PyModbus Documentation

Release 3.1.0

Sanjay

Dec 08, 2022

CONTENTS:

CHAPTER

ONE

PYMODBUS - A PYTHON MODBUS STACK

1.1 Supported versions

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

Version 3.0.2 is the current release (Supports Python >=3.8).

Remark: "Supports" means that we only test with those versions, lower versions (e.g. 3.7) might work depending on the functionality used.

Important: Note 3.0.0 is a major release with a number of incompatible changes.

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.

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 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

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://riptideio.github.io/pymodbus/ or 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:

```
from pymodbus.client import ModbusTcpClient

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

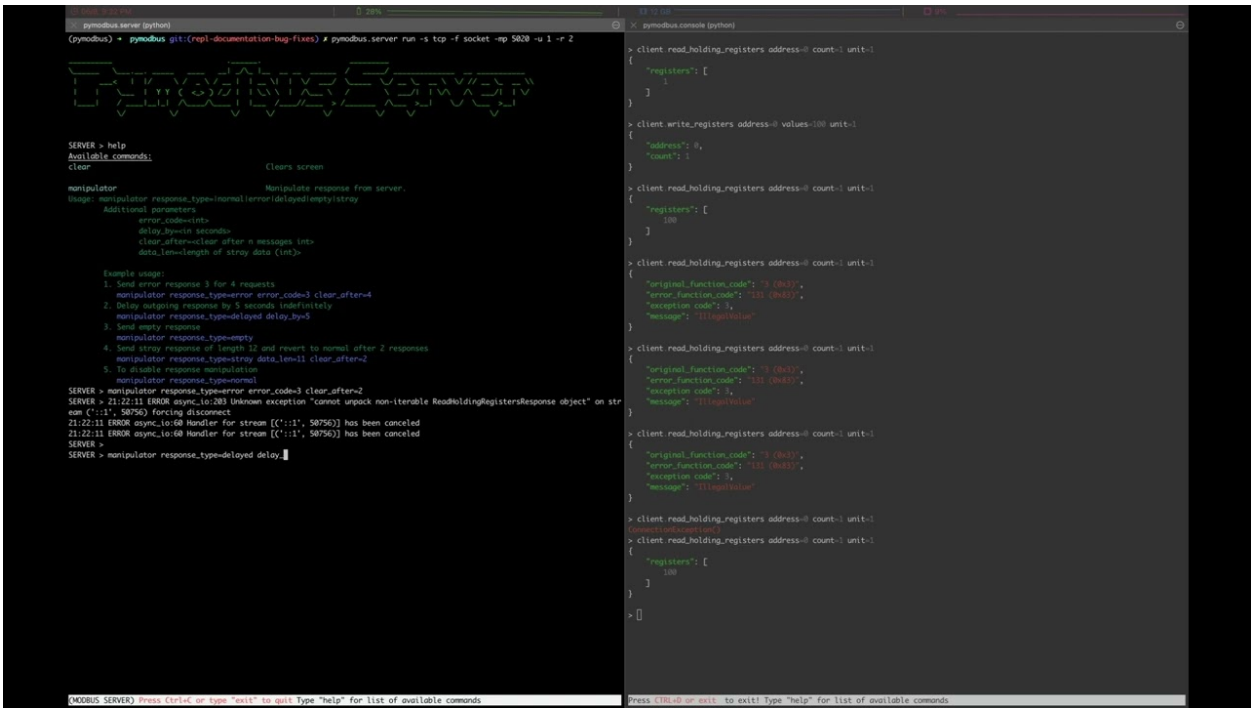
Pymodbus REPL comes with many handy features such as payload decoder to directly retrieve the values in desired format and supports all the diagnostic function codes directly .

For more info on REPL Client refer Pymodbus REPL Client

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



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/riptideio/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 .

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.

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

1.7 Repository structure

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.

1.8 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

tox -e py38 (or py39, py310, pypy38) Run pytest on source code

tox -e pylint Run pylint on source code

tox -e codespell Run codespell on source code

tox -e bandit Run bandit on source code

tox -e flake8 Run flake8 on source code

tox -e black Run black on source code

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/riptideio/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 3.0.2

- 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

2.2 version 3.0.1

- Faulty release!

2.3 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)

2.4 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)

2.5 version 3.0.0dev4

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

2.6 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

2.7 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)

2.8 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

2.9 version 3.0.0dev0

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

2.10 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

2.11 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.

2.12 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

2.13 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

2.14 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.

2.15 version 2.5.0rc2

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

2.16 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

2.17 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.

2.18 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

2.19 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.

2.20 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]
```

2.21 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.

2.22 Version 2.0.1

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

2.23 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.

2.24 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

2.25 Version 1.5.2

- Fix serial client *is_socket_open* method

2.26 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

2.27 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

2.28 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

2.29 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

2.30 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

2.31 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

2.32 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

2.33 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

2.34 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

2.35 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)

2.36 Version 0.9.0

Please view the git commit log

CHAPTER

THREE

PYMODBUS REPL

3.1 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
```

3.2 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 unit=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)

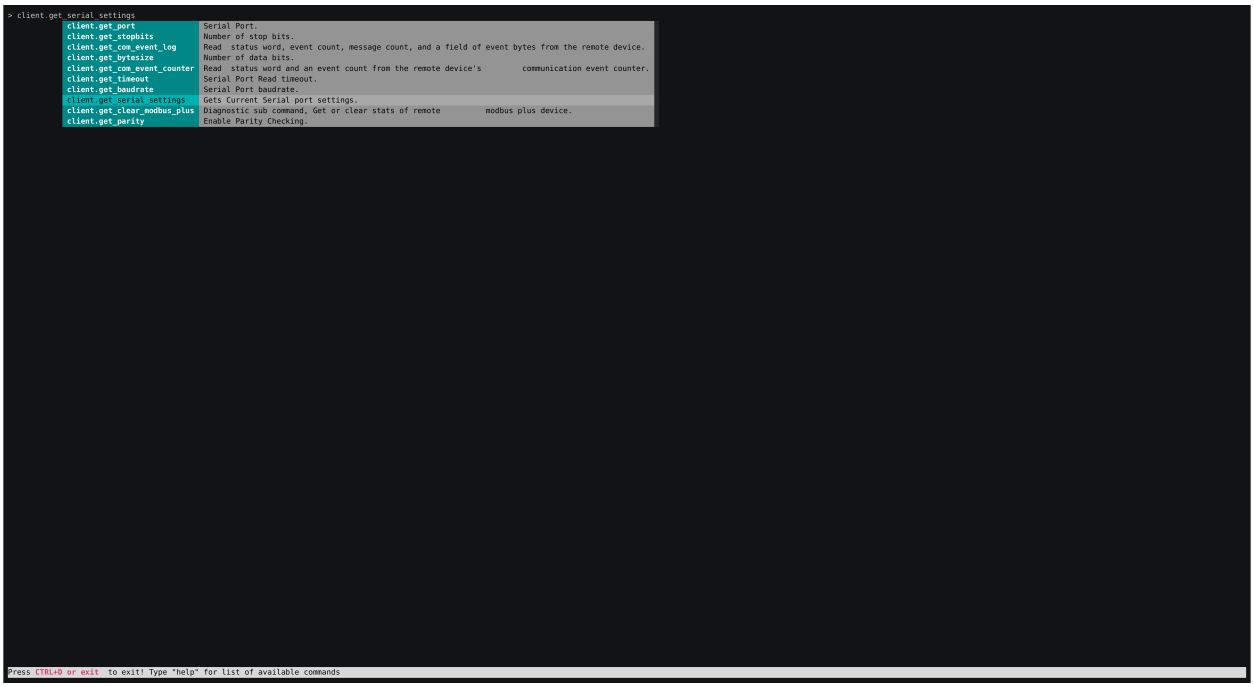
(continued from previous page)

```
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
}
```

3.3 DEMO


> client.get_serial_settings
client.get_port Serial Port.
client.get_stopbits Number of stop bits.
client.get_com_event_log Read status word, event count, message count, and a field of event bytes from the remote device.
client.get_bytesize Number of data bits.
client.get_com_event_counter Read status word and an event count from the remote device's communication event counter.
client.get_timeout Serial Port Read timeout.
client.get_baudrate Serial Port baudrate.
client.get_serial_settings Gets Current Serial port settings.
client.get_clear_modbus_plus Diagnostic sub command, Get or clear stats of remote modbus plus device.
client.get_parity Enable Parity Checking.
Press CTRL+D or exit to exit! Type "help" for list of available commands

CHAPTER

FOUR

EXAMPLES.

The examples are divived in 3 (4) parts:

- v2.5.3 examples

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

- v.2.5.3 tornado_twisted examples

  these examples uses the tornado/twisted frameworks which no longer are supported in pymodbus.

  These examples are only available in the repository.

- contrib examples

  these examples are supplied by users of pymodbus. The pymodbus sends thanks for making the examples available to the community.

- examples

  these examples are considered essential usage examples, and are guarenteed to work, because they are tested automatilly with each commit using CI.

4.1 Examples.

4.1.1 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}
```

(continues on next page)

(continued from previous page)

```
                        "critical", "error", "warning", "info" or "debug"
  --port PORT           the port to use

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

# --------------------------------------------------------------------------- #
# 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,
```

(continues on next page)

(continued from previous page)

```
            #    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,
            #    retry_on_empty=False,
            #    close_comm_on_error=False,
            #    strict=True,
            # Serial setup parameters
            #    baudrate=9600,
            #    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")
```

(continues on next page)

(continued from previous page)

```
    await client.connect()
    assert client.protocol
    if modbus_calls:
        await modbus_calls(client)
    await client.close()
    _logger.info("### End of Program")


if __name__ == "__main__":
    cmd_args = get_commandline(
        server=False,
        description="Run asynchronous client.",
    )
    testclient = setup_async_client(cmd_args)
    asyncio.run(run_async_client(testclient), debug=True)
```

4.1.2 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.
```

(continues on next page)

(continued from previous page)

```
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
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)
```

(continues on next page)

(continued from previous page)

```
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)
        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
```

(continues on next page)

(continued from previous page)

```
    _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
    # (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):
```

(continues on next page)

(continued from previous page)

```
    """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))
    )
    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()))
```

(continues on next page)

(continued from previous page)

```
    _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))
    )
    _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.",
```

(continues on next page)

(continued from previous page)

```
    )
    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)
```

4.1.3 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 pr 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 |
```

(continues on next page)

(continued from previous page)

```
    ++++++++++++++++++++++++++++++++++++++++
    """
    for word_endian, byte_endian in (
        (Endian.Big, Endian.Big),
        (Endian.Big, Endian.Little),
        (Endian.Little, Endian.Big),
        (Endian.Little, Endian.Little),
    ):
        print("-" * 60)
        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.
```

(continues on next page)

(continued from previous page)

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

        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))
```

4.1.4 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

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,
            #    retries=3,
```

(continues on next page)

(continued from previous page)

```
            #    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=9600,
            #    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,
            #    retries=3,
```

(continues on next page)

(continued from previous page)

```
            #    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)
```

4.1.5 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)

**WARNING** THIS EXAMPLE IS KNOWN TO HAVE PROBLEMS, a wrong solution.
```

(continues on next page)

(continued from previous page)

```
"""
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 unit=<unit_id>
    # in RemoteSlaveContext
    # For e.g to forward the requests to slave with unit address 1 use
    # store = RemoteSlaveContext(client, unit=1)
    if args.slaves:
        store = {}
        for i in args.slaves:
            store[i.to_bytes(1, "big")] = RemoteSlaveContext(args.client, unit=i)
    else:
        store = RemoteSlaveContext(args.client, unit=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.",
        extras=[
```

(continues on next page)

(continued from previous page)

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

4.1.6 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
  --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.datastore import (
    ModbusSequentialDataBlock,
    ModbusServerContext,
```

(continues on next page)

(continued from previous page)

```
    ModbusSlaveContext,
    ModbusSparseDataBlock,
)
from pymodbus.device import ModbusDeviceIdentification

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


_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 unit ids.
            # By default it will return the same context for every unit id supplied
            # (broadcast mode).
            # However, this can be overloaded by setting the single flag to False and
            # then supplying a dictionary of unit 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).
            # The default is False which is based on section 4.4 of the
            # specification, so address(0-7) will map to (1-8)::
```

(continues on next page)

(continued from previous page)

```
            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/riptideio/pymodbus/",
            "ProductName": "Pymodbus Server",
            "ModelName": "Pymodbus Server",
            "MajorMinorRevision": version.short(),
        }
    )
    return args


async def run_async_server(args):
    """Run server."""
    txt = f"### start ASYNC server, listening on {args.port} - {args.comm}"
```

(continues on next page)

(continued from previous page)

```
    _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 unit_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
            # TBD allow_reuse_address=True,  # allow the reuse of an address
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat unit_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=9600,  # 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
```

(continues on next page)

(continued from previous page)

```
            # broadcast_enable=False,  # treat unit_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 unit_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)
```

4.1.7 Modbus Payload Server Example

```
#!/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))
```

4.1.8 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
  --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
```

(continues on next page)

(continued from previous page)

```
**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 unit_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
            identity=args.identity,  # server identify
```

(continues on next page)

(continued from previous page)

```
            # 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
            # TBD allow_reuse_address=True,  # allow the reuse of an address
            # ignore_missing_slaves=True,  # ignore request to a missing slave
            # broadcast_enable=False,  # treat unit_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=9600,  # 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 unit_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 unit_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()
```

4.1.9 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")
    register = 3
    slave_id = 0x00
    address = 0x10
    values = context[slave_id].getValues(register, 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(register, 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)
```

4.2 Examples version 2.5.3

4.2.1 Asynchronous Asyncio Modbus TLS Client Example

```
#!/usr/bin/env python3
"""Simple Asynchronous Modbus TCP over TLS client.

This is a simple example of writing a asynchronous modbus TCP over TLS client
that uses Python builtin module ssl - TLS/SSL wrapper for socket objects for
the TLS feature and asyncio.
"""
import asyncio

# --------------------------------------------------------------------------- #
# import necessary libraries
# --------------------------------------------------------------------------- #
import ssl

from pymodbus.client import AsyncModbusTlsClient


# --------------------------------------------------------------------------- #
# the TLS detail security can be set in SSLContext which is the context here
# --------------------------------------------------------------------------- #
sslctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
sslctx.verify_mode = ssl.CERT_REQUIRED
sslctx.check_hostname = True

# Prepare client"s certificate which the server requires for TLS full handshake
# sslctx.load_cert_chain(certfile="client.crt", keyfile="client.key",
```

(continues on next page)

(continued from previous page)

```
#                           password="pwd")


async def start_async_test(client):
    """Start async test."""
    result = await client.read_coils(1, 8)
    print(result.bits)
    await client.write_coils(1, [False] * 3)
    result = await client.read_coils(1, 8)
    print(result.bits)


if __name__ == "__main__":
    # ------------------------------------------------------------------------ #
    # pass SSLContext which is the context here to ModbusTcpClient()
    # ------------------------------------------------------------------------ #
    new_client = AsyncModbusTlsClient(  # pylint: disable=unpacking-non-sequence
        "test.host.com",
        8020,
        sslctx=sslctx,
    )
    loop = asyncio.get_running_loop()
    loop.run_until_complete(start_async_test(new_client.protocol))
    loop.close()
```

4.2.2 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.interfaces import IPayloadBuilder
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(IPayloadBuilder):
    """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:
            nibble = value & 0xF
```

(continues on next page)

(continued from previous page)

```
            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")

    @staticmethod
```

(continues on next page)

(continued from previous page)

```
    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"]
```

4.2.3 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

from pymodbus.datastore import (
    ModbusServerContext,
    ModbusSlaveContext,
    ModbusSparseDataBlock,
)
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.server import StartTcpServer

# --------------------------------------------------------------------------- #
# import the modbus libraries we need
# --------------------------------------------------------------------------- #
from pymodbus.version import version


# 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
```

(continues on next page)

(continued from previous page)

```
        values = {k: 0 for k in devices.keys()}
        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/riptideio/pymodbus/",
            "ProductName": "pymodbus Server",
            "ModelName": "pymodbus Server",
            "MajorMinorRevision": version.short(),
        }
    )

    # ----------------------------------------------------------------------- #
    # 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()
```

4.2.4 Changing Framers Example

```
#!/usr/bin/env python3
"""Pymodbus Client Framer Overload.

All of the modbus clients are designed to have pluggable framers
so that the transport and protocol are decoupled. This allows a user
to define or plug in their custom protocols into existing transports
(like a binary framer over a serial connection).

It should be noted that although you are not limited to trying whatever
you would like, the library makes no guarantees that all framers with
all transports will produce predictable or correct results (for example
tcp transport with an RTU framer). However, please let us know of any
success cases that are not documented!
"""
import logging

# --------------------------------------------------------------------------- #
# import the modbus client and the framers
# --------------------------------------------------------------------------- #
from pymodbus.client import ModbusTcpClient as ModbusClient
from pymodbus.transaction import ModbusSocketFramer as ModbusFramer


# from pymodbus.transaction import ModbusRtuFramer as ModbusFramer
# from pymodbus.transaction import ModbusBinaryFramer as ModbusFramer
# from pymodbus.transaction import ModbusAsciiFramer as ModbusFramer

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

if __name__ == "__main__":
    # ------------------------------------------------------------------------ #
    # Initialize the client
    # ------------------------------------------------------------------------ #
    client = ModbusClient("localhost", port=5020, framer=ModbusFramer)
    client.connect()

    # ------------------------------------------------------------------------ #
    # perform your requests
    # ------------------------------------------------------------------------ #
    rq = client.write_coil(1, True)
    rr = client.read_coils(1, 1)
    assert not rq.isError()  # nosec test that we are not an error
    assert rr.bits[0]  # nosec test the expected value

    # ------------------------------------------------------------------------ #
    # close the client
    # ------------------------------------------------------------------------ #
```

(continues on next page)

(continued from previous page)

```
    client.close()
```

4.2.5 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).
```

(continues on next page)

(continued from previous page)

```
    """

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

(continues on next page)

(continued from previous page)

```
        try:
            workitem = input_queue.get(timeout=1)
            txt = f"dequeue worker request: {workitem}"
            log.debug(txt)
            if not workitem:
                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:  # nosec 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)
```

(continues on next page)

(continued from previous page)

```
            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)


# --------------------------------------------------------------------------- #
# 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)
```

(continues on next page)

(continued from previous page)

```
    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
        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")
```

(continues on next page)

(continued from previous page)

```
        for future in futures:
            txt = f"future result: {future.result(timeout=1)}"
            log.info(txt)
    finally:
        client.shutdown()
```

4.2.6 Custom Datablock Example

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

This is an example of performing custom logic after a value has been
written to the datastore.
"""
import logging

from pymodbus.datastore import (
    ModbusServerContext,
    ModbusSlaveContext,
    ModbusSparseDataBlock,
)
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.server import StartTcpServer

# --------------------------------------------------------------------------- #
# import the modbus libraries we need
# --------------------------------------------------------------------------- #
from pymodbus.version import version


# from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer

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

# --------------------------------------------------------------------------- #
# create your custom data block here
# --------------------------------------------------------------------------- #


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

    and performs a custom action after it has been stored.
    """
```

(continues on next page)

(continued from previous page)

```
    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)

        # whatever you want to do with the written value is done here,
        # however make sure not to do too much work here or it will
        # block the server, espectially if the server is being written
        # to very quickly
        print(f"wrote {value} to {address}")


def run_custom_db_server():
    """Run custom db server."""
    # ----------------------------------------------------------------------- #
    # initialize your data store
    # ----------------------------------------------------------------------- #
    block = CustomDataBlock([0] * 100)
    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/riptideio/pymodbus/",
            "ProductName": "pymodbus Server",
            "ModelName": "pymodbus Server",
            "MajorMinorRevision": version.short(),
        }
    )

    # ----------------------------------------------------------------------- #
    # run the server you want
    # ----------------------------------------------------------------------- #

    # p = Process(target=device_writer, args=(queue,))
    # p.start()
    StartTcpServer(context, identity=identity, address=("localhost", 5020))


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

4.2.7 Custom Message Example

```
#!/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, **kwargs):
        """Initialize."""
        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)

```
# 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, unit=1)
        result = client.execute(request)
        print(result.values)
```

4.2.8 Dbstore update Server Example

```
# pylint: disable=differing-param-doc,missing-any-param-doc
"""Pymodbus Server With Updating Thread.

This is an example of having a background thread updating the
context in an SQLite4 database while the server is operating.

This scrit generates a random address range (within 0 - 65000) and a random
value and stores it in a database. It then reads the same address to verify
that the process works as expected

This can also be done with a python thread::
    from threading import Thread
    thread = Thread(target=updating_writer, args=(context,))
    thread.start()
"""
import asyncio
import logging
import random
```

(continues on next page)

(continued from previous page)

```
from pymodbus.datastore import ModbusSequentialDataBlock, ModbusServerContext
from pymodbus.datastore.database import SqlSlaveContext
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.server import StartAsyncTcpServer

# --------------------------------------------------------------------------- #
# import the modbus libraries we need
# --------------------------------------------------------------------------- #
from pymodbus.version import version


# from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer

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

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


def updating_writer(parm1):
    """Run every so often,

    and updates live values of the context which resides in an SQLite3 database.
    It should be noted that there is a race condition for the update.
    :param arguments: The input arguments to the call
    """
    log.debug("Updating the database context")
    context = parm1[0]
    readfunction = 0x03  # read holding registers
    writefunction = 0x10
    slave_id = 0x01  # slave address
    count = 50

    # import pdb; pdb.set_trace()

    rand_value = random.randint(0, 9999)  # nosec
    rand_addr = random.randint(0, 65000)  # nosec
    txt = f"Writing to datastore: {rand_addr}, {rand_value}"
    log.debug(txt)
    # import pdb; pdb.set_trace()
    context[slave_id].setValues(writefunction, rand_addr, [rand_value], update=False)
    values = context[slave_id].getValues(readfunction, rand_addr, count)
    txt = f"Values from datastore: {values}"
    log.debug(txt)
```

(continues on next page)

(continued from previous page)

```
async def run_dbstore_update_server():
    """Run dbstore update server."""
    # ----------------------------------------------------------------------- #
    # initialize your data store
    # ----------------------------------------------------------------------- #

    block = ModbusSequentialDataBlock(0x00, [0] * 0xFF)
    store = SqlSlaveContext(block)

    context = ModbusServerContext(slaves={1: store}, single=False)

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

    # ----------------------------------------------------------------------- #
    # run the server you want
    # ----------------------------------------------------------------------- #
    time = 5  # 5 seconds delay
    loop = asyncio.get_event_loop()
    loop.start(time, now=False)  # initially delay by time
    loop.stop()
    await StartAsyncTcpServer(context, identity=identity, address=("", 5020))


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

4.2.9 Deviceinfo showcase client Example

```
#!/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

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

(continues on next page)

(continued from previous page)

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

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

(continues on next page)

(continued from previous page)

```
    # * 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)
    # 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)
```

(continues on next page)

(continued from previous page)

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


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

4.2.10 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

# ------------------------------------------------------------------------ #
# import the various server implementations
# ------------------------------------------------------------------------ #
from pymodbus.version import version


# ------------------------------------------------------------------------ #
# 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)
```

(continues on next page)

(continued from previous page)

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

    # ----------------------------------------------------------------------- #
    # 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 unit. 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.
```

(continues on next page)

(continued from previous page)

```
    # ------------------------------------------------------------------------ #
    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))

    # 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()
```

4.2.11 Libmodbus Client Example

```
#!/usr/bin/env python3
# pylint: disable=missing-type-doc,missing-param-doc,differing-param-doc,missing-raises-
↪doc
"""Libmodbus Protocol Wrapper.

What follows is an example wrapper of the libmodbus library
(https://libmodbus.org/documentation/) for use with pymodbus.
There are two utilities involved here:

* LibmodbusLevel1Client

  This is simply a python wrapper around the c library. It is
  mostly a clone of the pylibmodbus implementation, but I plan
```

(continues on next page)

(continued from previous page)

```
  on extending it to implement all the available protocol using
  the raw execute methods.

* LibmodbusClient

  This is just another modbus client that can be used just like
  any other client in pymodbus.

For these to work, you must have `cffi` and `libmodbus-dev` installed:

    sudo apt-get install libmodbus-dev
    pip install cffi
"""
# --------------------------------------------------------------------------- #
# import system libraries
# --------------------------------------------------------------------------- #
from cffi import FFI  # pylint: disable=import-error

from pymodbus.bit_read_message import (
    ReadCoilsResponse,
    ReadDiscreteInputsResponse,
)
from pymodbus.bit_write_message import (
    WriteMultipleCoilsResponse,
    WriteSingleCoilResponse,
)
from pymodbus.client.mixin import ModbusClientMixin
from pymodbus.constants import Defaults
from pymodbus.exceptions import ModbusException
from pymodbus.register_read_message import (
    ReadHoldingRegistersResponse,
    ReadInputRegistersResponse,
    ReadWriteMultipleRegistersResponse,
)
from pymodbus.register_write_message import (
    WriteMultipleRegistersResponse,
    WriteSingleRegisterResponse,
)


# --------------------------------------------------------------------------- #
# import pymodbus libraries
# --------------------------------------------------------------------------- #


# --------------------------------------------------------------------------- #
# create the C interface
# --------------------------------------------------------------------------- #
# * TODO add the protocol needed for the servers
# --------------------------------------------------------------------------- #

compiler = FFI()
```

(continues on next page)

(continued from previous page)

```
compiler.cdef(
    """
    typedef struct _modbus modbus_t;

    int modbus_connect(modbus_t *ctx);
    int modbus_flush(modbus_t *ctx);
    void modbus_close(modbus_t *ctx);

    const char *modbus_strerror(int errnum);
    int modbus_set_slave(modbus_t *ctx, int slave);

    void modbus_get_response_timeout(modbus_t *ctx, uint32_t *to_sec, uint32_t *to_usec);
    void modbus_set_response_timeout(modbus_t *ctx, uint32_t to_sec, uint32_t to_usec);

    int modbus_read_bits(modbus_t *ctx, int addr, int nb, uint8_t *dest);
    int modbus_read_input_bits(modbus_t *ctx, int addr, int nb, uint8_t *dest);
    int modbus_read_registers(modbus_t *ctx, int addr, int nb, uint16_t *dest);
    int modbus_read_input_registers(modbus_t *ctx, int addr, int nb, uint16_t *dest);

    int modbus_write_bit(modbus_t *ctx, int coil_addr, int status);
    int modbus_write_bits(modbus_t *ctx, int addr, int nb, const uint8_t *data);
    int modbus_write_register(modbus_t *ctx, int reg_addr, int value);
    int modbus_write_registers(modbus_t *ctx, int addr, int nb, const uint16_t *data);
    int modbus_write_and_read_registers(modbus_t *ctx, int write_addr, int write_nb,
                                        const uint16_t *src, int read_addr, int read_nb,␣
↪uint16_t *dest);

    int modbus_mask_write_register(modbus_t *ctx, int addr, uint16_t and_mask, uint16_t␣
↪or_mask);
    int modbus_send_raw_request(modbus_t *ctx, uint8_t *raw_req, int raw_req_length);

    float modbus_get_float(const uint16_t *src);
    void modbus_set_float(float f, uint16_t *dest);

    modbus_t* modbus_new_tcp(const char *ip_address, int port);
    modbus_t* modbus_new_rtu(const char *device, int baud, char parity, int data_bit,␣
↪int stop_bit);
    void modbus_free(modbus_t *ctx);

    int modbus_receive(modbus_t *ctx, uint8_t *req);
    int modbus_receive_from(modbus_t *ctx, int sockfd, uint8_t *req);
    int modbus_receive_confirmation(modbus_t *ctx, uint8_t *rsp);
"""
)
LIB = compiler.dlopen("modbus")  # create our bindings

# ------------------------------------------------------------------------ #
# helper utilities
# ------------------------------------------------------------------------ #


def get_float(data):
```

(continues on next page)

(continued from previous page)

```
    """Get float."""
    return LIB.modbus_get_float(data)


def set_float(value, data):
    """Set float."""
    LIB.modbus_set_float(value, data)


def cast_to_int16(data):
    """Cast to int16."""
    return int(compiler.cast("int16_t", data))


def cast_to_int32(data):
    """Cast to int32."""
    return int(compiler.cast("int32_t", data))


class NotImplementedException(Exception):
    """Not implemented exception."""


# --------------------------------------------------------------------------- #
# level1 client
# --------------------------------------------------------------------------- #


class LibmodbusLevel1Client:
    """A raw wrapper around the libmodbus c library.

    Feel free to use it if you want increased performance and don"t mind the
    entire protocol not being implemented.
    """

    @classmethod
    def create_tcp_client(cls, my_host="127.0.0.1", my_port=Defaults.TcpPort):
        """Create a TCP modbus client for the supplied parameters.

        :param host: The host to connect to
        :param port: The port to connect to on that host
        :returns: A new level1 client
        """
        my_client = LIB.modbus_new_tcp(my_host.encode(), my_port)
        return cls(my_client)

    @classmethod
    def create_rtu_client(cls, **kwargs):
        """Create a TCP modbus client for the supplied parameters.

        :param port: The serial port to attach to
        :param stopbits: The number of stop bits to use
```

(continues on next page)

(continued from previous page)

```
        :param bytesize: The bytesize of the serial messages
        :param parity: Which kind of parity to use
        :param baudrate: The baud rate to use for the serial device
        :returns: A new level1 client
        """
        my_port = kwargs.get("port", "/dev/ttyS0")
        baudrate = kwargs.get("baud", Defaults.Baudrate)
        parity = kwargs.get("parity", Defaults.Parity)
        bytesize = kwargs.get("bytesize", Defaults.Bytesize)
        stopbits = kwargs.get("stopbits", Defaults.Stopbits)
        my_client = LIB.modbus_new_rtu(my_port, baudrate, parity, bytesize, stopbits)
        return cls(my_client)

    def __init__(self, my_client):
        """Initialize a new instance of the LibmodbusLevel1Client.

        This method should not be used, instead new instances should be created
        using the two supplied factory methods:

        * LibmodbusLevel1Client.create_rtu_client(...)
        * LibmodbusLevel1Client.create_tcp_client(...)

        :param client: The underlying client instance to operate with.
        """
        self.client = my_client
        self.slave = Defaults.Slave

    def set_slave(self, slave):
        """Set the current slave to operate against.

        :param slave: The new slave to operate against
        :returns: The resulting slave to operate against
        """
        self.slave = self._execute(  # pylint: disable=no-member
            LIB.modbus_set_slave, slave
        )
        return self.slave

    def connect(self):
        """Attempt to connect to the client target.

        :returns: True if successful, throws otherwise
        """
        return not self.__execute(LIB.modbus_connect)

    def flush(self):
        """Discard the existing bytes on the wire.

        :returns: The number of flushed bytes, or throws
        """
        return self.__execute(LIB.modbus_flush)
```

(continues on next page)

(continued from previous page)

```
    def close(self):
        """Close and frees the underlying connection and context structure.

        :returns: Always True
        """
        LIB.modbus_close(self.client)
        LIB.modbus_free(self.client)
        return True

    def __execute(self, command, *args):
        """Run the supplied command against the currently instantiated client with the␣
↪supplied arguments.

        This will make sure to correctly handle resulting errors.

        :param command: The command to execute against the context
        :param *args: The arguments for the given command
        :returns: The result of the operation unless -1 which throws
        """
        if (result := command(self.client, *args)) == -1:
            message = LIB.modbus_strerror(compiler.errno)
            raise ModbusException(compiler.string(message))
        return result

    def read_bits(self, address, count=1):
        """Read bits.

        :param address: The starting address to read from
        :param count: The number of coils to read
        :returns: The resulting bits
        """
        result = compiler.new("uint8_t[]", count)
        self.__execute(LIB.modbus_read_bits, address, count, result)
        return result

    def read_input_bits(self, address, count=1):
        """Read input bits.

        :param address: The starting address to read from
        :param count: The number of discretes to read
        :returns: The resulting bits
        """
        result = compiler.new("uint8_t[]", count)
        self.__execute(LIB.modbus_read_input_bits, address, count, result)
        return result

    def write_bit(self, address, value):
        """Write bit.

        :param address: The starting address to write to
        :param value: The value to write to the specified address
        :returns: The number of written bits
```

(continues on next page)

(continued from previous page)

```
        """
        return self.__execute(LIB.modbus_write_bit, address, value)

    def write_bits(self, address, values):
        """Write bits.

        :param address: The starting address to write to
        :param values: The values to write to the specified address
        :returns: The number of written bits
        """
        count = len(values)
        return self.__execute(LIB.modbus_write_bits, address, count, values)

    def write_register(self, address, value):
        """Write register.

        :param address: The starting address to write to
        :param value: The value to write to the specified address
        :returns: The number of written registers
        """
        return self.__execute(LIB.modbus_write_register, address, value)

    def write_registers(self, address, values):
        """Write registers.

        :param address: The starting address to write to
        :param values: The values to write to the specified address
        :returns: The number of written registers
        """
        count = len(values)
        return self.__execute(LIB.modbus_write_registers, address, count, values)

    def read_registers(self, address, count=1):
        """Read registers.

        :param address: The starting address to read from
        :param count: The number of registers to read
        :returns: The resulting read registers
        """
        result = compiler.new("uint16_t[]", count)
        self.__execute(LIB.modbus_read_registers, address, count, result)
        return result

    def read_input_registers(self, address, count=1):
        """Read input registers.

        :param address: The starting address to read from
        :param count: The number of registers to read
        :returns: The resulting read registers
        """
        result = compiler.new("uint16_t[]", count)
        self.__execute(LIB.modbus_read_input_registers, address, count, result)
```

(continues on next page)

(continued from previous page)

```
        return result

    def read_and_write_registers(
        self, read_address, read_count, write_address, write_registers
    ):
        """Read/write registers.

        :param read_address: The address to start reading from
        :param read_count: The number of registers to read from address
        :param write_address: The address to start writing to
        :param write_registers: The registers to write to the specified address
        :returns: The resulting read registers
        """
        write_count = len(write_registers)
        read_result = compiler.new("uint16_t[]", read_count)
        self.__execute(
            LIB.modbus_write_and_read_registers,
            write_address,
            write_count,
            write_registers,
            read_address,
            read_count,
            read_result,
        )
        return read_result


# --------------------------------------------------------------------------- #
# level2 client
# --------------------------------------------------------------------------- #


class LibmodbusClient(ModbusClientMixin):
    """A facade around the raw level 1 libmodbus client.

    that implements the pymodbus protocol on top of the lower level client.
    """

    # ----------------------------------------------------------------------- #
    # these are used to convert from the pymodbus request types to the
    # libmodbus operations (overloaded operator).
    # ----------------------------------------------------------------------- #

    __methods = {
        "ReadCoilsRequest": lambda c, r: c.read_bits(r.address, r.count),
        "ReadDiscreteInputsRequest": lambda c, r: c.read_input_bits(r.address, r.count),
        "WriteSingleCoilRequest": lambda c, r: c.write_bit(r.address, r.value),
        "WriteMultipleCoilsRequest": lambda c, r: c.write_bits(r.address, r.values),
        "WriteSingleRegisterRequest": lambda c, r: c.write_register(r.address, r.value),
        "WriteMultipleRegistersRequest": lambda c, r: c.write_registers(
            r.address, r.values
        ),
```

(continues on next page)

(continued from previous page)

```
        "ReadHoldingRegistersRequest": lambda c, r: c.read_registers(
            r.address, r.count
        ),
        "ReadInputRegistersRequest": lambda c, r: c.read_input_registers(
            r.address, r.count
        ),
        "ReadWriteMultipleRegistersRequest": lambda c, r: c.read_and_write_registers(
            r.read_address, r.read_count, r.write_address, r.write_registers
        ),
    }

    # ------------------------------------------------------------------------ #
    # these are used to convert from the libmodbus result to the
    # pymodbus response type
    # ------------------------------------------------------------------------ #

    __adapters = {
        "ReadCoilsRequest": lambda tx, rx: ReadCoilsResponse(list(rx)),
        "ReadDiscreteInputsRequest": lambda tx, rx: ReadDiscreteInputsResponse(
            list(rx)
        ),
        "WriteSingleCoilRequest": lambda tx, rx: WriteSingleCoilResponse(
            tx.address, rx
        ),
        "WriteMultipleCoilsRequest": lambda tx, rx: WriteMultipleCoilsResponse(
            tx.address, rx
        ),
        "WriteSingleRegisterRequest": lambda tx, rx: WriteSingleRegisterResponse(
            tx.address, rx
        ),
        "WriteMultipleRegistersRequest": lambda tx, rx: WriteMultipleRegistersResponse(
            tx.address, rx
        ),
        "ReadHoldingRegistersRequest": lambda tx, rx: ReadHoldingRegistersResponse(
            list(rx)
        ),
        "ReadInputRegistersRequest": lambda tx, rx: ReadInputRegistersResponse(
            list(rx)
        ),
        "ReadWriteMultipleRegistersRequest": lambda tx, rx:␣
↪ReadWriteMultipleRegistersResponse(
            list(rx)
        ),
    }

    def __init__(self, my_client):
        """Initialize a new instance of the LibmodbusClient.

        This should be initialized with one of the LibmodbusLevel1Client instances:

        * LibmodbusLevel1Client.create_rtu_client(...)
        * LibmodbusLevel1Client.create_tcp_client(...)
```

(continues on next page)

(continued from previous page)

```
        :param client: The underlying client instance to operate with.
        """
        self.client = my_client

    # ---------------------------------------------------------------------- #
    # We use the client mixin to implement the api methods which are all
    # forwarded to this method. It is implemented using the previously
    # defined lookup tables. Any method not defined simply throws.
    # ---------------------------------------------------------------------- #

    def execute(self, request):
        """Execute the supplied request against the server.

        :param request: The request to process
        :returns: The result of the request execution
        """
        if self.client.slave != request.unit_id:
            self.client.set_slave(request.unit_id)

        method = request.__class__.__name__
        operation = self.__methods.get(method, None)
        adapter = self.__adapters.get(method, None)

        if not operation or not adapter:
            raise NotImplementedException("Method not implemented: " + operation)

        response = operation(self.client, request)
        return adapter(request, response)

    # ---------------------------------------------------------------------- #
    # Other methods can simply be forwarded using the decorator pattern
    # ---------------------------------------------------------------------- #

    def connect(self):
        """Connect."""
        return self.client.connect()

    def close(self):
        """Close."""
        return self.client.close()

    # ---------------------------------------------------------------------- #
    # magic methods
    # ---------------------------------------------------------------------- #

    def __enter__(self):
        """Implement the client with enter block

        :returns: The current instance of the client
        """
        self.client.connect()
        return self
```

(continues on next page)

(continued from previous page)

```
    def __exit__(self, klass, value, traceback):
        """Implement the client with exit block"""
        self.client.close()


# --------------------------------------------------------------------------- #
# main example runner
# --------------------------------------------------------------------------- #


if __name__ == "__main__":

    # create our low level client
    host = "127.0.0.1"  # pylint: disable=invalid-name
    port = 502  # pylint: disable=invalid-name
    protocol = LibmodbusLevel1Client.create_tcp_client(host, port)

    # operate with our high level client
    with LibmodbusClient(protocol) as client:
        registers = client.write_registers(0, [13, 12, 11])
        print(registers)
        registers = client.read_holding_registers(0, 10)
        print(registers.registers)
```

4.2.12 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,
)
```

(continues on next page)

(continued from previous page)

```
from pymodbus.bit_write_message import (
    WriteMultipleCoilsRequest,
    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,
```

(continues on next page)

(continued from previous page)

```
    ModbusSocketFramer,
)


# --------------------------------------------------------------------------- #
# 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,
]
```

(continues on next page)

(continued from previous page)

```
# ---------------------------------------------------------------------------- #
# enumerate all response messages
# ---------------------------------------------------------------------------- #
_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,
```

(continues on next page)

(continued from previous page)

```
    "count": 0x08,
    "value": 0x01,
    "values": [0x01] * 8,
    "read_address": 0x12,
    "read_count": 0x08,
    "write_address": 0x12,
    "write_registers": [0x01] * 8,
    "transaction": 0x01,
    "protocol": 0x00,
    "unit": 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)",
```

(continues on next page)

(continued from previous page)

```
        dest="framer",
        default="tcp",
    )

    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")
```

(continues on next page)

(continued from previous page)

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

    generate_messages(framer, option)


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

4.2.13 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 = (
```

(continues on next page)

(continued from previous page)

```
    "%(asctime)-15s %(threadName)-15s"
    " %(levelname)-8s %(module)-15s:%(lineno)-8s %(message)s"
)
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():
                    unit = decoder._header.get(  # pylint: disable=protected-access
                        "uid", 0x00
                    )
                    decoder.advanceFrame()
                    decoder.processIncomingPacket(message, self.report, unit)
                else:
                    self.check_errors(decoder, message)
            except Exception:  # pylint: disable=broad-except
                self.check_errors(decoder, message)
```

(continues on next page)

(continued from previous page)

```
    def check_errors(self, decoder, message):
        """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__,
            )
        )
```

(continues on next page)

(continued from previous page)

```
# --------------------------------------------------------------------------- #
# and decode our message
# --------------------------------------------------------------------------- #
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",
```

(continues on next page)

(continued from previous page)

```
        dest="file",
        default=None,
    )

    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()
```

(continues on next page)

(continued from previous page)

```
                    line = line.decode("hex")
                yield line


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()
```

4.2.14 Modbus Logging Example

```
#!/usr/bin/env python3
"""Pymodbus Logging Examples."""
import logging
import logging.handlers as Handlers


if __name__ == "__main__":
    # ------------------------------------------------------------------------ #
    # This will simply send everything logged to console
    # ------------------------------------------------------------------------ #
    log = logging.getLogger()
    log.setLevel(logging.DEBUG)

    # ------------------------------------------------------------------------ #
    # This will send the error messages in the specified namespace to a file.
    # The available namespaces in pymodbus are as follows:
    # ------------------------------------------------------------------------ #
    # * pymodbus.*          - The root namespace
    # * pymodbus.server.*   - all logging messages involving the modbus server
```

(continues on next page)

(continued from previous page)

```
    # * pymodbus.client.*   - all logging messages involving the client
    # * pymodbus.protocol.* - all logging messages inside the protocol layer
    # ------------------------------------------------------------------------- #
    log = logging.getLogger("pymodbus.server")
    log.setLevel(logging.ERROR)

    # ------------------------------------------------------------------------- #
    # This will send the error messages to the specified handlers:
    # * docs.python.org/library/logging.html
    # ------------------------------------------------------------------------- #
    log = logging.getLogger("pymodbus")
    log.setLevel(logging.ERROR)
    handlers = [
        Handlers.RotatingFileHandler("logfile", maxBytes=1024 * 1024),
        Handlers.SMTPHandler(
            "mx.host.com", "pymodbus@host.com", ["support@host.com"], "Pymodbus"
        ),
        Handlers.SysLogHandler(facility="daemon"),
        Handlers.DatagramHandler("localhost", 12345),
    ]
    [log.addHandler(h) for h in handlers]  # pylint: disable=expression-not-assigned
```

4.2.15 Modbus Mapper Example

```
# 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
    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
```

(continues on next page)

(continued from previous page)

```
    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::

    CSV:
    address,value,function,name,description
    0,0,hr,Comm. count PLC,Comm. count PLC
    1,10,hr,Comm. count PLC,Comm. count PLC
```

(continues on next page)

(continued from previous page)

```
    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/riptideio/pymodbus/",
            "ProductName": "Pymodbus Server",
            "ModelName": "Pymodbus Server",
            "MajorMinorRevision": version.short(),
        }
    )
    StartTcpServer(context=context, identity=identity, address=("localhost", 5020))

"""
import csv
import json
from collections import defaultdict
from io import StringIO
```

(continues on next page)

(continued from previous page)

```
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
    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.
```

(continues on next page)

(continued from previous page)

```
    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).

    :param mapping_blocks: The mapping blocks
    :returns: The initialized modbus slave context
    """
    sparse = ModbusSparseDataBlock()
```

(continues on next page)

(continued from previous page)

```
    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,
            "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,
```

(continues on next page)

(continued from previous page)

```
            "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():
        """Parse value."""
        return lambda d: d.decode_16bit_int()

    @staticmethod
    def parse_32bit_int():
```

(continues on next page)

(continued from previous page)

```
        """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.

    :param mapping_blocks: The mapping blocks
    :param decoder: The type decoder to use
    """
    decoder = decoder or ModbusTypeDecoder()
```

(continues on next page)

(continued from previous page)

```
    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
```

4.2.16 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  # nosec


class ModbusDatastoreSaver:
    """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
```

(continues on next page)

(continued from previous page)

```
    (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."""

    def handle_save_end(self):
        """Handle save end."""
```

(continues on next page)

(continued from previous page)

```
# --------------------------------------------------------------------------- #
# 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.
    """

    _context = None
    _store = None
    _line = None
    NEWLINE = "\r\n"
    HEADER = "slave,store,address,value" + NEWLINE
```

(continues on next page)

(continued from previous page)

```
    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",
    }

    def handle_save_start(self):
        """Handle save start."""
        self._context = xml.Element("context")
```

(continues on next page)

(continued from previous page)

```
        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)
```

4.2.17 Modbus Simulator Example

```
#!/usr/bin/env python3
# pylint: disable=missing-raises-doc
"""An example of creating a fully implemented modbus server.

with read/write data as well as user configurable base data
"""
import logging
import pickle  # nosec
from optparse import OptionParser  # pylint: disable=deprecated-module

from pymodbus.datastore import ModbusServerContext, ModbusSlaveContext
from pymodbus.server import StartTcpServer


# --------------------------------------------------------------------------- #
# Logging
# --------------------------------------------------------------------------- #
server_log = logging.getLogger("pymodbus.server")
protocol_log = logging.getLogger("pymodbus.protocol")
_logger = logging.getLogger(__name__)
```

(continues on next page)

(continued from previous page)

```
# --------------------------------------------------------------------------- #
# Extra Global Functions
# --------------------------------------------------------------------------- #
# These are extra helper functions that don"t belong in a class
# --------------------------------------------------------------------------- #
# import getpass


def root_test():
    """Check to see if we are running as root"""
    return True  # removed for the time being as it isn"t portable
    # return getpass.getuser() == "root"


# --------------------------------------------------------------------------- #
# Helper Classes
# --------------------------------------------------------------------------- #


class ConfigurationException(Exception):
    """Exception for configuration error"""

    def __init__(self, string):
        """Initialize the ConfigurationException instance

        :param string: The message to append to the exception
        """
        Exception.__init__(self, string)
        self.string = string

    def __str__(self):
        """Build a representation of the object

        :returns: A string representation of the object
        """
        return f"Configuration Error: {self.string}"


class Configuration:  # pylint: disable=too-few-public-methods
    """Class used to parse configuration file and create and modbus datastore.

    The format of the configuration file is actually just a
    python pickle, which is a compressed memory dump from
    the scraper.
    """

    def __init__(self, config):
        """Try to load a configuration file.

        lets the file not found exception fall through

        :param config: The pickled datastore
```

(continues on next page)

(continued from previous page)

```
        """
        try:
            self.file = open(config, "rb")  # pylint: disable=consider-using-with
        except Exception as exc:
            _logger.critical(str(exc))
            raise ConfigurationException(  # pylint: disable=raise-missing-from
                f"File not found {config}"
            )

    def parse(self):
        """Parse the config file and creates a server context"""
        handle = pickle.load(self.file)  # nosec
        try:  # test for existence, or bomb
            dsd = handle["di"]
            csd = handle["ci"]
            hsd = handle["hr"]
            isd = handle["ir"]
        except Exception:
            raise ConfigurationException(  # pylint: disable=raise-missing-from
                "Invalid Configuration"
            )
        slave = ModbusSlaveContext(d=dsd, c=csd, h=hsd, i=isd)
        return ModbusServerContext(slaves=slave)


# --------------------------------------------------------------------------- #
# Main start point
# --------------------------------------------------------------------------- #


def main():
    """Server launcher"""
    parser = OptionParser()
    parser.add_option(
        "-c", "--conf", help="The configuration file to load", dest="file"
    )
    parser.add_option(
        "-D",
        "--debug",
        help="Turn on to enable tracing",
        action="store_true",
        dest="debug",
        default=False,
    )
    (opt, _) = parser.parse_args()

    # enable debugging information
    if opt.debug:
        try:
            server_log.setLevel(logging.DEBUG)
            protocol_log.setLevel(logging.DEBUG)
        except Exception:  # pylint: disable=broad-except
```

(continues on next page)

(continued from previous page)

```
            print("Logging is not supported on this system")

    # parse configuration file and run
    try:
        conf = Configuration(opt.file)
        StartTcpServer(context=conf.parse())
    except ConfigurationException as err:
        print(err)
        parser.print_help()


# --------------------------------------------------------------------------- #
# Main jumper
# --------------------------------------------------------------------------- #


if __name__ == "__main__":
    if root_test():
        main()
    else:
        print("This script must be run as root!")
```

4.2.18 Modbus Tls client Example

```
#!/usr/bin/env python3
"""Simple Modbus TCP over TLS client.

This is a simple example of writing a modbus TCP over TLS client that uses
Python builtin module ssl - TLS/SSL wrapper for socket objects for the TLS
feature.
"""
# --------------------------------------------------------------------------- #
# import necessary libraries
# --------------------------------------------------------------------------- #
import ssl

from pymodbus.client import ModbusTlsClient


# --------------------------------------------------------------------------- #
# the TLS detail security can be set in SSLContext which is the context here
# --------------------------------------------------------------------------- #
sslctx = ssl.SSLContext(ssl.PROTOCOL_TLSv1_2)
sslctx.verify_mode = ssl.CERT_REQUIRED
sslctx.check_hostname = True

# Prepare client"s certificate which the server requires for TLS full handshake
# sslctx.load_cert_chain(certfile="client.crt", keyfile="client.key",
#                        password="pwd")
```

(continues on next page)

(continued from previous page)

```
# ------------------------------------------------------------------------ #
# pass SSLContext which is the context here to ModbusTcpClient()
# ------------------------------------------------------------------------ #
client = ModbusTlsClient("test.host.com", 8020, sslctx=sslctx)
client.connect()

result = client.read_coils(1, 8)
print(result.bits)

client.close()
```

4.2.19 Modicon Payload Example

```
# pylint: disable=missing-type-doc,missing-raises-doc
"""Modbus Modicon Payload Builder.

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

from pymodbus.constants import Endian
from pymodbus.exceptions import ParameterException
from pymodbus.interfaces import IPayloadBuilder
from pymodbus.utilities import pack_bitstring, unpack_bitstring


class ModiconPayloadBuilder(IPayloadBuilder):
    """A utility that helps build modicon encoded payload messages.

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

        builder = ModiconPayloadBuilder()
        builder.add_8bit_uint(1)
        builder.add_16bit_uint(2)
        payload = builder.build()
    """

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

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

    def __str__(self):
```

(continues on next page)

(continued from previous page)

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

    def add_8bit_uint(self, value):
        """Add a 8 bit unsigned int to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "B"
        self._payload.append(pack(fstring, value))

    def add_16bit_uint(self, value):
        """Add a 16 bit unsigned int to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "H"
        self._payload.append(pack(fstring, value))

    def add_32bit_uint(self, value):
```

(continues on next page)

(continued from previous page)

```
        """Add a 32 bit unsigned int to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "I"
        handle = pack(fstring, value)
        handle = handle[2:] + handle[:2]
        self._payload.append(handle)

    def add_8bit_int(self, value):
        """Add a 8 bit signed int to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "b"
        self._payload.append(pack(fstring, value))

    def add_16bit_int(self, value):
        """Add a 16 bit signed int to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "h"
        self._payload.append(pack(fstring, value))

    def add_32bit_int(self, value):
        """Add a 32 bit signed int to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "i"
        handle = pack(fstring, value)
        handle = handle[2:] + handle[:2]
        self._payload.append(handle)

    def add_32bit_float(self, value):
        """Add a 32 bit float to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "f"
        handle = pack(fstring, value)
        handle = handle[2:] + handle[:2]
        self._payload.append(handle)

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

        :param value: The value to add to the buffer
        """
        fstring = self._endian + "s"
        for i in value:
```

(continues on next page)

(continued from previous page)

```
            self._payload.append(pack(fstring, i))


class ModiconPayloadDecoder:
    """A utility that helps decode modicon encoded payload messages from a modbus␣
↪response message.

    What follows is a simple example::

        decoder = ModiconPayloadDecoder(payload)
        first   = decoder.decode_8bit_uint()
        second  = decoder.decode_16bit_uint()
    """

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

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

    @staticmethod
    def from_registers(registers, endian=Endian.Little):
        """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 ModiconPayloadDecoder(payload, endian)
        raise ParameterException("Invalid collection of registers supplied")

    @staticmethod
    def from_coils(coils, endian=Endian.Little):
        """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
```

(continues on next page)

(continued from previous page)

```
        :returns: An initialized PayloadDecoder
        """
        if isinstance(coils, list):
            payload = pack_bitstring(coils)
            return ModiconPayloadDecoder(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_8bit_uint(self):
        """Decode a 8 bit unsigned int from the buffer"""
        self._pointer += 1
        fstring = self._endian + "B"
        handle = self._payload[self._pointer - 1 : self._pointer]
        return unpack(fstring, handle)[0]

    def decode_16bit_uint(self):
        """Decode a 16 bit unsigned int from the buffer"""
        self._pointer += 2
        fstring = self._endian + "H"
        handle = self._payload[self._pointer - 2 : self._pointer]
        return unpack(fstring, handle)[0]

    def decode_32bit_uint(self):
        """Decode a 32 bit unsigned int from the buffer"""
        self._pointer += 4
        fstring = self._endian + "I"
        handle = self._payload[self._pointer - 4 : self._pointer]
        handle = handle[2:] + handle[:2]
        return unpack(fstring, handle)[0]

    def decode_8bit_int(self):
        """Decode a 8 bit signed int from the buffer"""
        self._pointer += 1
        fstring = self._endian + "b"
        handle = self._payload[self._pointer - 1 : self._pointer]
        return unpack(fstring, handle)[0]

    def decode_16bit_int(self):
        """Decode a 16 bit signed int from the buffer"""
        self._pointer += 2
        fstring = self._endian + "h"
        handle = self._payload[self._pointer - 2 : self._pointer]
        return unpack(fstring, handle)[0]

    def decode_32bit_int(self):
        """Decode a 32 bit signed int from the buffer"""
        self._pointer += 4
        fstring = self._endian + "i"
        handle = self._payload[self._pointer - 4 : self._pointer]
```

(continues on next page)

(continued from previous page)

```
        handle = handle[2:] + handle[:2]
        return unpack(fstring, handle)[0]

    def decode_32bit_float(self):
        """Decode a float from the buffer"""
        self._pointer += 4
        fstring = self._endian + "f"
        handle = self._payload[self._pointer - 4 : self._pointer]
        handle = handle[2:] + handle[:2]
        return unpack(fstring, handle)[0]

    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__ = ["ModiconPayloadBuilder", "ModiconPayloadDecoder"]
```

4.2.20 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
from time import time
```

(continues on next page)

(continued from previous page)

```
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
        client = ModbusTcpClient(n_host, port=5020)
```

(continues on next page)

(continued from previous page)

```
        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()
```

4.2.21 Remote Server Context Example

```
# pylint: disable=missing-type-doc,missing-param-doc,differing-param-doc,missing-raises-
↪doc
"""Although there is a remote server context already in the main library,

it works under the assumption that users would have a server context
of the following form::

    server_context = {
        0x00: client("host1.something.com"),
        0x01: client("host2.something.com"),
        0x02: client("host3.something.com")
    }

This example is how to create a server context where the client is
pointing to the same host, but the requested slave id is used as the
slave for the client::

    server_context = {
        0x00: client("host1.something.com", 0x00),
        0x01: client("host1.something.com", 0x01),
```

(continues on next page)

(continued from previous page)

```
        0x02: client("host1.something.com", 0x02)
    }
"""
import logging

from pymodbus.exceptions import NotImplementedException
from pymodbus.interfaces import IModbusSlaveContext


# --------------------------------------------------------------------------- #
# Logging
# --------------------------------------------------------------------------- #
_logger = logging.getLogger(__name__)

# --------------------------------------------------------------------------- #
# Slave Context
# --------------------------------------------------------------------------- #
# Basically we create a new slave context for the given slave identifier so
# that this slave context will only make requests to that slave with the
# client that the server is maintaining.
# --------------------------------------------------------------------------- #


class RemoteSingleSlaveContext(IModbusSlaveContext):
    """This is a remote server context,

    that allows one to create a server context backed by a single client that
    may be attached to many slave units. This can be used to
    effectively create a modbus forwarding server.
    """

    def __init__(self, context, unit_id):
        """Initialize the datastores

        :param context: The underlying context to operate with
        :param unit_id: The slave that this context will contact
        """
        self.context = context
        self.unit_id = unit_id

    def reset(self):
        """Reset all the datastores to their default values"""
        raise NotImplementedException()

    def validate(self, fx, address, count=1):
        """Validate the request to make sure it is in range

        :param fx: The function we are working with
        :param address: The starting address
        :param count: The number of values to test
        :returns: True if the request in within range, False otherwise
        """
```

(continues on next page)

(continued from previous page)

```
        txt = f"validate[{fx}] {address}:{count}"
        _logger.debug(txt)
        result = self.context.get_callbacks[self.decode(fx)](
            address, count, self.unit_id
        )
        return not result.isError()

    def getValues(self, fx, address, count=1):
        """Get `count` values from datastore

        :param fx: The function we are working with
        :param address: The starting address
        :param count: The number of values to retrieve
        :returns: The requested values from a:a+c
        """
        txt = f"get values[{fx}] {address}:{count}"
        _logger.debug(txt)
        result = self.context.get_callbacks[self.decode(fx)](
            address, count, self.unit_id
        )
        return self.__extract_result(self.decode(fx), result)

    def setValues(self, fx, address, values):
        """Set the datastore with the supplied values

        :param fx: The function we are working with
        :param address: The starting address
        :param values: The new values to be set
        """
        txt = f"set values[{fx}] {address}:{len(values)}"
        _logger.debug(txt)
        self.context.set_callbacks[self.decode(fx)](address, values, self.unit_id)

    def __str__(self):
        """Return a string representation of the context

        :returns: A string representation of the context
        """
        return f"Remote Single Slave Context({self.unit_id})"

    def __extract_result(self, f_code, result):
        """Extract the values out of a response.

        The future api should make the result consistent so we can just call `result.
↪getValues()`.

        :param fx: The function to call
        :param result: The resulting data
        """
        if not result.isError():
            if f_code in {"d", "c"}:
                return result.bits
```

(continues on next page)

(continued from previous page)

```
            if f_code in {"h", "i"}:
                return result.registers
            return None
        return result


# --------------------------------------------------------------------------- #
# Server Context
# --------------------------------------------------------------------------- #
# Think of this as simply a dictionary of { unit_id: client(req, unit_id) }
# --------------------------------------------------------------------------- #


class RemoteServerContext:
    """This is a remote server context,

    that allows one to create a server context backed by a single client that
    may be attached to many slave units. This can be used to
    effectively create a modbus forwarding server.
    """

    def __init__(self, client):
        """Initialize the datastores

        :param client: The client to retrieve values with
        """
        self.get_callbacks = {
            "d": lambda a, c, s: client.read_discrete_inputs(  # pylint:␣
↪disable=unnecessary-lambda
                a, c, s
            ),
            "c": lambda a, c, s: client.read_coils(  # pylint: disable=unnecessary-lambda
                a, c, s
            ),
            "h": lambda a, c, s: client.read_holding_registers(  # pylint:␣
↪disable=unnecessary-lambda
                a, c, s
            ),
            "i": lambda a, c, s: client.read_input_registers(  # pylint:␣
↪disable=unnecessary-lambda
                a, c, s
            ),
        }
        self.set_callbacks = {
            "d": lambda a, v, s: client.write_coils(  # pylint: disable=unnecessary-
↪lambda
                a, v, s
            ),
            "c": lambda a, v, s: client.write_coils(  # pylint: disable=unnecessary-
↪lambda
                a, v, s
            ),
```

(continues on next page)

(continued from previous page)

```
            "h": lambda a, v, s: client.write_registers(  # pylint: disable=unnecessary-
↪lambda
                a, v, s
            ),
            "i": lambda a, v, s: client.write_registers(  # pylint: disable=unnecessary-
↪lambda
                a, v, s
            ),
        }
        self._client = client
        self.slaves = {}  # simply a cache

    def __str__(self):
        """Return a string representation of the context

        :returns: A string representation of the context
        """
        return f"Remote Server Context{self._client}"

    def __iter__(self):
        """Iterate over the current collection of slave contexts.

        :returns: An iterator over the slave contexts
        """
        # note, this may not include all slaves
        return iter(self.slaves.items())

    def __contains__(self, slave):
        """Check if the given slave is in this list

        :param slave: slave The slave to check for existence
        :returns: True if the slave exists, False otherwise
        """
        # we don't want to check the cache here as the
        # slave may not exist yet or may not exist any
        # more. The best thing to do is try and fail.
        return True

    def __setitem__(self, slave, context):
        """Use to set a new slave context

        :param slave: The slave context to set
        :param context: The new context to set for this slave
        """
        raise NotImplementedException()  # doesn't make sense here

    def __delitem__(self, slave):
        """Use to access the slave context

        :param slave: The slave context to remove
        """
        raise NotImplementedException()  # doesn't make sense here
```

(continues on next page)

(continued from previous page)

```
    def __getitem__(self, slave):
        """Use to get access to a slave context

        :param slave: The slave context to get
        :returns: The requested slave context
        """
        if slave not in self.slaves:
            self.slaves[slave] = RemoteSingleSlaveContext(self, slave)
        return self.slaves[slave]
```

4.2.22 Thread Safe Datastore Example

```
# pylint: disable=missing-type-doc
"""Thread safe datastore."""
import threading
from contextlib import contextmanager

from pymodbus.datastore.store import BaseModbusDataBlock


class ContextWrapper:
    """This is a simple wrapper around enter and exit functions

    that conforms to the python context manager protocol:

    with ContextWrapper(enter, leave):
        do_something()
    """

    def __init__(self, enter=None, leave=None, factory=None):
        """Initialize."""
        self._enter = enter
        self._leave = leave
        self._factory = factory

    def __enter__(self):
        """Do on enter."""
        if self.enter:  # pylint: disable=no-member
            self._enter()
        return self if not self._factory else self._factory()

    def __exit__(self, *args):
        """Do on exit."""
        if self._leave:
            self._leave()


class ReadWriteLock:
    """This reader writer lock guarantees write order,
```

(continues on next page)

(continued from previous page)

```
    but not read order and is generally biased towards allowing writes
    if they are available to prevent starvation.
    TODO:
    * allow user to choose between read/write/random biasing
    - currently write biased
    - read biased allow N readers in queue
    - random is 50/50 choice of next
    """

    def __init__(self):
        """Initialize a new instance of the ReadWriteLock"""
        self.queue = []  # the current writer queue
        self.lock = threading.Lock()  # the underlying condition lock
        self.read_condition = threading.Condition(
            self.lock
        )  # the single reader condition
        self.readers = 0  # the number of current readers
        self.writer = False  # is there a current writer

    def __is_pending_writer(self):
        """Check is pending writer."""
        return self.writer or (  # if there is a current writer
            self.queue  # or if there is a waiting writer
            and (self.queue[0] != self.read_condition)
        )

    def acquire_reader(self):
        """Notify the lock that a new reader is requesting the underlying resource."""
        with self.lock:
            if self.__is_pending_writer():  # if there are existing writers waiting
                if (
                    self.read_condition not in self.queue
                ):  # do not pollute the queue with readers
                    self.queue.append(
                        self.read_condition
                    )  # add the readers in line for the queue
                while (
                    self.__is_pending_writer()
                ):  # until the current writer is finished
                    self.read_condition.wait(1)  # wait on our condition
                if self.queue and self.read_condition == self.queue[0]:
                    self.queue.pop(0)  # then go ahead and remove it
            self.readers += 1  # update the current number of readers

    def acquire_writer(self):
        """Notify the lock that a new writer is requesting the underlying resource."""
        with self.lock:
            if self.writer or self.readers:
                condition = threading.Condition(self.lock)
                # create a condition just for this writer
                self.queue.append(condition)  # and put it on the waiting queue
```

(continues on next page)

(continued from previous page)

```
                while self.writer or self.readers:  # until the write lock is free
                    condition.wait(1)
                self.queue.pop(0)
            self.writer = True  # stop other writers from operating

    def release_reader(self):
        """Notify the lock that an existing reader is finished with the underlying␣
↪resource."""
        with self.lock:
            self.readers = max(0, self.readers - 1)  # readers should never go below 0
            if not self.readers and self.queue:  # if there are no active readers
                self.queue[0].notify_all()  # then notify any waiting writers

    def release_writer(self):
        """Notify the lock that an existing writer is finished with the underlying␣
↪resource."""
        with self.lock:
            self.writer = False  # give up current writing handle
            if self.queue:  # if someone is waiting in the queue
                self.queue[0].notify_all()  # wake them up first
            else:
                self.read_condition.notify_all()  # otherwise wake up all possible␣
↪readers

    @contextmanager
    def get_reader_lock(self):
        """Wrap some code with a reader lock using the python context manager protocol::

        with rwlock.get_reader_lock():
            do_read_operation()
        """
        try:
            self.acquire_reader()
            yield self
        finally:
            self.release_reader()

    @contextmanager
    def get_writer_lock(self):
        """Wrap some code with a writer lock using the python context manager protocol::

        with rwlock.get_writer_lock():
            do_read_operation()
        """
        try:
            self.acquire_writer()
            yield self
        finally:
            self.release_writer()


class ThreadSafeDataBlock(BaseModbusDataBlock):
```

(continues on next page)

(continued from previous page)

```
    """This is a simple decorator for a data block.

    This allows a user to inject an existing data block which can then be
    safely operated on from multiple cocurrent threads.

    It should be noted that the choice was made to lock around the
    datablock instead of the manager as there is less source of
    contention (writes can occur to slave 0x01 while reads can
    occur to slave 0x02).
    """

    def __init__(self, block):
        """Initialize a new thread safe decorator

        :param block: The block to decorate
        """
        self.rwlock = ReadWriteLock()
        self.block = block

    def validate(self, address, count=1):
        """Check to see if the request is in range

        :param address: The starting address
        :param count: The number of values to test for
        :returns: True if the request in within range, False otherwise
        """
        with self.rwlock.get_reader_lock():
            return self.block.validate(address, count)

    def getValues(self, address, count=1):
        """Return the requested values of the datastore

        :param address: The starting address
        :param count: The number of values to retrieve
        :returns: The requested values from a:a+c
        """
        with self.rwlock.get_reader_lock():
            return self.block.getValues(address, count)

    def setValues(self, address, values):
        """Set the requested values of the datastore

        :param address: The starting address
        :param values: The new values to be set
        """
        with self.rwlock.get_writer_lock():
            return self.block.setValues(address, values)


if __name__ == "__main__":  # pylint: disable=too-complex

    class AtomicCounter:
```

(continues on next page)

(continued from previous page)

```
        """Atomic counter."""

        def __init__(self, **kwargs):
            """Init."""
            self.counter = kwargs.get("start", 0)
            self.finish = kwargs.get("finish", 1000)
            self.lock = threading.Lock()

        def increment(self, count=1):
            """Increment."""
            with self.lock:
                self.counter += count

        def is_running(self):
            """Is running."""
            return self.counter <= self.finish

    locker = ReadWriteLock()
    readers, writers = AtomicCounter(), AtomicCounter()

    def read():
        """Read."""
        while writers.is_running() and readers.is_running():
            with locker.get_reader_lock():
                readers.increment()

    def write():
        """Write."""
        while writers.is_running() and readers.is_running():
            with locker.get_writer_lock():
                writers.increment()

    rthreads = [threading.Thread(target=read) for i in range(50)]
    wthreads = [threading.Thread(target=write) for i in range(2)]
    for t in rthreads + wthreads:
        t.start()
    for t in rthreads + wthreads:
        t.join()
    print(f"readers[{readers.counter}] writers[{writers.counter}]")
```

4.3 Examples contributions

4.3.1 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
```

(continues on next page)

(continued from previous page)

```
"""
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):  # pylint: disable=unused-argument
    """Enters shutdown mode"""
    _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, unit=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:
```

(continues on next page)

(continued from previous page)

```
            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="+"
    )

    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())
```

CHAPTER

FIVE

PYMODBUS

5.1 pymodbus package

Pymodbus: Modbus Protocol Implementation.

Released under the the BSD license

pymodbus.**pymodbus_apply_logging_config**()

Apply basic logging configuration used by default by Pymodbus maintainers.

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

5.1.1 Subpackages

pymodbus.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)
```

(continues on next page)

(continued from previous page)

```
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)

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

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

Client setup.

class pymodbus.client.base.**ModbusBaseClient**(*framer: str = 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 = 300000*, ***kwargs: any*)

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 pr 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) Delay in milliseconds before reconnecting.
- **kwargs** – (optional) Experimental parameters.

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

Tip: **reconnect_delay** doubles automatically with each unsuccessful connect. 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() → None

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() → int

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**).

Serial RS-485 transport.

class pymodbus.client.serial.**AsyncModbusSerialClient**(*port: str*, *framer: ~pymodbus.framer.ModbusFramer = <class 'pymodbus.framer.rtu_framer.ModbusRtuFramer'>*, *baudrate: int = 19200*, *bytesize: int = 8*, *parity: chr = 'N'*, *stopbits: int = 1*, *handle_local_echo: bool = False*, ***kwargs: any*)

AsyncModbusSerialClient.

Parameters

- **port** – Serial port used for communication.
- **framer** – (optional) Framer class.
- **baudrate** – (optional) Bits pr second.
- **bytesize** – (optional) Number of bits pr 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 vith a usb RS485 dongle.

Example:

```
from pymodbus.client import AsyncModbusSerialClient

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

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

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

ModbusSerialClient.

Parameters

- **port** – Serial port used for communication.
- **framer** – (optional) Framer class.
- **baudrate** – (optional) Bits pr second.
- **bytesize** – (optional) Number of bits pr 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 vith a usb RS485 dongle.

Example:

```
from pymodbus.client import ModbusSerialClient

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

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

TCP transport.

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

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

Example:

```
from pymodbus.client import AsyncModbusTcpClient

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

    await client.connect()
```

(continues on next page)

(continued from previous page)

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

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

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

Example:

```
from pymodbus.client import ModbusTcpClient

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

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

TLS transport.

class pymodbus.client.tls.**AsyncModbusTlsClient**(*host: str*, *port: int = 802*, *framer: ~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: any*)

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.tls.**ModbusTlsClient**(*host: str*, *port: int = 802*, *framer: ~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: any*)

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 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()
```

UDP transport.

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

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

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()
```

Client device 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**

ModbusClientMixin.

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.

---

jan

diag_change_ascii_input_delimeter(*slave: int = 0*, ***kwargs: any*) → *ChangeAsciiInputDelimiterResponse*

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

Parameters

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

Returns

A deferred response handle

Raises

ModbusException –

diag_clear_counters(*slave: int = 0*, ***kwargs: any*) → *ClearCountersResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_clear_overrun_counter(*slave: int = 0*, ***kwargs: any*) → *ClearOverrunCountResponse*

Diagnose Clear Overrun Counter and Flag (code 0x08 sub 0x14).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_force_listen_only(*slave: int = 0*, ***kwargs: any*) → *ForceListenOnlyModeResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_get_comm_event_counter(***kwargs: any*) → *GetCommEventCounterResponse*

Diagnose get event counter (code 0x0B).

Parameters
kwargs – (optional) Experimental parameters.

Returns
A deferred response handle

Raises
ModbusException –

diag_get_comm_event_log(***kwargs: any*) → *GetCommEventLogResponse*

Diagnose get event counter (code 0x0C).

Parameters
kwargs – (optional) Experimental parameters.

Returns
A deferred response handle

Raises
ModbusException –

diag_getclear_modbus_response(*slave: int = 0*, ***kwargs: any*) → *GetClearModbusPlusResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_query_data(*msg: bytearray*, *slave: int = 0*, ***kwargs: any*) → *ReturnQueryDataResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_bus_char_overrun_count(*slave: int = 0*, ***kwargs: any*) → *ReturnSlaveBusCharacterOverrunCountResponse*

Diagnose read Bus Character Overrun Count (code 0x08 sub 0x12).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_bus_comm_error_count(*slave: int = 0*, ***kwargs: any*) → *ReturnBusCommunicationErrorCountResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_bus_exception_error_count(*slave: int = 0*, ***kwargs: any*) → *ReturnBusExceptionErrorCountResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_bus_message_count(*slave: int = 0*, ***kwargs: any*) → *ReturnBusMessageCountResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_diagnostic_register(*slave: int = 0*, ***kwargs: any*) → *ReturnDiagnosticRegisterResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_iop_overrun_count(*slave: int = 0*, ***kwargs: any*) → *ReturnIopOverrunCountResponse*

Diagnose read Iop overrun count (code 0x08 sub 0x13).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_slave_busy_count(*slave: int = 0*, ***kwargs: any*) → *ReturnSlaveBusyCountResponse*

Diagnose read Slave Busy Count (code 0x08 sub 0x11).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_slave_message_count(*slave: int = 0*, ***kwargs: any*) → *ReturnSlaveMessageCountResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_slave_nak_count(*slave: int = 0*, ***kwargs: any*) → *ReturnSlaveNAKCountResponse*

Diagnose read Slave NAK Count (code 0x08 sub 0x10).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_read_slave_no_response_count(*slave: int = 0*, ***kwargs: any*) → *ReturnSlaveNoReponseCountResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

diag_restart_communication(*toggle: bool*, *slave: int = 0*, ***kwargs: any*) → *RestartCommunicationsOptionResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

execute(*request: ModbusRequest*) → ModbusResponse

Execute request (code ???).

Parameters
request – Request to send

Returns
A deferred response handle

Raises
ModbusException –

Call with custom function codes.

Tip: Response is not interpreted.

---

mask_write_register(*address: int = 0*, *and_mask: int = 65535*, *or_mask: int = 0*, ***kwargs: any*) → *MaskWriteRegisterResponse*

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.

Returns
A deferred response handle

Raises
ModbusException –

read_coils(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: any*) → *ReadCoilsResponse*

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.

Returns
A deferred response handle

Raises
ModbusException –

read_device_information(*read_code: Optional[int] = None*, *object_id: int = 0*, ***kwargs: any*) → *ReadDeviceInformationResponse*

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

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

read_discrete_inputs(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: any*) → *ReadDiscreteInputsResponse*

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.

Returns
A deferred response handle

Raises
ModbusException –

read_exception_status(*slave: int = 0*, ***kwargs: any*) → *ReadExceptionStatusResponse*

Read Exception Status (code 0x07).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

read_fifo_queue(*address: int = 0*, ***kwargs: any*) → *ReadFifoQueueResponse*

Read FIFO queue (code 0x18).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

read_file_record(*records: List[Tuple]*, ***kwargs: any*) → *ReadFileRecordResponse*

Read file record (code 0x14).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

read_holding_registers(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: any*) → *ReadHoldingRegistersResponse*

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.

Returns
A deferred response handle

Raises
ModbusException –

read_input_registers(*address: int*, *count: int = 1*, *slave: int = 0*, ***kwargs: any*) → *ReadInputRegistersResponse*

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.

Returns
A deferred response handle

Raises
ModbusException –

readwrite_registers(*read_address: int = 0*, *read_count: int = 0*, *write_address: int = 0*, *values: int = 0*, *slave: int = 0*, ***kwargs*) → *ReadWriteMultipleRegistersResponse*

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** – The registers to write to the specified address
- **slave** – (optional) Modbus slave unit ID

- **kwargs** –

Returns
A deferred response handle

Raises
ModbusException –

report_slave_id(*slave: int = 0*, ***kwargs: any*) → *ReportSlaveIdResponse*

Report slave ID (code 0x11).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

write_coil(*address: int*, *value: bool*, *slave: int = 0*, ***kwargs: any*) → *WriteSingleCoilResponse*

Write single coil (code 0x05).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

write_coils(*address: int*, *values: List[bool]*, *slave: int = 0*, ***kwargs: any*) → *WriteMultipleCoilsResponse*

Write coils (code 0x0F).

Parameters

- **address** – Start address to read from
- **values** – List of booleans to write
- **slave** – (optional) Modbus slave ID
- **kwargs** – (optional) Experimental parameters.

Returns
A deferred response handle

Raises
ModbusException –

write_file_record(*records: List[Tuple]*, ***kwargs: any*) → *ReadFileRecordResponse*

Write file record (code 0x15).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

write_register(*address: int*, *value: Union[int, float, str]*, *slave: int = 0*, ***kwargs: any*) → *WriteSingleRegisterResponse*

Write register (code 0x06).

Parameters

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

Returns
A deferred response handle

Raises
ModbusException –

write_registers(*address: int*, *values: List[Union[int, float, str]]*, *slave: int = 0*, ***kwargs: any*) → *WriteMultipleRegistersResponse*

Write registers (code 0x10).

Parameters

- **address** – Start address to read from
- **values** – List of booleans to write
- **slave** – (optional) Modbus slave unit ID
- **kwargs** – (optional) Experimental parameters.

Returns
A deferred response handle

Raises
ModbusException –

pymodbus.constants package

Constants For Modbus Server/Client.

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

class pymodbus.constants.**Defaults**(**args*, ***kwargs*)

Bases: *Singleton*

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)

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.

broadcast_enable

When False unit_id 0 will be treated as any other unit_id. When True and the unit_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 = 300000

Reconnects = 0

Retries = 3

RetryOnEmpty = False

RetryOnInvalid = False

Slave = 0

Stopbits = 1

Strict = True

TcpPort = 502

Timeout = 3

TlsPort = 802

TransactionId = 0

UdpPort = 502

ZeroMode = False

class pymodbus.constants.**DeviceInformation**(**args*, ***kwargs*)

Bases: *Singleton*

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**(**args*, ***kwargs*)

Bases: *Singleton*

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 little endian format

Little

This indicates that the bytes are in big 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**(**args*, ***kwargs*)

Bases: *Singleton*

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**(**args*, ***kwargs*)

Bases: *Singleton*

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**(**args*, ***kwargs*)

Bases: *Singleton*

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

Submodules

pymodbus.datastore package

Define datastore.

class pymodbus.datastore.**ModbusSequentialDataBlock**(*address*, *values*)

Bases: *BaseModbusDataBlock*

Creates a sequential modbus datastore.

classmethod create()

Create a datastore.

With the full address space initialized to 0x00

Returns
An initialized datastore

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*)

Set the requested values of the datastore.

Parameters

- **address** – The starting address
- **values** – The new values to be set

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

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

Bases: `object`

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]*, *actions: Dict[str, Callable]*)

Bases: object

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,
            },
        },
```

(continues on next page)

(continued from previous page)

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

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

start_time = 1670520884

class pymodbus.datastore.**ModbusSlaveContext**(**args*, ***kwargs*)

Bases: *IModbusSlaveContext*

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

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

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

reset()

Reset all the datastores to their default values.

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

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

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

Bases: *BaseModbusDataBlock*

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

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

reset()

Reset the store to the initially provided defaults.

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 –

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

Subpackages

pymodbus.datastore.database package

Define Datastore.

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

Bases: *IModbusSlaveContext*

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

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

Get *count* values from datastore.

Parameters

- **fx** – 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(*fx*, *address*, *values*)

Set the datastore with the supplied values.

Parameters

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

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

Validate the request to make sure it is in range.

Parameters

- **fx** – 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.database.**SqlSlaveContext**(**args*, ***kwargs*)

Bases: *IModbusSlaveContext*

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

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

Get *count* values from datastore.

Parameters

- **fx** – 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(*fx*, *address*, *values*, *update=True*)
Set the datastore with the supplied values.

Parameters

- **fx** – 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(*fx*, *address*, *count=1*)
Validate the request to make sure it is in range.

Parameters

- **fx** – 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

Submodules

pymodbus.datastore.database.redis_datastore module

Datastore using redis.

class pymodbus.datastore.database.redis_datastore.**RedisSlaveContext**(***kwargs*)
Bases: *IModbusSlaveContext*

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

getValues(*fx*, *address*, *count=1*)
Get *count* values from datastore.

Parameters

- **fx** – 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(*fx*, *address*, *values*)

Set the datastore with the supplied values.

Parameters

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

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

Validate the request to make sure it is in range.

Parameters

- **fx** – 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

pymodbus.datastore.database.sql_datastore module

Datastore using SQL.

class pymodbus.datastore.database.sql_datastore.**SqlSlaveContext**(**args*, ***kwargs*)

Bases: *IModbusSlaveContext*

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

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

Get *count* values from datastore.

Parameters

- **fx** – 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(*fx*, *address*, *values*, *update=True*)

Set the datastore with the supplied values.

Parameters

- **fx** – 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(*fx*, *address*, *count=1*)

Validate the request to make sure it is in range.

Parameters

- **fx** – 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

Submodules

pymodbus.datastore.context module

Context for datastore.

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

Bases: `object`

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.context.**ModbusSlaveContext**(**args*, ***kwargs*)

Bases: *IModbusSlaveContext*

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

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

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

reset()

Reset all the datastores to their default values.

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

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

pymodbus.datastore.remote module

Remote datastore.

class pymodbus.datastore.remote.**RemoteSlaveContext**(*client*, *unit=None*)

Bases: *IModbusSlaveContext*

TODO.

This creates a modbus data model that connects to a remote device (depending on the client used)

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

Get values from real call in validate

reset()

Reset all the datastores to their default values.

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

Set the datastore with the supplied values.

Already done in validate

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

pymodbus.datastore.simulator module

Pymodbus ModbusSimulatorContext.

class pymodbus.datastore.simulator.**Cell**(*type: int = ' '*, *access: bool = False*, *value: int = 0*, *action: Optional[Callable] = None*)

Handle a single cell.

class pymodbus.datastore.simulator.**ModbusSimulatorContext**(*config: Dict[str, any]*, *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)
```

(continues on next page)

(continued from previous page)

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

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

pymodbus.datastore.store module

Modbus Server Datastore.

For each server, you will create a ModbusServerContext and pass in the default address space for each data access. The class will create and manage the data.

Further modification of said data accesses should be performed with [get,set][access]Values(address, count)

Datastore Implementation

There are two ways that the server datastore can be implemented. The first is a complete range from "address" start to "count" number of indices. This can be thought of as a straight array:

```
data = range(1, 1 + count)
[1,2,3,...,count]
```

The other way that the datastore can be implemented (and how many devices implement it) is a associate-array:

```
data = {1:"1", 3:"3", ..., count:"count"}
[1,3,...,count]
```

The difference between the two is that the latter will allow arbitrary gaps in its datastore while the former will not. This is seen quite commonly in some modbus implementations. What follows is a clear example from the field:

Say a company makes two devices to monitor power usage on a rack. One works with three-phase and the other with a single phase. The company will dictate a modbus data mapping such that registers:

```
n:      phase 1 power
n+1:    phase 2 power
n+2:    phase 3 power
```

Using this, layout, the first device will implement n, n+1, and n+2, however, the second device may set the latter two values to 0 or will simply not implemented the registers thus causing a single read or a range read to fail.

I have both methods implemented, and leave it up to the user to change based on their preference.

class pymodbus.datastore.store.**BaseModbusDataBlock**

Bases: object

Base class for a modbus datastore

Derived classes must create the following fields:
@address The starting address point @defult_value The default value of the datastore @values The actual datastore values

Derived classes must implemented the following methods:
validate(self, address, count=1) getValues(self, address, count=1) setValues(self, address, values)

default(*count*, *value=False*)

Use to initialize a store to one value.

Parameters

- **count** – The number of fields to set
- **value** – The default value to set to the fields

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

Return the requested values from the datastore.

Parameters

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

Raises

NotImplementedException –

reset()

Reset the datastore to the initialized default value.

setValues(*address*, *values*)

Return the requested values from the datastore.

Parameters

- **address** – The starting address
- **values** – The values to store

Raises

NotImplementedException –

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

Raises

NotImplementedException –

class pymodbus.datastore.store.**ModbusSequentialDataBlock**(*address*, *values*)

Bases: *BaseModbusDataBlock*

Creates a sequential modbus datastore.

classmethod create()

Create a datastore.

With the full address space initialized to 0x00

Returns

An initialized datastore

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*)

Set the requested values of the datastore.

Parameters

- **address** – The starting address
- **values** – The new values to be set

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

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

Bases: *BaseModbusDataBlock*

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

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

reset()

Reset the store to the initially provided defaults.

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 –

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

pymodbus.framer package

Submodules

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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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).

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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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).

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

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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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).

Module contents

Framer start.

class pymodbus.framer.**ModbusFramer**

Bases: *IModbusFramer*

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:

pymodbus.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.

pymodbus.server package

Server.

import external classes, to make them easier to use:

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

Terminate server.

pymodbus.server.**ServerStop**()

Terminate server.

async pymodbus.server.**StartAsyncSerialServer**(*context=None*, *identity=None*, *custom_functions=[]*, *defer_start=False*, ***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.
- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** – The rest

async pymodbus.server.**StartAsyncTcpServer**(*context=None*, *identity=None*, *address=None*, *custom_functions=[]*, *defer_start=False*, ***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.
- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** – The rest

Returns
an initialized but inactive server object coroutine

async pymodbus.server.**StartAsyncTlsServer**(*context=None*, *identity=None*, *address=None*, *sslctx=None*, *certfile=None*, *keyfile=None*, *password=None*, *reqclicert=False*, *allow_reuse_address=False*, *allow_reuse_port=False*, *custom_functions=[]*, *defer_start=False*, ***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.
- **allow_reuse_port** – Whether the server will allow the reuse of a port.
- **custom_functions** – An optional list of custom function classes supported by server instance.
- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** – The rest

Returns
an initialized but inactive server object coroutine

async pymodbus.server.**StartAsyncUdpServer**(*context=None*, *identity=None*, *address=None*, *custom_functions=[]*, *defer_start=False*, ***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.

- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** –

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

Start and run a serial modbus server.

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

Start and run a serial modbus server.

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

Start and run a serial modbus server.

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

Start and run a serial modbus server.

Submodules

pymodbus.server module

Implementation of a Threaded Modbus Server.

class pymodbus.server.async_io.**ModbusBaseRequestHandler**(*owner*)

Bases: BaseProtocol

Implements modbus slave wire protocol.

This uses the asyncio.Protocol to implement the client handler.

When a connection is established, the asyncio.Protocol.connection_made callback is called. This callback will setup the connection and create and schedule an asyncio.Task and assign it to running_task.

running_task will be canceled upon connection_lost event.

connection_lost(*call_exc*)

Call for socket tear down.

For streamed protocols any break in the network connection will be reported here; for datagram protocols, only a teardown of the socket itself will result in this call.

connection_made(*transport*)

Call for socket establish

For streamed protocols (TCP) this will also correspond to an entire conversation; however for datagram protocols (UDP) this corresponds to the socket being opened

execute(*request*, **addr*)

Call with the resulting message.

Parameters

- **request** – The decoded request message
- **addr** – the address

async handle()

Return Asyncio coroutine which represents a single conversation.

between the modbus slave and master

Once the client connection is established, the data chunks will be fed to this coroutine via the asyncio.Queue object which is fed by the ModbusBaseRequestHandler class"s callback Future.

This callback future gets data from either asyncio.DatagramProtocol.datagram_received or from asyncio.BaseProtocol.data_received.

This function will execute without blocking in the while-loop and yield to the asyncio event loop when the frame is exhausted. As a result, multiple clients can be interleaved without any interference between them.

For ModbusConnectedRequestHandler, each connection will be given an instance of the handle() coroutine and this instance will be put in the active_connections dict. Calling server_close will individually cancel each running handle() task.

For ModbusDisconnectedRequestHandler, a single handle() coroutine will be started and maintained. Calling server_close will cancel that task.

send(*message*, **addr*, ***kwargs*)

Send message.

class pymodbus.server.async_io.ModbusConnectedRequestHandler(*owner*)

Bases: *ModbusBaseRequestHandler*, Protocol

Implements the modbus server protocol

This uses asyncio.Protocol to implement the client handler for a connected protocol (TCP).

connection_lost(*call_exc*)

Call when the connection is lost or closed.

connection_made(*transport*)

Call when a connection is made.

data_received(*data*)

Call when some data is received.

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

class pymodbus.server.async_io.ModbusDisconnectedRequestHandler(*owner*)

Bases: *ModbusBaseRequestHandler*, DatagramProtocol

Implements the modbus server protocol

This uses the socketserver.BaseRequestHandler to implement the client handler for a disconnected protocol (UDP). The only difference is that we have to specify who to send the resulting packet data to.

connection_lost(*call_exc*)

Handle connection lost.

datagram_received(*data*, *addr*)

Call when a datagram is received.

data is a bytes object containing the incoming data. addr is the address of the peer sending the data; the exact format depends on the transport.

error_received(*exc*)

Call when a previous send/receive raises an OSError.

exc is the OSError instance.

This method is called in rare conditions, when the transport (e.g. UDP) detects that a datagram could not be delivered to its recipient. In many conditions though, undeliverable datagrams will be silently dropped.

class pymodbus.server.async_io.**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 = 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.async_io.**ModbusSingleRequestHandler**(*owner*)

Bases: *ModbusBaseRequestHandler*, `Protocol`

Implement the modbus server protocol.

This uses asyncio.Protocol to implement the client handler for a serial connection.

connection_lost(*call_exc*)

Handle connection lost.

connection_made(*transport*)

Handle connect made.

data_received(*data*)

Receive data.

class pymodbus.server.async_io.**ModbusTcpServer**(*context*, *framer=None*, *identity=None*, *address=None*, *handler=None*, *allow_reuse_address=False*, *allow_reuse_port=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.async_io.**ModbusTlsServer**(*context*, *framer=None*, *identity=None*, *address=None*, *sslctx=None*, *certfile=None*, *keyfile=None*, *password=None*, *reqclicert=False*, *handler=None*, *allow_reuse_address=False*, *allow_reuse_port=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.async_io.**ModbusUdpServer**(*context*, *framer=None*, *identity=None*, *address=None*, *handler=None*, *allow_reuse_address=False*, *allow_reuse_port=False*, *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.

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

Terminate server.

pymodbus.server.async_io.**ServerStop**()

Terminate server.

async pymodbus.server.async_io.**StartAsyncSerialServer**(*context=None*, *identity=None*, *custom_functions=[]*, *defer_start=False*, ***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.

- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** – The rest

async pymodbus.server.async_io.**StartAsyncTcpServer**(*context=None*, *identity=None*, *address=None*, *custom_functions=[]*, *defer_start=False*, ***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.
- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** – The rest

Returns

an initialized but inactive server object coroutine

async pymodbus.server.async_io.**StartAsyncTlsServer**(*context=None*, *identity=None*, *address=None*, *sslctx=None*, *certfile=None*, *keyfile=None*, *password=None*, *reqclicert=False*, *allow_reuse_address=False*, *allow_reuse_port=False*, *custom_functions=[]*, *defer_start=False*, ***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.
- **allow_reuse_port** – Whether the server will allow the reuse of a port.
- **custom_functions** – An optional list of custom function classes supported by server instance.
- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** – The rest

Returns
an initialized but inactive server object coroutine

async pymodbus.server.async_io.**StartAsyncUdpServer**(*context=None*, *identity=None*, *address=None*, *custom_functions=[]*, *defer_start=False*, ***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.
- **defer_start** – if set, the server object will be returned ready to start. Otherwise, the server will be immediately spun up without the ability to shut it off
- **kwargs** –

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

Start and run a serial modbus server.

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

Start and run a serial modbus server.

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

Start and run a serial modbus server.

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

Start and run a serial modbus server.

pymodbus.server.async_io.**sslctx_provider**(*sslctx=None*, *certfile=None*, *keyfile=None*, *password=None*, *reqclicert=False*)

Provide the SSLContext for ModbusTlsServer.

If the user defined SSLContext is not passed in, sslctx_provider will produce a default one.

Parameters

- **sslctx** – The user defined 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

pymodbus.repl package

Pymodbus REPL Module.

Subpackages

pymodbus.repl.client package

Repl client.

Submodules

pymodbus.repl.client.completer module

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.

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

A list of completions.

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.

pymodbus.repl.client.helper module

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 = 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 = 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.client.main module

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.

pymodbus.repl.client.mclient module

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*, *unit=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
- **unit** – 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(***kwargs*)

Read tcontents of eight Exception Status output.

In a remote device.

Parameters

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*, *read_count*, *write_address*, *write_registers*, *unit=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
- **write_registers** – List of register values to write (comma separated)
- **unit** – Modbus slave unit ID
- **kwargs** –

Returns

report_slave_id(*unit=0*, ***kwargs*)

Report information about remote slave ID.

Parameters

- **unit** – 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.

pymodbus.repl..server package

Repl server.

Submodules

pymodbus.repl.server.cli module

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.

pymodbus.repl.server.main module

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: str = <typer.models.OptionInfo object>*, *modbus_unit_id: ~typing.List[int] = <typer.models.OptionInfo object>*, *modbus_config: ~pathlib.Path = <typer.models.OptionInfo object>*, *randomize: 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.

5.1.2 Submodules

5.1.3 pymodbus.bit_read_message module

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

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

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

5.1.4 pymodbus.bit_write_message module

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

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

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

5.1.5 pymodbus.device module

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**(**args*, ***kwargs*)

Bases: *Singleton*

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: *Singleton*

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*)

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

5.1.6 pymodbus.diag_message module

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

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.**ReturnSlaveNoReponseCountResponse**(*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

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

5.1.7 pymodbus.events module

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

5.1.8 pymodbus.exceptions module

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.

5.1.9 pymodbus.factory module

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: *IModbusDecoder*

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

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: *IModbusDecoder*

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

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 –

5.1.10 pymodbus.file_message module

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.

Parameters
context – The datastore to request from

Returns
The populated response

function_code = 24

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.

Parameters
context – The datastore to request from

Returns
The populated response

function_code = 20

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.

Parameters
context – The datastore to request from

Returns
The populated response

function_code = 21

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

5.1.11 pymodbus.interfaces module

Pymodbus Interfaces.

A collection of base classes that are used throughout the pymodbus library.

class pymodbus.interfaces.**IModbusDecoder**

Bases: object

Modbus Decoder Base Class.

This interface must be implemented by a modbus message decoder factory. These factories are responsible for abstracting away converting a raw packet into a request / response message object.

decode(*message*)

Decode a given packet.

Parameters
message – The raw modbus request packet

Raises
NotImplementedException –

lookupPduClass(*function_code*)

Use *function_code* to determine the class of the PDU.

Parameters
function_code – The function code specified in a frame.

Raises
NotImplementedException –

register(*function*)

Register a function and sub function class with the decoder.

Parameters
function – Custom function class to register

Raises
NotImplementedException –

class pymodbus.interfaces.**IModbusFramer**

Bases: object

A framer strategy interface.

The idea is that we abstract away all the detail about how to detect if a current message frame exists, decoding it, sending it, etc so that we can plug in a new Framer object (tcp, rtu, 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

Raises
NotImplementedException –

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.

The raw packet is built off of a fully populated modbus request / response message.

Parameters
message – The request/response to send

Raises
NotImplementedException –

checkFrame()

Check and decode the next frame.

Raises
NotImplementedException –

getFrame()

Get the next frame from the buffer.

Raises
NotImplementedException –

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.

Raises
NotImplementedException –

populateResult(*result*)

Populate the modbus result with current frame header.

We basically copy the data back over from the current header to the result header. This may not be needed for serial messages.

Parameters
result – The response packet

Raises
NotImplementedException –

processIncomingPacket(*data*, *callback*)

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

Raises
NotImplementedException –

class pymodbus.interfaces.**IModbusSlaveContext**

Bases: object

Interface for a modbus slave data context.

Derived classes must implemented the following methods:
reset(self) validate(self, fx, address, count=1) getValues(self, fx, address, count=1) setValues(self, fx, address, values)

decode(*fx*)

Convert the function code to the datastore to.

Parameters
fx – The function we are working with

Returns
one of [d(iscretes),i(nputs),h(olding),c(oils)

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

Get *count* values from datastore.

Parameters

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

Raises

NotImplementedException –

reset()

Reset all the datastores to their default values.

setValues(*fx*, *address*, *values*)

Set the datastore with the supplied values.

Parameters

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

Raises

NotImplementedException –

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

Validate the request to make sure it is in range.

Parameters

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

Raises

NotImplementedException –

class pymodbus.interfaces.**IPayloadBuilder**

Bases: object

This is an interface to a class that can build a payload for a modbus register write command.

It should abstract the codec for encoding data to the required format (bcd, binary, char, etc).

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.

Raises

NotImplementedException –

class pymodbus.interfaces.**Singleton**(**args*, ***kwargs*)

Bases: object

Singleton base class.

https://mail.python.org/pipermail/python-list/2007-July/450681.html

5.1.12 pymodbus.mei_message module

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.

Parameters
context – The datastore to request from

Returns
The populated response

function_code = 43

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

5.1.13 pymodbus.other_message module

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

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

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

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

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

5.1.14 pymodbus.payload module

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: *IPayloadBuilder*

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.

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

Parameters

- **coils** – The coil results to initialize with
- **byteorder** – The endianness of the payload

- **wordorder** – The endianness of the payload

Returns
An initialized PayloadDecoder

Raises
ParameterException –

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

5.1.15 pymodbus.pdu module

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.

Parameters
data – Not used

execute(*context*)

Build an illegal function request error response.

Parameters
context – The current context for the message

Returns
The error response packet

class pymodbus.pdu.**ModbusExceptions**(**args*, ***kwargs*)

Bases: *Singleton*

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

5.1.16 pymodbus.register_read_message module

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

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

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

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

5.1.17 pymodbus.register_write_message module

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

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

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

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:

5.1.18 pymodbus.transaction module

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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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.

With the transport specific header information (no header before PDU in decrypted message)

Parameters
result – The response packet

processIncomingPacket(*data*, *callback*, *unit*, ***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
- **unit** – Process if unit id matches, ignore otherwise (could be a list of unit ids (server) or single unit 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).

5.1.19 pymodbus.utilities module

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)
```

5.1.20 pymodbus.version module

Handle the version information here.

you should only have to change the version tuple.

CHAPTER

SIX

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

p

INDEX

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

U

V

W

Z