpServer(
            context=args.context,  # Data storage
            identity=args.identity,  # server identify
            # TBD host=
            # TBD port=
            address=address,  # listen address
            # custom_functions=[],  # allow custom handling
            framer=args.framer,  # The framer strategy to use
            # TBD handler=None,  # handler for each session
            allow_reuse_address=True,  # allow the reuse of an address
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # timeout=1,  # waiting time for request to complete
            # TBD strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    elif args.comm == "udp":
        address = ("", args.port) if args.port else None
        server = StartUdpServer(
            context=args.context,  # Data storage
```

(continues on next page)

(continued from previous page)

```
            identity=args.identity,  # server identify
            # TBD host=
            # TBD port=
            address=address,  # listen address
            # custom_functions=[],  # allow custom handling
            framer=args.framer,  # The framer strategy to use
            # TBD handler=None,  # handler for each session
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # timeout=1,  # waiting time for request to complete
            # TBD strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    elif args.comm == "serial":
        # socat -d -d PTY,link=/tmp/ptyp0,raw,echo=0,ispeed=9600
        #             PTY,link=/tmp/ttyp0,raw,echo=0,ospeed=9600
        server = StartSerialServer(
            context=args.context,  # Data storage
            identity=args.identity,  # server identify
            # timeout=1,  # waiting time for request to complete
            port=args.port,  # serial port
            # custom_functions=[],  # allow custom handling
            framer=args.framer,  # The framer strategy to use
            # handler=None,  # handler for each session
            # stopbits=1,  # The number of stop bits to use
            # bytesize=7,  # The bytesize of the serial messages
            # parity="E",  # Which kind of parity to use
            baudrate=args.baudrate,  # The baud rate to use for the serial device
            # handle_local_echo=False,  # Handle local echo of the USB-to-RS485 adaptor
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    elif args.comm == "tls":
        address = ("", args.port) if args.port else None
        cwd = os.getcwd().split("/")[-1]
        if cwd == "examples":
            path = "."
        elif cwd == "test":
            path = "../examples"
        else:
            path = "examples"
        server = StartTlsServer(
            context=args.context,  # Data storage
            host="localhost",  # define tcp address where to connect to.
            # port=port,  # on which port
            identity=args.identity,  # server identify
            # custom_functions=[],  # allow custom handling
            address=None,  # listen address
            framer=args.framer,  # The framer strategy to use
            # handler=None,  # handler for each session
```

(continues on next page)

(continued from previous page)

```
            allow_reuse_address=True,  # allow the reuse of an address
            certfile=f"{path}/certificates/pymodbus.crt",  # The cert file path for TLS␣
↪(used if sslctx is None)
            # sslctx=None,  # The SSLContext to use for TLS (default None and auto␣
↪create)
            keyfile=f"{path}/certificates/pymodbus.key",  # The key file path for TLS␣
↪(used if sslctx is None)
            # password=None,  # The password for for decrypting the private key file
            # reqclicert=False,  # Force the sever request client"s certificate
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat slave_id 0 as broadcast address,
            # timeout=1,  # waiting time for request to complete
            # TBD strict=True,  # use strict timing, t1.5 for Modbus RTU
            # defer_start=False,  # Only define server do not activate
        )
    return server


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=True,
        description="Run synchronous server.",
    )
    run_args = setup_server(cmd_args)
    server = run_sync_server(run_args)
    server.shutdown()
```

11.1.9 Updating server example

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

Command line options for examples

options:
  -h, --help            show this help message and exit
  --comm {tcp,udp,serial,tls}
                        "serial", "tcp", "udp" or "tls"
  --framer {ascii,binary,rtu,socket,tls}
                        "ascii", "binary", "rtu", "socket" or "tls"
  --log {critical,error,warning,info,debug}
                        "critical", "error", "warning", "info" or "debug"
```

(continues on next page)

(continued from previous page)

```
  --port PORT           the port to use
  --store {sequential,sparse,factory,none}
                        "sequential", "sparse", "factory" or "none"
  --slaves SLAVES       number of slaves to respond to

The corresponding client can be started as:
    python3 client_sync.py
"""
import asyncio
import logging

from examples.helper import get_commandline
from examples.server_async import run_async_server, setup_server
from pymodbus.datastore import (
    ModbusSequentialDataBlock,
    ModbusServerContext,
    ModbusSlaveContext,
)


_logger = logging.getLogger()


async def updating_task(context):
    """Run every so often,

    and updates live values of the context. It should be noted
    that there is a lrace condition for the update.
    """
    _logger.debug("updating the context")
    fc_as_hex = 3
    slave_id = 0x00
    address = 0x10
    values = context[slave_id].getValues(fc_as_hex, address, count=5)
    values = [v + 1 for v in values]  # increment by 1.
    txt = f"new values: {str(values)}"
    _logger.debug(txt)
    context[slave_id].setValues(fc_as_hex, address, values)
    await asyncio.sleep(1)


def setup_updating_server(args):
    """Run server setup."""
    # The datastores only respond to the addresses that are initialized
    # If you initialize a DataBlock to addresses of 0x00 to 0xFF, a request to
    # 0x100 will respond with an invalid address exception.
    # This is because many devices exhibit this kind of behavior (but not all)

    # Continuing, use a sequential block without gaps.
    datablock = ModbusSequentialDataBlock(0x00, [17] * 100)
    context = ModbusSlaveContext(
        di=datablock, co=datablock, hr=datablock, ir=datablock, unit=1
```

(continues on next page)

(continued from previous page)

```
    )
    args.context = ModbusServerContext(slaves=context, single=True)
    return setup_server(args)


async def run_updating_server(args):
    """Start updater task and async server."""
    asyncio.create_task(updating_task(args.context))
    await run_async_server(args)


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=True,
        description="Run asynchronous server.",
    )
    run_args = setup_updating_server(cmd_args)
    asyncio.run(run_updating_server(run_args), debug=True)
```

11.2 Examples contributions

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

11.2.1 Serial Forwarder example

```
"""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
"""
import argparse
import asyncio
import logging
import signal

from pymodbus.client import ModbusSerialClient
from pymodbus.datastore import ModbusServerContext
from pymodbus.datastore.remote import RemoteSlaveContext
from pymodbus.server.async_io import ModbusTcpServer


FORMAT = "%(asctime)-15s %(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
logging.basicConfig(format=FORMAT)
_logger = logging.getLogger()


def raise_graceful_exit(*_args):
    """Enters shutdown mode"""
```

(continues on next page)

(continued from previous page)

```
    _logger.info("receiving shutdown signal now")
    raise SystemExit


class SerialForwarderTCPServer:
    """SerialRTU2TCP Forwarder Server"""

    def __init__(self):
        """Initialize the server"""
        self.server = None

    async def run(self):
        """Run the server"""
        port, baudrate, server_port, server_ip, slaves = get_commandline()
        client = ModbusSerialClient(method="rtu", port=port, baudrate=baudrate)
        message = f"RTU bus on {port} - baudrate {baudrate}"
        _logger.info(message)
        store = {}
        for i in slaves:
            store[i] = RemoteSlaveContext(client, slave=i)
        context = ModbusServerContext(slaves=store, single=False)
        self.server = ModbusTcpServer(
            context, address=(server_ip, server_port), allow_reuse_address=True
        )
        message = f"serving on {server_ip} port {server_port}"
        _logger.info(message)
        message = f"listening to slaves {context.slaves()}"
        _logger.info(message)
        await self.server.serve_forever()

    async def stop(self):
        """Stop the server"""
        if self.server:
            await self.server.shutdown()
            _logger.info("TCP server is down")


def get_commandline():
    """Read and validate command line arguments"""
    logchoices = ["critical", "error", "warning", "info", "debug"]

    parser = argparse.ArgumentParser(description="Command line options")
    parser.add_argument("--log", help=",".join(logchoices), default="info", type=str)
    parser.add_argument(
        "--port", help="RTU serial port", default="/dev/ttyUSB0", type=str
    )
    parser.add_argument("--baudrate", help="RTU baudrate", default=9600, type=int)
    parser.add_argument("--server_port", help="server port", default=5020, type=int)
    parser.add_argument("--server_ip", help="server IP", default="127.0.0.1", type=str)
    parser.add_argument(
        "--slaves", help="list of slaves to forward", type=int, nargs="+"
    )
```

(continues on next page)

(continued from previous page)

```
    args = parser.parse_args()

    # set defaults
    _logger.setLevel(
        args.log.upper() if args.log.lower() in logchoices else logging.INFO
    )
    if not args.slaves:
        args.slaves = {1, 2, 3}
    return args.port, args.baudrate, args.server_port, args.server_ip, args.slaves


if __name__ == "__main__":
    server = SerialForwarderTCPServer()
    try:
        signal.signal(signal.SIGINT, raise_graceful_exit)
        asyncio.run(server.run())
    finally:
        asyncio.run(server.stop())
```

11.3 Examples version 2.5.3

These examples have not been upgraded to v3.0.0 but are still relevant.

Help is wanted to upgrade the examples.

11.3.1 Bcd Payload example

```
# pylint: disable=missing-type-doc,missing-param-doc,differing-param-doc,missing-raises-
↪doc,missing-any-param-doc
"""Modbus BCD Payload Builder.

This is an example of building a custom payload builder
that can be used in the pymodbus library. Below is a
simple binary coded decimal builder and decoder.
"""
from struct import pack

from pymodbus.constants import Endian
from pymodbus.exceptions import ParameterException
from pymodbus.payload import BinaryPayloadDecoder
from pymodbus.utilities import pack_bitstring, unpack_bitstring


def convert_to_bcd(decimal):
    """Convert a decimal value to a bcd value

    :param value: The decimal value to to pack into bcd
    :returns: The number in bcd form
```

(continues on next page)

(continued from previous page)

```
    """
    place, bcd = 0, 0
    while decimal > 0:
        nibble = decimal % 10
        bcd += nibble << place
        decimal /= 10
        place += 4
    return bcd


def convert_from_bcd(bcd):
    """Convert a bcd value to a decimal value

    :param value: The value to unpack from bcd
    :returns: The number in decimal form
    """
    place, decimal = 1, 0
    while bcd > 0:
        nibble = bcd & 0xF
        decimal += nibble * place
        bcd >>= 4
        place *= 10
    return decimal


def count_bcd_digits(bcd):
    """Count the number of digits in a bcd value

    :param bcd: The bcd number to count the digits of
    :returns: The number of digits in the bcd string
    """
    count = 0
    while bcd > 0:
        count += 1
        bcd >>= 4
    return count


class BcdPayloadBuilder:
    """A utility that helps build binary coded decimal payload messages

    to be written with the various modbus messages.
    example::

        builder = BcdPayloadBuilder()
        builder.add_number(1)
        builder.add_number(int(2.234 * 1000))
        payload = builder.build()
    """

    def __init__(self, payload=None, endian=Endian.Little):
        """Initialize a new instance of the payload builder
```

(continues on next page)

(continued from previous page)

```
        :param payload: Raw payload data to initialize with
        :param endian: The endianness of the payload
        """
        self._endian = endian
        self._payload = payload or []

    def __str__(self):
        """Return the payload buffer as a string

        :returns: The payload buffer as a string
        """
        return "".join(self._payload)

    def reset(self):
        """Reset the payload buffer"""
        self._payload = []

    def build(self):
        """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
        """
        string = str(self)
        length = len(string)
        string = string + ("\x00" * (length % 2))
        return [string[i : i + 2] for i in range(0, length, 2)]

    def add_bits(self, values):
        """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.

        :param value: The value to add to the buffer
        """
        value = pack_bitstring(values)
        self._payload.append(value)

    def add_number(self, value, size=None):
        """Add any 8bit numeric type to the buffer

        :param value: The value to add to the buffer
        """
        encoded = []
        value = convert_to_bcd(value)
        size = size or count_bcd_digits(value)
        while size > 0:
```

(continues on next page)

(continued from previous page)

```
            nibble = value & 0xF
            encoded.append(pack("B", nibble))
            value >>= 4
            size -= 1
        self._payload.extend(encoded)

    def add_string(self, value):
        """Add a string to the buffer

        :param value: The value to add to the buffer
        """
        self._payload.append(value)


class BcdPayloadDecoder:
    """A utility that helps decode binary coded decimal payload messages from a modbus␣
↪response message.

    What follows is a simple example::

        decoder = BcdPayloadDecoder(payload)
        first   = decoder.decode_int(2)
        second  = decoder.decode_int(5) / 100
    """

    def __init__(self, payload):
        """Initialize a new payload decoder

        :param payload: The payload to decode with
        """
        self._payload = payload
        self._pointer = 0x00

    @staticmethod
    def fromRegisters(registers, endian=Endian.Little):  # pylint: disable=invalid-name
        """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.

        :param registers: The register results to initialize with
        :param endian: The endianness of the payload
        :returns: An initialized PayloadDecoder
        """
        if isinstance(registers, list):  # repack into flat binary
            payload = "".join(pack(">H", x) for x in registers)
            return BinaryPayloadDecoder(payload, endian)
        raise ParameterException("Invalid collection of registers supplied")
```

(continues on next page)

(continued from previous page)

```
    @staticmethod
    def fromCoils(coils, endian=Endian.Little):  # pylint: disable=invalid-name
        """Initialize a payload decoder.

        with the result of reading a collection of coils from a modbus device.

        The coils are treated as a list of bit(boolean) values.

        :param coils: The coil results to initialize with
        :param endian: The endianness of the payload
        :returns: An initialized PayloadDecoder
        """
        if isinstance(coils, list):
            payload = pack_bitstring(coils)
            return BinaryPayloadDecoder(payload, endian)
        raise ParameterException("Invalid collection of coils supplied")

    def reset(self):
        """Reset the decoder pointer back to the start"""
        self._pointer = 0x00

    def decode_int(self, size=1):
        """Decode a int or long from the buffer"""
        self._pointer += size
        handle = self._payload[self._pointer - size : self._pointer]
        return convert_from_bcd(handle)

    def decode_bits(self):
        """Decode a byte worth of bits from the buffer"""
        self._pointer += 1
        handle = self._payload[self._pointer - 1 : self._pointer]
        return unpack_bitstring(handle)

    def decode_string(self, size=1):
        """Decode a string from the buffer

        :param size: The size of the string to decode
        """
        self._pointer += size
        return self._payload[self._pointer - size : self._pointer]


# --------------------------------------------------------------------------- #
# Exported Identifiers
# --------------------------------------------------------------------------- #

__all__ = ["BcdPayloadBuilder", "BcdPayloadDecoder"]
```

11.3.2 Callback Server example

```
#!/usr/bin/env python3
# pylint: disable=missing-type-doc,missing-param-doc,differing-param-doc
"""Pymodbus Server With Callbacks.

This is an example of adding callbacks to a running modbus server
when a value is written to it. In order for this to work, it needs
a device-mapping file.
"""
import logging
from multiprocessing import Queue
from threading import Thread

# --------------------------------------------------------------------------- #
# import the modbus libraries we need
# --------------------------------------------------------------------------- #
from pymodbus import __version__ as pymodbus_version
from pymodbus.datastore import (
    ModbusServerContext,
    ModbusSlaveContext,
    ModbusSparseDataBlock,
)
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.server import StartTcpServer


# from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer


# --------------------------------------------------------------------------- #
# configure the service logging
# --------------------------------------------------------------------------- #
log = logging.getLogger()
log.setLevel(logging.DEBUG)

# --------------------------------------------------------------------------- #
# create your custom data block with callbacks
# --------------------------------------------------------------------------- #


class CallbackDataBlock(ModbusSparseDataBlock):
    """A datablock that stores the new value in memory,

    and passes the operation to a message queue for further processing.
    """

    def __init__(self, devices, queue):
        """Initialize."""
        self.devices = devices
        self.queue = queue

        values = {k: 0 for k in devices.keys()}
```

(continues on next page)

(continued from previous page)

```
        values[0xBEEF] = len(values)  # the number of devices
        super().__init__(values)

    def setValues(self, address, value):  # pylint: disable=arguments-differ
        """Set the requested values of the datastore

        :param address: The starting address
        :param values: The new values to be set
        """
        super().setValues(address, value)
        self.queue.put((self.devices.get(address, None), value))


# --------------------------------------------------------------------------- #
# define your callback process
# --------------------------------------------------------------------------- #


def rescale_value(value):
    """Rescale the input value from the range of 0..100 to -3200..3200.

    :param value: The input value to scale
    :returns: The rescaled value
    """
    scale = 1 if value >= 50 else -1
    cur = value if value < 50 else (value - 50)
    return scale * (cur * 64)


def device_writer(queue):
    """Process new messages from a queue to write to device outputs

    :param queue: The queue to get new messages from
    """
    while True:
        device, value = queue.get()
        rescale_value(value[0])
        txt = f"Write({device}) = {value}"
        log.debug(txt)
        if not device:
            continue
        # do any logic here to update your devices


# --------------------------------------------------------------------------- #
# initialize your device map
# --------------------------------------------------------------------------- #


def read_device_map(path):
    """Read the device path to address mapping from file::
```

(continues on next page)

(continued from previous page)

```
        0x0001,/dev/device1
        0x0002,/dev/device2

    :param path: The path to the input file
    :returns: The input mapping file
    """
    devices = {}
    with open(path, "r") as stream:  # pylint: disable=unspecified-encoding
        for line in stream:
            piece = line.strip().split(",")
            devices[int(piece[0], 16)] = piece[1]
    return devices


def run_callback_server():
    """Run callback server."""
    # ----------------------------------------------------------------------- #
    # initialize your data store
    # ----------------------------------------------------------------------- #
    queue = Queue()
    devices = read_device_map("device-mapping")
    block = CallbackDataBlock(devices, queue)
    store = ModbusSlaveContext(di=block, co=block, hr=block, ir=block)
    context = ModbusServerContext(slaves=store, single=True)

    # ----------------------------------------------------------------------- #
    # initialize the server information
    # ----------------------------------------------------------------------- #
    identity = ModbusDeviceIdentification(
        info_name={
            "VendorName": "pymodbus",
            "ProductCode": "PM",
            "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
            "ProductName": "pymodbus Server",
            "ModelName": "pymodbus Server",
            "MajorMinorRevision": pymodbus_version,
        }
    )

    # ----------------------------------------------------------------------- #
    # run the server you want
    # ----------------------------------------------------------------------- #
    thread = Thread(target=device_writer, args=(queue,))
    thread.start()
    StartTcpServer(context, identity=identity, address=("localhost", 5020))


if __name__ == "__main__":
    run_callback_server()
```

11.3.3 Concurrent Client example

```
#!/usr/bin/env python3
# pylint: disable=missing-type-doc,missing-param-doc,differing-param-doc
"""Concurrent Modbus Client.

This is an example of writing a high performance modbus client that allows
a high level of concurrency by using worker threads/processes to handle
writing/reading from one or more client handles at once.
"""
import itertools

# -------------------------------------------------------------------------- #
# import system libraries
# -------------------------------------------------------------------------- #
import logging
import multiprocessing
import threading
from collections import namedtuple
from concurrent.futures import Future
from multiprocessing import Event as mEvent
from multiprocessing import Process as mProcess
from multiprocessing import Queue as mQueue
from queue import Queue as qQueue
from threading import Event, Thread

# -------------------------------------------------------------------------- #
# import necessary modbus libraries
# -------------------------------------------------------------------------- #
from pymodbus.client.mixin import ModbusClientMixin


# -------------------------------------------------------------------------- #
# configure the client logging
# -------------------------------------------------------------------------- #
log = logging.getLogger("pymodbus")
log.setLevel(logging.DEBUG)
logging.basicConfig()


# -------------------------------------------------------------------------- #
# Initialize out concurrency primitives
# -------------------------------------------------------------------------- #
class _Primitives:  # pylint: disable=too-few-public-methods)
    """This is a helper class.

    used to group the threading primitives depending on the type of
    worker situation we want to run (threads or processes).
    """

    def __init__(self, **kwargs):
        self.queue = kwargs.get("queue")
        self.event = kwargs.get("event")
```

(continues on next page)

(continued from previous page)

```
        self.worker = kwargs.get("worker")

    @classmethod
    def create(cls, in_process=False):
        """Initialize a new instance of the concurrency primitives.

        :param in_process: True for threaded, False for processes
        :returns: An initialized instance of concurrency primitives
        """
        if in_process:
            return cls(queue=qQueue, event=Event, worker=Thread)
        return cls(queue=mQueue, event=mEvent, worker=mProcess)


# --------------------------------------------------------------------------- #
# Define our data transfer objects
# --------------------------------------------------------------------------- #
# These will be used to serialize state between the various workers.
# We use named tuples here as they are very lightweight while giving us
# all the benefits of classes.
# --------------------------------------------------------------------------- #
WorkRequest = namedtuple("WorkRequest", "request, work_id")
WorkResponse = namedtuple("WorkResponse", "is_exception, work_id, response")


# --------------------------------------------------------------------------- #
# Define our worker processes
# --------------------------------------------------------------------------- #
def _client_worker_process(factory, input_queue, output_queue, is_shutdown):
    """Take input requests,

    issues them on its
    client handle, and then sends the client response (success or failure)
    to the manager to deliver back to the application.

    It should be noted that there are N of these workers and they can
    be run in process or out of process as all the state serializes.

    :param factory: A client factory used to create a new client
    :param input_queue: The queue to pull new requests to issue
    :param output_queue: The queue to place client responses
    :param is_shutdown: Condition variable marking process shutdown
    """
    txt = f"starting up worker : {threading.current_thread()}"
    log.info(txt)
    my_client = factory()
    while not is_shutdown.is_set():
        try:
            workitem = input_queue.get(timeout=1)
            txt = f"dequeue worker request: {workitem}"
            log.debug(txt)
            if not workitem:
```

(continues on next page)

(continued from previous page)

```
                continue
            try:
                txt = f"executing request on thread: {workitem}"
                log.debug(txt)
                result = my_client.execute(workitem.request)
                output_queue.put(WorkResponse(False, workitem.work_id, result))
            except Exception as exc:  # pylint: disable=broad-except
                txt = f"error in worker thread: {threading.current_thread()}"
                log.exception(txt)
                output_queue.put(WorkResponse(True, workitem.work_id, exc))
        except Exception:  # pylint: disable=broad-except
            pass
    txt = f"request worker shutting down: {threading.current_thread()}"
    log.info(txt)


def _manager_worker_process(output_queue, my_futures, is_shutdown):
    """Take output responses and tying them back to the future.

    keyed on the initial transaction id.

    Basically this can be thought of as the delivery worker.

    It should be noted that there are one of these threads and it must
    be an in process thread as the futures will not serialize across
    processes..

    :param output_queue: The queue holding output results to return
    :param futures: The mapping of tid -> future
    :param is_shutdown: Condition variable marking process shutdown
    """
    txt = f"starting up manager worker: {threading.current_thread()}"
    log.info(txt)
    while not is_shutdown.is_set():
        try:
            workitem = output_queue.get()
            my_future = my_futures.get(workitem.work_id, None)
            txt = f"dequeue manager response: {workitem}"
            log.debug(txt)
            if not my_future:
                continue
            if workitem.is_exception:
                my_future.set_exception(workitem.response)
            else:
                my_future.set_result(workitem.response)
            txt = f"updated future result: {my_future}"
            log.debug(txt)
            del futures[workitem.work_id]
        except Exception:  # pylint: disable=broad-except
            log.exception("error in manager")
    txt = f"manager worker shutting down: {threading.current_thread()}"
    log.info(txt)
```

(continues on next page)

(continued from previous page)

```python
# -------------------------------------------------------------------------- #
# Define our concurrent client
# -------------------------------------------------------------------------- #
class ConcurrentClient(ModbusClientMixin):
    """This is a high performance client.

    that can be used to read/write a large number of requests at once asynchronously.
    This operates with a backing worker pool of processes or threads
    to achieve its performance.
    """

    def __init__(self, **kwargs):
        """Initialize a new instance of the client."""
        worker_count = kwargs.get("count", multiprocessing.cpu_count())
        self.factory = kwargs.get("factory")
        primitives = _Primitives.create(kwargs.get("in_process", False))
        self.is_shutdown = primitives.event()  # process shutdown condition
        self.input_queue = primitives.queue()  # input requests to process
        self.output_queue = primitives.queue()  # output results to return
        self.futures = {}  # mapping of tid -> future
        self.workers = []  # handle to our worker threads
        self.counter = itertools.count()

        # creating the response manager
        self.manager = threading.Thread(
            target=_manager_worker_process,
            args=(self.output_queue, self.futures, self.is_shutdown),
        )
        self.manager.start()
        self.workers.append(self.manager)

        # creating the request workers
        for i in range(worker_count):
            worker = primitives.worker(
                target=_client_worker_process,
                args=(
                    self.factory,
                    self.input_queue,
                    self.output_queue,
                    self.is_shutdown,
                ),
            )
            worker.start()
            self.workers.append(worker)

    def shutdown(self):
        """Shutdown all the workersbeing used to concurrently process the requests."""
        log.info("stating to shut down workers")
        self.is_shutdown.set()
        # to wake up the manager
```

(continues on next page)

(continued from previous page)

```
        self.output_queue.put(WorkResponse(None, None, None))
        for worker in self.workers:
            worker.join()
        log.info("finished shutting down workers")

    def execute(self, request):
        """Given a request-

        enqueue it to be processed
        and then return a future linked to the response
        of the call.

        :param request: The request to execute
        :returns: A future linked to the call's response
        """
        fut, work_id = Future(), next(self.counter)
        self.input_queue.put(WorkRequest(request, work_id))
        self.futures[work_id] = fut
        return fut

    def execute_silently(self, request):
        """Given a write request.

        enqueue it to be processed without worrying about calling the
        application back (fire and forget)

        :param request: The request to execute
        """
        self.input_queue.put(WorkRequest(request, None))


if __name__ == "__main__":
    from pymodbus.client import ModbusTcpClient

    def client_factory():
        """Client factory."""
        txt = f"creating client for: {threading.current_thread()}"
        log.debug(txt)
        my_client = ModbusTcpClient("127.0.0.1", port=5020)
        my_client.connect()
        return client

    client = ConcurrentClient(factory=client_factory)
    try:
        log.info("issuing concurrent requests")
        futures = [client.read_coils(i * 8, 8) for i in range(10)]
        log.info("waiting on futures to complete")
        for future in futures:
            txt = f"future result: {future.result(timeout=1)}"
            log.info(txt)
    finally:
        client.shutdown()
```

11.3.4 Custom Message example

```python
#!/usr/bin/env python3
# pylint: disable=missing-type-doc
"""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
"""
import logging
import struct

from pymodbus.bit_read_message import ReadCoilsRequest
from pymodbus.client import ModbusTcpClient as ModbusClient

# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.pdu import ModbusExceptions, ModbusRequest, ModbusResponse


# --------------------------------------------------------------------------- #
# configure the client logging
# --------------------------------------------------------------------------- #
log = logging.getLogger()
log.setLevel(logging.DEBUG)

# --------------------------------------------------------------------------- #
# create your custom message
# --------------------------------------------------------------------------- #
# The following is simply a read coil request that always reads 16 coils.
# Since the function code is already registered with the decoder factory,
# this will be decoded as a read coil response. If you implement a new
# method that is not currently implemented, you must register the request
# and response with a ClientDecoder factory.
# --------------------------------------------------------------------------- #


class CustomModbusResponse(ModbusResponse):
    """Custom modbus response."""

    function_code = 55
    _rtu_byte_count_pos = 2

    def __init__(self, values=None, **kwargs):
        """Initialize."""
        ModbusResponse.__init__(self, **kwargs)
        self.values = values or []
```

(continues on next page)

(continued from previous page)

```
    def encode(self):
        """Encode response pdu

        :returns: The encoded packet message
        """
        res = struct.pack(">B", len(self.values) * 2)
        for register in self.values:
            res += struct.pack(">H", register)
        return res

    def decode(self, data):
        """Decode response pdu

        :param data: The packet data to decode
        """
        byte_count = int(data[0])
        self.values = []
        for i in range(1, byte_count + 1, 2):
            self.values.append(struct.unpack(">H", data[i : i + 2])[0])


class CustomModbusRequest(ModbusRequest):
    """Custom modbus request."""

    function_code = 55
    _rtu_frame_size = 8

    def __init__(self, address=None, slave=0, **kwargs):
        """Initialize."""
        kwargs["unit"] = slave
        ModbusRequest.__init__(self, **kwargs)
        self.address = address
        self.count = 16

    def encode(self):
        """Encode."""
        return struct.pack(">HH", self.address, self.count)

    def decode(self, data):
        """Decode."""
        self.address, self.count = struct.unpack(">HH", data)

    def execute(self, context):
        """Execute."""
        if not 1 <= self.count <= 0x7D0:
            return self.doException(ModbusExceptions.IllegalValue)
        if not context.validate(self.function_code, self.address, self.count):
            return self.doException(ModbusExceptions.IllegalAddress)
        values = context.getValues(self.function_code, self.address, self.count)
        return CustomModbusResponse(values)
```

(continues on next page)

(continued from previous page)

```python
# --------------------------------------------------------------------------- #
# This could also have been defined as
# --------------------------------------------------------------------------- #


class Read16CoilsRequest(ReadCoilsRequest):
    """Read 16 coils in one request."""

    def __init__(self, address, **kwargs):
        """Initialize a new instance

        :param address: The address to start reading from
        """
        ReadCoilsRequest.__init__(self, address, 16, **kwargs)


# --------------------------------------------------------------------------- #
# execute the request with your client
# --------------------------------------------------------------------------- #
# using the with context, the client will automatically be connected
# and closed when it leaves the current scope.
# --------------------------------------------------------------------------- #


if __name__ == "__main__":
    with ModbusClient(host="localhost", port=5020) as client:
        client.register(CustomModbusResponse)
        request = CustomModbusRequest(1, slave=1)
        result = client.execute(request)
        print(result.values)
```

11.3.5 Deviceinfo showcase client example

```python
#!/usr/bin/env python3
"""Pymodbus Synchronous Client Example to showcase Device Information.

This client demonstrates the use of Device Information to get information
about servers connected to the client. This is part of the MODBUS specification,
and uses the MEI 0x2B 0x0E request / response.
"""
import logging

# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.client import ModbusTcpClient as ModbusClient
from pymodbus.device import ModbusDeviceIdentification

# --------------------------------------------------------------------------- #
# import the request
```

(continues on next page)

(continued from previous page)

```
# --------------------------------------------------------------------------- #
from pymodbus.mei_message import ReadDeviceInformationRequest


# from pymodbus.client import ModbusUdpClient as ModbusClient
# from pymodbus.client import ModbusSerialClient as ModbusClient


# --------------------------------------------------------------------------- #
# configure the client logging
# --------------------------------------------------------------------------- #
FORMAT = (
    "%(asctime)-15s %(threadName)-15s "
    "%(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
)
logging.basicConfig(format=FORMAT)
log = logging.getLogger()
log.setLevel(logging.DEBUG)

UNIT = 0x1


def run_sync_client():
    """Run sync client."""
    # ------------------------------------------------------------------------#
    # choose the client you want
    # ------------------------------------------------------------------------#
    # make sure to start an implementation to hit against. For this
    # you can use an existing device, the reference implementation in the tools
    # directory, or start a pymodbus server.
    #
    # If you use the UDP or TCP clients, you can override the framer being used
    # to use a custom implementation (say RTU over TCP). By default they use
    # the socket framer::
    #
    #    client = ModbusClient("localhost", port=5020, framer=ModbusRtuFramer)
    #
    # It should be noted that you can supply an ipv4 or an ipv6 host address
    # for both the UDP and TCP clients.
    #
    # There are also other options that can be set on the client that controls
    # how transactions are performed. The current ones are:
    #
    # * retries - Specify how many retries to allow per transaction (default=3)
    # * retry_on_empty - Is an empty response a retry (default = False)
    # * source_address - Specifies the TCP source address to bind to
    #
    # Here is an example of using these options::
    #
    #    client = ModbusClient("localhost", retries=3, retry_on_empty=True)
    # ------------------------------------------------------------------------#
    client = ModbusClient("localhost", port=5020)
```

(continues on next page)

(continued from previous page)

```
    # from pymodbus.transaction import ModbusRtuFramer
    # client = ModbusClient("localhost", port=5020, framer=ModbusRtuFramer)
    # client = ModbusClient(method="binary", port="/dev/ptyp0", timeout=1)
    # client = ModbusClient(method="ascii", port="/dev/ptyp0", timeout=1)
    # client = ModbusClient(method="rtu", port="/dev/ptyp0", timeout=1,
    #                       baudrate=9600)
    client.connect()

    # ------------------------------------------------------------------------#
    # specify slave to query
    # ------------------------------------------------------------------------#
    # The slave to query is specified in an optional parameter for each
    # individual request. This can be done by specifying the `unit` parameter
    # which defaults to `0x00`
    # ----------------------------------------------------------------------- #
    log.debug("Reading Device Information")
    information = {}
    rr = None

    while not rr or rr.more_follows:
        next_object_id = rr.next_object_id if rr else 0
        rq = ReadDeviceInformationRequest(
            read_code=0x03, unit=UNIT, object_id=next_object_id
        )
        rr = client.execute(rq)
        information.update(rr.information)
        log.debug(rr)

    print("Device Information : ")
    for (
        key
    ) in (
        information.keys()
    ):  # pylint: disable=consider-iterating-dictionary,consider-using-dict-items
        print(key, information[key])

    # ----------------------------------------------------------------------- #
    # You can also have the information parsed through the
    # ModbusDeviceIdentificiation class, which gets you a more usable way
    # to access the Basic and Regular device information objects which are
    # specifically listed in the Modbus specification
    # ----------------------------------------------------------------------- #
    device_id = ModbusDeviceIdentification(info=information)
    print("Product Name : ", device_id.ProductName)

    # ----------------------------------------------------------------------- #
    # close the client
    # ----------------------------------------------------------------------- #
    client.close()


if __name__ == "__main__":
```

(continues on next page)

(continued from previous page)

```
    run_sync_client()
```

11.3.6 Deviceinfo showcase server example

```
#!/usr/bin/env python3
"""Pymodbus Synchronous Server Example to showcase Device Information.

This server demonstrates the use of Device Information to provide information
to clients about the device. This is part of the MODBUS specification, and
uses the MEI 0x2B 0x0E request / response. This example creates an otherwise
empty server.
"""
import logging

from serial import __version__ as pyserial_version

from pymodbus import __version__ as pymodbus_version
from pymodbus.datastore import ModbusServerContext, ModbusSlaveContext

# from pymodbus.server import StartUdpServer
# from pymodbus.server import StartSerialServer
# from pymodbus.transaction import ModbusRtuFramer, ModbusBinaryFramer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.server import StartTcpServer


# --------------------------------------------------------------------------- #
# configure the service logging
# --------------------------------------------------------------------------- #
FORMAT = (
    "%(asctime)-15s %(threadName)-15s"
    " %(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
)
logging.basicConfig(format=FORMAT)
log = logging.getLogger()
log.setLevel(logging.DEBUG)


def run_server():
    """Run server."""
    # ----------------------------------------------------------------------- #
    # initialize your data store
    # ----------------------------------------------------------------------- #
    store = ModbusSlaveContext()
    context = ModbusServerContext(slaves=store, single=True)

    # ----------------------------------------------------------------------- #
    # initialize the server information
    # ----------------------------------------------------------------------- #
    # If you don't set this or any fields, they are defaulted to empty strings.
```

(continues on next page)

(continued from previous page)

```
    # ----------------------------------------------------------------------- #
    identity = ModbusDeviceIdentification(
        info_name={
            "VendorName": "Pymodbus",
            "ProductCode": "PM",
            "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
            "ProductName": "Pymodbus Server",
            "ModelName": "Pymodbus Server",
            "MajorMinorRevision": pymodbus_version,
        }
    )

    # ----------------------------------------------------------------------- #
    # Add an example which is long enough to force the ReadDeviceInformation
    # request / response to require multiple responses to send back all of the
    # information.
    # ----------------------------------------------------------------------- #

    identity[0x80] = (
        "Lorem ipsum dolor sit amet, consectetur adipiscing "
        "elit. Vivamus rhoncus massa turpis, sit amet "
        "ultrices orci semper ut. Aliquam tristique sapien in "
        "lacus pharetra, in convallis nunc consectetur. Nunc "
        "velit elit, vehicula tempus tempus sed. "
    )

    # ----------------------------------------------------------------------- #
    # Add an example with repeated object IDs. The MODBUS specification is
    # entirely silent on whether or not this is allowed. In practice, this
    # should be assumed to be contrary to the MODBUS specification and other
    # clients (other than pymodbus) might behave differently when presented
    # with an object ID occurring twice in the returned information.
    #
    # Use this at your discretion, and at the very least ensure that all
    # objects which share a single object ID can fit together within a single
    # ADU slave. In the case of Modbus RTU, this is about 240 bytes or so. In
    # other words, when the spec says "An object is indivisible, therefore
    # any object must have a size consistent with the size of transaction
    # response", if you use repeated OIDs, apply that rule to the entire
    # grouping of objects with the repeated OID.
    # ----------------------------------------------------------------------- #
    identity[0x81] = [f"pymodbus {pymodbus_version}", f"pyserial {pyserial_version}"]

    # ----------------------------------------------------------------------- #
    # run the server you want
    # ----------------------------------------------------------------------- #
    # Tcp:
    StartTcpServer(context, identity=identity, address=("localhost", 5020))

    # TCP with different framer
    # StartTcpServer(context, identity=identity,
    #                framer=ModbusRtuFramer, address=("0.0.0.0", 5020))
```

(continues on next page)

(continued from previous page)

```
    # Udp:
    # StartUdpServer(context, identity=identity, address=("0.0.0.0", 5020))

    # Ascii:
    # StartSerialServer(context, identity=identity,
    #                    port="/dev/ttyp0", timeout=1)

    # RTU:
    # StartSerialServer(context, framer=ModbusRtuFramer, identity=identity,
    #                   port="/dev/ttyp0", timeout=.005, baudrate=9600)

    # Binary
    # StartSerialServer(context,
    #                   identity=identity,
    #                   framer=ModbusBinaryFramer,
    #                   port="/dev/ttyp0",
    #                   timeout=1)


if __name__ == "__main__":
    run_server()
```

11.3.7 Message Generator example

```
#!/usr/bin/env python3
# pylint: disable=missing-type-doc
"""Modbus Message Generator.

The following is an example of how to generate example encoded messages
for the supplied modbus format:

* tcp    - `./generate-messages.py -f tcp -m rx -b`
* ascii  - `./generate-messages.py -f ascii -m tx -a`
* rtu    - `./generate-messages.py -f rtu -m rx -b`
* binary - `./generate-messages.py -f binary -m tx -b`
"""
import codecs as c
import logging
from optparse import OptionParser  # pylint: disable=deprecated-module

import pymodbus.diag_message as modbus_diag
from pymodbus.bit_read_message import (
    ReadCoilsRequest,
    ReadCoilsResponse,
    ReadDiscreteInputsRequest,
    ReadDiscreteInputsResponse,
)
from pymodbus.bit_write_message import (
    WriteMultipleCoilsRequest,
```

(continues on next page)

(continued from previous page)

```
    WriteMultipleCoilsResponse,
    WriteSingleCoilRequest,
    WriteSingleCoilResponse,
)
from pymodbus.file_message import (
    ReadFifoQueueRequest,
    ReadFifoQueueResponse,
    ReadFileRecordRequest,
    ReadFileRecordResponse,
    WriteFileRecordRequest,
    WriteFileRecordResponse,
)
from pymodbus.mei_message import (
    ReadDeviceInformationRequest,
    ReadDeviceInformationResponse,
)
from pymodbus.other_message import (
    GetCommEventCounterRequest,
    GetCommEventCounterResponse,
    GetCommEventLogRequest,
    GetCommEventLogResponse,
    ReadExceptionStatusRequest,
    ReadExceptionStatusResponse,
    ReportSlaveIdRequest,
    ReportSlaveIdResponse,
)
from pymodbus.register_read_message import (
    ReadHoldingRegistersRequest,
    ReadHoldingRegistersResponse,
    ReadInputRegistersRequest,
    ReadInputRegistersResponse,
    ReadWriteMultipleRegistersRequest,
    ReadWriteMultipleRegistersResponse,
)
from pymodbus.register_write_message import (
    MaskWriteRegisterRequest,
    MaskWriteRegisterResponse,
    WriteMultipleRegistersRequest,
    WriteMultipleRegistersResponse,
    WriteSingleRegisterRequest,
    WriteSingleRegisterResponse,
)

# -------------------------------------------------------------------------- #
# import all the available framers
# -------------------------------------------------------------------------- #
from pymodbus.transaction import (
    ModbusAsciiFramer,
    ModbusBinaryFramer,
    ModbusRtuFramer,
    ModbusSocketFramer,
)
```

(continues on next page)

(continued from previous page)

```
# --------------------------------------------------------------------------- #
# initialize logging
# --------------------------------------------------------------------------- #
modbus_log = logging.getLogger("pymodbus")


# --------------------------------------------------------------------------- #
# enumerate all request messages
# --------------------------------------------------------------------------- #
_request_messages = [
    ReadHoldingRegistersRequest,
    ReadDiscreteInputsRequest,
    ReadInputRegistersRequest,
    ReadCoilsRequest,
    WriteMultipleCoilsRequest,
    WriteMultipleRegistersRequest,
    WriteSingleRegisterRequest,
    WriteSingleCoilRequest,
    ReadWriteMultipleRegistersRequest,
    ReadExceptionStatusRequest,
    GetCommEventCounterRequest,
    GetCommEventLogRequest,
    ReportSlaveIdRequest,
    ReadFileRecordRequest,
    WriteFileRecordRequest,
    MaskWriteRegisterRequest,
    ReadFifoQueueRequest,
    ReadDeviceInformationRequest,
    modbus_diag.ReturnQueryDataRequest,
    modbus_diag.RestartCommunicationsOptionRequest,
    modbus_diag.ReturnDiagnosticRegisterRequest,
    modbus_diag.ChangeAsciiInputDelimiterRequest,
    modbus_diag.ForceListenOnlyModeRequest,
    modbus_diag.ClearCountersRequest,
    modbus_diag.ReturnBusMessageCountRequest,
    modbus_diag.ReturnBusCommunicationErrorCountRequest,
    modbus_diag.ReturnBusExceptionErrorCountRequest,
    modbus_diag.ReturnSlaveMessageCountRequest,
    modbus_diag.ReturnSlaveNoResponseCountRequest,
    modbus_diag.ReturnSlaveNAKCountRequest,
    modbus_diag.ReturnSlaveBusyCountRequest,
    modbus_diag.ReturnSlaveBusCharacterOverrunCountRequest,
    modbus_diag.ReturnIopOverrunCountRequest,
    modbus_diag.ClearOverrunCountRequest,
    modbus_diag.GetClearModbusPlusRequest,
]


# --------------------------------------------------------------------------- #
# enumerate all response messages
```

(continues on next page)

(continued from previous page)

```
# --------------------------------------------------------------------------- #
_response_messages = [
    ReadHoldingRegistersResponse,
    ReadDiscreteInputsResponse,
    ReadInputRegistersResponse,
    ReadCoilsResponse,
    WriteMultipleCoilsResponse,
    WriteMultipleRegistersResponse,
    WriteSingleRegisterResponse,
    WriteSingleCoilResponse,
    ReadWriteMultipleRegistersResponse,
    ReadExceptionStatusResponse,
    GetCommEventCounterResponse,
    GetCommEventLogResponse,
    ReportSlaveIdResponse,
    ReadFileRecordResponse,
    WriteFileRecordResponse,
    MaskWriteRegisterResponse,
    ReadFifoQueueResponse,
    ReadDeviceInformationResponse,
    modbus_diag.ReturnQueryDataResponse,
    modbus_diag.RestartCommunicationsOptionResponse,
    modbus_diag.ReturnDiagnosticRegisterResponse,
    modbus_diag.ChangeAsciiInputDelimiterResponse,
    modbus_diag.ForceListenOnlyModeResponse,
    modbus_diag.ClearCountersResponse,
    modbus_diag.ReturnBusMessageCountResponse,
    modbus_diag.ReturnBusCommunicationErrorCountResponse,
    modbus_diag.ReturnBusExceptionErrorCountResponse,
    modbus_diag.ReturnSlaveMessageCountResponse,
    modbus_diag.ReturnSlaveNoReponseCountResponse,
    modbus_diag.ReturnSlaveNAKCountResponse,
    modbus_diag.ReturnSlaveBusyCountResponse,
    modbus_diag.ReturnSlaveBusCharacterOverrunCountResponse,
    modbus_diag.ReturnIopOverrunCountResponse,
    modbus_diag.ClearOverrunCountResponse,
    modbus_diag.GetClearModbusPlusResponse,
]


# --------------------------------------------------------------------------- #
# build an arguments singleton
# --------------------------------------------------------------------------- #
# Feel free to override any values here to generate a specific message
# in question. It should be noted that many argument names are reused
# between different messages, and a number of messages are simply using
# their default values.
# --------------------------------------------------------------------------- #
_arguments = {
    "address": 0x12,
    "count": 0x08,
    "value": 0x01,
```

(continues on next page)

(continued from previous page)

```
    "values": [0x01] * 8,
    "read_address": 0x12,
    "read_count": 0x08,
    "write_address": 0x12,
    "write_registers": [0x01] * 8,
    "transaction": 0x01,
    "protocol": 0x00,
    "slave": 0xFF,
}


# -------------------------------------------------------------------------- #
# generate all the requested messages
# -------------------------------------------------------------------------- #
def generate_messages(framer, options):
    """Parse the command line options

    :param framer: The framer to encode the messages with
    :param options: The message options to use
    """
    if options.messages == "tx":
        messages = _request_messages
    else:
        messages = _response_messages
    for message in messages:
        message = message(**_arguments)
        print(
            "%-44s = "  # pylint: disable=consider-using-f-string
            % message.__class__.__name__
        )
        packet = framer.buildPacket(message)
        if not options.ascii:
            packet = c.encode(packet, "hex_codec").decode("utf-8")
        print(f"{packet}\n")  # because ascii ends with a \r\n


# -------------------------------------------------------------------------- #
# initialize our program settings
# -------------------------------------------------------------------------- #
def get_options():
    """Parse the command line options

    :returns: The options manager
    """
    parser = OptionParser()

    parser.add_option(
        "-f",
        "--framer",
        help="The type of framer to use (tcp, rtu, binary, ascii)",
        dest="framer",
        default="tcp",
```

(continues on next page)

(continued from previous page)

```
    )

    parser.add_option(
        "-D",
        "--debug",
        help="Enable debug tracing",
        action="store_true",
        dest="debug",
        default=False,
    )

    parser.add_option(
        "-a",
        "--ascii",
        help="The indicates that the message is ascii",
        action="store_true",
        dest="ascii",
        default=True,
    )

    parser.add_option(
        "-b",
        "--binary",
        help="The indicates that the message is binary",
        action="store_false",
        dest="ascii",
    )

    parser.add_option(
        "-m",
        "--messages",
        help="The messages to encode (rx, tx)",
        dest="messages",
        default="rx",
    )

    (opt, _) = parser.parse_args()
    return opt


def main():
    """Run main runner function"""
    option = get_options()

    if option.debug:
        try:
            modbus_log.setLevel(logging.DEBUG)
        except Exception:  # pylint: disable=broad-except
            print("Logging is not supported on this system")

    framer = {
        "tcp": ModbusSocketFramer,
```

(continues on next page)

(continued from previous page)

```
        "rtu": ModbusRtuFramer,
        "binary": ModbusBinaryFramer,
        "ascii": ModbusAsciiFramer,
    }.get(option.framer, ModbusSocketFramer)(None)

    generate_messages(framer, option)


if __name__ == "__main__":
    main()
```

11.3.8 Message Parser example

```
#!/usr/bin/env python3
# pylint: disable=missing-type-doc,missing-param-doc,differing-param-doc,missing-any-
↪param-doc
"""Modbus Message Parser.

The following is an example of how to parse modbus messages
using the supplied framers for a number of protocols:

* tcp
* ascii
* rtu
* binary
"""
# -------------------------------------------------------------------------- #
# import needed libraries
# -------------------------------------------------------------------------- #

import codecs as c
import collections
import logging
import textwrap
from optparse import OptionParser  # pylint: disable=deprecated-module

from pymodbus.factory import ClientDecoder, ServerDecoder
from pymodbus.transaction import (
    ModbusAsciiFramer,
    ModbusBinaryFramer,
    ModbusRtuFramer,
    ModbusSocketFramer,
)


# -------------------------------------------------------------------------- #
# -------------------------------------------------------------------------- #
FORMAT = (
    "%(asctime)-15s %(threadName)-15s"
    " %(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
```

(continues on next page)

(continued from previous page)

```
)
logging.basicConfig(format=FORMAT)
log = logging.getLogger()


# -------------------------------------------------------------------------- #
# build a quick wrapper around the framers
# -------------------------------------------------------------------------- #


class Decoder:
    """Decoder."""

    def __init__(self, framer, encode=False):
        """Initialize a new instance of the decoder

        :param framer: The framer to use
        :param encode: If the message needs to be encoded
        """
        self.framer = framer
        self.encode = encode

    def decode(self, message):
        """Attempt to decode the supplied message

        :param message: The message to decode
        """
        value = message if self.encode else c.encode(message, "hex_codec")
        print("=" * 80)
        print(f"Decoding Message {value}")
        print("=" * 80)
        decoders = [
            self.framer(ServerDecoder(), client=None),
            self.framer(ClientDecoder(), client=None),
        ]
        for decoder in decoders:
            print(f"{decoder.decoder.__class__.__name__}")
            print("-" * 80)
            try:
                decoder.addToFrame(message)
                if decoder.checkFrame():
                    slave = decoder._header.get(  # pylint: disable=protected-access
                        "uid", 0x00
                    )
                    decoder.advanceFrame()
                    decoder.processIncomingPacket(message, self.report, slave)
                else:
                    self.check_errors(decoder, message)
            except Exception:  # pylint: disable=broad-except
                self.check_errors(decoder, message)

    def check_errors(self, decoder, message):
```

(continues on next page)

(continued from previous page)

```
        """Attempt to find message errors

        :param message: The message to find errors in
        """
        txt = f"Unable to parse message - {message} with {decoder}"
        log.error(txt)

    def report(self, message):
        """Print the message information

        :param message: The message to print
        """
        print(
            "%-15s = %s"  # pylint: disable=consider-using-f-string
            % (
                "name",
                message.__class__.__name__,
            )
        )
        for (k_dict, v_dict) in message.__dict__.items():
            if isinstance(v_dict, dict):
                print("%-15s =" % k_dict)  # pylint: disable=consider-using-f-string
                for k_item, v_item in v_dict.items():
                    print(
                        "  %-12s => %s"  # pylint: disable=consider-using-f-string
                        % (k_item, v_item)
                    )

            elif isinstance(v_dict, collections.abc.Iterable):
                print("%-15s =" % k_dict)  # pylint: disable=consider-using-f-string
                value = str([int(x) for x in v_dict])
                for line in textwrap.wrap(value, 60):
                    print(
                        "%-15s . %s"  # pylint: disable=consider-using-f-string
                        % ("", line)
                    )
            else:
                print(
                    "%-15s = %s"  # pylint: disable=consider-using-f-string
                    % (k_dict, hex(v_dict))
                )
        print(
            "%-15s = %s"  # pylint: disable=consider-using-f-string
            % (
                "documentation",
                message.__doc__,
            )
        )


# -------------------------------------------------------------------------- #
# and decode our message
```

(continues on next page)

(continued from previous page)

```python
# -------------------------------------------------------------------------- #
def get_options():
    """Parse the command line options

    :returns: The options manager
    """
    parser = OptionParser()

    parser.add_option(
        "-p",
        "--parser",
        help="The type of parser to use (tcp, rtu, binary, ascii)",
        dest="parser",
        default="tcp",
    )

    parser.add_option(
        "-D",
        "--debug",
        help="Enable debug tracing",
        action="store_true",
        dest="debug",
        default=False,
    )

    parser.add_option(
        "-m", "--message", help="The message to parse", dest="message", default=None
    )

    parser.add_option(
        "-a",
        "--ascii",
        help="The indicates that the message is ascii",
        action="store_true",
        dest="ascii",
        default=False,
    )

    parser.add_option(
        "-b",
        "--binary",
        help="The indicates that the message is binary",
        action="store_false",
        dest="ascii",
    )

    parser.add_option(
        "-f",
        "--file",
        help="The file containing messages to parse",
        dest="file",
        default=None,
```

(continues on next page)

(continued from previous page)

```
    )

    parser.add_option(
        "-t",
        "--transaction",
        help="If the incoming message is in hexadecimal format",
        action="store_true",
        dest="transaction",
        default=False,
    )
    parser.add_option(
        "--framer",
        help="Framer to use",
        dest="framer",
        default=None,
    )

    (opt, arg) = parser.parse_args()

    if not opt.message and len(arg) > 0:
        opt.message = arg[0]

    return opt


def get_messages(option):
    """Do a helper method to generate the messages to parse

    :param options: The option manager
    :returns: The message iterator to parse
    """
    if option.message:
        if option.transaction:
            msg = ""
            for segment in option.message.split():
                segment = segment.replace("0x", "")
                segment = "0" + segment if len(segment) == 1 else segment
                msg = msg + segment
            option.message = msg

        if not option.ascii:
            option.message = c.decode(option.message.encode(), "hex_codec")
        yield option.message
    elif option.file:
        with open(option.file, "r") as handle:  # pylint: disable=unspecified-encoding
            for line in handle:
                if line.startswith("#"):
                    continue
                if not option.ascii:
                    line = line.strip()
                    line = line.decode("hex")
                yield line
```

(continues on next page)

(continued from previous page)

```python


def main():
    """Run main runner function"""
    option = get_options()

    if option.debug:
        try:
            log.setLevel(logging.DEBUG)
        except Exception as exc:  # pylint: disable=broad-except
            print(f"Logging is not supported on this system- {exc}")

    framer = {
        "tcp": ModbusSocketFramer,
        "rtu": ModbusRtuFramer,
        "binary": ModbusBinaryFramer,
        "ascii": ModbusAsciiFramer,
    }.get(option.framer or option.parser, ModbusSocketFramer)

    decoder = Decoder(framer, option.ascii)
    for message in get_messages(option):
        decoder.decode(message)


if __name__ == "__main__":
    main()
```

11.3.9 Modbus Mapper example

```python
# pylint: disable=missing-type-doc
r"""This is used to generate decoder blocks.

so that non-programmers can define the
register values and then decode a modbus device all
without having to write a line of code for decoding.

Currently supported formats are:

* csv
* json
* xml

Here is an example of generating and using a mapping decoder
(note that this is still in the works and will be greatly
simplified in the final api; it is just an example of the
requested functionality)::

    CSV:
    address,type,size,name,function
    0,int16,1,Comm. count PLC,hr
```

(continues on next page)

(continued from previous page)

```
    1,int16,1,Comm. count PLC,hr
    2,int16,1,Comm. count PLC,hr
    3,int16,1,Comm. count PLC,hr
    4,int16,1,Comm. count PLC,hr
    5,int16,1,Comm. count PLC,hr
    6,int16,1,Comm. count PLC,hr
    7,int16,1,Comm. count PLC,hr
    8,int16,1,Comm. count PLC,hr
    9,int16,1,Comm. count PLC,hr
    10,int32,2,Comm. count PLC,hr
    12,int32,2,Comm. count PLC,hr

    from modbus_mapper import csv_mapping_parser
    from modbus_mapper import mapping_decoder
    from pymodbus.client import ModbusTcpClient
    from pymodbus.payload import BinaryPayloadDecoder
    from pymodbus.constants import Endian

    from pprint import pprint
    import logging

    # FORMAT = "%(asctime)-15s %(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
    # logging.basicConfig(format=FORMAT)
    # _logger = logging.getLogger()
    # _logger.setLevel(logging.DEBUG)

    template = ["address", "type", "size", "name", "function"]
    raw_mapping = csv_mapping_parser("simple_mapping_client.csv", template)
    # raw_mapping = csv_mapping_parser("Naust_Comm_to_scr_client.csv", template)
    mapping = mapping_decoder(raw_mapping)

    client = ModbusTcpClient(host="localhost", port=5020)

    response = client.read_holding_registers(address=int(0), count=14)
    decoder = BinaryPayloadDecoder.fromRegisters(
        response.registers, byteorder=Endian.Big, wordorder=Endian.Little
    )

    for block in mapping.items():
        for mac in block:
            if type(mac) == dict:
                # response = client.read_holding_registers(
                #     address=int(mac["address"]), count=mac["size"]
                # )
                # decoder = BinaryPayloadDecoder.fromRegisters(
                #     response.registers, byteorder=Endian.Big, wordorder=Endian.Little
                # )
                print("[{}]\t{}".format(mac["address"], mac["type"]()(decoder)))
                # decoder._payload # remove mac["size"] bytes from beginning

Also, using the same input mapping parsers, we can generate
populated slave contexts that can be run behind a modbus server::
```

(continues on next page)

(continued from previous page)

```
CSV:
address,value,function,name,description
0,0,hr,Comm. count PLC,Comm. count PLC
1,10,hr,Comm. count PLC,Comm. count PLC
2,20,hr,Comm. count PLC,Comm. count PLC
3,30,hr,Comm. count PLC,Comm. count PLC
4,40,hr,Comm. count PLC,Comm. count PLC
5,50,hr,Comm. count PLC,Comm. count PLC
6,60,hr,Comm. count PLC,Comm. count PLC
7,70,hr,Comm. count PLC,Comm. count PLC
8,80,hr,Comm. count PLC,Comm. count PLC
9,90,hr,Comm. count PLC,Comm. count PLC
10,100,hr,Comm. count PLC,Comm. count PLC
11,0,hr,Comm. count PLC,Comm. count PLC
12,120,hr,Comm. count PLC,Comm. count PLC
13,0,hr,Comm. count PLC,Comm. count PLC

from modbus_mapper import csv_mapping_parser
from modbus_mapper import modbus_context_decoder

from pymodbus.server import StartTcpServer
from pymodbus.datastore.context import ModbusServerContext
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.version import version


from pprint import pprint
import logging

FORMAT = "%(asctime)-15s %(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
logging.basicConfig(format=FORMAT)
_logger = logging.getLogger()
_logger.setLevel(logging.DEBUG)

template = ["address", "value", "function", "name", "description"]
raw_mapping = csv_mapping_parser("simple_mapping_server.csv", template)

slave_context = modbus_context_decoder(raw_mapping)
context = ModbusServerContext(slaves=slave_context, single=True)
identity = ModbusDeviceIdentification(
    info_name={
        "VendorName": "Pymodbus",
        "ProductCode": "PM",
        "VendorUrl": "https://github.com/pymodbus-dev/pymodbus/",
        "ProductName": "Pymodbus Server",
        "ModelName": "Pymodbus Server",
        "MajorMinorRevision": version.short(),
    }
)
StartTcpServer(context=context, identity=identity, address=("localhost", 5020))
```

(continues on next page)

(continued from previous page)

```
"""
import csv
import json
from collections import defaultdict
from io import StringIO
from tokenize import generate_tokens

from pymodbus.datastore import ModbusSlaveContext, ModbusSparseDataBlock


# --------------------------------------------------------------------------- #
# raw mapping input parsers
# --------------------------------------------------------------------------- #
# These generate the raw mapping_blocks from some form of input
# which can then be passed to the decoder in question to supply
# the requested output result.
# --------------------------------------------------------------------------- #


def csv_mapping_parser(path, template):
    """Given a csv file of the the mapping data for a modbus device,

    return a mapping layout that can be used to decode an new block.

    .. note:: For the template, a few values are required
    to be defined: address, size, function, and type. All the remaining
    values will be stored, but not formatted by the application.
    So for example::

        template = ["address", "type", "size", "name", "function"]
        mappings = json_mapping_parser("mapping.json", template)

    :param path: The path to the csv input file
    :param template: The row value template
    :returns: The decoded csv dictionary
    """
    mapping_blocks = defaultdict(dict)
    with open(path, "r") as handle:  # pylint: disable=unspecified-encoding
        reader = csv.reader(handle)
        next(reader)  # skip the csv header
        for row in reader:
            mapping = dict(zip(template, row))
            # mapping.pop("function")
            aid = mapping["address"]
            mapping_blocks[aid] = mapping
    return mapping_blocks


def json_mapping_parser(path, template):
    """Given a json file of the the mapping data for a modbus device,

    return a mapping layout that can
```

(continues on next page)

(continued from previous page)

```
    be used to decode an new block.

    .. note:: For the template, a few values are required
    to be mapped: address, size, and type. All the remaining
    values will be stored, but not formatted by the application.
    So for example::

        template = {
            "Start": "address",
            "DataType": "type",
            "Length": "size"
            # the remaining keys will just pass through
        }
        mappings = json_mapping_parser("mapping.json", template)

    :param path: The path to the csv input file
    :param template: The row value template
    :returns: The decoded csv dictionary
    """
    mapping_blocks = {}
    with open(path, "r") as handle:  # pylint: disable=unspecified-encoding
        for tid, rows in json.load(handle).iteritems():
            mappings = {}
            for key, values in rows.iteritems():
                mapping = {template.get(k, k): v for k, v in values.iteritems()}
                mappings[int(key)] = mapping
            mapping_blocks[tid] = mappings
    return mapping_blocks


def xml_mapping_parser():
    """Given an xml file of the the mapping data for a modbus device,

    return a mapping layout that can be used to decode an new block.

    :returns: The decoded csv dictionary
    """


# --------------------------------------------------------------------------- #
# modbus context decoders
# --------------------------------------------------------------------------- #
# These are used to decode a raw mapping_block into a slave context with
# populated function data blocks.
# --------------------------------------------------------------------------- #
def modbus_context_decoder(mapping_blocks):
    """Generate a backing slave context with initialized data blocks.

    .. note:: This expects the following for each block:
    address, value, and function where function is one of
    di (discretes), co (coils), hr (holding registers), or
    ir (input registers).
```

(continues on next page)

(continued from previous page)

```
    :param mapping_blocks: The mapping blocks
    :returns: The initialized modbus slave context
    """
    sparse = ModbusSparseDataBlock()
    sparse.create()
    for block in mapping_blocks.items():
        for mapping in block:
            if type(mapping) == dict:
                value = mapping["value"]
                address = mapping["address"]
                sparse.setValues(address=int(address), values=int(value))
    return ModbusSlaveContext(
        di=sparse, co=sparse, hr=sparse, ir=sparse, zero_mode=True
    )


# --------------------------------------------------------------------------- #
# modbus mapping decoder
# --------------------------------------------------------------------------- #
# These are used to decode a raw mapping_block into a request decoder.
# So this allows one to simply grab a number of registers, and then
# pass them to this decoder which will do the rest.
# --------------------------------------------------------------------------- #
class ModbusTypeDecoder:
    """This is a utility to determine the correct decoder to use given a type name.

    By default this supports all the types available in the default modbus
    decoder, however this can easily be extended this class
    and adding new types to the mapper::

        class CustomTypeDecoder(ModbusTypeDecoder):
            def __init__(self):
                ModbusTypeDecode.__init__(self)
                self.mapper["type-token"] = self.callback

            def parse_my_bitfield(self, tokens):
                return lambda d: d.decode_my_type()

    """

    def __init__(self):
        """Initialize a new instance of the decoder"""
        self.default = lambda m: self.parse_16bit_uint
        self.parsers = {
            "uint": self.parse_16bit_uint,
            "uint8": self.parse_8bit_uint,
            "uint16": self.parse_16bit_uint,
            "uint32": self.parse_32bit_uint,
            "uint64": self.parse_64bit_uint,
            "int": self.parse_16bit_int,
            "int8": self.parse_8bit_int,
```

(continues on next page)

(continued from previous page)

```
            "int16": self.parse_16bit_int,
            "int32": self.parse_32bit_int,
            "int64": self.parse_64bit_int,
            "float": self.parse_32bit_float,
            "float32": self.parse_32bit_float,
            "float64": self.parse_64bit_float,
            "string": self.parse_32bit_int,
            "bits": self.parse_bits,
        }

    # ------------------------------------------------------------ #
    # Type parsers
    # ------------------------------------------------------------ #
    @staticmethod
    def parse_string(tokens):
        """Parse value."""
        _ = next(tokens)
        size = int(next(tokens))
        return lambda d: d.decode_string(size=size)

    @staticmethod
    def parse_bits():
        """Parse value."""
        return lambda d: d.decode_bits()

    @staticmethod
    def parse_8bit_uint():
        """Parse value."""
        return lambda d: d.decode_8bit_uint()

    @staticmethod
    def parse_16bit_uint():
        """Parse value."""
        return lambda d: d.decode_16bit_uint()

    @staticmethod
    def parse_32bit_uint():
        """Parse value."""
        return lambda d: d.decode_32bit_uint()

    @staticmethod
    def parse_64bit_uint():
        """Parse value."""
        return lambda d: d.decode_64bit_uint()

    @staticmethod
    def parse_8bit_int():
        """Parse value."""
        return lambda d: d.decode_8bit_int()

    @staticmethod
    def parse_16bit_int():
```

(continues on next page)

(continued from previous page)

```
        """Parse value."""
        return lambda d: d.decode_16bit_int()

    @staticmethod
    def parse_32bit_int():
        """Parse value."""
        return lambda d: d.decode_32bit_int()

    @staticmethod
    def parse_64bit_int():
        """Parse value."""
        return lambda d: d.decode_64bit_int()

    @staticmethod
    def parse_32bit_float():
        """Parse value."""
        return lambda d: d.decode_32bit_float()

    @staticmethod
    def parse_64bit_float():
        """Parse value."""
        return lambda d: d.decode_64bit_float()

    # ------------------------------------------------------------
    # Public Interface
    # ------------------------------------------------------------
    def tokenize(self, value):
        """Return the tokens

        :param value: The value to tokenize
        :returns: A token generator
        """
        tokens = generate_tokens(StringIO(value).readline)
        for _, tokval, _, _, _ in tokens:
            yield tokval

    def parse(self, value):
        """Return a function that supplied with a decoder,

        will decode the correct value.

        :param value: The type of value to parse
        :returns: The decoder method to use
        """
        tokens = self.tokenize(value)
        token = next(tokens).lower()  # pylint: disable=no-member
        parser = self.parsers.get(token, self.default)
        return parser


def mapping_decoder(mapping_blocks, decoder=None):
    """Convert them into modbus value decoder map.
```

(continues on next page)

(continued from previous page)

```
    :param mapping_blocks: The mapping blocks
    :param decoder: The type decoder to use
    """
    decoder = decoder or ModbusTypeDecoder()
    map = defaultdict(dict)
    for block in mapping_blocks.items():
        for mapping in block:
            if type(mapping) == dict:
                mapping["address"] = mapping["address"]
                mapping["size"] = mapping["size"]
                mapping["type"] = decoder.parse(mapping["type"])
                map[mapping["address"]] = mapping
    return map
```

11.3.10 Modbus Saver example

```
"""These are a collection of helper methods.

that can be used to save a modbus server context to file for backup,
checkpointing, or any other purpose. There use is very
simple::

    context = server.context
    saver   = JsonDatastoreSaver(context)
    saver.save()

These can then be re-opened by the parsers in the
modbus_mapping module. At the moment, the supported
output formats are:

* csv
* json
* xml

To implement your own, simply subclass ModbusDatastoreSaver
and supply the needed callbacks for your given format:

* handle_store_start(self, store)
* handle_store_end(self, store)
* handle_slave_start(self, slave)
* handle_slave_end(self, slave)
* handle_save_start(self)
* handle_save_end(self)
"""
import json
import xml.etree.ElementTree as xml


class ModbusDatastoreSaver:
```

(continues on next page)

(continued from previous page)

```
    """An abstract base class.

    that can be used to implement
    a persistence format for the modbus server context. In
    order to use it, just complete the necessary callbacks
    (SAX style) that your persistence format needs.
    """

    def __init__(self, context, path=None):
        """Initialize a new instance of the saver.

        :param context: The modbus server context
        :param path: The output path to save to
        """
        self.context = context
        self.path = path or "modbus-context-dump"

    def save(self):
        """Save the context to file.

        which calls the various callbacks which the sub classes will implement.
        """
        with open(  # pylint: disable=unspecified-encoding
            self.path, "w"
        ) as self.file_handle:  # pylint: disable=attribute-defined-outside-init
            self.handle_save_start()
            for slave_name, slave in self.context:
                self.handle_slave_start(slave_name)
                for store_name, store in slave.store.iteritems():
                    self.handle_store_start(store_name)
                    self.handle_store_values(iter(store))  # pylint: disable=no-member
                    self.handle_store_end(store_name)
                self.handle_slave_end(slave_name)
            self.handle_save_end()

    # ------------------------------------------------------------
    # predefined state machine callbacks
    # ------------------------------------------------------------
    def handle_save_start(self):
        """Handle save start."""

    def handle_store_start(self, store):
        """Handle store start."""

    def handle_store_end(self, store):
        """Handle store end."""

    def handle_slave_start(self, slave):
        """Handle slave start."""

    def handle_slave_end(self, slave):
        """Handle slave end."""
```

(continues on next page)

(continued from previous page)

```
    def handle_save_end(self):
        """Handle save end."""


# ------------------------------------------------------------------ #
# Implementations of the data store savers
# ------------------------------------------------------------------ #
class JsonDatastoreSaver(ModbusDatastoreSaver):
    """An implementation of the modbus datastore saver.

    that persists the context as a json document.
    """

    _context = None
    _store = None
    _slave = None

    STORE_NAMES = {
        "i": "input-registers",
        "d": "discretes",
        "h": "holding-registers",
        "c": "coils",
    }

    def handle_save_start(self):
        """Handle save start."""
        self._context = {}

    def handle_slave_start(self, slave):
        """Handle slave start."""
        self._context[hex(slave)] = self._slave = {}

    def handle_store_start(self, store):
        """Handle store start."""
        self._store = self.STORE_NAMES[store]

    def handle_store_values(self, values):
        """Handle store values."""
        self._slave[self._store] = dict(values)

    def handle_save_end(self):
        """Handle save end."""
        json.dump(self._context, self.file_handle)


class CsvDatastoreSaver(ModbusDatastoreSaver):
    """An implementation of the modbus datastore saver.

    that persists the context as a csv document.
    """
```

(continues on next page)

(continued from previous page)

```
    _context = None
    _store = None
    _line = None
    NEWLINE = "\r\n"
    HEADER = "slave,store,address,value" + NEWLINE
    STORE_NAMES = {
        "i": "i",
        "d": "d",
        "h": "h",
        "c": "c",
    }

    def handle_save_start(self):
        """Handle save start."""
        self.file_handle.write(self.HEADER)

    def handle_slave_start(self, slave):
        """Handle slave start."""
        self._line = [str(slave)]

    def handle_store_start(self, store):
        """Handle store start."""
        self._line.append(self.STORE_NAMES[store])

    def handle_store_values(self, values):
        """Handle store values."""
        self.file_handle.writelines(self.handle_store_value(values))

    def handle_store_end(self, store):
        """Handle store end."""
        self._line.pop()

    def handle_store_value(self, values):
        """Handle store value."""
        for val_a, val_v in values:
            yield ",".join(self._line + [str(val_a), str(val_v)]) + self.NEWLINE


class XmlDatastoreSaver(ModbusDatastoreSaver):
    """An implementation of the modbus datastore saver.

    that persists the context as a XML document.
    """

    _context = None
    _store = None

    STORE_NAMES = {
        "i": "input-registers",
        "d": "discretes",
        "h": "holding-registers",
        "c": "coils",
```

(continues on next page)

(continued from previous page)

```
    }

    def handle_save_start(self):
        """Handle save start."""
        self._context = xml.Element("context")
        self._root = xml.ElementTree(  # pylint: disable=attribute-defined-outside-init
            self._context
        )

    def handle_slave_start(self, slave):
        """Handle slave start."""
        self._slave = xml.SubElement(  # pylint: disable=attribute-defined-outside-init
            self._context, "slave"
        )
        self._slave.set("id", str(slave))

    def handle_store_start(self, store):
        """Handle store start."""
        self._store = xml.SubElement(self._slave, "store")
        self._store.set("function", self.STORE_NAMES[store])

    def handle_store_values(self, values):
        """Handle store values."""
        for address, value in values:
            entry = xml.SubElement(self._store, "entry")
            entry.text = str(value)
            entry.set("address", str(address))

    def handle_save_end(self):
        """Handle save end."""
        self._root.write(self.file_handle)
```

11.3.11 performance module

```
#!/usr/bin/env python3
# pylint: disable=missing-type-doc
"""Pymodbus Performance Example.

The following is an quick performance check of the synchronous
modbus client.
"""
# --------------------------------------------------------------------------- #
# import the necessary modules
# --------------------------------------------------------------------------- #
import logging
import os
from concurrent.futures import ThreadPoolExecutor as eWorker
from concurrent.futures import as_completed
from threading import Lock
from threading import Thread as tWorker
```

(continues on next page)

(continued from previous page)

```python
from time import time

from pymodbus.client import ModbusTcpClient


try:
    from multiprocessing import Process as mWorker
    from multiprocessing import log_to_stderr
except ImportError:
    log_to_stderr = logging.getLogger

# --------------------------------------------------------------------------- #
# choose between threads or processes
# --------------------------------------------------------------------------- #

# from multiprocessing import Process as Worker
# from threading import Thread as Worker
_thread_lock = Lock()
# --------------------------------------------------------------------------- #
# initialize the test
# --------------------------------------------------------------------------- #
# Modify the parameters below to control how we are testing the client:
#
# * workers - the number of workers to use at once
# * cycles  - the total number of requests to send
# * host    - the host to send the requests to
# --------------------------------------------------------------------------- #
workers = 10  # pylint: disable=invalid-name
cycles = 1000  # pylint: disable=invalid-name
host = "127.0.0.1"  # pylint: disable=invalid-name


# --------------------------------------------------------------------------- #
# perform the test
# --------------------------------------------------------------------------- #
# This test is written such that it can be used by many threads of processes
# although it should be noted that there are performance penalties
# associated with each strategy.
# --------------------------------------------------------------------------- #
def single_client_test(n_host, n_cycles):
    """Perform a single threaded test of a synchronous client against the specified host

    :param n_host: The host to connect to
    :param n_cycles: The number of iterations to perform
    """
    logger = log_to_stderr()
    logger.setLevel(logging.WARNING)
    txt = f"starting worker: {os.getpid()}"
    logger.debug(txt)

    try:
        count = 0
```

(continues on next page)

(continued from previous page)

```
        client = ModbusTcpClient(n_host, port=5020)
        while count < n_cycles:
            client.read_holding_registers(10, 123, slave=1)
            count += 1
    except Exception:  # pylint: disable=broad-except
        logger.exception("failed to run test successfully")
    txt = f"finished worker: {os.getpid()}"
    logger.debug(txt)


def multiprocessing_test(func, extras):
    """Multiprocessing test."""
    start_time = time()
    procs = [mWorker(target=func, args=extras) for _ in range(workers)]

    any(p.start() for p in procs)  # start the workers
    any(p.join() for p in procs)  # wait for the workers to finish
    return start_time


def thread_test(func, extras):
    """Thread test."""
    start_time = time()
    procs = [tWorker(target=func, args=extras) for _ in range(workers)]

    any(p.start() for p in procs)  # start the workers
    any(p.join() for p in procs)  # wait for the workers to finish
    return start_time


def thread_pool_exe_test(func, extras):
    """Thread pool exe."""
    start_time = time()
    with eWorker(max_workers=workers, thread_name_prefix="Perform") as exe:
        futures = {exe.submit(func, *extras): job for job in range(workers)}
        for future in as_completed(futures):
            future.result()
    return start_time


# --------------------------------------------------------------------------- #
# run our test and check results
# --------------------------------------------------------------------------- #
# We shard the total number of requests to perform between the number of
# threads that was specified. We then start all the threads and block on
# them to finish. This may need to switch to another mechanism to signal
# finished as the process/thread start up/shut down may skew the test a bit.

# RTU 32 requests/second @9600
# TCP 31430 requests/second

# --------------------------------------------------------------------------- #
```

(continues on next page)

(continued from previous page)

```


if __name__ == "__main__":
    args = (host, int(cycles * 1.0 / workers))
    # with Worker(max_workers=workers, thread_name_prefix="Perform") as exe:
    #     futures = {exe.submit(single_client_test, *args): job for job in␣
↪range(workers)}
    #     for future in as_completed(futures):
    #         data = future.result()
    # for _ in range(workers):
    #    futures.append(Worker.submit(single_client_test, args=args))
    # procs = [Worker(target=single_client_test, args=args)
    #           for _ in range(workers)]

    # any(p.start() for p in procs)   # start the workers
    # any(p.join() for p in procs)   # wait for the workers to finish
    # start = multiprocessing_test(single_client_test, args)
    # start = thread_pool_exe_test(single_client_test, args)
    for tester in (multiprocessing_test, thread_test, thread_pool_exe_test):
        print(tester.__name__)
        start = tester(single_client_test, args)
        stop = time()
        print(f"{(1.0 * cycles) / (stop - start)} requests/second")
        print(
            f"time taken to complete {cycles} cycle by "
            f"{workers} workers is {stop - start} seconds"
        )
        print()
```

CHAPTER

TWELVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX