PyModbus Documentation

Release 1.4.0

Sanjay

Jan 03, 2018

Contents:

CHAPTER 1

Summary

Pymodbus is a full Modbus protocol implementation using twisted for its asynchronous communications core. It can also be used without any third party dependencies (aside from pyserial) if a more lightweight project is needed. Furthermore, it should work fine under any python version > 2.3 with a python 3.0 branch currently being maintained as well.

CHAPTER 2

Features

2.1 Client Features

- Full read/write protocol on discrete and register
- Most of the extended protocol (diagnostic/file/pipe/setting/information)
- TCP, UDP, Serial ASCII, Serial RTU, and Serial Binary
- asynchronous(powered by twisted) and synchronous versions
- Payload builder/decoder utilities

2.2 Server Features

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

CHAPTER 3

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:

http://riptideio.github.io/pymodbus/ or http://readthedocs.org/docs/pymodbus/en/latest/index.html

3.1 Example Code

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

```
from pymodbus.client.sync 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 respository. If you have created any utilities that meet a specific need, feel free to submit them so others can benefit.

Also, if you have questions, please ask them on the mailing list so that others can benefit from the results and so that I can trace them. I get a lot of email and sometimes these requests get lost in the noise: http://groups.google.com/group/pymodbus or at gitter: https://gitter.im/pymodbus_dev/Lobby

3.2 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):

```
easy_install -U pymodbus
pip install  -U pymodbus
```

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

```
git clone git://github.com/bashwork/pymodbus.git
cd pymodbus
python setup.py install
```

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

If you would like to install pymodbus without the twisted dependency, simply edit the setup.py file before running easy_install and comment out all mentions of twisted. It should be noted that without twisted, one will only be able to run the synchronized version as the asynchronous versions uses twisted for its event loop.

3.3 Current Work In Progress

Since I don't have access to any live modbus devices anymore it is a bit hard to test on live hardware. However, if you would like your device tested, I accept devices via mail or by IP address.

That said, the current work mainly involves polishing the library as I get time doing such tasks as:

- Make PEP-8 compatible and flake8 ready
- Fixing bugs/feature requests
- Architecture documentation
- Functional testing against any reference I can find
- The remaining edges of the protocol (that I think no one uses)

3.4 Development Instructions

The current code base is compatible with both py2 and py3. Use make to perform a range of activities

```
$ make
   Makefile for pymodbus

Usage:

 make install    install the package in a virtual environment
 make reset      recreate the virtual environment
 make check      check coding style (PEP-8, PEP-257)
 make test       run the test suite, report coverage
 make tox        run the tests on all Python versions
 make clean      cleanup all temporary files
```

3.5 Contributing

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

3.6 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
- Twisted Matrix

Released under the BSD License

CHAPTER 4

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 Asyncronous 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

CHAPTER 5

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 recieved without header and CRC
- Fix tests

CHAPTER 6

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

CHAPTER 7

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

CHAPTER 8

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

CHAPTER 9

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

CHAPTER 10

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

CHAPTER 11

Version 1.0.0

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

CHAPTER 12

Version 0.9.0

Please view the git commit log

CHAPTER 13

Examples

===

13.1 Asynchronous Client Example

```
#!/usr/bin/env python
"""
Pymodbus Asynchronous Client Examples
--------------------------------------------------------------------------

The following is an example of how to use the asynchronous modbus
client implementation from pymodbus.
"""
# --------------------------------------------------------------------------- #
# import needed libraries
# --------------------------------------------------------------------------- #
from twisted.internet import reactor, protocol
from pymodbus.constants import Defaults

# --------------------------------------------------------------------------- #
# choose the requested modbus protocol
# --------------------------------------------------------------------------- #
from pymodbus.client.async import ModbusClientProtocol
#from pymodbus.client.async import ModbusUdpClientProtocol

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

```
# --------------------------------------------------------------------------- #
# helper method to test deferred callbacks
# --------------------------------------------------------------------------- #


def dassert(deferred, callback):
    def _assertor(value):
        assert value

    deferred.addCallback(lambda r: _assertor(callback(r)))
    deferred.addErrback(lambda _: _assertor(False))

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


def exampleRequests(client):
    rr = client.read_coils(1, 1, unit=0x02)

# --------------------------------------------------------------------------- #
# example requests
# --------------------------------------------------------------------------- #
# simply call the methods that you would like to use. An example session
# is displayed below along with some assert checks. Note that unlike the
# synchronous version of the client, the asynchronous version returns
# deferreds which can be thought of as a handle to the callback to send
# the result of the operation.  We are handling the result using the
# deferred assert helper(dassert).
# --------------------------------------------------------------------------- #

UNIT = 0x01

def beginAsynchronousTest(client):
    rq = client.write_coil(1, True, unit=UNIT)
    rr = client.read_coils(1, 1, unit=UNIT)
    dassert(rq, lambda r: r.function_code < 0x80)     # test for no error
    dassert(rr, lambda r: r.bits[0] == True)          # test the expected value

    rq = client.write_coils(1, [True]*8, unit=UNIT)
    rr = client.read_coils(1, 8, unit=UNIT)
    dassert(rq, lambda r: r.function_code < 0x80)     # test for no error
    dassert(rr, lambda r: r.bits == [True]*8)         # test the expected value

    rq = client.write_coils(1, [False]*8, unit=UNIT)
    rr = client.read_discrete_inputs(1, 8, unit=UNIT)
    dassert(rq, lambda r: r.function_code < 0x80)     # test for no error
    dassert(rr, lambda r: r.bits == [True]*8)         # test the expected value

    rq = client.write_register(1, 10, unit=UNIT)
    rr = client.read_holding_registers(1, 1, unit=UNIT)
    dassert(rq, lambda r: r.function_code < 0x80)     # test for no error
    dassert(rr, lambda r: r.registers[0] == 10)       # test the expected value
```

```
    rq = client.write_registers(1, [10]*8, unit=UNIT)
    rr = client.read_input_registers(1, 8, unit=UNIT)
    dassert(rq, lambda r: r.function_code < 0x80)     # test for no error
    dassert(rr, lambda r: r.registers == [17]*8)      # test the expected value

    arguments = {
        'read_address':    1,
        'read_count':      8,
        'write_address':   1,
        'write_registers': [20]*8,
    }
    rq = client.readwrite_registers(**arguments, unit=UNIT)
    rr = client.read_input_registers(1, 8, unit=UNIT)
    dassert(rq, lambda r: r.registers == [20]*8)      # test the expected value
    dassert(rr, lambda r: r.registers == [17]*8)      # test the expected value

    # ------------------------------------------------------------------------ #
    # close the client at some time later
    # ------------------------------------------------------------------------ #
    reactor.callLater(1, client.transport.loseConnection)
    reactor.callLater(2, reactor.stop)

# ---------------------------------------------------------------------------- #
# extra requests
# ---------------------------------------------------------------------------- #
# 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
#
# ---------------------------------------------------------------------------- #

# ---------------------------------------------------------------------------- #
# 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 __name__ == "__main__":
    defer = protocol.ClientCreator(
        reactor, ModbusClientProtocol).connectTCP("localhost", 5020)
    defer.addCallback(beginAsynchronousTest)
    reactor.run()
```

13.2 Asynchronous Processor Example

```
#!/usr/bin/env python
"""
Pymodbus Asynchronous Processor Example
--------------------------------------------------------------------------

The following is a full example of a continuous client processor. Feel
free to use it as a skeleton guide in implementing your own.
"""
# --------------------------------------------------------------------------- #
# import the neccessary modules
# --------------------------------------------------------------------------- #
from twisted.internet import serialport, reactor
from twisted.internet.protocol import ClientFactory
from pymodbus.factory import ClientDecoder
from pymodbus.client.async import ModbusClientProtocol

# --------------------------------------------------------------------------- #
# Choose the framer you want to use
# --------------------------------------------------------------------------- #
#from pymodbus.transaction import ModbusBinaryFramer as ModbusFramer
#from pymodbus.transaction import ModbusAsciiFramer as ModbusFramer
from pymodbus.transaction import ModbusRtuFramer as ModbusFramer
# from pymodbus.transaction import ModbusSocketFramer as ModbusFramer

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

# --------------------------------------------------------------------------- #
# state a few constants
# --------------------------------------------------------------------------- #
SERIAL_PORT = "/dev/ttyp0"
STATUS_REGS = (1, 2)
STATUS_COILS = (1, 3)
CLIENT_DELAY = 1
UNIT = 0x01

# --------------------------------------------------------------------------- #
# an example custom protocol
# --------------------------------------------------------------------------- #
# Here you can perform your main procesing loop utilizing defereds and timed
# callbacks.
# --------------------------------------------------------------------------- #
class ExampleProtocol(ModbusClientProtocol):

    def __init__(self, framer, endpoint):
        """ Initializes our custom protocol

        :param framer: The decoder to use to process messages
        :param endpoint: The endpoint to send results to
        """
        ModbusClientProtocol.__init__(self, framer)
```

```
        self.endpoint = endpoint
        log.debug("Beginning the processing loop")
        reactor.callLater(CLIENT_DELAY, self.fetch_holding_registers)

    def fetch_holding_registers(self):
        """ Defer fetching holding registers
        """
        log.debug("Starting the next cycle")
        d = self.read_holding_registers(*STATUS_REGS, unit=UNIT)
        d.addCallbacks(self.send_holding_registers, self.error_handler)

    def send_holding_registers(self, response):
        """ Write values of holding registers, defer fetching coils

        :param response: The response to process
        """
        self.endpoint.write(response.getRegister(0))
        self.endpoint.write(response.getRegister(1))
        d = self.read_coils(*STATUS_COILS, unit=UNIT)
        d.addCallbacks(self.start_next_cycle, self.error_handler)

    def start_next_cycle(self, response):
        """ Write values of coils, trigger next cycle

        :param response: The response to process
        """
        self.endpoint.write(response.getBit(0))
        self.endpoint.write(response.getBit(1))
        self.endpoint.write(response.getBit(2))
        reactor.callLater(CLIENT_DELAY, self.fetch_holding_registers)

    def error_handler(self, failure):
        """ Handle any twisted errors

        :param failure: The error to handle
        """
        log.error(failure)


# --------------------------------------------------------------------------- #
# a factory for the example protocol
# --------------------------------------------------------------------------- #
# This is used to build client protocol's if you tie into twisted's method
# of processing. It basically produces client instances of the underlying
# protocol::
#
#     Factory(Protocol) -> ProtocolInstance
#
# It also persists data between client instances (think protocol singelton).
# --------------------------------------------------------------------------- #
class ExampleFactory(ClientFactory):

    protocol = ExampleProtocol

    def __init__(self, framer, endpoint):
        """ Remember things necessary for building a protocols """
        self.framer = framer
        self.endpoint = endpoint
```

```
    def buildProtocol(self, _):
        """ Create a protocol and start the reading cycle """
        proto = self.protocol(self.framer, self.endpoint)
        proto.factory = self
        return proto


# --------------------------------------------------------------------------- #
# a custom client for our device
# --------------------------------------------------------------------------- #
# Twisted provides a number of helper methods for creating and starting
# clients:
# - protocol.ClientCreator
# - reactor.connectTCP
#
# How you start your client is really up to you.
# --------------------------------------------------------------------------- #
class SerialModbusClient(serialport.SerialPort):

    def __init__(self, factory, *args, **kwargs):
        """ Setup the client and start listening on the serial port

        :param factory: The factory to build clients with
        """
        protocol = factory.buildProtocol(None)
        self.decoder = ClientDecoder()
        serialport.SerialPort.__init__(self, protocol, *args, **kwargs)


# --------------------------------------------------------------------------- #
# a custom endpoint for our results
# --------------------------------------------------------------------------- #
# An example line reader, this can replace with:
# - the TCP protocol
# - a context recorder
# - a database or file recorder
# --------------------------------------------------------------------------- #
class LoggingLineReader(object):

    def write(self, response):
        """ Handle the next modbus response

        :param response: The response to process
        """
        log.info("Read Data: %d" % response)

# --------------------------------------------------------------------------- #
# start running the processor
# --------------------------------------------------------------------------- #
# This initializes the client, the framer, the factory, and starts the
# twisted event loop (the reactor). It should be noted that a number of
# things could be chanegd as one sees fit:
# - The ModbusRtuFramer could be replaced with a ModbusAsciiFramer
# - The SerialModbusClient could be replaced with reactor.connectTCP
# - The LineReader endpoint could be replaced with a database store
# --------------------------------------------------------------------------- #
```

```
def main():
    log.debug("Initializing the client")
    framer = ModbusFramer(ClientDecoder())
    reader = LoggingLineReader()
    factory = ExampleFactory(framer, reader)
    SerialModbusClient(factory, SERIAL_PORT, reactor)
    # factory = reactor.connectTCP("localhost", 502, factory)
    log.debug("Starting the client")
    reactor.run()


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

13.3 Asynchronous Server Example

```
#!/usr/bin/env python
"""
Pymodbus Asynchronous Server Example
--------------------------------------------------------------------------

The asynchronous server is a high performance implementation using the
twisted library as its backend.  This allows it to scale to many thousands
of nodes which can be helpful for testing monitoring software.
"""
# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.server.async import StartTcpServer
from pymodbus.server.async import StartUdpServer
from pymodbus.server.async import StartSerialServer

from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext
from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer

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


def run_async_server():
    # --------------------------------------------------------------------------- #
    # initialize your data store
    # --------------------------------------------------------------------------- #
    # The datastores only respond to the addresses that they are initialized to
    # Therefore, if you initialize a DataBlock to addresses from 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)
    #
    #     block = ModbusSequentialDataBlock(0x00, [0]*0xff)
    #
    # Continuing, you can choose to use a sequential or a sparse DataBlock in
    # your data context.  The difference is that the sequential has no gaps in
    # the data while the sparse can. Once again, there are devices that exhibit
    # both forms of behavior::
    #
    #     block = ModbusSparseDataBlock({0x00: 0, 0x05: 1})
    #     block = ModbusSequentialDataBlock(0x00, [0]*5)
    #
    # Alternately, you can 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::
    #
    #     store = ModbusSlaveContext(di = ModbusSequentialDataBlock.create())
    #     store = ModbusSlaveContext()
    #
    # Finally, you are allowed to use the same DataBlock reference for every
    # table or you you may use a seperate DataBlock for each table.
    # This depends if you would like functions to be able to access and modify
    # the same data or not::
    #
    #     block = ModbusSequentialDataBlock(0x00, [0]*0xff)
    #     store = ModbusSlaveContext(di=block, co=block, hr=block, ir=block)
    #
    # 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::
    #
    #     slaves  = {
    #         0x01: ModbusSlaveContext(...),
    #         0x02: ModbusSlaveContext(...),
    #         0x03: ModbusSlaveContext(...),
    #     }
    #     context = ModbusServerContext(slaves=slaves, single=False)
    #
    # 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)::
    #
    #     store = ModbusSlaveContext(..., zero_mode=True)
    # ----------------------------------------------------------------------- #
    store = ModbusSlaveContext(
        di=ModbusSequentialDataBlock(0, [17]*100),
        co=ModbusSequentialDataBlock(0, [17]*100),
        hr=ModbusSequentialDataBlock(0, [17]*100),
        ir=ModbusSequentialDataBlock(0, [17]*100))
    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()
    identity.VendorName = 'Pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/bashwork/pymodbus/'
    identity.ProductName = 'Pymodbus Server'
    identity.ModelName = 'Pymodbus Server'
    identity.MajorMinorRevision = '1.0'

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

    StartTcpServer(context, identity=identity, address=("localhost", 5020))
    # StartUdpServer(context, identity=identity, address=("localhost", 502))
    # StartSerialServer(context, identity=identity,
    #                   port='/dev/pts/3', framer=ModbusRtuFramer)
    # StartSerialServer(context, identity=identity,
    #                   port='/dev/pts/3', framer=ModbusAsciiFramer)


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

13.4 Callback Server Example

```
#!/usr/bin/env python
"""
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 the modbus libraries we need
# ------------------------------------------------------------------------ #
from pymodbus.server.async import StartTcpServer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSparseDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext
from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer


# ------------------------------------------------------------------------ #
# import the python libraries we need
# ------------------------------------------------------------------------ #
from multiprocessing import Queue, Process

# ------------------------------------------------------------------------ #
# configure the service logging
# ------------------------------------------------------------------------ #
```

```
import logging
logging.basicConfig()
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):
        """
        """
        self.devices = devices
        self.queue = queue

        values = {k: 0 for k in devices.keys()}
        values[0xbeef] = len(values)  # the number of devices
        super(CallbackDataBlock, self).__init__(values)

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

        :param address: The starting address
        :param values: The new values to be set
        """
        super(CallbackDataBlock, self).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
    """
    s = 1 if value >= 50 else -1
    c = value if value < 50 else (value - 50)
    return s * (c * 64)


def device_writer(queue):
    """ A worker process that processes 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()
        scaled = rescale_value(value[0])
        log.debug("Write(%s) = %s" % (device, value))
        if not device: continue
        # do any logic here to update your devices

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


def read_device_map(path):
    """ A helper method to read the device
    path to address mapping from file::

       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:
        for line in stream:
            piece = line.strip().split(',')
            devices[int(piece[0], 16)] = piece[1]
    return devices


def 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()
    identity.VendorName = 'pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/bashwork/pymodbus/'
    identity.ProductName = 'pymodbus Server'
    identity.ModelName = 'pymodbus Server'
    identity.MajorMinorRevision = '1.0'

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

13.5 Changing Framers Example

```
#!/usr/bin/env python
"""
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 gurantees 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 the modbus client and the framers
# --------------------------------------------------------------------------- #
from pymodbus.client.sync import ModbusTcpClient as ModbusClient

# --------------------------------------------------------------------------- #
# Import the modbus framer that you want
# --------------------------------------------------------------------------- #
# --------------------------------------------------------------------------- #
#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
# --------------------------------------------------------------------------- #
import logging
logging.basicConfig()
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(rq.function_code < 0x80)     # test that we are not an error
    assert(rr.bits[0] == True)          # test the expected value

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

13.6 Custom Datablock Example

```
#!/usr/bin/env python
"""
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 the modbus libraries we need
# ------------------------------------------------------------------------ #
from __future__ import print_function
from pymodbus.server.async import StartTcpServer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSparseDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext
from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer

# ------------------------------------------------------------------------ #
# configure the service logging
# ------------------------------------------------------------------------ #

import logging
logging.basicConfig()
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.
    """

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

        :param address: The starting address
        :param values: The new values to be set
        """
```

```
        super(ModbusSparseDataBlock, self).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("wrote {} to {}".format(value, address))


def 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()
    identity.VendorName = 'pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/bashwork/pymodbus/'
    identity.ProductName = 'pymodbus Server'
    identity.ModelName = 'pymodbus Server'
    identity.MajorMinorRevision = '1.0'

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

13.7 Custom Message Example

```
#!/usr/bin/env python
"""
Pymodbus Synchrnonous Client Examples
--------------------------------------------------------------------------

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

It should be noted that the client can also be used with
the guard construct that is available in python 2.5 and up::
```

```
    with ModbusClient('127.0.0.1') as client:
        result = client.read_coils(1,10)
        print result
"""
import struct
# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.pdu import ModbusRequest, ModbusResponse, ModbusExceptions
from pymodbus.client.sync import ModbusTcpClient as ModbusClient
from pymodbus.bit_read_message import ReadCoilsRequest
# --------------------------------------------------------------------------- #
# configure the client logging
# --------------------------------------------------------------------------- #
import logging
logging.basicConfig()
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):
    pass


class CustomModbusRequest(ModbusRequest):

    function_code = 1

    def __init__(self, address):
        ModbusRequest.__init__(self)
        self.address = address
        self.count = 16

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

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

    def execute(self, context):
        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)
```

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


class Read16CoilsRequest(ReadCoilsRequest):

    def __init__(self, address):
        """ Initializes a new instance

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

# --------------------------------------------------------------------------- #
# 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('127.0.0.1') as client:
        request = CustomModbusRequest(0)
        result = client.execute(request)
        print(result)
```

13.8 Dbstore Update Server Example

```
"""
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 the modbus libraries we need
# --------------------------------------------------------------------------- #
from pymodbus.server.async import StartTcpServer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusServerContext
from pymodbus.datastore.database import SqlSlaveContext
from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer
```

```
import random

# --------------------------------------------------------------------------- #
# import the twisted libraries we need
# --------------------------------------------------------------------------- #
from twisted.internet.task import LoopingCall

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

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


def updating_writer(a):
    """ A worker process that runs 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  = a[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)
    rand_addr = random.randint(0, 65000)
    log.debug("Writing to datastore: {}, {}".format(rand_addr, rand_value))
    # import pdb; pdb.set_trace()
    context[slave_id].setValues(writefunction, rand_addr, [rand_value])
    values = context[slave_id].getValues(readfunction, rand_addr, count)
    log.debug("Values from datastore: " + str(values))


def 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()
```

```
    identity.VendorName = 'pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/bashwork/pymodbus/'
    identity.ProductName = 'pymodbus Server'
    identity.ModelName = 'pymodbus Server'
    identity.MajorMinorRevision = '1.0'

    # ------------------------------------------------------------------------ #
    # run the server you want
    # ------------------------------------------------------------------------ #
    time = 5  # 5 seconds delay
    loop = LoopingCall(f=updating_writer, a=(context,))
    loop.start(time, now=False)  # initially delay by time
    StartTcpServer(context, identity=identity, address=("", 5020))


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

13.9 Modbus Logging Example

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

if __name__ == "__main__":
    # ------------------------------------------------------------------------ #
    # This will simply send everything logged to console
    # ------------------------------------------------------------------------ #
    logging.basicConfig()
    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
    # * pymodbus.client.*   - all logging messages involving the client
    # * pymodbus.protocol.* - all logging messages inside the protocol layer
    # ------------------------------------------------------------------------ #
    logging.basicConfig()
    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]
```

13.10 Modbus Payload Example

```
#!/usr/bin/env python
"""
Pymodbus Payload Building/Decoding Example
--------------------------------------------------------------------------

# Run modbus-payload-server.py or synchronous-server.py to check the behavior
"""
from pymodbus.constants import Endian
from pymodbus.payload import BinaryPayloadDecoder
from pymodbus.payload import BinaryPayloadBuilder
from pymodbus.client.sync import ModbusTcpClient as ModbusClient
from pymodbus.compat import iteritems

# ------------------------------------------------------------------------ #
# configure the client logging
# ------------------------------------------------------------------------ #
import logging
logging.basicConfig()
log = logging.getLogger()
log.setLevel(logging.INFO)


def run_binary_payload_ex():
    # ------------------------------------------------------------------------ #
    # We are going to use a simple client to send our requests
    # ------------------------------------------------------------------------ #
    client = ModbusClient('127.0.0.1', port=5440)
    client.connect()

    # ------------------------------------------------------------------------ #
    # If you need to build a complex message to send, you can use the payload
    # builder to simplify the packing logic.
    #
    # Here we demonstrate packing a random payload layout, unpacked it looks
    # like the following:
    #
    # - a 8 byte string 'abcdefgh'
    # - a 32 bit float 22.34
```

```
    # - a 16 bit unsigned int 0x1234
    # - another 16 bit unsigned int 0x5678
    # - an 8 bit int 0x12
    # - an 8 bit bitstring [0,1,0,1,1,0,1,0]
    # - an 32 bit uint 0x12345678
    # - an 32 bit signed int -0x1234
    # - an 64 bit signed int 0x12345678

    # The packing can also be applied to the word (wordorder) and bytes in each
    # word (byteorder)

    # 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 - Big                      Byte Order - Big
    # word1 =0x1234 word2 = 0x5678

    # Word Order - Big                      Byte Order - Little
    # word1 =0x3412 word2 = 0x7856

    # Word Order - Little                   Byte Order - Big
    # word1 = 0x5678 word2 = 0x1234

    # Word Order - Little                   Byte Order - Little
    # word1 =0x7856 word2 = 0x3412
    # ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++ #

    # ---------------------------------------------------------------------- #
    builder = BinaryPayloadBuilder(byteorder=Endian.Little,
                                   wordorder=Endian.Big)
    builder.add_string('abcdefgh')
    builder.add_32bit_float(22.34)
    builder.add_16bit_uint(0x1234)
    builder.add_16bit_uint(0x5678)
    builder.add_8bit_int(0x12)
    builder.add_bits([0, 1, 0, 1, 1, 0, 1, 0])
    builder.add_32bit_uint(0x12345678)
    builder.add_32bit_int(-0x1234)
    builder.add_64bit_int(0x1234567890ABCDEF)
    payload = builder.build()
    address = 0
    client.write_registers(address, payload, skip_encode=True, unit=1)
    # ---------------------------------------------------------------------- #
    # If you need to decode a collection of registers in a weird layout, the
    # payload decoder can help you as well.
    #
    # Here we demonstrate decoding a random register layout, unpacked it looks
    # like the following:
    #
    # - a 8 byte string 'abcdefgh'
    # - a 32 bit float 22.34
    # - a 16 bit unsigned int 0x1234
    # - another 16 bit unsigned int which we will ignore
    # - an 8 bit int 0x12
    # - an 8 bit bitstring [0,1,0,1,1,0,1,0]
```

```
    # ------------------------------------------------------------------------ #
    address = 0x00
    count = len(payload)
    result = client.read_holding_registers(address, count,  unit=1)
    print("-" * 60)
    print("Registers")
    print("-" * 60)
    print(result.registers)
    print("\n")
    decoder = BinaryPayloadDecoder.fromRegisters(result.registers,
                                                 byteorder=Endian.Little,
                                                 wordorder=Endian.Big)
    decoded = {
        'string': decoder.decode_string(8),
        'float': decoder.decode_32bit_float(),
        '16uint': decoder.decode_16bit_uint(),
        'ignored': decoder.skip_bytes(2),
        '8int': decoder.decode_8bit_int(),
        'bits': decoder.decode_bits(),
        "32uints": decoder.decode_32bit_uint(),
        "32ints": decoder.decode_32bit_int(),
        "64ints": decoder.decode_64bit_int(),
    }

    print("-" * 60)
    print("Decoded Data")
    print("-" * 60)
    for name, value in iteritems(decoded):
        print("%s\t" % name, hex(value) if isinstance(value, int) else value)

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


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

13.11 Modbus Payload Server Example

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

If you want to initialize a server context with a complicated memory
layout, you can actually use the payload builder.
"""
# ------------------------------------------------------------------------ #
# import the various server implementations
# ------------------------------------------------------------------------ #
from pymodbus.server.sync import StartTcpServer

from pymodbus.device import ModbusDeviceIdentification
```

```
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext

# --------------------------------------------------------------------------- #
# import the payload builder
# --------------------------------------------------------------------------- #

from pymodbus.constants import Endian
from pymodbus.payload import BinaryPayloadDecoder
from pymodbus.payload import BinaryPayloadBuilder

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


def run_payload_server():
    # --------------------------------------------------------------------------- #
    # build your payload
    # --------------------------------------------------------------------------- #
    builder = BinaryPayloadBuilder(byteorder=Endian.Little)
    # builder.add_string('abcdefgh')
    # builder.add_32bit_float(22.34)
    # builder.add_16bit_uint(4660)
    # builder.add_8bit_int(18)
    builder.add_bits([0, 1, 0, 1, 1, 0, 1, 0])

    # --------------------------------------------------------------------------- #
    # 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)
    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()
    identity.VendorName = 'Pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/bashwork/pymodbus/'
    identity.ProductName = 'Pymodbus Server'
    identity.ModelName = 'Pymodbus Server'
    identity.MajorMinorRevision = '1.0'
    # --------------------------------------------------------------------------- #
    # run the server you want
    # --------------------------------------------------------------------------- #
    StartTcpServer(context, identity=identity, address=("localhost", 5020))
```

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

13.12 performance module

```
#!/usr/bin/env python
"""
Pymodbus Performance Example
--------------------------------------------------------------------------

The following is an quick performance check of the synchronous
modbus client.
"""
# --------------------------------------------------------------------------- #
# import the necessary modules
#---------------------------------------------------------------------------#
from __future__ import print_function
import logging, os
from time import time
# from pymodbus.client.sync import ModbusTcpClient
from pymodbus.client.sync import ModbusSerialClient

try:
    from multiprocessing import log_to_stderr
except ImportError:
    import logging
    logging.basicConfig()
    log_to_stderr = logging.getLogger

# --------------------------------------------------------------------------- #
# choose between threads or processes
# --------------------------------------------------------------------------- #

#from multiprocessing import Process as Worker
from threading import Thread as Worker
from threading import Lock
_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
cycles = 1000
host = '127.0.0.1'


# --------------------------------------------------------------------------- #
# 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(host, cycles):
    """ Performs a single threaded test of a synchronous
    client against the specified host

    :param host: The host to connect to
    :param cycles: The number of iterations to perform
    """
    logger = log_to_stderr()
    logger.setLevel(logging.DEBUG)
    logger.debug("starting worker: %d" % os.getpid())

    try:
        count = 0
        # client = ModbusTcpClient(host, port=5020)
        client = ModbusSerialClient(method="rtu",
                                    port="/dev/ttyp0", baudrate=9600)
        while count < cycles:
            with _thread_lock:
                client.read_holding_registers(10, 1, unit=1).registers[0]
                count += 1
    except:
        logger.exception("failed to run test successfully")
    logger.debug("finished worker: %d" % os.getpid())

# ---------------------------------------------------------------------------- #
# 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

# ---------------------------------------------------------------------------- #


if __name__ == "__main__":
    args = (host, int(cycles * 1.0 / workers))
    procs = [Worker(target=single_client_test, args=args)
             for _ in range(workers)]
    start = time()
    any(p.start() for p in procs)    # start the workers
    any(p.join() for p in procs)   # wait for the workers to finish
    stop = time()
    print("%d requests/second" % ((1.0 * cycles) / (stop - start)))
    print("time taken to complete %s cycle by "
          "%s workers is %s seconds" % (cycles, workers, stop-start))
```

13.13 Synchronous Client Example

```
#!/usr/bin/env python
"""
Pymodbus Synchronous Client Examples
--------------------------------------------------------------------------

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

It should be noted that the client can also be used with
the guard construct that is available in python 2.5 and up::

    with ModbusClient('127.0.0.1') as client:
        result = client.read_coils(1,10)
        print result
"""
# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.client.sync import ModbusTcpClient as ModbusClient
#from pymodbus.client.sync import ModbusUdpClient as ModbusClient
# from pymodbus.client.sync import ModbusSerialClient as ModbusClient

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

UNIT = 0x1


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

```
    #
    # Here is an example of using these options::
    #
    #    client = ModbusClient('localhost', retries=3, retry_on_empty=True)
    # -----------------------------------------------------------------------#
    client = ModbusClient('localhost', port=5020)
    # client = ModbusClient(method='ascii', port='/dev/pts/2', timeout=1)
    # client = ModbusClient(method='rtu', port='/dev/ttyp0', timeout=1)
    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 Coils")
    rr = client.read_coils(1, 1, unit=0x01)
    print(rr)

    # ----------------------------------------------------------------------- #
    # example requests
    # ----------------------------------------------------------------------- #
    # simply call the methods that you would like to use. An example session
    # is displayed below along with some assert checks. Note that some modbus
    # implementations differentiate holding/input discrete/coils and as such
    # you will not be able to write to these, therefore the starting values
    # are not known to these tests. Furthermore, some use the same memory
    # blocks for the two sets, so a change to one is a change to the other.
    # Keep both of these cases in mind when testing as the following will
    # _only_ pass with the supplied async modbus server (script supplied).
    # ----------------------------------------------------------------------- #
    log.debug("Write to a Coil and read back")
    rq = client.write_coil(0, True, unit=UNIT)
    rr = client.read_coils(0, 1, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error
    assert(rr.bits[0] == True)          # test the expected value

    log.debug("Write to multiple coils and read back- test 1")
    rq = client.write_coils(1, [True]*8, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error
    rr = client.read_coils(1, 21, unit=UNIT)
    assert(rr.function_code < 0x80)     # test that we are not an error
    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

    log.debug("Write to multiple coils and read back - test 2")
    rq = client.write_coils(1, [False]*8, unit=UNIT)
    rr = client.read_coils(1, 8, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error
    assert(rr.bits == [False]*8)         # test the expected value
```

```
    log.debug("Read discrete inputs")
    rr = client.read_discrete_inputs(0, 8, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error

    log.debug("Write to a holding register and read back")
    rq = client.write_register(1, 10, unit=UNIT)
    rr = client.read_holding_registers(1, 1, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error
    assert(rr.registers[0] == 10)       # test the expected value

    log.debug("Write to multiple holding registers and read back")
    rq = client.write_registers(1, [10]*8, unit=UNIT)
    rr = client.read_holding_registers(1, 8, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error
    assert(rr.registers == [10]*8)      # test the expected value

    log.debug("Read input registers")
    rr = client.read_input_registers(1, 8, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error

    arguments = {
        'read_address':    1,
        'read_count':      8,
        'write_address':   1,
        'write_registers': [20]*8,
    }
    log.debug("Read write registeres simulataneously")
    rq = client.readwrite_registers(unit=UNIT, **arguments)
    rr = client.read_holding_registers(1, 8, unit=UNIT)
    assert(rq.function_code < 0x80)     # test that we are not an error
    assert(rq.registers == [20]*8)      # test the expected value
    assert(rr.registers == [20]*8)      # test the expected value

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


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

13.14 Synchronous Client Ext Example

```
#!/usr/bin/env python
"""
Pymodbus Synchronous Client Extended Examples
--------------------------------------------------------------------------

The following is an example of how to use the synchronous modbus client
implementation from pymodbus to perform the extended portions of the
modbus protocol.
"""
# ------------------------------------------------------------------------ #
```

```
# import the various server implementations
# --------------------------------------------------------------------------- #
# from pymodbus.client.sync import ModbusTcpClient as ModbusClient
# from pymodbus.client.sync import ModbusUdpClient as ModbusClient
from pymodbus.client.sync import ModbusSerialClient as ModbusClient

# --------------------------------------------------------------------------- #
# import the extended messages to perform
# --------------------------------------------------------------------------- #
from pymodbus.diag_message import *
from pymodbus.file_message import *
from pymodbus.other_message import *
from pymodbus.mei_message import *

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

UNIT = 0x01


def execute_extended_requests():
    # ------------------------------------------------------------------------#
    # 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.
    #
    # It should be noted that you can supply an ipv4 or an ipv6 host address
    # for both the UDP and TCP clients.
    # ------------------------------------------------------------------------#
    client = ModbusClient(method='rtu', port="/dev/ttyp0")
    # client = ModbusClient('127.0.0.1', port=5020)
    client.connect()

    # --------------------------------------------------------------------------- #
    # extra requests
    # --------------------------------------------------------------------------- #
    # 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
    #
    #
    # What follows is a listing of all the supported methods. Feel free to
    # comment, uncomment, or modify each result set to match with your ref.
    # --------------------------------------------------------------------------- #
```

```
    # ------------------------------------------------------------------------ #
    # information requests
    # ------------------------------------------------------------------------ #
    log.debug("Running ReadDeviceInformationRequest")
    rq = ReadDeviceInformationRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)              # not supported by reference
    # assert (rr.function_code < 0x80)  # test that we are not an error
    # assert (rr.information[0] == b'Pymodbus')  # test the vendor name
    # assert (rr.information[1] == b'PM')  # test the product code
    # assert (rr.information[2] == b'1.0')  # test the code revision

    log.debug("Running ReportSlaveIdRequest")
    rq = ReportSlaveIdRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                          # not supported by reference
    # assert(rr.function_code < 0x80)             # test that we are not an error
    # assert(rr.identifier  == 0x00)              # test the slave identifier
    # assert(rr.status  == 0x00)                  # test that the status is ok

    log.debug("Running ReadExceptionStatusRequest")
    rq = ReadExceptionStatusRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                          # not supported by reference
    # assert(rr.function_code < 0x80)             # test that we are not an error
    # assert(rr.status == 0x55)                   # test the status code

    log.debug("Running GetCommEventCounterRequest")
    rq = GetCommEventCounterRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                         # not supported by reference
    # assert(rr.function_code < 0x80)            # test that we are not an error
    # assert(rr.status == True)                  # test the status code
    # assert(rr.count == 0x00)                   # test the status code

    log.debug("Running GetCommEventLogRequest")
    rq = GetCommEventLogRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                         # not supported by reference
    # assert(rr.function_code < 0x80)            # test that we are not an error
    # assert(rr.status == True)                  # test the status code
    # assert(rr.event_count == 0x00)             # test the number of events
    # assert(rr.message_count == 0x00)           # test the number of messages
    # assert(len(rr.events) == 0x00)             # test the number of events

    # ------------------------------------------------------------------------#
    # diagnostic requests
    # ------------------------------------------------------------------------#
    log.debug("Running ReturnQueryDataRequest")
    rq = ReturnQueryDataRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
```

```
    # assert(rr == None)                        # not supported by reference
    # assert(rr.message[0] == 0x0000)           # test the resulting message

    log.debug("Running RestartCommunicationsOptionRequest")
    rq = RestartCommunicationsOptionRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                      # not supported by reference
    # assert(rr.message == 0x0000)            # test the resulting message

    log.debug("Running ReturnDiagnosticRegisterRequest")
    rq = ReturnDiagnosticRegisterRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                      # not supported by reference

    log.debug("Running ChangeAsciiInputDelimiterRequest")
    rq = ChangeAsciiInputDelimiterRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                     # not supported by reference

    log.debug("Running ForceListenOnlyModeRequest")
    rq = ForceListenOnlyModeRequest(unit=UNIT)
    rr = client.execute(rq)  # does not send a response
    print(rr)

    log.debug("Running ClearCountersRequest")
    rq = ClearCountersRequest()
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                    # not supported by reference

    log.debug("Running ReturnBusCommunicationErrorCountRequest")
    rq = ReturnBusCommunicationErrorCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                     # not supported by reference

    log.debug("Running ReturnBusExceptionErrorCountRequest")
    rq = ReturnBusExceptionErrorCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                    # not supported by reference

    log.debug("Running ReturnSlaveMessageCountRequest")
    rq = ReturnSlaveMessageCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                   # not supported by reference

    log.debug("Running ReturnSlaveNoResponseCountRequest")
    rq = ReturnSlaveNoResponseCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                   # not supported by reference

    log.debug("Running ReturnSlaveNAKCountRequest")
```

```
    rq = ReturnSlaveNAKCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                # not supported by reference

    log.debug("Running ReturnSlaveBusyCountRequest")
    rq = ReturnSlaveBusyCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                # not supported by reference

    log.debug("Running ReturnSlaveBusCharacterOverrunCountRequest")
    rq = ReturnSlaveBusCharacterOverrunCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                # not supported by reference

    log.debug("Running ReturnIopOverrunCountRequest")
    rq = ReturnIopOverrunCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                # not supported by reference

    log.debug("Running ClearOverrunCountRequest")
    rq = ClearOverrunCountRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                # not supported by reference

    log.debug("Running GetClearModbusPlusRequest")
    rq = GetClearModbusPlusRequest(unit=UNIT)
    rr = client.execute(rq)
    print(rr)
    # assert(rr == None)                # not supported by reference

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


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

13.15 Synchronous Server Example

```
#!/usr/bin/env python
"""
Pymodbus Synchronous Server Example
--------------------------------------------------------------------------
```

```
The synchronous server is implemented in pure python without any third
party libraries (unless you need to use the serial protocols which require
pyserial). This is helpful in constrained or old environments where using
twisted just is not feasable. What follows is an examle of its use:
"""
# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.server.sync import StartTcpServer
from pymodbus.server.sync import StartUdpServer
from pymodbus.server.sync import StartSerialServer

from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext

from pymodbus.transaction import ModbusRtuFramer
# --------------------------------------------------------------------------- #
# configure the service logging
# --------------------------------------------------------------------------- #
import logging
logging.basicConfig()
log = logging.getLogger()
log.setLevel(logging.DEBUG)


def run_server():
    # --------------------------------------------------------------------------- #
    # initialize your data store
    # --------------------------------------------------------------------------- #
    # The datastores only respond to the addresses that they are initialized to
    # Therefore, 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)::
    #
    #     block = ModbusSequentialDataBlock(0x00, [0]*0xff)
    #
    # Continuing, you can choose to use a sequential or a sparse DataBlock in
    # your data context.  The difference is that the sequential has no gaps in
    # the data while the sparse can. Once again, there are devices that exhibit
    # both forms of behavior::
    #
    #     block = ModbusSparseDataBlock({0x00: 0, 0x05: 1})
    #     block = ModbusSequentialDataBlock(0x00, [0]*5)
    #
    # Alternately, you can 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::
    #
    #     store = ModbusSlaveContext(di = ModbusSequentialDataBlock.create())
    #     store = ModbusSlaveContext()
    #
    # Finally, you are allowed to use the same DataBlock reference for every
    # table or you you may use a seperate DataBlock for each table.
    # This depends if you would like functions to be able to access and modify
    # the same data or not::
    #
```

```
    #     block = ModbusSequentialDataBlock(0x00, [0]*0xff)
    #     store = ModbusSlaveContext(di=block, co=block, hr=block, ir=block)
    #
    # 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::
    #
    #     slaves  = {
    #         0x01: ModbusSlaveContext(...),
    #         0x02: ModbusSlaveContext(...),
    #         0x03: ModbusSlaveContext(...),
    #     }
    #     context = ModbusServerContext(slaves=slaves, single=False)
    #
    # 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)::
    #
    #     store = ModbusSlaveContext(..., zero_mode=True)
    # --------------------------------------------------------------------------- #
    store = ModbusSlaveContext(
        di = ModbusSequentialDataBlock(0, [17]*100),
        co = ModbusSequentialDataBlock(0, [17]*100),
        hr = ModbusSequentialDataBlock(0, [17]*100),
        ir = ModbusSequentialDataBlock(0, [17]*100))
    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()
    identity.VendorName = 'Pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/riptideio/pymodbus/'
    identity.ProductName = 'Pymodbus Server'
    identity.ModelName = 'Pymodbus Server'
    identity.MajorMinorRevision = '1.0'

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

    # Udp:
    # StartUdpServer(context, identity=identity, address=("localhost", 502))

    # Ascii:
    #  StartSerialServer(context, identity=identity,
    #                    port='/dev/pts/3', timeout=1)

    # RTU:
```

```
    # StartSerialServer(context, framer=ModbusRtuFramer, identity=identity,
    #                   port='/dev/ptyp0', timeout=.005, baudrate=9600)


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

13.16 Updating Server Example

```
#!/usr/bin/env python
"""
Pymodbus Server With Updating Thread
--------------------------------------------------------------------------

This is an example of having a background thread updating the
context while the server is operating. This can also be done with
a python thread::

    from threading import Thread

    thread = Thread(target=updating_writer, args=(context,))
    thread.start()
"""
# --------------------------------------------------------------------------- #
# import the modbus libraries we need
# --------------------------------------------------------------------------- #
from pymodbus.server.async import StartTcpServer
from pymodbus.device import ModbusDeviceIdentification
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext
from pymodbus.transaction import ModbusRtuFramer, ModbusAsciiFramer

# --------------------------------------------------------------------------- #
# import the twisted libraries we need
# --------------------------------------------------------------------------- #
from twisted.internet.task import LoopingCall

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

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


def updating_writer(a):
    """ A worker process that runs every so often and
    updates live values of the context. 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 context")
    context = a[0]
    register = 3
    slave_id = 0x00
    address = 0x10
    values = context[slave_id].getValues(register, address, count=5)
    values = [v + 1 for v in values]
    log.debug("new values: " + str(values))
    context[slave_id].setValues(register, address, values)


def run_updating_server():
    # ----------------------------------------------------------------------- #
    # initialize your data store
    # ----------------------------------------------------------------------- #

    store = ModbusSlaveContext(
        di=ModbusSequentialDataBlock(0, [17]*100),
        co=ModbusSequentialDataBlock(0, [17]*100),
        hr=ModbusSequentialDataBlock(0, [17]*100),
        ir=ModbusSequentialDataBlock(0, [17]*100))
    context = ModbusServerContext(slaves=store, single=True)

    # ----------------------------------------------------------------------- #
    # initialize the server information
    # ----------------------------------------------------------------------- #
    identity = ModbusDeviceIdentification()
    identity.VendorName = 'pymodbus'
    identity.ProductCode = 'PM'
    identity.VendorUrl = 'http://github.com/bashwork/pymodbus/'
    identity.ProductName = 'pymodbus Server'
    identity.ModelName = 'pymodbus Server'
    identity.MajorMinorRevision = '1.0'

    # ----------------------------------------------------------------------- #
    # run the server you want
    # ----------------------------------------------------------------------- #
    time = 5  # 5 seconds delay
    loop = LoopingCall(f=updating_writer, a=(context,))
    loop.start(time, now=False) # initially delay by time
    StartTcpServer(context, identity=identity, address=("localhost", 5020))


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

13.17 Bcd Payload Example

```
"""
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, unpack
from pymodbus.constants import Endian
from pymodbus.interfaces import IPayloadBuilder
from pymodbus.utilities import pack_bitstring
from pymodbus.utilities import unpack_bitstring
from pymodbus.exceptions import ParameterException
from pymodbus.payload import BinaryPayloadDecoder


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

    :param value: The decimal value to to pack into bcd
    :returns: The number in bcd form
    """
    place, bcd = 0, 0
    while decimal > 0:
        nibble = decimal % 10
        bcd += nibble << place
        decimal /= 10
        place += 4
    return bcd


def convert_from_bcd(bcd):
    """ Converts 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

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

    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):
        """ Adds 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):
        """ Adds 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
            encoded.append(pack('B', nibble))
            value >>= 4
            size -= 1
        self._payload.extend(encoded)

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

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


class BcdPayloadDecoder(object):
    """
    A utility that helps decode binary coded decimal payload
    messages from a modbus reponse 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):
        """ 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 endianess 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
    def fromCoils(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 endianess 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):
        """ Decodes 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):
        """ Decodes 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):
        """ Decodes 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"]
```

13.18 Concurrent Client Example

```
#!/usr/bin/env python
"""
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 system libraries
# --------------------------------------------------------------------------- #
import multiprocessing
import threading
import itertools
from collections import namedtuple

from pymodbus.compat import IS_PYTHON3

# we are using the future from the concurrent.futures released with
# python3. Alternatively we will try the backported library::
#   pip install futures
try:
    from concurrent.futures import Future
except ImportError:
    from futures import Future

# --------------------------------------------------------------------------- #
# import neccessary modbus libraries
# --------------------------------------------------------------------------- #
from pymodbus.client.common import ModbusClientMixin

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


# --------------------------------------------------------------------------- #
# Initialize out concurrency primitives
# --------------------------------------------------------------------------- #
class _Primitives(object):
    """ This is a helper class used to group the
    threading primitives depending on the type of
    worker situation we want to run (threads or processes).
    """

    def __init__(self, **kwargs):
        self.queue  = kwargs.get('queue')
        self.event  = kwargs.get('event')
        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:
            if IS_PYTHON3:
                from queue import Queue
            else:
                from Queue import Queue
            from threading import Thread
            from threading import Event
            return cls(queue=Queue, event=Event, worker=Thread)
        else:
            from multiprocessing import Queue
            from multiprocessing import Event
            from multiprocessing import Process
            return cls(queue=Queue, event=Event, worker=Process)


# --------------------------------------------------------------------------- #
# 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):
    """ This worker process takes 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
    """
    log.info("starting up worker : %s", threading.current_thread())
    client = factory()
    while not is_shutdown.is_set():
        try:
            workitem = input_queue.get(timeout=1)
            log.debug("dequeue worker request: %s", workitem)
            if not workitem: continue
            try:
                log.debug("executing request on thread: %s", workitem)
                result = client.execute(workitem.request)
                output_queue.put(WorkResponse(False, workitem.work_id, result))
            except Exception as exception:
                log.exception("error in worker "
                              "thread: %s", threading.current_thread())
                output_queue.put(WorkResponse(True,
```

```
                                            workitem.work_id, exception))
        except Exception as ex:
            pass
    log.info("request worker shutting down: %s", threading.current_thread())


def _manager_worker_process(output_queue, futures, is_shutdown):
    """ This worker process manages taking 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
    """
    log.info("starting up manager worker: %s", threading.current_thread())
    while not is_shutdown.is_set():
        try:
            workitem = output_queue.get()
            future = futures.get(workitem.work_id, None)
            log.debug("dequeue manager response: %s", workitem)
            if not future: continue
            if workitem.is_exception:
                future.set_exception(workitem.response)
            else: future.set_result(workitem.response)
            log.debug("updated future result: %s", future)
            del futures[workitem.work_id]
        except Exception as ex:
            log.exception("error in manager")
    log.info("manager worker shutting down: %s", threading.current_thread())


# --------------------------------------------------------------------------- #
# Define our concurrent client
# --------------------------------------------------------------------------- #
class ConcurrentClient(ModbusClientMixin):
    """ This is a high performance client that can be used
    to read/write a large number of reqeusts at once asyncronously.
    This operates with a backing worker pool of processes or threads
    to achieve its performance.
    """

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

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

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

    def shutdown(self):
        """ Shutdown all the workers being 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(), self.counter.next()
        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.sync import ModbusTcpClient

    def client_factory():
```

```
        log.debug("creating client for: %s", threading.current_thread())
        client = ModbusTcpClient('127.0.0.1', port=5020)
        client.connect()
        return client

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

13.19 Libmodbus Client Example

```
#!/usr/bin/env python
"""
Libmodbus Protocol Wrapper
------------------------------------------------------------

What follows is an example wrapper of the libmodbus library
(http://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
  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

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

from pymodbus.constants import Defaults
from pymodbus.exceptions import ModbusException
```

```
from pymodbus.client.common import ModbusClientMixin
from pymodbus.bit_read_message import ReadCoilsResponse, ReadDiscreteInputsResponse
from pymodbus.register_read_message import ReadHoldingRegistersResponse,␣
↪ReadInputRegistersResponse
from pymodbus.register_read_message import ReadWriteMultipleRegistersResponse
from pymodbus.bit_write_message import WriteSingleCoilResponse,␣
↪WriteMultipleCoilsResponse
from pymodbus.register_write_message import WriteSingleRegisterResponse,␣
↪WriteMultipleRegistersResponse

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

compiler = FFI()
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 utilites
# --------------------------------------------------------------------------- #


def get_float(data):
    return LIB.modbus_get_float(data)


def set_float(value, data):
    LIB.modbus_set_float(value, data)


def cast_to_int16(data):
    return int(compiler.cast('int16_t', data))


def cast_to_int32(data):
    return int(compiler.cast('int32_t', data))


class NotImplementedException(Exception):
    pass

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


class LibmodbusLevel1Client(object):
    """ 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(klass, host='127.0.0.1', port=Defaults.Port):
        """ 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
        """
        client = LIB.modbus_new_tcp(host.encode(), port)
        return klass(client)

    @classmethod
    def create_rtu_client(klass, **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
            :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
        """
        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)
        client = LIB.modbus_new_rtu(port, baudrate, parity, bytesize, stopbits)
        return klass(client)

    def __init__(self, client):
        """ Initalize 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 = client
        self.slave  = Defaults.UnitId

    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(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 (self.__execute(LIB.modbus_connect) == 0)

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

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

    def close(self):
        """ Closes 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
        """
        result = command(self.client, *args)
        if result == -1:
            message = LIB.modbus_strerror(compiler.errno)
            raise ModbusException(compiler.string(message))
        return result

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

        :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):
        """

        :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):
        """

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

    def write_bits(self, address, values):
        """

        :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):
        """
```

```
        :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):
        """

        :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):
        """

        :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):
        """

        :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)
        return result

    def read_and_write_registers(self, read_address, read_count, write_address, 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),
        '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, client):
        """ Initalize a new instance of the LibmodbusClient. This should
        be initialized with one of the LibmodbusLevel1Client instances:

        * LibmodbusLevel1Client.create_rtu_client(...)
        * LibmodbusLevel1Client.create_tcp_client(...)
        :param client: The underlying client instance to operate with.
        """
        self.client = 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):
        return self.client.connect()

    def close(self):
        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

    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'
    port = 502
    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)
```

13.20 Message Generator Example

```
#!/usr/bin/env python
"""
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`
"""
from optparse import OptionParser
import codecs as c
# --------------------------------------------------------------------------- #
# import all the available framers
# --------------------------------------------------------------------------- #
from pymodbus.transaction import ModbusSocketFramer
from pymodbus.transaction import ModbusBinaryFramer
from pymodbus.transaction import ModbusAsciiFramer
from pymodbus.transaction import ModbusRtuFramer
# --------------------------------------------------------------------------- #
# import all available messages
# --------------------------------------------------------------------------- #
from pymodbus.bit_read_message import *
```

```
from pymodbus.bit_write_message import *
from pymodbus.diag_message import *
from pymodbus.file_message import *
from pymodbus.other_message import *
from pymodbus.mei_message import *
from pymodbus.register_read_message import *
from pymodbus.register_write_message import *
from pymodbus.compat import IS_PYTHON3

# --------------------------------------------------------------------------- #
# initialize logging
# --------------------------------------------------------------------------- #
import 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,

    ReturnQueryDataRequest,
    RestartCommunicationsOptionRequest,
    ReturnDiagnosticRegisterRequest,
    ChangeAsciiInputDelimiterRequest,
    ForceListenOnlyModeRequest,
    ClearCountersRequest,
    ReturnBusMessageCountRequest,
    ReturnBusCommunicationErrorCountRequest,
    ReturnBusExceptionErrorCountRequest,
    ReturnSlaveMessageCountRequest,
    ReturnSlaveNoResponseCountRequest,
    ReturnSlaveNAKCountRequest,
    ReturnSlaveBusyCountRequest,
    ReturnSlaveBusCharacterOverrunCountRequest,
    ReturnIopOverrunCountRequest,
    ClearOverrunCountRequest,
```

```
    GetClearModbusPlusRequest
]


# --------------------------------------------------------------------------- #
# enumerate all response messages
# --------------------------------------------------------------------------- #
_response_messages = [
    ReadHoldingRegistersResponse,
    ReadDiscreteInputsResponse,
    ReadInputRegistersResponse,
    ReadCoilsResponse,
    WriteMultipleCoilsResponse,
    WriteMultipleRegistersResponse,
    WriteSingleRegisterResponse,
    WriteSingleCoilResponse,
    ReadWriteMultipleRegistersResponse,

    ReadExceptionStatusResponse,
    GetCommEventCounterResponse,
    GetCommEventLogResponse,
    ReportSlaveIdResponse,

    ReadFileRecordResponse,
    WriteFileRecordResponse,
    MaskWriteRegisterResponse,
    ReadFifoQueueResponse,

    ReadDeviceInformationResponse,

    ReturnQueryDataResponse,
    RestartCommunicationsOptionResponse,
    ReturnDiagnosticRegisterResponse,
    ChangeAsciiInputDelimiterResponse,
    ForceListenOnlyModeResponse,
    ClearCountersResponse,
    ReturnBusMessageCountResponse,
    ReturnBusCommunicationErrorCountResponse,
    ReturnBusExceptionErrorCountResponse,
    ReturnSlaveMessageCountResponse,
    ReturnSlaveNoReponseCountResponse,
    ReturnSlaveNAKCountResponse,
    ReturnSlaveBusyCountResponse,
    ReturnSlaveBusCharacterOverrunCountResponse,
    ReturnIopOverrunCountResponse,
    ClearOverrunCountResponse,
    GetClearModbusPlusResponse
]


# --------------------------------------------------------------------------- #
# build an arguments singleton
# --------------------------------------------------------------------------- #
# Feel free to override any values here to generate a specific message
# in question. It should be noted that many argument names are reused
# between different messages, and a number of messages are simply using
# their default values.
# --------------------------------------------------------------------------- #
```

```
_arguments = {
    'address': 0x12,
    'count': 0x08,
    'value': 0x01,
    '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):
    """ A helper method to 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 = " % message.__class__.__name__)
        packet = framer.buildPacket(message)
        if not options.ascii:
            if not IS_PYTHON3:
                packet = packet.encode('hex')
            else:
                packet = c.encode(packet, 'hex_codec').decode('utf-8')
        print ("{}\n".format(packet))    # because ascii ends with a \r\n


# ---------------------------------------------------------------------------- #
# initialize our program settings
# ---------------------------------------------------------------------------- #
def get_options():
    """ A helper method to parse the command line options

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

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

    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, arg) = parser.parse_args()
    return opt


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

    if option.debug:
        try:
            modbus_log.setLevel(logging.DEBUG)
            logging.basicConfig()
        except Exception as e:
            print("Logging is not supported on this system")

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

    generate_messages(framer, option)


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

13.21 Message Parser Example

```
#!/usr/bin/env python
"""
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
# --------------------------------------------------------------------------- #
from __future__ import print_function
import collections
import textwrap
from optparse import OptionParser
import codecs as c

from pymodbus.factory import ClientDecoder, ServerDecoder
from pymodbus.transaction import ModbusSocketFramer
from pymodbus.transaction import ModbusBinaryFramer
from pymodbus.transaction import ModbusAsciiFramer
from pymodbus.transaction import ModbusRtuFramer
from pymodbus.compat import  IS_PYTHON3


# --------------------------------------------------------------------------- #
# Logging
# --------------------------------------------------------------------------- #
import logging
modbus_log = logging.getLogger("pymodbus")


# --------------------------------------------------------------------------- #
# build a quick wrapper around the framers
# --------------------------------------------------------------------------- #
class Decoder(object):

    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 messge to decode
        """
        if IS_PYTHON3:
            value = message if self.encode else c.encode(message, 'hex_codec')
        else:
            value = message if self.encode else message.encode('hex')
        print("="*80)
        print("Decoding Message %s" % value)
        print("="*80)
        decoders = [
            self.framer(ServerDecoder()),
            self.framer(ClientDecoder()),
        ]
        for decoder in decoders:
            print("%s" % decoder.decoder.__class__.__name__)
```

```
            print("-"*80)
            try:
                decoder.addToFrame(message)
                if decoder.checkFrame():
                    decoder.advanceFrame()
                    decoder.processIncomingPacket(message, self.report)
                else:
                    self.check_errors(decoder, message)
            except Exception as ex:
                self.check_errors(decoder, message)

    def check_errors(self, decoder, message):
        """ Attempt to find message errors

        :param message: The message to find errors in
        """
        pass

    def report(self, message):
        """ The callback to print the message information

        :param message: The message to print
        """
        print("%-15s = %s" % ('name', message.__class__.__name__))
        for (k, v) in message.__dict__.items():
            if isinstance(v, dict):
                print("%-15s =" % k)
                for kk,vv in v.items():
                    print("  %-12s => %s" % (kk, vv))

            elif isinstance(v, collections.Iterable):
                print("%-15s =" % k)
                value = str([int(x) for x  in v])
                for line in textwrap.wrap(value, 60):
                    print("%-15s . %s" % ("", line))
            else:
                print("%-15s = %s" % (k, hex(v)))
        print("%-15s = %s" % ('documentation', message.__doc__))


# --------------------------------------------------------------------------- #
# and decode our message
# --------------------------------------------------------------------------- #
def get_options():
    """ A helper method to 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=True)

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

    parser.add_option("-f", "--file",
                      help="The file containing messages to parse",
                      dest="file", default=None)

    parser.add_option("-t", "--transaction",
                      help="If the incoming message is in hexadecimal format",
                      action="store_true", dest="transaction", default=False)

    parser.add_option("-t", "--transaction",
        help="If the incoming message is in hexadecimal format",
        action="store_true", dest="transaction", default=False)

    (opt, arg) = parser.parse_args()

    if not opt.message and len(arg) > 0:
        opt.message = arg[0]

    return opt


def get_messages(option):
    """ 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:
            if not IS_PYTHON3:
                option.message = option.message.decode('hex')
            else:
                option.message = c.decode(option.message.encode(), 'hex_codec')
        yield option.message
    elif option.file:
        with open(option.file, "r") as handle:
            for line in handle:
                if line.startswith('#'): continue
```

```
                if not option.ascii:
                    line = line.strip()
                    line = line.decode('hex')
                yield line


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

    if option.debug:
        try:
            modbus_log.setLevel(logging.DEBUG)
            logging.basicConfig()
        except Exception as e:
            print("Logging is not supported on this system- {}".format(e))

    framer = lookup = {
        'tcp':    ModbusSocketFramer,
        'rtu':    ModbusRtuFramer,
        'binary': ModbusBinaryFramer,
        'ascii':  ModbusAsciiFramer,
    }.get(option.parser, ModbusSocketFramer)

    decoder = Decoder(framer, option.ascii)
    for message in get_messages(option):
        decoder.decode(message)


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

13.22 Modbus Mapper Example

```
"""
Given a modbus mapping file, 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)::

    from modbus_mapper import csv_mapping_parser
    from modbus_mapper import mapping_decoder
    from pymodbus.client.sync import ModbusTcpClient
```

```
    from pymodbus.payload import BinaryModbusDecoder

    template = ['address', 'size', 'function', 'name', 'description']
    raw_mapping = csv_mapping_parser('input.csv', template)
    mapping = mapping_decoder(raw_mapping)

    index, size = 1, 100
    client = ModbusTcpClient('localhost')
    response = client.read_holding_registers(index, size)
    decoder = BinaryModbusDecoder.fromRegisters(response.registers)
    while index < size:
        print "[{}]\t{}".format(i, mapping[i]['type'](decoder))
        index += mapping[i]['size']

Also, using the same input mapping parsers, we can generate
populated slave contexts that can be run behing a modbus server::

    from modbus_mapper import csv_mapping_parser
    from modbus_mapper import modbus_context_decoder
    from pymodbus.client.ssync import StartTcpServer
    from pymodbus.datastore.context import ModbusServerContext

    template = ['address', 'value', 'function', 'name', 'description']
    raw_mapping = csv_mapping_parser('input.csv', template)
    slave_context = modbus_context_decoder(raw_mapping)
    context = ModbusServerContext(slaves=slave_context, single=True)
    StartTcpServer(context)
"""
import csv
import json
from collections import defaultdict

from tokenize import generate_tokens
from pymodbus.payload import BinaryPayloadDecoder
from pymodbus.datastore.store import ModbusSparseDataBlock
from pymodbus.compat import IS_PYTHON3
from pymodbus.datastore.context import ModbusSlaveContext
if IS_PYTHON3:
    from io import StringIO
else:
    from StringIO import StringIO

# --------------------------------------------------------------------------- #
# 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:
        reader = csv.reader(handle)
        reader.next() # skip the csv header
        for row in reader:
            mapping = dict(zip(template, row))
            fid = mapping.pop('function')
            aid = int(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.
    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:
        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(path):
    """ 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.

    .. note:: The input of the xml file is defined as
    follows::

    :param path: The path to the xml input file
    :returns: The decoded csv dictionary
    """
    pass


# --------------------------------------------------------------------------- #
# 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):
    """ Given a mapping block input, 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
    """
    blocks = defaultdict(dict)
    for block in mapping_blocks.itervalues():
        for mapping in block.itervalues():
            value    = int(mapping['value'])
            address  = int(mapping['address'])
            function = mapping['function']
            blocks[function][address] = value
    return ModbusSlaveContext(**blocks)


# --------------------------------------------------------------------------- #
# 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(object):
    """ 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):
        """ Initializes 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,
            'float64': self.parse_64bit_float,
            'string':  self.parse_32bit_int,
            'bits':    self.parse_bits,
        }

    # ------------------------------------------------------------- #
    # Type parsers
    # ------------------------------------------------------------- #
    @staticmethod
    def parse_string(tokens):
        _ = tokens.next()
        size = int(tokens.next())
        return lambda d: d.decode_string(size=size)

    @staticmethod
    def parse_bits(tokens):
        return lambda d: d.decode_bits()

    @staticmethod
    def parse_8bit_uint(tokens):
        return lambda d: d.decode_8bit_uint()

    @staticmethod
    def parse_16bit_uint(tokens):
        return lambda d: d.decode_16bit_uint()

    @staticmethod
    def parse_32bit_uint(tokens):
        return lambda d: d.decode_32bit_uint()

    @staticmethod
    def parse_64bit_uint(tokens):
        return lambda d: d.decode_64bit_uint()

    @staticmethod
    def parse_8bit_int(tokens):
        return lambda d: d.decode_8bit_int()
```

```
    @staticmethod
    def parse_16bit_int(tokens):
        return lambda d: d.decode_16bit_int()

    @staticmethod
    def parse_32bit_int(tokens):
        return lambda d: d.decode_32bit_int()

    @staticmethod
    def parse_64bit_int(tokens):
        return lambda d: d.decode_64bit_int()

    @staticmethod
    def parse_32bit_float(tokens):
        return lambda d: d.decode_32bit_float()

    @staticmethod
    def parse_64bit_float(tokens):
        return lambda d: d.decode_64bit_float()

    #--------------------------------------------------------------
    # Public Interface
    #--------------------------------------------------------------
    def tokenize(self, value):
        """ Given a value, return the tokens

        :param value: The value to tokenize
        :returns: A token generator
        """
        tokens = generate_tokens(StringIO(value).readline)
        for toknum, tokval, _, _, _ in tokens:
            yield tokval

    def parse(self, value):
        """ Given a type 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  = tokens.next().lower()
        parser = self.parsers.get(token, self.default)
        return parser(tokens)


def mapping_decoder(mapping_blocks, decoder=None):
    """ Given the raw mapping blocks, convert
    them into modbus value decoder map.

    :param mapping_blocks: The mapping blocks
    :param decoder: The type decoder to use
    """
    decoder = decoder or ModbusTypeDecoder()
    for block in mapping_blocks.itervalues():
        for mapping in block.itervalues():
```

```
            mapping['address'] = int(mapping['address'])
            mapping['size'] = int(mapping['size'])
            mapping['type'] = decoder.parse(mapping['type'])
```

13.23 Modbus Saver Example

```
"""
These are a collection of helper methods that can be
used to save a modbus server context to file for backup,
checkpointing, or any other purpose. There use is very
simple::

    context = server.context
    saver   = JsonDatastoreSaver(context)
    saver.save()

These can then be re-opened by the parsers in the
modbus_mapping module. At the moment, the supported
output formats are:

* csv
* json
* xml

To implement your own, simply subclass ModbusDatastoreSaver
and supply the needed callbacks for your given format:

* handle_store_start(self, store)
* handle_store_end(self, store)
* handle_slave_start(self, slave)
* handle_slave_end(self, slave)
* handle_save_start(self)
* handle_save_end(self)
"""
import json
import xml.etree.ElementTree as xml


class ModbusDatastoreSaver(object):
    """ An abstract base class that can be used to implement
    a persistance format for the modbus server context. In
    order to use it, just complete the neccessary callbacks
    (SAX style) that your persistance 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):
```

```
        """ The main runner method to save the
        context to file which calls the various
        callbacks which the sub classes will
        implement.
        """
        with open(self.path, 'w') as self.file_handle:
            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))
                    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):
        pass

    def handle_store_start(self, store):
        pass

    def handle_store_end(self, store):
        pass

    def handle_slave_start(self, slave):
        pass

    def handle_slave_end(self, slave):
        pass

    def handle_save_end(self):
        pass


# ---------------------------------------------------------------- #
# 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):
        self._context = dict()
```

```
    def handle_slave_start(self, slave):
        self._context[hex(slave)] = self._slave = dict()

    def handle_store_start(self, store):
        self._store = self.STORE_NAMES[store]

    def handle_store_values(self, values):
        self._slave[self._store] = dict(values)

    def handle_save_end(self):
        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
    STORE_NAMES = {
        'i': 'i',
        'd': 'd',
        'h': 'h',
        'c': 'c',
    }

    def handle_save_start(self):
        self.file_handle.write(self.HEADER)

    def handle_slave_start(self, slave):
        self._line = [str(slave)]

    def handle_store_start(self, store):
        self._line.append(self.STORE_NAMES[store])

    def handle_store_values(self, values):
        self.file_handle.writelines(self.handle_store_value(values))

    def handle_store_end(self, store):
        self._line.pop()

    def handle_store_value(self, values):
        for a, v in values:
            yield ','.join(self._line + [str(a), str(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):
        self._context = xml.Element("context")
        self._root = xml.ElementTree(self._context)

    def handle_slave_start(self, slave):
        self._slave = xml.SubElement(self._context, "slave")
        self._slave.set("id", str(slave))

    def handle_store_start(self, store):
        self._store = xml.SubElement(self._slave, "store")
        self._store.set("function", self.STORE_NAMES[store])

    def handle_store_values(self, 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):
        self._root.write(self.file_handle)
```

13.24 Modbus Scraper Example

```
#!/usr/bin/env python
"""
This is a simple scraper that can be pointed at a
modbus device to pull down all its values and store
them as a collection of sequential data blocks.
"""
import pickle
from optparse import OptionParser
from twisted.internet import serialport, reactor
from twisted.internet.protocol import ClientFactory
from pymodbus.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext
from pymodbus.factory import ClientDecoder
from pymodbus.client.async import ModbusClientProtocol

#---------------------------------------------------------------------------#
# Configure the client logging
#---------------------------------------------------------------------------#
import logging
log = logging.getLogger("pymodbus")

# --------------------------------------------------------------------------- #
# Choose the framer you want to use
# --------------------------------------------------------------------------- #
from pymodbus.transaction import ModbusBinaryFramer
from pymodbus.transaction import ModbusAsciiFramer
```

```
from pymodbus.transaction import ModbusRtuFramer
from pymodbus.transaction import ModbusSocketFramer

# --------------------------------------------------------------------------- #
# Define some constants
# --------------------------------------------------------------------------- #
COUNT = 8    # The number of bits/registers to read at once
DELAY = 0    # The delay between subsequent reads
SLAVE = 0x01 # The slave unit id to read from

# --------------------------------------------------------------------------- #
# A simple scraper protocol
# --------------------------------------------------------------------------- #
# I tried to spread the load across the device, but feel free to modify the
# logic to suit your own purpose.
# --------------------------------------------------------------------------- #
class ScraperProtocol(ModbusClientProtocol):

    address = None

    def __init__(self, framer, endpoint):
        """ Initializes our custom protocol

        :param framer: The decoder to use to process messages
        :param endpoint: The endpoint to send results to
        """
        ModbusClientProtocol.__init__(self, framer)
        self.endpoint = endpoint

    def connectionMade(self):
        """ Callback for when the client has connected
        to the remote server.
        """
        super(ScraperProtocol, self).connectionMade()
        log.debug("Beginning the processing loop")
        self.address  = self.factory.starting
        reactor.callLater(DELAY, self.scrape_holding_registers)

    def connectionLost(self, reason):
        """ Callback for when the client disconnects from the
        server.

        :param reason: The reason for the disconnection
        """
        reactor.callLater(DELAY, reactor.stop)

    def scrape_holding_registers(self):
        """ Defer fetching holding registers
        """
        log.debug("reading holding registers: %d" % self.address)
        d = self.read_holding_registers(self.address, count=COUNT, unit=SLAVE)
        d.addCallbacks(self.scrape_discrete_inputs, self.error_handler)

    def scrape_discrete_inputs(self, response):
        """ Defer fetching holding registers
        """
        log.debug("reading discrete inputs: %d" % self.address)
        self.endpoint.write((3, self.address, response.registers))
```

```
        d = self.read_discrete_inputs(self.address, count=COUNT, unit=SLAVE)
        d.addCallbacks(self.scrape_input_registers, self.error_handler)

    def scrape_input_registers(self, response):
        """ Defer fetching holding registers
        """
        log.debug("reading discrete inputs: %d" % self.address)
        self.endpoint.write((2, self.address, response.bits))
        d = self.read_input_registers(self.address, count=COUNT, unit=SLAVE)
        d.addCallbacks(self.scrape_coils, self.error_handler)

    def scrape_coils(self, response):
        """ Write values of holding registers, defer fetching coils

        :param response: The response to process
        """
        log.debug("reading coils: %d" % self.address)
        self.endpoint.write((4, self.address, response.registers))
        d = self.read_coils(self.address, count=COUNT, unit=SLAVE)
        d.addCallbacks(self.start_next_cycle, self.error_handler)

    def start_next_cycle(self, response):
        """ Write values of coils, trigger next cycle

        :param response: The response to process
        """
        log.debug("starting next round: %d" % self.address)
        self.endpoint.write((1, self.address, response.bits))
        self.address += COUNT
        if self.address >= self.factory.ending:
            self.endpoint.finalize()
            self.transport.loseConnection()
        else: reactor.callLater(DELAY, self.scrape_holding_registers)

    def error_handler(self, failure):
        """ Handle any twisted errors

        :param failure: The error to handle
        """
        log.error(failure)


# --------------------------------------------------------------------------- #
# a factory for the example protocol
# --------------------------------------------------------------------------- #
# This is used to build client protocol's if you tie into twisted's method
# of processing. It basically produces client instances of the underlying
# protocol::
#
#     Factory(Protocol) -> ProtocolInstance
#
# It also persists data between client instances (think protocol singelton).
# --------------------------------------------------------------------------- #
class ScraperFactory(ClientFactory):

    protocol = ScraperProtocol

    def __init__(self, framer, endpoint, query):
```

```
        """ Remember things necessary for building a protocols """
        self.framer   = framer
        self.endpoint = endpoint
        self.starting, self.ending = query

    def buildProtocol(self, _):
        """ Create a protocol and start the reading cycle """
        protocol = self.protocol(self.framer, self.endpoint)
        protocol.factory = self
        return protocol


# --------------------------------------------------------------------------- #
# a custom client for our device
# --------------------------------------------------------------------------- #
# Twisted provides a number of helper methods for creating and starting
# clients:
# - protocol.ClientCreator
# - reactor.connectTCP
#
# How you start your client is really up to you.
# --------------------------------------------------------------------------- #
class SerialModbusClient(serialport.SerialPort):

    def __init__(self, factory, *args, **kwargs):
        """ Setup the client and start listening on the serial port

        :param factory: The factory to build clients with
        """
        protocol = factory.buildProtocol(None)
        self.decoder = ClientDecoder()
        serialport.SerialPort.__init__(self, protocol, *args, **kwargs)


# --------------------------------------------------------------------------- #
# a custom endpoint for our results
# --------------------------------------------------------------------------- #
# An example line reader, this can replace with:
# - the TCP protocol
# - a context recorder
# - a database or file recorder
# --------------------------------------------------------------------------- #
class LoggingContextReader(object):

    def __init__(self, output):
        """ Initialize a new instance of the logger

        :param output: The output file to save to
        """
        self.output  = output
        self.context = ModbusSlaveContext(
            di = ModbusSequentialDataBlock.create(),
            co = ModbusSequentialDataBlock.create(),
            hr = ModbusSequentialDataBlock.create(),
            ir = ModbusSequentialDataBlock.create())

    def write(self, response):
        """ Handle the next modbus response
```

```
        :param response: The response to process
        """
        log.info("Read Data: %s" % str(response))
        fx, address, values = response
        self.context.setValues(fx, address, values)

    def finalize(self):
        with open(self.output, "w") as handle:
            pickle.dump(self.context, handle)


# --------------------------------------------------------------------------- #
# Main start point
# --------------------------------------------------------------------------- #
def get_options():
    """ A helper method to parse the command line options

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

    parser.add_option("-o", "--output",
        help="The resulting output file for the scrape",
        dest="output", default="datastore.pickle")

    parser.add_option("-p", "--port",
        help="The port to connect to", type='int',
        dest="port", default=502)

    parser.add_option("-s", "--server",
        help="The server to scrape",
        dest="host", default="127.0.0.1")

    parser.add_option("-r", "--range",
        help="The address range to scan",
        dest="query", default="0:1000")

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

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


def main():
    """ The main runner function """
    options = get_options()

    if options.debug:
        try:
            log.setLevel(logging.DEBUG)
            logging.basicConfig()
        except Exception as ex:
            print("Logging is not supported on this system")

    # split the query into a starting and ending range
```

```
    query = [int(p) for p in options.query.split(':')]

    try:
        log.debug("Initializing the client")
        framer  = ModbusSocketFramer(ClientDecoder())
        reader  = LoggingContextReader(options.output)
        factory = ScraperFactory(framer, reader, query)

        # how to connect based on TCP vs Serial clients
        if isinstance(framer, ModbusSocketFramer):
            reactor.connectTCP(options.host, options.port, factory)
        else:
            SerialModbusClient(factory, options.port, reactor)

        log.debug("Starting the client")
        reactor.run()
        log.debug("Finished scraping the client")
    except Exception as ex:
        print(ex)

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


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

13.25 Modbus Simulator Example

```
#!/usr/bin/env python
"""
An example of creating a fully implemented modbus server
with read/write data as well as user configurable base data
"""

import pickle
from optparse import OptionParser
from twisted.internet import reactor

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

# ---------------------------------------------------------------------------- #
# Logging
# ---------------------------------------------------------------------------- #
import logging
logging.basicConfig()

server_log = logging.getLogger("pymodbus.server")
protocol_log = logging.getLogger("pymodbus.protocol")

# ---------------------------------------------------------------------------- #
# Extra Global Functions
# ---------------------------------------------------------------------------- #
```

```
# These are extra helper functions that don't belong in a class
# --------------------------------------------------------------------------- #
import getpass


def root_test():
    """ Simple test 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):
        """ Initializes the ConfigurationException instance

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

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

        :returns: A string representation of the object
        """
        return 'Configuration Error: %s' % self.string



class Configuration:
    """
    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):
        """
        Trys to load a configuration file, lets the file not
        found exception fall through

        :param config: The pickled datastore
        """
        try:
            self.file = open(config, "r")
        except Exception:
            raise ConfigurationException("File not found %s" % config)

    def parse(self):
```

```
        """ Parses the config file and creates a server context
        """
        handle = pickle.load(self.file)
        try:  # test for existance, or bomb
            dsd = handle['di']
            csd = handle['ci']
            hsd = handle['hr']
            isd = handle['ir']
        except Exception:
            raise ConfigurationException("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, arg) = parser.parse_args()

    # enable debugging information
    if opt.debug:
        try:
            server_log.setLevel(logging.DEBUG)
            protocol_log.setLevel(logging.DEBUG)
        except Exception as e:
            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!")
```

13.26 Modicon Payload Example

```
"""
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.interfaces import IPayloadBuilder
from pymodbus.utilities import pack_bitstring
from pymodbus.utilities import unpack_bitstring
from pymodbus.exceptions import ParameterException


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 endianess of the payload
        """
        self._payload = payload or []
        self._endian = endian

    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):
        """ Adds 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):
        """ Adds 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):
        """ Adds 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):
        """ Adds 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):
        """ Adds 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):
        """ Adds 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):
        """ Adds 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):
        """ Adds 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):
        """ Adds a string to the buffer

        :param value: The value to add to the buffer
        """
        fstring = self._endian + 's'
        for c in value:
            self._payload.append(pack(fstring, c))


class ModiconPayloadDecoder(object):
    """
    A utility that helps decode modicon encoded payload
    messages from a modbus reponse 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 endianess 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 endianess of the payload
        :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):
        """ Decodes 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):
        """ Decodes 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):
        """ Decodes 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):
        """ Decodes 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):
        """ Decodes 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):
        """ Decodes a 32 bit signed 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_32bit_float(self, size=1):
        """ Decodes 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):
        """ Decodes 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):
        """ Decodes 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"]
```

13.27 Remote Server Context Example

```
"""
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),
        0x02: client('host1.something.com', 0x02)
    }
"""
from pymodbus.exceptions import NotImplementedException
from pymodbus.interfaces import IModbusSlaveContext

# --------------------------------------------------------------------------- #
# Logging
# --------------------------------------------------------------------------- #

import 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):
        """ Initializes 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):
        """ Resets all the datastores to their default values """
        raise NotImplementedException()

    def validate(self, fx, address, count=1):
        """ Validates 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
        """
        _logger.debug("validate[%d] %d:%d" % (fx, address, count))
        result = self.context.get_callbacks[self.decode(fx)](address,
                                                             count,
                                                             self.unit_id)
        return result.function_code < 0x80

    def getValues(self, fx, address, count=1):
        """ Validates 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 retrieve
        :returns: The requested values from a:a+c
        """
        _logger.debug("get values[%d] %d:%d" % (fx, address, count))
        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):
        """ Sets 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
        """
        _logger.debug("set values[%d] %d:%d" % (fx, address, len(values)))
        self.context.set_callbacks[self.decode(fx)](address,
                                                    values,
                                                    self.unit_id)

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

        :returns: A string representation of the context
        """
        return "Remote Single Slave Context(%s)" % self.unit_id

    def __extract_result(self, fx, result):
        """ A helper method to 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 result.function_code < 0x80:
            if fx in ['d', 'c']:
                return result.bits
            if fx in ['h', 'i']:
                return result.registers
        else: return result

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


class RemoteServerContext(object):
    """ 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):
        """ Initializes the datastores

        :param client: The client to retrieve values with
        """
        self.get_callbacks = {
            'd': lambda a, c, s: client.read_discrete_inputs(a, c, s),
            'c': lambda a, c, s: client.read_coils(a, c, s),
            'h': lambda a, c, s: client.read_holding_registers(a, c, s),
            'i': lambda a, c, s: client.read_input_registers(a, c, s),
        }
        self.set_callbacks = {
            'd': lambda a, v, s: client.write_coils(a, v, s),
            'c': lambda a, v, s: client.write_coils(a, v, s),
            'h': lambda a, v, s: client.write_registers(a, v, s),
            'i': lambda a, v, s: client.write_registers(a, v, s),
        }
        self._client = client
        self.slaves = {} # simply a cache

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

        :returns: A string representation of the context
        """
        return "Remote Server Context(%s)" % self._client

    def __iter__(self):
        """ Iterater 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 existance
        :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):
        """ Used 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):
        """ Wrapper used to access the slave context

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

    def __getitem__(self, slave):
        """ Used 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]
```

13.28 Serial Forwarder Example

```
#!/usr/bin/env python
"""
Pymodbus Synchronous Serial Forwarder
--------------------------------------------------------------------------

We basically set the context for the tcp serial server to be that of a
serial client! This is just an example of how clever you can be with
the data context (basically anything can become a modbus device).
"""
# --------------------------------------------------------------------------- #
# import the various server implementations
# --------------------------------------------------------------------------- #
from pymodbus.server.sync import StartTcpServer as StartServer
from pymodbus.client.sync import ModbusSerialClient as ModbusClient

from pymodbus.datastore.remote import RemoteSlaveContext
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext
```

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


def run_serial_forwarder():
    # --------------------------------------------------------------------------- #
    # initialize the datastore(serial client)
    # --------------------------------------------------------------------------- #
    client = ModbusClient(method='ascii', port='/dev/pts/14')
    store = RemoteSlaveContext(client)
    context = ModbusServerContext(slaves=store, single=True)

    # --------------------------------------------------------------------------- #
    # run the server you want
    # --------------------------------------------------------------------------- #
    StartServer(context)


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

13.29 Sunspec Client Example

```
from pymodbus.constants import Endian
from pymodbus.client.sync import ModbusTcpClient
from pymodbus.payload import BinaryPayloadDecoder
from twisted.internet.defer import Deferred


#---------------------------------------------------------------------------#
# Logging
#---------------------------------------------------------------------------#
import logging
_logger = logging.getLogger(__name__)
_logger.setLevel(logging.DEBUG)
logging.basicConfig()


# --------------------------------------------------------------------------- #
# Sunspec Common Constants
# --------------------------------------------------------------------------- #
class SunspecDefaultValue(object):
    """ A collection of constants to indicate if
    a value is not implemented.
    """
    Signed16        = 0x8000
    Unsigned16      = 0xffff
    Accumulator16   = 0x0000
    Scale           = 0x8000
```

```
    Signed32        = 0x80000000
    Float32         = 0x7fc00000
    Unsigned32      = 0xffffffff
    Accumulator32   = 0x00000000
    Signed64        = 0x8000000000000000
    Unsigned64      = 0xffffffffffffffff
    Accumulator64   = 0x0000000000000000
    String          = '\x00'


class SunspecStatus(object):
    """ Indicators of the current status of a
    sunspec device
    """
    Normal  = 0x00000000
    Error   = 0xfffffffe
    Unknown = 0xffffffff


class SunspecIdentifier(object):
    """ Assigned identifiers that are pre-assigned
    by the sunspec protocol.
    """
    Sunspec = 0x53756e53


class SunspecModel(object):
    """ Assigned device indentifiers that are pre-assigned
    by the sunspec protocol.
    """
    #------------------------------------------
    # 0xx Common Models
    #------------------------------------------
    CommonBlock                         = 1
    AggregatorBlock                     = 2

    #------------------------------------------
    # 1xx Inverter Models
    #------------------------------------------
    SinglePhaseIntegerInverter          = 101
    SplitPhaseIntegerInverter           = 102
    ThreePhaseIntegerInverter           = 103
    SinglePhaseFloatsInverter           = 103
    SplitPhaseFloatsInverter            = 102
    ThreePhaseFloatsInverter            = 103

    #------------------------------------------
    # 2xx Meter Models
    #------------------------------------------
    SinglePhaseMeter                    = 201
    SplitPhaseMeter                     = 201
    WyeConnectMeter                     = 201
    DeltaConnectMeter                   = 201

    #------------------------------------------
    # 3xx Environmental Models
    #------------------------------------------
    BaseMeteorological                  = 301
```

```
    Irradiance                                  = 302
    BackOfModuleTemperature                     = 303
    Inclinometer                                = 304
    Location                                    = 305
    ReferencePoint                              = 306
    BaseMeteorological                          = 307
    MiniMeteorological                          = 308

    #--------------------------------------------
    # 4xx String Combiner Models
    #--------------------------------------------
    BasicStringCombiner                         = 401
    AdvancedStringCombiner                      = 402

    #--------------------------------------------
    # 5xx Panel Models
    #--------------------------------------------
    PanelFloat                                  = 501
    PanelInteger                                = 502

    #--------------------------------------------
    # 641xx Outback Blocks
    #--------------------------------------------
    OutbackDeviceIdentifier                     = 64110
    OutbackChargeController                     = 64111
    OutbackFMSeriesChargeController             = 64112
    OutbackFXInverterRealTime                   = 64113
    OutbackFXInverterConfiguration              = 64114
    OutbackSplitPhaseRadianInverter             = 64115
    OutbackRadianInverterConfiguration          = 64116
    OutbackSinglePhaseRadianInverterRealTime    = 64117
    OutbackFLEXNetDCRealTime                    = 64118
    OutbackFLEXNetDCConfiguration               = 64119
    OutbackSystemControl                        = 64120

    #--------------------------------------------
    # 64xxx Vender Extension Block
    #--------------------------------------------
    EndOfSunSpecMap                             = 65535

    @classmethod
    def lookup(klass, code):
        """ Given a device identifier, return the
        device model name for that identifier

        :param code: The device code to lookup
        :returns: The device model name, or None if none available
        """
        values = dict((v, k) for k, v in klass.__dict__.iteritems()
            if not callable(v))
        return values.get(code, None)


class SunspecOffsets(object):
    """ Well known offsets that are used throughout
    the sunspec protocol
    """
    CommonBlock             = 40000
```

```
    CommonBlockLength       = 69
    AlternateCommonBlock    = 50000


# --------------------------------------------------------------------------- #
# Common Functions
# --------------------------------------------------------------------------- #
def defer_or_apply(func):
    """ Decorator to apply an adapter method
    to a result regardless if it is a deferred
    or a concrete response.

    :param func: The function to decorate
    """
    def closure(future, adapt):
        if isinstance(future, Deferred):
            d = Deferred()
            future.addCallback(lambda r: d.callback(adapt(r)))
            return d
        return adapt(future)
    return closure


def create_sunspec_sync_client(host):
    """ A quick helper method to create a sunspec
    client.

    :param host: The host to connect to
    :returns: an initialized SunspecClient
    """
    modbus = ModbusTcpClient(host)
    modbus.connect()
    client = SunspecClient(modbus)
    client.initialize()
    return client


# --------------------------------------------------------------------------- #
# Sunspec Client
# --------------------------------------------------------------------------- #
class SunspecDecoder(BinaryPayloadDecoder):
    """ A decoder that deals correctly with the sunspec
    binary format.
    """

    def __init__(self, payload, byteorder):
        """ Initialize a new instance of the SunspecDecoder

        .. note:: This is always set to big endian byte order
        as specified in the protocol.
        """
        byteorder = Endian.Big
        BinaryPayloadDecoder.__init__(self, payload, byteorder)

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

        :param size: The size of the string to decode
```

```
        """
        self._pointer += size
        string = self._payload[self._pointer - size:self._pointer]
        return string.split(SunspecDefaultValue.String)[0]


class SunspecClient(object):

    def __init__(self, client):
        """ Initialize a new instance of the client

        :param client: The modbus client to use
        """
        self.client = client
        self.offset = SunspecOffsets.CommonBlock

    def initialize(self):
        """ Initialize the underlying client values

        :returns: True if successful, false otherwise
        """
        decoder  = self.get_device_block(self.offset, 2)
        if decoder.decode_32bit_uint() == SunspecIdentifier.Sunspec:
            return True
        self.offset = SunspecOffsets.AlternateCommonBlock
        decoder  = self.get_device_block(self.offset, 2)
        return decoder.decode_32bit_uint() == SunspecIdentifier.Sunspec

    def get_common_block(self):
        """ Read and return the sunspec common information
        block.

        :returns: A dictionary of the common block information
        """
        length  = SunspecOffsets.CommonBlockLength
        decoder = self.get_device_block(self.offset, length)
        return {
            'SunSpec_ID':       decoder.decode_32bit_uint(),
            'SunSpec_DID':      decoder.decode_16bit_uint(),
            'SunSpec_Length':   decoder.decode_16bit_uint(),
            'Manufacturer':     decoder.decode_string(size=32),
            'Model':            decoder.decode_string(size=32),
            'Options':          decoder.decode_string(size=16),
            'Version':          decoder.decode_string(size=16),
            'SerialNumber':     decoder.decode_string(size=32),
            'DeviceAddress':    decoder.decode_16bit_uint(),
            'Next_DID':         decoder.decode_16bit_uint(),
            'Next_DID_Length':  decoder.decode_16bit_uint(),
        }

    def get_device_block(self, offset, size):
        """ A helper method to retrieve the next device block

        .. note:: We will read 2 more registers so that we have
        the information for the next block.

        :param offset: The offset to start reading at
        :param size: The size of the offset to read
```

```
        :returns: An initialized decoder for that result
        """
        _logger.debug("reading device block[{}..{}]".format(offset, offset + size))
        response = self.client.read_holding_registers(offset, size + 2)
        return SunspecDecoder.fromRegisters(response.registers)

    def get_all_device_blocks(self):
        """ Retrieve all the available blocks in the supplied
        sunspec device.

        .. note:: Since we do not know how to decode the available
        blocks, this returns a list of dictionaries of the form:

            decoder: the-binary-decoder,
            model:   the-model-identifier (name)

        :returns: A list of the available blocks
        """
        blocks = []
        offset = self.offset + 2
        model  = SunspecModel.CommonBlock
        while model != SunspecModel.EndOfSunSpecMap:
            decoder = self.get_device_block(offset, 2)
            model   = decoder.decode_16bit_uint()
            length  = decoder.decode_16bit_uint()
            blocks.append({
                'model' : model,
                'name'  : SunspecModel.lookup(model),
                'length': length,
                'offset': offset + length + 2
            })
            offset += length + 2
        return blocks


#---------------------------------------------------------------
# A quick test runner
#---------------------------------------------------------------
if __name__ == "__main__":
    client = create_sunspec_sync_client("YOUR.HOST.GOES.HERE")

    # print out all the device common block
    common = client.get_common_block()
    for key, value in common.iteritems():
        if key == "SunSpec_DID":
            value = SunspecModel.lookup(value)
        print("{:<20}: {}".format(key, value))

    # print out all the available device blocks
    blocks = client.get_all_device_blocks()
    for block in blocks:
        print(block)

    client.client.close()
```

13.30 Thread Safe Datastore Example

```
import threading
from contextlib import contextmanager
from pymodbus.datastore.store import BaseModbusDataBlock


class ContextWrapper(object):
    """ This is a simple wrapper around enter
    and exit functions that conforms to the pyhton
    context manager protocol:

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

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

    def __enter__(self):
        if self.enter: self._enter()
        return self if not self._factory else self._factory()

    def __exit__(self, args):
        if self._leave: self._leave()


class ReadWriteLock(object):
    """ This reader writer lock gurantees write order, 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):
        """ Initializes 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):
        return (self.writer                      # if there is a current writer
            or (self.queue                      # or if there is a waiting writer
           and (self.queue[0] != self.read_condition)))     # or if the queue head is␣
↪not a reader
```

```
    def acquire_reader(self):
        """ Notifies 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]: # if the read␣
↪condition is at the queue head
                    self.queue.pop(0)                       # then go ahead and remove␣
↪it
            self.readers += 1                               # update the current␣
↪number of readers

    def acquire_writer(self):
        """ Notifies the lock that a new writer is requesting
        the underlying resource.
        """
        with self.lock:
            if self.writer or self.readers:                 # if we need to wait on a␣
↪writer or readers
                condition = threading.Condition(self.lock) # create a condition just␣
↪for this writer
                self.queue.append(condition)                # and put it on the␣
↪waiting queue
                while self.writer or self.readers:          # until the write lock is␣
↪free
                    condition.wait(1)                       # wait on our condition
                self.queue.pop(0)                           # remove our condition␣
↪after our condition is met
            self.writer = True                              # stop other writers from␣
↪operating

    def release_reader(self):
        """ Notifies 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):
        """ Notifies 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):
    """ 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):
        """ Checks 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):
        """ Returns 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):
        """ Sets 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__":

    class AtomicCounter(object):
        def __init__(self, **kwargs):
            self.counter = kwargs.get('start', 0)
            self.finish  = kwargs.get('finish', 1000)
            self.lock    = threading.Lock()

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

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

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

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

    def 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("readers[%d] writers[%d]" % (readers.counter, writers.counter))
```

13.31 Gui Common Example

```
#!/usr/bin/env python
# --------------------------------------------------------------------------- #
# System
# --------------------------------------------------------------------------- #
import os
import getpass
import pickle
from threading import Thread

# --------------------------------------------------------------------------- #
# SNMP Simulator
# --------------------------------------------------------------------------- #
from twisted.internet import reactor
from twisted.internet import error as twisted_error
from pymodbus.server.async import ModbusServerFactory
from pymodbus.datastore import ModbusServerContext,ModbusSlaveContext

# --------------------------------------------------------------------------- #
# Logging
# --------------------------------------------------------------------------- #
import logging
log = logging.getLogger("pymodbus")

# --------------------------------------------------------------------------- #
# Application Error
# --------------------------------------------------------------------------- #


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

# --------------------------------------------------------------------------- #
# Extra Global Functions
# --------------------------------------------------------------------------- #
# These are extra helper functions that don't belong in a class
# --------------------------------------------------------------------------- #


def root_test():
    """ Simple test to see if we are running as root """
    return getpass.getuser() == "root"

# --------------------------------------------------------------------------- #
# Simulator Class
# --------------------------------------------------------------------------- #


class Simulator(object):
    """
    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):
        """
        Trys to load a configuration file, lets the file not
        found exception fall through

        :param config: The pickled datastore
        """
        try:
            self.file = open(config, "r")
        except Exception:
            raise ConfigurationException("File not found %s" % config)

    def _parse(self):
        """ Parses the config file and creates a server context """
        try:
            handle = pickle.load(self.file)
            dsd = handle['di']
            csd = handle['ci']
            hsd = handle['hr']
            isd = handle['ir']
        except KeyError:
            raise ConfigurationException("Invalid Configuration")
        slave = ModbusSlaveContext(d=dsd, c=csd, h=hsd, i=isd)
        return ModbusServerContext(slaves=slave)

    def _simulator(self):
        """ Starts the snmp simulator """
        ports = [502]+range(20000,25000)
        for port in ports:
            try:
                reactor.listenTCP(port, ModbusServerFactory(self._parse()))
                log.debug('listening on port %d' % port)
                return port
            except twisted_error.CannotListenError:
                pass

    def run(self):
        """ Used to run the simulator """
        log.debug('simulator started')
        reactor.callWhenRunning(self._simulator)

# -------------------------------------------------------------------------- #
# Network reset thread
# -------------------------------------------------------------------------- #
# This is linux only, maybe I should make a base class that can be filled
# in for linux(debian/redhat)/windows/nix
# -------------------------------------------------------------------------- #


class NetworkReset(Thread):
    """
    This class is simply a daemon that is spun off at the end of the
    program to call the network restart function (an easy way to
    remove all the virtual interfaces)
    """
```

```
    def __init__(self):
        """ Initialize a new network reset thread """
        Thread.__init__(self)
        self.setDaemon(True)

    def run(self):
        """ Run the network reset """
        os.system("/etc/init.d/networking restart")
```

CHAPTER 14

Pymodbus

14.1 Pymodbus package

14.1.1 Subpackages

pymodbus.client package

Submodules

pymodbus.client.async module

Implementation of a Modbus Client Using Twisted

Example run:

```
from twisted.internet import reactor, protocol
from pymodbus.client.async import ModbusClientProtocol

def printResult(result):
    print "Result: %d" % result.bits[0]

def process(client):
    result = client.write_coil(1, True)
    result.addCallback(printResult)
    reactor.callLater(1, reactor.stop)

defer = protocol.ClientCreator(reactor, ModbusClientProtocol
        ).connectTCP("localhost", 502)
defer.addCallback(process)
```

Another example:

```
from twisted.internet import reactor
from pymodbus.client.async import ModbusClientFactory

def process():
    factory = reactor.connectTCP("localhost", 502, ModbusClientFactory())
    reactor.stop()

if __name__ == "__main__":
   reactor.callLater(1, process)
   reactor.run()
```

class pymodbus.client.async.**ModbusClientProtocol**(*framer=None*, ***kwargs*)

Bases: twisted.internet.protocol.Protocol, *pymodbus.client.common.ModbusClientMixin*

This represents the base modbus client protocol. All the application layer code is deferred to a higher level wrapper.

connectionLost(*reason*)

Called upon a client disconnect

Parameters **reason** – The reason for the disconnect

connectionMade()

Called upon a successful client connection.

dataReceived(*data*)

Get response, check for valid message, decode result

Parameters **data** – The data returned from the server

execute(*request*)

Starts the producer to send the next request to consumer.write(Frame(request))

class pymodbus.client.async.**ModbusUdpClientProtocol**(*framer=None*, ***kwargs*)

Bases: twisted.internet.protocol.DatagramProtocol, *pymodbus.client.common.ModbusClientMixin*

This represents the base modbus client protocol. All the application layer code is deferred to a higher level wrapper.

datagramReceived(*data*, *params*)

Get response, check for valid message, decode result

Parameters

- **data** – The data returned from the server
- **params** – The host parameters sending the datagram

execute(*request*)

Starts the producer to send the next request to consumer.write(Frame(request))

class pymodbus.client.async.**ModbusClientFactory**

Bases: twisted.internet.protocol.ReconnectingClientFactory

Simple client protocol factory

protocol

alias of *ModbusClientProtocol*

pymodbus.client.common module

Modbus Client Common

This is a common client mixin that can be used by both the synchronous and asynchronous clients to simplify the interface.

class pymodbus.client.common.**ModbusClientMixin**
Bases: object

This is a modbus client mixin that provides additional factory methods for all the current modbus methods. This can be used instead of the normal pattern of:

```
# instead of this
client = ModbusClient(...)
request = ReadCoilsRequest(1,10)
response = client.execute(request)

# now like this
client = ModbusClient(...)
response = client.read_coils(1, 10)
```

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

Parameters

- **address** – The address of the register to write
- **and_mask** – The and bitmask to apply to the register address
- **or_mask** – The or bitmask to apply to the register address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

read_coils(*address*, *count=1*, ***kwargs*)

Parameters

- **address** – The starting address to read from
- **count** – The number of coils to read
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

read_discrete_inputs(*address*, *count=1*, ***kwargs*)

Parameters

- **address** – The starting address to read from
- **count** – The number of discretes to read
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

read_holding_registers(*address*, *count=1*, ***kwargs*)

Parameters

- **address** – The starting address to read from

- **count** – The number of registers to read
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

read_input_registers(*address*, *count=1*, ***kwargs*)

Parameters

- **address** – The starting address to read from
- **count** – The number of registers to read
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

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
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

write_coil(*address*, *value*, ***kwargs*)

Parameters

- **address** – The starting address to write to
- **value** – The value to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

write_coils(*address*, *values*, ***kwargs*)

Parameters

- **address** – The starting address to write to
- **values** – The values to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

write_register(*address*, *value*, ***kwargs*)

Parameters

- **address** – The starting address to write to
- **value** – The value to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

write_registers(*address*, *values*, ***kwargs*)

Parameters

- **address** – The starting address to write to
- **values** – The values to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

pymodbus.client.sync module

class pymodbus.client.sync.**ModbusTcpClient**(*host='127.0.0.1'*, *port=502*, *framer=<class 'pymodbus.transaction.ModbusSocketFramer'>*, ***kwargs*)

Bases: pymodbus.client.sync.BaseModbusClient

Implementation of a modbus tcp client

close()

Closes the underlying socket connection

connect()

Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

class pymodbus.client.sync.**ModbusUdpClient**(*host='127.0.0.1'*, *port=502*, *framer=<class 'pymodbus.transaction.ModbusSocketFramer'>*, ***kwargs*)

Bases: pymodbus.client.sync.BaseModbusClient

Implementation of a modbus udp client

close()

Closes the underlying socket connection

connect()

Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

class pymodbus.client.sync.**ModbusSerialClient**(*method='ascii'*, ***kwargs*)

Bases: pymodbus.client.sync.BaseModbusClient

Implementation of a modbus serial client

close()

Closes the underlying socket connection

connect()

Connect to the modbus serial server

Returns True if connection succeeded, False otherwise

Module contents

pymodbus.datastore package

Subpackages

pymodbus.datastore.database package

Submodules

pymodbus.datastore.database.redis_datastore module

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

Bases: *pymodbus.interfaces.IModbusSlaveContext*

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

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

Validates 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 retrieve

Returns The requested values from a:a+c

reset()

Resets all the datastores to their default values

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

Sets 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*)

Validates 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

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

Bases: *pymodbus.interfaces.IModbusSlaveContext*

This creates a modbus data model with each data access stored in its own personal block

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

Validates 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 retrieve

Returns The requested values from a:a+c

reset()

Resets all the datastores to their default values

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

Sets 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*)

Validates 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

Module contents

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

Bases: *pymodbus.interfaces.IModbusSlaveContext*

This creates a modbus data model with each data access stored in its own personal block

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

Validates 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 retrieve

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets 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*)
Validates 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.**RedisSlaveContext**(***kwargs*)
Bases: *pymodbus.interfaces.IModbusSlaveContext*

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

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

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

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.

class pymodbus.datastore.context.**ModbusSlaveContext**(**args*, ***kwargs*)
Bases: *pymodbus.interfaces.IModbusSlaveContext*

This creates a modbus data model with each data access stored in its own personal block

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets 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*)
Validates 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.remote module

class pymodbus.datastore.remote.**RemoteSlaveContext**(*client*, *unit=None*)
Bases: *pymodbus.interfaces.IModbusSlaveContext*

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

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets 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*)
Validates 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.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 indecies. 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 implmented 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*)

Used 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*)

Returns the requested values from the datastore

Parameters

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

Returns The requested values from a:a+c

reset()

Resets the datastore to the initialized default value

setValues(*address*, *values*)

Returns the requested values from the datastore

Parameters

- **address** – The starting address

- **values** – The values to store

validate(*address*, *count=1*)
Checks 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.**ModbusSequentialDataBlock**(*address*, *values*)
Bases: *pymodbus.datastore.store.BaseModbusDataBlock*

Creates a sequential modbus datastore

classmethod create(*klass*)
Factory method to create a datastore with the full address space initialized to 0x00

Returns An initialized datastore

getValues(*address*, *count=1*)
Returns 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*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks 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*)
Bases: *pymodbus.datastore.store.BaseModbusDataBlock*

Creates a sparse modbus datastore

classmethod create(*klass*)
Factory method to create a datastore with the full address space initialized to 0x00

Returns An initialized datastore

getValues(*address*, *count=1*)
Returns 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*)
Sets the requested values of the datastore

Parameters

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

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

Module contents

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

Creates a sequential modbus datastore

classmethod create(*klass*)
Factory method to create a datastore with the full address space initialized to 0x00

Returns An initialized datastore

getValues(*address*, *count=1*)
Returns 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*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks 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.**ModbusSparseDataBlock**(*values*)
Bases: *pymodbus.datastore.store.BaseModbusDataBlock*

Creates a sparse modbus datastore

classmethod create(*klass*)
Factory method to create a datastore with the full address space initialized to 0x00

Returns An initialized datastore

getValues(*address*, *count=1*)
Returns 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*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks 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.**ModbusSlaveContext**(**args*, ***kwargs*)
Bases: *pymodbus.interfaces.IModbusSlaveContext*

This creates a modbus data model with each data access stored in its own personal block

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets 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*)
Validates 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.**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.

pymodbus.internal package

Submodules

pymodbus.internal.ptwisted module

A collection of twisted utility code

pymodbus.internal.ptwisted.**InstallManagementConsole**(*namespace*, *users={'admin': 'admin'}*, *port=503*)

Helper method to start an ssh management console for the modbus server.

Parameters

- **namespace** – The data to constrain the server to
- **users** – The users to login with
- **port** – The port to host the server on

Module contents

pymodbus.server package

Submodules

pymodbus.server.async module

Implementation of a Twisted Modbus Server

pymodbus.server.async.**StartTcpServer**(*context*, *identity=None*, *address=None*, *console=False*, ***kwargs*)

Helper method to start the Modbus Async TCP server

Parameters

- **context** – The server data context
- **identify** – The server identity to use (default empty)
- **address** – An optional (interface, port) to bind to.
- **console** – A flag indicating if you want the debug console
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.async.**StartUdpServer**(*context*, *identity=None*, *address=None*, ***kwargs*)

Helper method to start the Modbus Async Udp server

Parameters

- **context** – The server data context
- **identify** – The server identity to use (default empty)
- **address** – An optional (interface, port) to bind to.
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.async.**StartSerialServer**(*context*, *identity=None*, *framer=<class 'pymodbus.transaction.ModbusAsciiFramer'>*, ***kwargs*)

Helper method to start the Modbus Async Serial server

Parameters

- **context** – The server data context
- **identify** – The server identity to use (default empty)
- **framer** – The framer to use (default ModbusAsciiFramer)
- **port** – The serial port to attach to
- **baudrate** – The baud rate to use for the serial device
- **console** – A flag indicating if you want the debug console
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.async.**StopServer**()

Helper method to stop Async Server

pymodbus.server.sync module

Implementation of a Threaded Modbus Server

pymodbus.server.sync.**StartTcpServer**(*context=None*, *identity=None*, *address=None*, ***kwargs*)

A factory to 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.
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.sync.**StartUdpServer**(*context=None*, *identity=None*, *address=None*, ***kwargs*)

A factory to 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.
- **framer** – The framer to operate with (default ModbusSocketFramer)
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.sync.**StartSerialServer**(*context=None*, *identity=None*, ***kwargs*)

A factory to start and run a serial modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **framer** – The framer to operate with (default ModbusAsciiFramer)
- **port** – The serial port to attach to
- **stopbits** – The number of stop bits to use
- **bytesize** – The bytesize of the serial messages
- **parity** – Which kind of parity to use
- **baudrate** – The baud rate to use for the serial device
- **timeout** – The timeout to use for the serial device
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

Module contents

14.1.2 Submodules

14.1.3 pymodbus.bit_read_message module

Bit Reading Request/Response messages

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

Bases: pymodbus.bit_read_message.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 The initializes response message, exception message otherwise

function_code = 1

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

Bases: pymodbus.bit_read_message.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.

function_code = 1

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

Bases: pymodbus.bit_read_message.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 The initializes response message, exception message otherwise

function_code = 2

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

Bases: pymodbus.bit_read_message.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.

function_code = 2

14.1.4 pymodbus.bit_write_message module

Bit Writing Request/Response

TODO write mask request/response

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

Bases: *pymodbus.pdu.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*)

Decodes a write coil request

Parameters **data** – The packet data to decode

encode()

Encodes 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()

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: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request, returned after the coil state has been written.

decode(*data*)
Decodes a write coil response

Parameters **data** – The packet data to decode

encode()
Encodes write coil response

Returns The byte encoded message

function_code = 5

class pymodbus.bit_write_message.**WriteMultipleCoilsRequest**(*address=None*, *values=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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

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

decode(*data*)
Decodes a write coils request

Parameters **data** – The packet data to decode

encode()
Encodes 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

class pymodbus.bit_write_message.**WriteMultipleCoilsResponse**(*address=None*, *count=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

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

decode(*data*)
Decodes a write coils response

Parameters **data** – The packet data to decode

encode()
Encodes write coils response

Returns The byte encoded message

function_code = 15

14.1.5 pymodbus.compat module

Python 2.x/3.x Compatibility Layer

This is mostly based on the jinja2 compat code:

> Some py2/py3 compatibility support based on a stripped down version of six so we don't have to depend on a specific version of it.
>
> **copyright** Copyright 2013 by the Jinja team, see AUTHORS.
>
> **license** BSD, see LICENSE for details.

pymodbus.compat.**get_next**(*x*)

pymodbus.compat.**implements_to_string**(*klass*)

pymodbus.compat.**iteritems**(*d*)

pymodbus.compat.**iterkeys**(*d*)

pymodbus.compat.**itervalues**(*d*)

14.1.6 pymodbus.constants module

Constants For Modbus Server/Client

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

class pymodbus.constants.**Defaults**
Bases: *pymodbus.interfaces.Singleton*

A collection of modbus default values

Port
The default modbus tcp server port (502)

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 requst.

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.

UnitId
The modbus slave addrss. 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 respons).

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-existant slave on a bus will never respond. The client in this case will simply timeout.

Baudrate = 19200

Bytesize = 8

IgnoreMissingSlaves = False

Parity = 'N'

Port = 502

ProtocolId = 0

Reconnects = 0

Retries = 3

RetryOnEmpty = False

Stopbits = 1

Timeout = 3

TransactionId = 0

UnitId = 0

ZeroMode = False

class pymodbus.constants.**ModbusStatus**
Bases: *pymodbus.interfaces.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.**Endian**
Bases: *pymodbus.interfaces.Singleton*

An enumeration representing the various byte endianess.

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**
Bases: *pymodbus.interfaces.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.**DeviceInformation**
Bases: *pymodbus.interfaces.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 optinoal 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.**MoreData**
Bases: *pymodbus.interfaces.Singleton*

Represents the more follows condition

Nothing
This indiates that no more objects are going to be returned.

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

KeepReading = 255

Nothing = 0

14.1.7 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.**ModbusAccessControl**
Bases: *pymodbus.interfaces.Singleton*

This is a simple implementation of a Network Management System table. Its purpose is to control access to the server (if it is used). We assume that if an entry is in the table, it is allowed accesses to resources. However, if the host does not appear in the table (all unknown hosts) its connection will simply be closed.

Since it is a singleton, only one version can possible exist and all instances pull from here.

add(*host*)
Add allowed host(s) from the NMS table

Parameters **host** – The host to add

check(*host*)
Check if a host is allowed to access resources

Parameters **host** – The host to check

remove(*host*)
Remove allowed host(s) from the NMS table

Parameters **host** – The host to remove

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()
Returns a summary of the modbus plus statistics

Returns 54 16-bit words representing the status

reset()
This clears all of the modbus plus statistics

summary()
Returns a summary of the modbus plus statistics

Returns 54 16-bit words representing the status

class pymodbus.device.**ModbusDeviceIdentification**(*info=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.

MajorMinorRevision

ModelName

ProductCode

ProductName

UserApplicationName

VendorName

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.**DeviceInformationFactory**
Bases: *pymodbus.interfaces.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**

Bases: *pymodbus.interfaces.Singleton*

This is a global singleotn that controls all system information

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

Counter

Delimiter

Events

Identity

ListenOnly

Mode

Plus

addEvent(*event*)

Adds a new event to the event log

Parameters **event** – A new event to add to the log

clearEvents()

Clears the current list of events

getDiagnostic(*bit*)

This gets the value in the diagnostic register

Parameters **bit** – The bit to get

Returns The current value of the requested bit

getDiagnosticRegister()

This gets the entire diagnostic register

Returns The diagnostic register collection

getEvents()

Returns an encoded collection of the event log.

Returns The encoded events packet

reset()

This clears all of the system counters and the diagnostic register

setDiagnostic(*mapping*)

This sets the value in the diagnostic register

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

14.1.8 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.**DiagnosticStatusRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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

decode(*data*)
Base decoder for a diagnostic request

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

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

function_code = 8

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: *pymodbus.pdu.ModbusResponse*

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*)
Base decoder for a diagnostic response

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

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

function_code = 8

class pymodbus.diag_message.**ReturnQueryDataRequest**(*message=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusRequest*

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*)
Executes 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: *pymodbus.diag_message.DiagnosticStatusResponse*

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.**RestartCommunicationsOptionRequest**(*toggle=False*, ***kwargs*)

Bases: *pymodbus.diag_message.DiagnosticStatusRequest*

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: *pymodbus.diag_message.DiagnosticStatusResponse*

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.**ReturnDiagnosticRegisterRequest**(*data=0*, ***kwargs*)

Bases: pymodbus.diag_message.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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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

sub_function_code = 2

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

Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ForceListenOnlyModeRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.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: *pymodbus.diag_message.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.**ClearCountersRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

The goal is to 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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

The goal is to clear ll counters and the diagnostic register. Also, counters are cleared upon power-up

sub_function_code = 10

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

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ReturnBusCommunicationErrorCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ReturnSlaveMessageCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ReturnSlaveNoResponseCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

sub_function_code = 15

class pymodbus.diag_message.**ReturnSlaveNoReponseCountResponse**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ReturnSlaveNAKCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ReturnSlaveBusyCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ReturnSlaveBusCharacterOverrunCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.

sub_function_code = 18

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

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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

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.**ClearOverrunCountRequest**(*data=0*, ***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

Clears 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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

Clears the overrun error counter and reset the error flag

sub_function_code = 20

class pymodbus.diag_message.**GetClearModbusPlusRequest**(***kwargs*)
Bases: pymodbus.diag_message.DiagnosticStatusSimpleRequest

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()
Base encoder for 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()
Returns 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: pymodbus.diag_message.DiagnosticStatusSimpleResponse

Returns 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

14.1.9 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: *pymodbus.events.ModbusEvent*

Remote device Initiated Communication Restart

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*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

value = 0

class pymodbus.events.**EnteredListenModeEvent**
Bases: *pymodbus.events.ModbusEvent*

Remote device Entered 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*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

value = 4

class pymodbus.events.**ModbusEvent**
Bases: object

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteReceiveEvent**(***kwargs*)
Bases: *pymodbus.events.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*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteSendEvent**(***kwargs*)
Bases: *pymodbus.events.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*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

14.1.10 pymodbus.exceptions module

Pymodbus Exceptions

Custom exceptions to be used in the Modbus code.

exception pymodbus.exceptions.**ModbusException**(*string*)
Bases: exceptions.Exception

Base modbus exception

exception pymodbus.exceptions.**ModbusIOException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from data i/o

exception pymodbus.exceptions.**ParameterException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from invalid parameter

exception pymodbus.exceptions.**NotImplementedException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from not implemented function

exception pymodbus.exceptions.**ConnectionException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from a bad connection

exception pymodbus.exceptions.**NoSuchSlaveException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from making a request to a slave that does not exist

14.1.11 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.**ServerDecoder**
Bases: *pymodbus.interfaces.IModbusDecoder*

Request Message Factory (Server)

To add more implemented functions, simply add them to the list

decode(*message*)
Wrapper to 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*.

class pymodbus.factory.**ClientDecoder**
Bases: *pymodbus.interfaces.IModbusDecoder*

Response Message Factory (Client)

To add more implemented functions, simply add them to the list

decode(*message*)
Wrapper to 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*.

14.1.12 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.**ReadFileRecordRequest**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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 seperate '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*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception 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: *pymodbus.pdu.ModbusResponse*

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*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

function_code = 20

class pymodbus.file_message.**WriteFileRecordRequest**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()
Encodes 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: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()
Encodes the response

Returns The byte encoded message

function_code = 21

class pymodbus.file_message.**ReadFifoQueueRequest**(*address=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception 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: *pymodbus.pdu.ModbusResponse*

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*)
Calculates 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*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

function_code = 24

14.1.13 pymodbus.interfaces module

Pymodbus Interfaces

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

class pymodbus.interfaces.**Singleton**
Bases: object

Singleton base class http://mail.python.org/pipermail/python-list/2007-July/450681.html

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*)
Wrapper to decode a given 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*.

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

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*)
Creates 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

Returns The built packet

checkFrame()
Check and decode the next frame

Returns True if we successful, False otherwise

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

populateResult(*result*)
Populates 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

processIncomingPacket(*data*, *callback*)
The new packet processing 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

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*)
Converts the function code to the datastore to

Parameters **fx** – The function we are working with

Returns one of [d(iscretes),i(inputs),h(oliding),c(oils)

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

Returns The requested values from a:a+c

i = 15

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets 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*)
Validates 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.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.

Returns The payload buffer as a list

14.1.14 pymodbus.mei_message module

Encapsulated Interface (MEI) Transport Messages

class pymodbus.mei_message.**ReadDeviceInformationRequest**(*read_code=None*, *object_id=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception 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: *pymodbus.pdu.ModbusResponse*

classmethod calculateRtuFrameSize(*buffer*)

Calculates 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*)

Decodes a the response

Parameters **data** – The packet data to decode

encode()

Encodes the response

Returns The byte encoded message

function_code = 43

sub_function_code = 14

14.1.15 pymodbus.other_message module

Diagnostic record read/write

Currently not all implemented

class pymodbus.other_message.**ReadExceptionStatusRequest**(***kwargs*)

Bases: *pymodbus.pdu.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*)

Decodes data part of the message.

Parameters **data** – The incoming data

encode()

Encodes the message

execute(*context=None*)

Run a read exeception status request against the store

Returns The populated response

function_code = 7

class pymodbus.other_message.**ReadExceptionStatusResponse**(*status=0*, ***kwargs*)

Bases: *pymodbus.pdu.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*)

Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

function_code = 7

class pymodbus.other_message.**GetCommEventCounterRequest**(**kwargs)
Bases: *pymodbus.pdu.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*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the message

execute(*context=None*)
Run a read exeception status request against the store

Returns The populated response

function_code = 11

class pymodbus.other_message.**GetCommEventCounterResponse**(*count=0*, **kwargs)
Bases: *pymodbus.pdu.ModbusResponse*

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*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

function_code = 11

class pymodbus.other_message.**GetCommEventLogRequest**(**kwargs)
Bases: *pymodbus.pdu.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*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the message

execute(*context=None*)
Run a read exeception status request against the store

Returns The populated response

function_code = 12

class pymodbus.other_message.**GetCommEventLogResponse**(***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

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*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

function_code = 12

class pymodbus.other_message.**ReportSlaveIdRequest**(***kwargs*)
Bases: *pymodbus.pdu.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*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the message

execute(*context=None*)
Run a read exeception status request against the store

Returns The populated response

function_code = 17

class pymodbus.other_message.**ReportSlaveIdResponse**(*identifier='x00'*, *status=True*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The format of a normal response is shown in the following example. The data contents are specific to each type of device.

decode(*data*)
Decodes 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()
Encodes the response

Returns The byte encoded message

function_code = 17

14.1.16 pymodbus.payload module

Modbus Payload Builders

A collection of utilities for building and decoding modbus messages payloads.

class pymodbus.payload.**BinaryPayloadBuilder**(*payload=None*, *byteorder='<'*, *worder='>'*)
Bases: *pymodbus.interfaces.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(endian=Endian.Little)
builder.add_8bit_uint(1)
builder.add_16bit_uint(2)
payload = builder.build()
```

add_16bit_int(*value*)
Adds a 16 bit signed int to the buffer

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

add_16bit_uint(*value*)
Adds a 16 bit unsigned int to the buffer

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

add_32bit_float(*value*)
Adds a 32 bit float to the buffer

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

add_32bit_int(*value*)
Adds a 32 bit signed int to the buffer

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

add_32bit_uint(*value*)
Adds a 32 bit unsigned int to the buffer

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

add_64bit_float(*value*)
Adds a 64 bit float(double) to the buffer

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

add_64bit_int(*value*)
Adds a 64 bit signed int to the buffer

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

add_64bit_uint(*value*)
Adds a 64 bit unsigned int to the buffer

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

add_8bit_int(*value*)
Adds a 8 bit signed int to the buffer

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

add_8bit_uint(*value*)
Adds a 8 bit unsigned int to the buffer

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

add_bits(*values*)
Adds 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 **value** – The value to add to the buffer

add_string(*value*)
Adds 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_registers()
Convert the payload buffer into a 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 reponse 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()
```

decode_16bit_int()
Decodes a 16 bit signed int from the buffer

decode_16bit_uint()
Decodes a 16 bit unsigned int from the buffer

decode_32bit_float()
Decodes a 32 bit float from the buffer

decode_32bit_int()
Decodes a 32 bit signed int from the buffer

decode_32bit_uint()
Decodes a 32 bit unsigned int from the buffer

decode_64bit_float()
Decodes a 64 bit float(double) from the buffer

decode_64bit_int()
Decodes a 64 bit signed int from the buffer

decode_64bit_uint()
Decodes a 64 bit unsigned int from the buffer

decode_8bit_int()
Decodes a 8 bit signed int from the buffer

decode_8bit_uint()
Decodes a 8 bit unsigned int from the buffer

decode_bits()
Decodes a byte worth of bits from the buffer

decode_string(*size=1*)
Decodes a string from the buffer

Parameters **size** – The size of the string to decode

classmethod fromCoils(*klass*, *coils*, *byteorder='<'*)
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.

Parameters

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

Returns An initialized PayloadDecoder

classmethod fromRegisters(*klass*, *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 endianess of the word (when wordcount is >= 2)

Returns An initialized PayloadDecoder

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

14.1.17 pymodbus.pdu module

Contains base classes for modbus request/response/error packets

class pymodbus.pdu.**ModbusRequest**(**kwargs*)
Bases: pymodbus.pdu.ModbusPDU

Base class for a modbus request PDU

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

class pymodbus.pdu.**ModbusResponse**(**kwargs*)
Bases: pymodbus.pdu.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.

should_respond = True

class pymodbus.pdu.**ModbusExceptions**
Bases: *pymodbus.interfaces.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*)
Given an error code, translate it to a string error name.

Parameters **code** – The code number to translate

class pymodbus.pdu.**ExceptionResponse**(*function_code*, *exception_code=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Base class for a modbus exception PDU

ExceptionOffset = 128

decode(*data*)
Decodes a modbus exception response

Parameters **data** – The packet data to decode

encode()
Encodes a modbus exception response

Returns The encoded exception packet

class pymodbus.pdu.**IllegalFunctionRequest**(*function_code*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

Defines 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*)
This is here so this failure will run correctly

Parameters **data** – Not used

execute(*context*)
Builds an illegal function request error response

Parameters **context** – The current context for the message

Returns The error response packet

14.1.18 pymodbus.register_read_message module

Register Reading Request/Response

class pymodbus.register_read_message.**ReadHoldingRegistersRequest**(*address=None*, *count=None*, ***kwargs*)
Bases: pymodbus.register_read_message.ReadRegistersRequestBase

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 response, exception message otherwise

function_code = 3

class pymodbus.register_read_message.**ReadHoldingRegistersResponse**(*values=None*, ***kwargs*)

Bases: pymodbus.register_read_message.ReadRegistersResponseBase

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.

function_code = 3

class pymodbus.register_read_message.**ReadInputRegistersRequest**(*address=None*, *count=None*, ***kwargs*)

Bases: pymodbus.register_read_message.ReadRegistersRequestBase

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 response, exception message otherwise

function_code = 4

class pymodbus.register_read_message.**ReadInputRegistersResponse**(*values=None*, ***kwargs*)

Bases: pymodbus.register_read_message.ReadRegistersResponseBase

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.

function_code = 4

class pymodbus.register_read_message.**ReadWriteMultipleRegistersRequest**(***kwargs*)

Bases: *pymodbus.pdu.ModbusRequest*

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()

Encodes 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 response, exception message otherwise

function_code = 23

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: *pymodbus.pdu.ModbusResponse*

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.

decode(*data*)
Decode the register response packet

Parameters **data** – The response to decode

encode()
Encodes the response packet

Returns The encoded packet

function_code = 23

14.1.19 pymodbus.register_write_message module

Register Writing Request/Response Messages

class pymodbus.register_write_message.**WriteSingleRegisterRequest**(*address=None*, *value=None*, ***kwargs*)

Bases: *pymodbus.pdu.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()
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: *pymodbus.pdu.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

class pymodbus.register_write_message.**WriteMultipleRegistersRequest**(*address=None*, *values=None*, ***kwargs*)

Bases: *pymodbus.pdu.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

class pymodbus.register_write_message.**WriteMultipleRegistersResponse**(*address=None*, *count=None*, ***kwargs*)

Bases: *pymodbus.pdu.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.**MaskWriteRegisterRequest**(*address=0*, *and_mask=65535*, *or_mask=0*, ***kwargs*)

Bases: *pymodbus.pdu.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*)

Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()

Encodes 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: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request. The response is returned after the register has been written.

decode(*data*)

Decodes a the response

Parameters **data** – The packet data to decode

encode()

Encodes the response

Returns The byte encoded message

function_code = 22

14.1.20 pymodbus.transaction module

Collection of transaction based abstractions

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

Bases: pymodbus.transaction.ModbusTransactionManager

Impelements a transaction for a manager where the results are returned in a FIFO manner.

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

Adds a transaction to the handler

This holds the requets 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*)
Removes a transaction matching the referenced tid

Parameters **tid** – The transaction to remove

getTransaction(*tid*)
Returns a transaction matching the referenced tid

If the transaction does not exist, None is returned

Parameters **tid** – The transaction to retrieve

class pymodbus.transaction.**DictTransactionManager**(*client*, ***kwargs*)
Bases: pymodbus.transaction.ModbusTransactionManager

Impelements a transaction for a manager where the results are keyed based on the supplied transaction id.

addTransaction(*request*, *tid=None*)
Adds a transaction to the handler

This holds the requets 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*)
Removes a transaction matching the referenced tid

Parameters **tid** – The transaction to remove

getTransaction(*tid*)
Returns a transaction matching the referenced tid

If the transaction does not exist, None is returned

Parameters **tid** – The transaction to retrieve

class pymodbus.transaction.**ModbusSocketFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

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*)
Adds 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*)
Creates 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

getFrame()
Return the next frame from the buffered data

Returns The next full frame buffer

getRawFrame()
Returns 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

populateResult(*result*)
Populates the modbus result with the transport specific header information (pid, tid, uid, checksum, etc)

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing 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

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*)
Bases: *pymodbus.interfaces.IModbusFramer*

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 transmist at least x many characters. In this case it is 3.5 characters. Also, if we recieve 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*)
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*)
Creates 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.

getFrame()
Get the next frame from the buffer

Returns The frame data or ‘’

getRawFrame()
Returns 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 know that there is still data in the buffer.

Returns True if ready, False otherwise

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

This method examines *self._buffer* and writes meta information into *self._header*. It calculates only the values for headers that are not already in the dictionary.

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

populateResult(*result*)
Populates the modbus result header

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

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing 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

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.**ModbusAsciiFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

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*)
Creates 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

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

populateResult(*result*)
Populates the modbus result header

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

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing 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

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*)
Bases: *pymodbus.interfaces.IModbusFramer*

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*)
Creates 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

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

populateResult(*result*)
Populates the modbus result header

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

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing 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

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).

14.1.21 pymodbus.utilities module

Modbus Utilities

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

pymodbus.utilities.**pack_bitstring**(*bits*)
Creates 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.**unpack_bitstring**(*string*)

Creates bit array out of a string

Parameters **string** – The modbus data packet to decode

example:

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

pymodbus.utilities.**default**(*value*)

Given a python object, return the default value of that object.

Parameters **value** – The value to get the default of

Returns The default value

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

Computes 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.**checkCRC**(*data*, *check*)

Checks 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.**computeLRC**(*data*)

Used 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 appendex B of the serial line modbus description.

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

Returns The calculated LRC

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

Checks 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.**rtuFrameSize**(*data*, *byte_count_pos*)

Calculates 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).

14.1.22 pymodbus.version module

Handle the version information here; you should only have to change the version tuple.

Since we are using twisted's version class, we can also query the svn version as well using the local .entries file.

14.1.23 Module contents

Pymodbus: Modbus Protocol Implementation

TwistedModbus is built on top of the code developed 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 <hynek@swac.cz>

Released under the the BSD license

CHAPTER 15

Indices and tables

- genindex
- modindex
- search

Python Module Index

p

Index

Symbols

A

B

C

D

E

F

G

I

K

L

M

N

O

P

R

S