Pymodbus Documentation

Release 1.0

Galen Collins

January 04, 2017

Contents

Contents:

CHAPTER 1

Pymodbus Library Examples

What follows is a collection of examples using the pymodbus library in various ways

1.1 Example Library Code

1.1.1 Asynchronous Client Example

The asynchronous client functions in the same way as the synchronous client, however, the asynchronous client uses twisted to return deferreds for the response result. Just like the synchronous version, it works against TCP, UDP, serial ASCII, and serial RTU devices.

Below an asynchronous tcp client is demonstrated running against a reference server. If you do not have a device to test with, feel free to run a pymodbus server instance or start the reference tester in the tools directory.

```
#!/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).
#---------------------------------------------------------------------------#
def beginAsynchronousTest(client):
    rq = client.write_coil(1, True)
    rr = client.read_coils(1,1)
    dassert(rq, lambda r: r.function_code < 0x80)     # test that we are not an error
    dassert(rr, lambda r: r.bits[0] == True)          # test the expected value

    rq = client.write_coils(1, [True]*8)
    rr = client.read_coils(1,8)
    dassert(rq, lambda r: r.function_code < 0x80)     # test that we are not an error
    dassert(rr, lambda r: r.bits == [True]*8)         # test the expected value

    rq = client.write_coils(1, [False]*8)
    rr = client.read_discrete_inputs(1,8)
    dassert(rq, lambda r: r.function_code < 0x80)     # test that we are not an error
    dassert(rr, lambda r: r.bits == [True]*8)         # test the expected value

    rq = client.write_register(1, 10)
    rr = client.read_holding_registers(1,1)
    dassert(rq, lambda r: r.function_code < 0x80)     # test that we are not an error
    dassert(rr, lambda r: r.registers[0] == 10)       # test the expected value

    rq = client.write_registers(1, [10]*8)
    rr = client.read_input_registers(1,8)
    dassert(rq, lambda r: r.function_code < 0x80)     # test that we are not an 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)
    rr = client.read_input_registers(1,8)
    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.
#---------------------------------------------------------------------------#
defer = protocol.ClientCreator(reactor, ModbusClientProtocol
        ).connectTCP("localhost", Defaults.Port)
defer.addCallback(beginAsynchronousTest)
reactor.run()
```

1.1.2 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)

#---------------------------------------------------------------------------#
# 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/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", 502))
#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)
```

1.1.3 Asynchronous Processor Example

Below is a simplified asynchronous client skeleton that was submitted by a user of the library. It can be used as a guide for implementing more complex pollers or state machines.

Feel free to test it against whatever device you currently have available. If you do not have a device to test with, feel free to run a pymodbus server instance or start the reference tester in the tools directory.

```
#!/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/ttyS0"
STATUS_REGS  = (1, 2)
STATUS_COILS = (1, 3)
CLIENT_DELAY = 1


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

1.1.4 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
from pymodbus.client.sync import ModbusTcpClient as ModbusClient

#---------------------------------------------------------------------------#
# 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 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(merror.IllegalValue)
        if not context.validate(self.function_code, self.address, self.count):
            return self.doException(merror.IllegalAddress)
        values = context.getValues(self.function_code, self.address, self.count)
        return CustomModbusResponse(values)

#---------------------------------------------------------------------------#
# This could also have been defined as
#---------------------------------------------------------------------------#
from pymodbus.bit_read_message import ReadCoilsRequest

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

1.1.5 Modbus Logging Example

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

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

1.1.6 Modbus Payload Building/Decoding Example

```
#!/usr/bin/env python
'''
Pymodbus Payload Building/Decoding Example
--------------------------------------------------------------------------
'''
from pymodbus.constants import Endian
from pymodbus.payload import BinaryPayloadDecoder
from pymodbus.payload import BinaryPayloadBuilder
from pymodbus.client.sync import ModbusTcpClient as ModbusClient

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

#---------------------------------------------------------------------------#
# We are going to use a simple client to send our requests
#---------------------------------------------------------------------------#
client = ModbusClient('127.0.0.1')
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
# - an 8 bit int 0x12
```

```
# - an 8 bit bitstring [0,1,0,1,1,0,1,0]
#---------------------------------------------------------------------------#
builder = BinaryPayloadBuilder(endian=Endian.Little)
builder.add_string('abcdefgh')
builder.add_32bit_float(22.34)
builder.add_16bit_uint(0x1234)
builder.add_8bit_int(0x12)
builder.add_bits([0,1,0,1,1,0,1,0])
payload = builder.build()
address = 0x01
result  = client.write_registers(address, payload, skip_encode=True)

#---------------------------------------------------------------------------#
# 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
# - an 8 bit int 0x12
# - an 8 bit bitstring [0,1,0,1,1,0,1,0]
#---------------------------------------------------------------------------#
address = 0x01
count   = 8
result  = client.read_input_registers(address, count)
decoder = BinaryPayloadDecoder.fromRegisters(result.registers, endian=Endian.Little)
decoded = {
    'string': decoder.decode_string(8),
    'float': decoder.decode_32bit_float(),
    '16uint': decoder.decode_16bit_uint(),
    '8int': decoder.decode_8bit_int(),
    'bits': decoder.decode_bits(),
}

print "-" * 60
print "Decoded Data"
print "-" * 60
for name, value in decoded.iteritems():
    print ("%s\t" % name), value

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

1.1.7 Modbus Payload Server Context Building 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)

#---------------------------------------------------------------------------#
# build your payload
#---------------------------------------------------------------------------#
builder = BinaryPayloadBuilder(endian=Endian.Little)
builder.add_string('abcdefgh')
builder.add_32bit_float(22.34)
builder.add_16bit_uint(0x1234)
builder.add_8bit_int(0x12)
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))
```

1.1.8 Synchronous Client Example

It should be noted that each request will block waiting for the result. If asynchronous behaviour is required, please use the asynchronous client implementations. The synchronous client, works against TCP, UDP, serial ASCII, and serial RTU devices.

The synchronous client exposes the most popular methods of the modbus protocol, however, if you want to execute other methods against the device, simple create a request instance and pass it to the execute method.

Below an synchronous tcp client is demonstrated running against a reference server. If you do not have a device to test with, feel free to run a pymodbus server instance or start the reference tester in the tools directory.

```
#!/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)

#---------------------------------------------------------------------------#
# 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=502)
#client = ModbusClient(method='ascii', port='/dev/pts/2', timeout=1)
#client = ModbusClient(method='rtu', port='/dev/pts/2', 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`
#---------------------------------------------------------------------------#
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 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).
#---------------------------------------------------------------------------#
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

rq = client.write_coils(1, [True]*8)
rr = client.read_coils(1,8)
assert(rq.function_code < 0x80)     # test that we are not an error
assert(rr.bits == [True]*8)         # test the expected value

rq = client.write_coils(1, [False]*8)
rr = client.read_discrete_inputs(1,8)
assert(rq.function_code < 0x80)     # test that we are not an error
assert(rr.bits == [False]*8)         # test the expected value
```

```
rq = client.write_register(1, 10)
rr = client.read_holding_registers(1,1)
assert(rq.function_code < 0x80)     # test that we are not an error
assert(rr.registers[0] == 10)       # test the expected value

rq = client.write_registers(1, [10]*8)
rr = client.read_input_registers(1,8)
assert(rq.function_code < 0x80)     # test that we are not an error
assert(rr.registers == [10]*8)      # test the expected value

arguments = {
    'read_address':    1,
    'read_count':      8,
    'write_address':   1,
    'write_registers': [20]*8,
}
rq = client.readwrite_registers(**arguments)
rr = client.read_input_registers(1,8)
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()
```

1.1.9 Synchronous Client Extended 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

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

#---------------------------------------------------------------------------#
# 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('127.0.0.1')
client.connect()

#---------------------------------------------------------------------------#
# 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 *

#---------------------------------------------------------------------------#
# 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 reference.
#---------------------------------------------------------------------------#

#---------------------------------------------------------------------------#
# information requests
#---------------------------------------------------------------------------#
rq = ReadDeviceInformationRequest()
rr = client.execute(rq)
#assert(rr == None)                              # not supported by reference
assert(rr.function_code < 0x80)                 # test that we are not an error
assert(rr.information[0] == 'proconX Pty Ltd')   # test the vendor name
assert(rr.information[1] == 'FT-MBSV')          # test the product code
assert(rr.information[2] == 'EXPERIMENTAL')     # test the code revision

rq = ReportSlaveIdRequest()
rr = client.execute(rq)
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

rq = ReadExceptionStatusRequest()
rr = client.execute(rq)
#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

rq = GetCommEventCounterRequest()
rr = client.execute(rq)
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

rq = GetCommEventLogRequest()
rr = client.execute(rq)
#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
#-----------------------------------------------------------------------#
rq = ReturnQueryDataRequest()
rr = client.execute(rq)
assert(rr == None)                                 # not supported by reference
#assert(rr.message[0] == 0x0000)                   # test the resulting message

rq = RestartCommunicationsOptionRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference
#assert(rr.message == 0x0000)                      # test the resulting message

rq = ReturnDiagnosticRegisterRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference

rq = ChangeAsciiInputDelimiterRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference

rq = ForceListenOnlyModeRequest()
client.execute(rq)                                 # does not send a response

rq = ClearCountersRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference

rq = ReturnBusCommunicationErrorCountRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference

rq = ReturnBusExceptionErrorCountRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference

rq = ReturnSlaveMessageCountRequest()
rr = client.execute(rq)
#assert(rr == None)                                # not supported by reference
```

```
rq = ReturnSlaveNoResponseCountRequest()
rr = client.execute(rq)
#assert(rr == None)                            # not supported by reference

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

rq = ReturnSlaveBusyCountRequest()
rr = client.execute(rq)
#assert(rr == None)                            # not supported by reference

rq = ReturnSlaveBusCharacterOverrunCountRequest()
rr = client.execute(rq)
#assert(rr == None)                            # not supported by reference

rq = ReturnIopOverrunCountRequest()
rr = client.execute(rq)
#assert(rr == None)                            # not supported by reference

rq = ClearOverrunCountRequest()
rr = client.execute(rq)
#assert(rr == None)                            # not supported by reference

rq = GetClearModbusPlusRequest()
rr = client.execute(rq)
#assert(rr == None)                            # not supported by reference

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

1.1.10 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)

#---------------------------------------------------------------------------#
# 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/bashwork/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/pts/3', timeout=.0
```

1.1.11 Synchronous Client Performance Check

Below is a quick example of how to test the performance of a tcp modbus device using the synchronous tcp client. If you do not have a device to test with, feel free to run a pymodbus server instance or start the reference tester in the tools directory.

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

The following is an quick performance check of the synchronous
modbus client.
'''
#---------------------------------------------------------------------------#
# import the necessary modules
#---------------------------------------------------------------------------#
import logging, os
from time import time
```

```
from multiprocessing import log_to_stderr
from pymodbus.client.sync import ModbusTcpClient

#---------------------------------------------------------------------------#
# choose between threads or processes
#---------------------------------------------------------------------------#
#from multiprocessing import Process as Worker
from threading import Thread as Worker

#---------------------------------------------------------------------------#
# 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 = 1
cycles  = 10000
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)
        while count < cycles:
            result = client.read_holding_registers(10, 1).getRegister(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.
#---------------------------------------------------------------------------#
```

```
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))
```

1.1.12 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)

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

1.1.13 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.iterkeys()}
        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

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

1.1.14 Changing Default Framers

```
#!/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)

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

1.1.15 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
                    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 reader
                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
            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)
```

1.2 Custom Pymodbus Code

1.2.1 Redis Datastore Example

```
import redis
from pymodbus.interfaces import IModbusSlaveContext
from pymodbus.utilities import pack_bitstring, unpack_bitstring

#---------------------------------------------------------------------------#
# Logging
#---------------------------------------------------------------------------#
```

```
import logging;
_logger = logging.getLogger(__name__)


#---------------------------------------------------------------------------#
# Context
#---------------------------------------------------------------------------#
class RedisSlaveContext(IModbusSlaveContext):
    '''
    This is a modbus slave context using redis as a backing
    store.
    '''

    def __init__(self, **kwargs):
        ''' Initializes the datastores

        :param host: The host to connect to
        :param port: The port to connect to
        :param prefix: A prefix for the keys
        '''
        host = kwargs.get('host', 'localhost')
        port = kwargs.get('port', 6379)
        self.prefix = kwargs.get('prefix', 'pymodbus')
        self.client = kwargs.get('client', redis.Redis(host=host, port=port))
        self.__build_mapping()

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

        :returns: A string representation of the context
        '''
        return "Redis Slave Context %s" % self.client

    def reset(self):
        ''' Resets all the datastores to their default values '''
        self.client.flushall()

    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
        '''
        address = address + 1  # section 4.4 of specification
        _logger.debug("validate[%d] %d:%d" % (fx, address, count))
        return self.__val_callbacks[self.decode(fx)](address, count)

    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
        '''
        address = address + 1  # section 4.4 of specification
```

```
        _logger.debug("getValues[%d] %d:%d" % (fx, address, count))
        return self.__get_callbacks[self.decode(fx)](address, count)

    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
        '''
        address = address + 1  # section 4.4 of specification
        _logger.debug("setValues[%d] %d:%d" % (fx, address, len(values)))
        self.__set_callbacks[self.decode(fx)](address, values)

    #--------------------------------------------------------------------------#
    # Redis Helper Methods
    #--------------------------------------------------------------------------#
    def __get_prefix(self, key):
        ''' This is a helper to abstract getting bit values

        :param key: The key prefix to use
        :returns: The key prefix to redis
        '''
        return "%s:%s" % (self.prefix, key)

    def __build_mapping(self):
        '''
        A quick helper method to build the function
        code mapper.
        '''
        self.__val_callbacks = {
            'd' : lambda o, c: self.__val_bit('d', o, c),
            'c' : lambda o, c: self.__val_bit('c', o, c),
            'h' : lambda o, c: self.__val_reg('h', o, c),
            'i' : lambda o, c: self.__val_reg('i', o, c),
        }
        self.__get_callbacks = {
            'd' : lambda o, c: self.__get_bit('d', o, c),
            'c' : lambda o, c: self.__get_bit('c', o, c),
            'h' : lambda o, c: self.__get_reg('h', o, c),
            'i' : lambda o, c: self.__get_reg('i', o, c),
        }
        self.__set_callbacks = {
            'd' : lambda o, v: self.__set_bit('d', o, v),
            'c' : lambda o, v: self.__set_bit('c', o, v),
            'h' : lambda o, v: self.__set_reg('h', o, v),
            'i' : lambda o, v: self.__set_reg('i', o, v),
        }

    #--------------------------------------------------------------------------#
    # Redis discrete implementation
    #--------------------------------------------------------------------------#
    __bit_size    = 16
    __bit_default = '\x00' * (__bit_size % 8)

    def __get_bit_values(self, key, offset, count):
        ''' This is a helper to abstract getting bit values
```

```
        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        '''
        key = self.__get_prefix(key)
        s = divmod(offset, self.__bit_size)[0]
        e = divmod(offset + count, self.__bit_size)[0]

        request  = ('%s:%s' % (key, v) for v in range(s, e + 1))
        response = self.client.mget(request)
        return response

    def __val_bit(self, key, offset, count):
        ''' Validates that the given range is currently set in redis.
        If any of the keys return None, then it is invalid.

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        '''
        response = self.__get_bit_values(key, offset, count)
        return None not in response

    def __get_bit(self, key, offset, count):
        '''

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        '''
        response = self.__get_bit_values(key, offset, count)
        response = (r or self.__bit_default for r in response)
        result = ''.join(response)
        result = unpack_bitstring(result)
        return result[offset:offset + count]

    def __set_bit(self, key, offset, values):
        '''

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param values: The values to set
        '''
        count = len(values)
        s = divmod(offset, self.__bit_size)[0]
        e = divmod(offset + count, self.__bit_size)[0]
        value = pack_bitstring(values)

        current = self.__get_bit_values(key, offset, count)
        current = (r or self.__bit_default for r in current)
        current = ''.join(current)
        current = current[0:offset] + value + current[offset + count:]
        final   = (current[s:s + self.__bit_size] for s in range(0, count, self.__bit_size))

        key = self.__get_prefix(key)
        request = ('%s:%s' % (key, v) for v in range(s, e + 1))
        request = dict(zip(request, final))
        self.client.mset(request)
```

```
    #---------------------------------------------------------------------------#
    # Redis register implementation
    #---------------------------------------------------------------------------#
    __reg_size    = 16
    __reg_default = '\x00' * (__reg_size % 8)

    def __get_reg_values(self, key, offset, count):
        ''' This is a helper to abstract getting register values

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        '''
        key = self.__get_prefix(key)
        #s = divmod(offset, self.__reg_size)[0]
        #e = divmod(offset+count, self.__reg_size)[0]

        #request  = ('%s:%s' % (key, v) for v in range(s, e + 1))
        request  = ('%s:%s' % (key, v) for v in range(offset, count + 1))
        response = self.client.mget(request)
        return response

    def __val_reg(self, key, offset, count):
        ''' Validates that the given range is currently set in redis.
        If any of the keys return None, then it is invalid.

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        '''
        response = self.__get_reg_values(key, offset, count)
        return None not in response

    def __get_reg(self, key, offset, count):
        '''

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        '''
        response = self.__get_reg_values(key, offset, count)
        response = [r or self.__reg_default for r in response]
        return response[offset:offset + count]

    def __set_reg(self, key, offset, values):
        '''

        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param values: The values to set
        '''
        count = len(values)
        #s = divmod(offset, self.__reg_size)
        #e = divmod(offset+count, self.__reg_size)

        #current = self.__get_reg_values(key, offset, count)

        key = self.__get_prefix(key)
```

```
        request = ('%s:%s' % (key, v) for v in range(offset, count + 1))
        request = dict(zip(request, values))
        self.client.mset(request)
```

1.2.2 Database Datastore Example

```
import sqlalchemy
import sqlalchemy.types as sqltypes
from sqlalchemy.sql import and_
from sqlalchemy.schema import UniqueConstraint
from sqlalchemy.sql.expression import bindparam

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

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


#---------------------------------------------------------------------------#
# Context
#---------------------------------------------------------------------------#
class DatabaseSlaveContext(IModbusSlaveContext):
    '''
    This creates a modbus data model with each data access
    stored in its own personal block
    '''

    def __init__(self, *args, **kwargs):
        ''' Initializes the datastores

        :param kwargs: Each element is a ModbusDataBlock
        '''
        self.table = kwargs.get('table', 'pymodbus')
        self.database = kwargs.get('database', 'sqlite:///pymodbus.db')
        self.__db_create(self.table, self.database)

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

        :returns: A string representation of the context
        '''
        return "Modbus Slave Context"

    def reset(self):
        ''' Resets all the datastores to their default values '''
        self._metadata.drop_all()
        self.__db_create(self.table, self.database)
        raise NotImplementedException()  # TODO drop table?

    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
        '''
        address = address + 1  # section 4.4 of specification
        _logger.debug("validate[%d] %d:%d" % (fx, address, count))
        return self.__validate(self.decode(fx), address, count)

    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
        '''
        address = address + 1  # section 4.4 of specification
        _logger.debug("get-values[%d] %d:%d" % (fx, address, count))
        return self.__get(self.decode(fx), address, count)

    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
        '''
        address = address + 1  # section 4.4 of specification
        _logger.debug("set-values[%d] %d:%d" % (fx, address, len(values)))
        self.__set(self.decode(fx), address, values)

    #--------------------------------------------------------------------------#
    # Sqlite Helper Methods
    #--------------------------------------------------------------------------#
    def __db_create(self, table, database):
        ''' A helper method to initialize the database and handles

        :param table: The table name to create
        :param database: The database uri to use
        '''
        self._engine = sqlalchemy.create_engine(database, echo=False)
        self._metadata = sqlalchemy.MetaData(self._engine)
        self._table = sqlalchemy.Table(table, self._metadata,
            sqlalchemy.Column('type', sqltypes.String(1)),
            sqlalchemy.Column('index', sqltypes.Integer),
            sqlalchemy.Column('value', sqltypes.Integer),
            UniqueConstraint('type', 'index', name='key'))
        self._table.create(checkfirst=True)
        self._connection = self._engine.connect()

    def __get(self, type, offset, count):
        '''

        :param type: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        :returns: The resulting values
```

```
        '''
        query  = self._table.select(and_(
            self._table.c.type == type,
            self._table.c.index >= offset,
            self._table.c.index <= offset + count))
        query = query.order_by(self._table.c.index.asc())
        result = self._connection.execute(query).fetchall()
        return [row.value for row in result]

    def __build_set(self, type, offset, values, p=''):
        ''' A helper method to generate the sql update context

        :param type: The key prefix to use
        :param offset: The address offset to start at
        :param values: The values to set
        '''
        result = []
        for index, value in enumerate(values):
            result.append({
                p + 'type'  : type,
                p + 'index' : offset + index,
                    'value' : value
            })
        return result

    def __set(self, type, offset, values):
        '''

        :param key: The type prefix to use
        :param offset: The address offset to start at
        :param values: The values to set
        '''
        context = self.__build_set(type, offset, values)
        query   = self._table.insert()
        result  = self._connection.execute(query, context)
        return result.rowcount == len(values)

    def __update(self, type, offset, values):
        '''

        :param type: The type prefix to use
        :param offset: The address offset to start at
        :param values: The values to set
        '''
        context = self.__build_set(type, offset, values, p='x_')
        query   = self._table.update().values(name='value')
        query   = query.where(and_(
            self._table.c.type  == bindparam('x_type'),
            self._table.c.index == bindparam('x_index')))
        result  = self._connection.execute(query, context)
        return result.rowcount == len(values)

    def __validate(self, key, offset, count):
        '''
        :param key: The key prefix to use
        :param offset: The address offset to start at
        :param count: The number of bits to read
        :returns: The result of the validation
```

```
        '''
        query  = self._table.select(and_(
            self._table.c.type == type,
            self._table.c.index >= offset,
            self._table.c.index <= offset + count))
        result = self._connection.execute(query)
        return result.rowcount == count
```

1.2.3 Binary Coded Decimal 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

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 xrange(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"]
```

1.2.4 Modicon Encoded 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 xrange(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_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 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 ModiconPayloadDecoder(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 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__ = ["BcdPayloadBuilder", "BcdPayloadDecoder"]
```

1.2.5 Modbus Message Generator Example

This is an example of a utility that will build examples of modbus messages in all the available formats in the pymodbus package.

Program Source

```
#!/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 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 *

#---------------------------------------------------------------------------#
# 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'              : 0x01,
}


#---------------------------------------------------------------------------#
# 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
    '''
    messages = _request_messages if options.messages == 'tx' else _response_messages
    for message in messages:
```

```
        message = message(**_arguments)
        print "%-44s = " % message.__class__.__name__,
        packet = framer.buildPacket(message)
        if not options.ascii:
            packet = packet.encode('hex') + '\n'
        print 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, 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()
```

Example Request Messages

```
# --------------------------------------------------------------
# What follows is a collection of encoded messages that can
# be used to test the message-parser.  Simply uncomment the
# messages you want decoded and run the message parser with
# the given arguments. What follows is the listing of messages
# that are encoded in each format:
#
# - 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
# - ReturnSlaveNoReponseCountRequest
# - ReturnSlaveNAKCountRequest
# - ReturnSlaveBusyCountRequest
# - ReturnSlaveBusCharacterOverrunCountRequest
# - ReturnIopOverrunCountRequest
# - ClearOverrunCountRequest
# - GetClearModbusPlusRequest
# --------------------------------------------------------------
# Modbus TCP Messages
# --------------------------------------------------------------
# [         MBAP Header         ] [ Function Code] [ Data ]
# [ tid ][ pid ][ length ][ uid ]
#   2b     2b     2b        1b           1b           Nb
```

```
#
# ./message-parser -b -p tcp -f messages
# ---------------------------------------------------------------
#00010000000601030012000 8
#00010000000601020012000 8
#00010000000601040012000 8
#00010000000601010012000 8
#00010000000801 0f0012000801ff
#0001000000170110001200081000010001000100010001000100010001
#00010000000601060012000 1
#00010000000601050012ff00
#0001000000 1b0117001200080000000810000100010001000100010001000100010001
#0001000000020107
#0001000000020 10b
#0001000000020 10c
#0001000000020111
#0001000000030114 00
#0001000000030115 00
#0001000000080116 0012ffff0000
#0001000000040118 0012
#0001000000050 12b0e0100
#0001000000060108 00000000
#0001000000060108 00010000
#0001000000060108 00020000
#0001000000060108 00030000
#0001000000060108 00040000
#000100000006010 8000a0000
#000100000006010 8000b0000
#000100000006010 8000c0000
#000100000006010 8000d0000
#000100000006010 8000e0000
#000100000006010 8000f0000
#0001000000060108 00100000
#0001000000060108 00110000
#0001000000060108 00120000
#0001000000060108 00130000
#0001000000060108 00140000
#0001000000060108 00150000
# ---------------------------------------------------------------
# Modbus RTU Messages
# ---------------------------------------------------------------
# [Address ][ Function Code] [ Data ][ CRC ]
#  1b         1b               Nb      2b
#
# ./message-parser -b -p rtu -f messages
# ---------------------------------------------------------------
#010300120008e409
#010200120008d9c9
#01040012000851c9
#0101001200089dc9
#010f0012000801ff06d6
#0110001200081000010001000100010001000100010001d551
#010600120001e80f
#01050012ff002c3f
#0117001200080000000810000100010001000100010001000100010001e6f8
#010741e2
#010b41e7
#010c0025
```

```
#0111c02c
#0114002f00
#0115002e90
#01160012ffff00004e21
#0118001201d2
#012b0e01007077
#010800000000e00b
#010800010000b1cb
#01080002000041cb
#010800030000100b
#010800040000a1ca
#0108000a0000c009
#0108000b000091c9
#0108000c00002008
#0108000d000071c8
#0108000e000081c8
#0108000f0000d008
#010800100000e1ce
#010800110000b00e
#010800120000400e
#01080013000011ce
#010800140000a00f
#010800150000f1cf
# ------------------------------------------------------------
# Modbus ASCII Messages
# ------------------------------------------------------------
# [ Start ][Address ][ Function ][ Data ][ LRC ][ End ]
#   1c        2c         2c         Nc     2c      2c
#
# ./message-parser -a -p ascii -f messages
# ------------------------------------------------------------
#:010300120008E2
#:010200120008E3
#:010400120008E1
#:010100120008E4
#:010F0012000801FFD6
#:0110001200081000010001000100010001000100010001BD
#:010600120001E6
#:01050012FF00E9
#:0117001200080000000810000100010001000100010001000100010001AE
#:0107F8
#:010BF4
#:010CF3
#:0111EE
#:011400EB
#:011500EA
#:01160012FFFF0000D9
#:01180012D5
#:012B0E0100C5
#:010800000000F7
#:010800010000F6
#:010800020000F5
#:010800030000F4
#:010800040000F3
#:0108000A0000ED
#:0108000B0000EC
#:0108000C0000EB
#:0108000D0000EA
```

```
#:0108000E0000E9
#:0108000F0000E8
#:010800100000E7
#:010800110000E6
#:010800120000E5
#:010800130000E4
#:010800140000E3
#:010800150000E2
# ------------------------------------------------------------
# Modbus Binary Messages
# ------------------------------------------------------------
# [ Start ][Address ][ Function ][ Data ][ CRC ][ End ]
#   1b        1b          1b          Nb     2b     1b
#
# ./message-parser -b -p binary -f messages
# ------------------------------------------------------------
#7b010300120008e4097d
#7b010200120008d9c97d
#7b01040012000851c97d
#7b0101001200089dc97d
#7b010f0012000801ff06d67d
#7b011000120008100001000100010001000100010001000100010001d5517d
#7b010600120001e80f7d
#7b01050012ff002c3f7d
#7b011700120008000000081000010001000100010001000100010001e6f87d
#7b010741e27d
#7b010b41e77d
#7b010c00257d
#7b0111c02c7d
#7b0114002f007d
#7b0115002e907d
#7b01160012ffff00004e217d
#7b0118001201d27d
#7b012b0e010070777d
#7b010800000000e00b7d
#7b010800010000b1cb7d
#7b01080002000041cb7d
#7b010800030000100b7d
#7b010800040000a1ca7d
#7b0108000a0000c0097d
#7b0108000b000091c97d
#7b0108000c000020087d
#7b0108000d000071c87d
#7b0108000e000081c87d
#7b0108000f0000d0087d
#7b010800100000e1ce7d
#7b010800110000b00e7d
#7b010800120000400e7d
#7b01080013000011ce7d
#7b010800140000a00f7d
#7b010800150000f1cf7d
```

Example Response Messages

```
# ------------------------------------------------------------
# What follows is a collection of encoded messages that can
# be used to test the message-parser.  Simply uncomment the
```

```
# messages you want decoded and run the message parser with
# the given arguments. What follows is the listing of messages
# that are encoded in each format:
#
# - 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
# -------------------------------------------------------------
# Modbus TCP Messages
# -------------------------------------------------------------
# [         MBAP Header          ] [ Function Code] [ Data ]
# [ tid ][ pid ][ length ][ uid ]
#   2b     2b     2b        1b           1b           Nb
#
# ./message-parser -b -p tcp -f messages
# -------------------------------------------------------------
#00010000001301031000010001000100010001000100010001
#000100000004010201ff
#00010000001301041000010001000100010001000100010001
#000100000004010101ff
#00010000000601 0f00120008
#00010000000601 1000120008
#00010000000601 0600120001
#00010000000601 050012ff00
#00010000001301171000010001000100010001000100010001
#000100000003010700
```

```
#000100000006010b00000008
#000100000009010c06000000000000
#00010000000501110300ff
#000100000003011400
#000100000003011500
#00010000000801160012ffff0000
#0001000000160118001200100001000100010001000100010001
#000100000008012b0e0183000000
#000100000006010800000000
#000100000006010800010000
#000100000006010800020000
#000100000006010800030000
#000100000000401080004
#000100000060108000a0000
#000100000060108000b0000
#000100000060108000c0000
#000100000060108000d0000
#000100000060108000e0000
#000100000060108000f0000
#000100000006010800100000
#000100000006010800110000
#000100000006010800120000
#000100000006010800130000
#000100000006010800140000
#000100000006010800150000
# ---------------------------------------------------------------
# Modbus RTU Messages
# ---------------------------------------------------------------
# [Address ][ Function Code] [ Data ][ CRC ]
#  1b         1b               Nb      2b
#
# ./message-parser -b -p rtu -f messages
# ---------------------------------------------------------------
#0103100001000100010001000100010001000193b4
#010201ffe1c8
#0104100001000100010001000100010001000122c1
#010101ff11c8
#010f00120008f408
#01100012000861ca
#010600120001e80f
#01050012ff002c3f
#0117100001000100010001000100010001000100010001d640
#0107002230
#010b00000008a5cd
#010c060000000000006135
#01110300ffacbc
#0114002f00
#0115002e90
#01160012ffff00004e21
#011800120010000100010001000100010001000100010001d74d
#012b0e01830000000faf
#010800000000e00b
#010800010000b1cb
#01080002000041cb
#010800030000100b
#0108000481d9
#0108000a0000c009
#0108000b000091c9
```

```
#0108000c00002008
#0108000d000071c8
#0108000e000081c8
#0108000f0000d008
#010800100000e1ce
#010800110000b00e
#010800120000400e
#01080013000011ce
#010800140000a00f
#010800150000f1cf
# ------------------------------------------------------------
# Modbus ASCII Messages
# ------------------------------------------------------------
# [ Start ][Address ][ Function ][ Data ][ LRC ][ End ]
#   1c        2c         2c         Nc     2c      2c
#
# ./message-parser -a -p ascii -f messages
# ------------------------------------------------------------
#:0103100001000100010001000100010001000100010001E4
#:010201FFFD
#:0104100001000100010001000100010001000100010001E3
#:010101FFFE
#:010F00120008D6
#:011000120008D5
#:010600120001E6
#:01050012FF00E9
#:0117100001000100010001000100010001000100010001D0
#:010700F8
#:010B00000008EC
#:010C060000000000000ED
#:01110300FFEC
#:011400EB
#:011500EA
#:01160012FFFF0000D9
#:0118001200100001000100010001000100010001000100010001BD
#:012B0E018300000042
#:010800000000F7
#:010800010000F6
#:010800020000F5
#:010800030000F4
#:01080004F3
#:0108000A0000ED
#:0108000B0000EC
#:0108000C0000EB
#:0108000D0000EA
#:0108000E0000E9
#:0108000F0000E8
#:010800100000E7
#:010800110000E6
#:010800120000E5
#:010800130000E4
#:010800140000E3
#:010800150000E2
# ------------------------------------------------------------
# Modbus Binary Messages
# ------------------------------------------------------------
# [ Start ][Address ][ Function ][ Data ][ CRC ][ End ]
#   1b        1b         1b         Nb     2b     1b
```

```
#
# ./message-parser -b -p binary -f messages
# ------------------------------------------------------------
#7b0103100001000100010001000100010001000193b47d
#7b010201ffe1c87d
#7b0104100001000100010001000100010001000122c17d
#7b010101ff11c87d
#7b010f00120008f4087d
#7b01100012000861ca7d
#7b010600120001e80f7d
#7b01050012ff002c3f7d
#7b0117100001000100010001000100010001d6407d
#7b01070022307d
#7b010b00000008a5cd7d
#7b010c0600000000000061357d
#7b01110300ffacbc7d
#7b0114002f007d
#7b0115002e907d
#7b01160012ffff00004e217d
#7b011800120010000100010001000100010001000100010001d74d7d
#7b012b0e01830000000faf7d
#7b010800000000e00b7d
#7b010800010000b1cb7d
#7b01080002000041cb7d
#7b010800030000100b7d
#7b0108000481d97d
#7b0108000a0000c0097d
#7b0108000b000091c97d
#7b0108000c000020087d
#7b0108000d000071c87d
#7b0108000e000081c87d
#7b0108000f0000d0087d
#7b010800100000e1ce7d
#7b010800110000b00e7d
#7b010800120000400e7d
#7b01080013000011ce7d
#7b010800140000a00f7d
#7b010800150000f1cf7d
```

1.2.6 Modbus Message Parsing Example

This is an example of a parser to decode raw messages to a readable description. It will attempt to decode a message to the request and response version of a message if possible. Here is an example output:

```
$./message-parser.py -b -m 000112340006ff076d
============================================================================
Decoding Message 000112340006ff076d
============================================================================
ServerDecoder
----------------------------------------------------------------------------
name            = ReadExceptionStatusRequest
check           = 0x0
unit_id         = 0xff
transaction_id  = 0x1
protocol_id     = 0x1234
documentation   =
    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).

ClientDecoder
------------------------------------------------------------------------------
name            = ReadExceptionStatusResponse
check           = 0x0
status          = 0x6d
unit_id         = 0xff
transaction_id  = 0x1
protocol_id     = 0x1234
documentation   =
    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.
```

Program Source

```
#!/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
#---------------------------------------------------------------------------#
import sys
import collections
import textwrap
from optparse import OptionParser
from pymodbus.utilities import computeCRC, computeLRC
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

#---------------------------------------------------------------------------#
# 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
        '''
        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, 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__.iteritems():
            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)

    (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 not option.ascii:
            option.message = option.message.decode('hex')
        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, e:
                print "Logging is not supported on this system"

    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()
```

Example Messages

See the documentation for the message generator for a collection of messages that can be parsed by this utility.

1.2.7 Synchronous Serial Forwarder

```
#!/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)

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

1.2.8 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):

    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, 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, ex:
        print ex

#---------------------------------------------------------------------------#
# Main jumper
#---------------------------------------------------------------------------#
if __name__ == "__main__":
    main()
```

1.2.9 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, 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, err:
        print err
        parser.print_help()

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

1.2.10 Modbus 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 logging
import time
import itertools
from collections import namedtuple

# 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(klass, 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:
            from Queue import Queue
            from threading import Thread
            from threading import Event
            return klass(queue=Queue, event=Event, worker=Thread)
        else:
            from multiprocessing import Queue
            from multiprocessing import Event
            from multiprocessing import Process
            return klass(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, exception:
                log.exception("error in worker thread: %s", threading.current_thread())
                output_queue.put(WorkResponse(True, workitem.work_id, exception))
        except Exception, 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, 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() # condition marking process shutdown
        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()
        self.output_queue.put(WorkResponse(None, None, None)) # to wake up the manager
        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
        '''
        future, work_id = Future(), self.counter.next()
        self.input_queue.put(WorkRequest(request, work_id))
        self.futures[work_id] = future
        return future

    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()
```

1.2.11 Libmodbus Client Facade

```
#!/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, WriteMultipleRegistersRespon

#---------------------------------------------------------------------------
# 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 *
```

```
    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))

#---------------------------------------------------------------------------#
# 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,
    }

    #-----------------------------------------------------------------------#
    # 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, r
        'WriteSingleRegisterRequest'        : lambda tx, rx: WriteSingleRegisterResponse(tx.address,
        'WriteMultipleRegistersRequest'     : lambda tx, rx: WriteMultipleRegistersResponse(tx.addres
        'ReadHoldingRegistersRequest'       : lambda tx, rx: ReadHoldingRegistersResponse(list(rx)),
        'ReadInputRegistersRequest'         : lambda tx, rx: ReadInputRegistersResponse(list(rx)),
        'ReadWriteMultipleRegistersRequest' : lambda tx, rx: ReadWriteMultipleRegistersResponse(list
    }

    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: " + name)

        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
```

1.2.12 Remote Single Server Context

```
'''
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 = 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 = 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)))
        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.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 self.__slaves.iteritems()

    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]
```

1.3 Example Frontend Code

1.3.1 Glade/GTK Frontend Example

Main Program

This is an example simulator that is written using the pygtk bindings. Although it currently does not have a frontend for modifying the context values, it does allow one to expose N virtual modbus devices to a network which is useful for testing data center monitoring tools.

Note: The virtual networking code will only work on linux

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

#---------------------------------------------------------------------------#
# For Gui
```

```
#---------------------------------------------------------------------------#
from twisted.internet import gtk2reactor
gtk2reactor.install()
import gtk
from gtk import glade

#---------------------------------------------------------------------------#
# 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(__name__)

#---------------------------------------------------------------------------#
# Application Error
#---------------------------------------------------------------------------#
class ConfigurationException(Exception):
    ''' Exception for configuration error '''

    def __init__(self, string):
        Exception.__init__(self, string)
        self.string = string

    def __str__(self):
        return 'Configuration Error: %s' % self.string

#---------------------------------------------------------------------------#
# 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()))
                print 'listening on port', port
                return port
            except twisted_error.CannotListenError:
                pass

    def run(self):
        ''' Used to run the simulator '''
        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):
        Thread.__init__(self)
        self.setDaemon(True)

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

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

```
# Main Gui Class
#---------------------------------------------------------------------------#
# Note, if you are using gtk2 before 2.12, the file_set signal is not
# introduced.  To fix this, you need to apply the following patch
#---------------------------------------------------------------------------#
#Index: simulator.py
#===================================================================
#--- simulator.py       (revision 60)
#+++ simulator.py       (working copy)
#@@ -158,7 +161,7 @@
#                       "on_helpBtn_clicked"    : self.help_clicked,
#                       "on_quitBtn_clicked"    : self.close_clicked,
#                       "on_startBtn_clicked"   : self.start_clicked,
#-                      "on_file_changed"       : self.file_changed,
#+                      #"on_file_changed"      : self.file_changed,
#                       "on_window_destroy"     : self.close_clicked
#               }
#               self.tree.signal_autoconnect(actions)
#@@ -235,6 +238,7 @@
#                       return False
#
#               # check input file
#+              self.file_changed(self.tdevice)
#               if os.path.exists(self.file):
#                       self.grey_out()
#                       handle = Simulator(config=self.file)
#---------------------------------------------------------------------------#
class SimulatorApp(object):
    '''
    This class implements the GUI for the flasher application
    '''
    file = "none"
    subnet = 205
    number = 1
    restart = 0

    def __init__(self, xml):
        ''' Sets up the gui, callback, and widget handles '''

        #-----------------------------------------------------------------------#
        # Action Handles
        #-----------------------------------------------------------------------#
        self.tree    = glade.XML(xml)
        self.bstart  = self.tree.get_widget("startBtn")
        self.bhelp   = self.tree.get_widget("helpBtn")
        self.bclose  = self.tree.get_widget("quitBtn")
        self.window  = self.tree.get_widget("window")
        self.tdevice = self.tree.get_widget("fileTxt")
        self.tsubnet = self.tree.get_widget("addressTxt")
        self.tnumber = self.tree.get_widget("deviceTxt")

        #-----------------------------------------------------------------------#
        # Actions
        #-----------------------------------------------------------------------#
        actions = {
            "on_helpBtn_clicked"  : self.help_clicked,
            "on_quitBtn_clicked"  : self.close_clicked,
            "on_startBtn_clicked" : self.start_clicked,
```

```
            "on_file_changed"     : self.file_changed,
            "on_window_destroy"   : self.close_clicked
        }
        self.tree.signal_autoconnect(actions)
        if not root_test():
            self.error_dialog("This program must be run with root permissions!", True)

#---------------------------------------------------------------------------#
# Gui helpers
#---------------------------------------------------------------------------#
# Not callbacks, but used by them
#---------------------------------------------------------------------------#
    def show_buttons(self, state=False, all=0):
        ''' Greys out the buttons '''
        if all:
            self.window.set_sensitive(state)
        self.bstart.set_sensitive(state)
        self.tdevice.set_sensitive(state)
        self.tsubnet.set_sensitive(state)
        self.tnumber.set_sensitive(state)

    def destroy_interfaces(self):
        ''' This is used to reset the virtual interfaces '''
        if self.restart:
            n = NetworkReset()
            n.start()

    def error_dialog(self, message, quit=False):
        ''' Quick pop-up for error messages '''
        dialog = gtk.MessageDialog(
            parent         = self.window,
            flags          = gtk.DIALOG_DESTROY_WITH_PARENT | gtk.DIALOG_MODAL,
            type           = gtk.MESSAGE_ERROR,
            buttons        = gtk.BUTTONS_CLOSE,
            message_format = message)
        dialog.set_title('Error')
        if quit:
            dialog.connect("response", lambda w, r: gtk.main_quit())
        else:
            dialog.connect("response", lambda w, r: w.destroy())
        dialog.show()

#---------------------------------------------------------------------------#
# Button Actions
#---------------------------------------------------------------------------#
# These are all callbacks for the various buttons
#---------------------------------------------------------------------------#
    def start_clicked(self, widget):
        ''' Starts the simulator '''
        start = 1
        base = "172.16"

        # check starting network
        net = self.tsubnet.get_text()
        octets = net.split('.')
        if len(octets) == 4:
            base = "%s.%s" % (octets[0], octets[1])
            net = int(octets[2]) % 255
```

```
            start = int(octets[3]) % 255
        else:
            self.error_dialog("Invalid starting address!");
            return False

        # check interface size
        size = int(self.tnumber.get_text())
        if (size >= 1):
            for i in range(start, (size + start)):
                j = i % 255
                cmd = "/sbin/ifconfig eth0:%d %s.%d.%d" % (i, base, net, j)
                os.system(cmd)
                if j == 254: net = net + 1
            self.restart = 1
        else:
            self.error_dialog("Invalid number of devices!");
            return False

        # check input file
        if os.path.exists(self.file):
            self.show_buttons(state=False)
            try:
                handle = Simulator(config=self.file)
                handle.run()
            except ConfigurationException, ex:
                self.error_dialog("Error %s" % ex)
                self.show_buttons(state=True)
        else:
            self.error_dialog("Device to emulate does not exist!");
            return False

    def help_clicked(self, widget):
        ''' Quick pop-up for about page '''
        data = gtk.AboutDialog()
        data.set_version("0.1")
        data.set_name(('Modbus Simulator'))
        data.set_authors(["Galen Collins"])
        data.set_comments(('First Select a device to simulate,\n'
            + 'then select the starting subnet of the new devices\n'
            + 'then select the number of device to simulate and click start'))
        data.set_website("http://code.google.com/p/pymodbus/")
        data.connect("response", lambda w,r: w.hide())
        data.run()

    def close_clicked(self, widget):
        ''' Callback for close button '''
        self.destroy_interfaces()
        reactor.stop()          # quit twisted

    def file_changed(self, widget):
        ''' Callback for the filename change '''
        self.file = widget.get_filename()

#---------------------------------------------------------------------------#
# Main handle function
#---------------------------------------------------------------------------#
# This is called when the application is run from a console
# We simply start the gui and start the twisted event loop
```

```
#---------------------------------------------------------------------------#
def main():
    '''
    Main control function
    This either launches the gui or runs the command line application
    '''
    debug = True
    if debug:
        try:
            log.setLevel(logging.DEBUG)
                logging.basicConfig()
        except Exception, e:
                print "Logging is not supported on this system"
    simulator = SimulatorApp('./simulator.glade')
    reactor.run()

#---------------------------------------------------------------------------#
# Library/Console Test
#---------------------------------------------------------------------------#
# If this is called from console, we start main
#---------------------------------------------------------------------------#
if __name__ == "__main__":
    main()
```

Glade Layout File

The following is the glade layout file that is used by this script:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.0 on Thu Nov 20 10:51:52 2008 -->
<glade-interface>
  <widget class="GtkWindow" id="window">
    <property name="visible">True</property>
    <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_PRESS
    <property name="title" translatable="yes">Modbus Simulator</property>
    <property name="resizable">False</property>
    <property name="window_position">GTK_WIN_POS_CENTER</property>
    <signal name="destroy" handler="on_window_destroy"/>
    <child>
      <widget class="GtkVBox" id="vbox1">
        <property name="width_request">400</property>
        <property name="height_request">200</property>
        <property name="visible">True</property>
        <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_
        <child>
          <widget class="GtkHBox" id="hbox1">
            <property name="visible">True</property>
            <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUT
            <child>
              <widget class="GtkLabel" id="label1">
                <property name="visible">True</property>
                <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
                <property name="label" translatable="yes">Device to Simulate</property>
              </widget>
            </child>
            <child>
```

```
            <widget class="GtkHButtonBox" id="hbuttonbox2">
              <property name="visible">True</property>
              <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
              <child>
                <widget class="GtkFileChooserButton" id="fileTxt">
                  <property name="width_request">220</property>
                  <property name="visible">True</property>
                  <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK |
                  <signal name="file_set" handler="on_file_changed"/>
                </widget>
              </child>
            </widget>
            <packing>
              <property name="expand">False</property>
              <property name="fill">False</property>
              <property name="padding">20</property>
              <property name="position">1</property>
            </packing>
          </child>
        </widget>
      </child>
      <child>
        <widget class="GtkHBox" id="hbox2">
          <property name="visible">True</property>
          <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTT
          <child>
            <widget class="GtkLabel" id="label2">
              <property name="visible">True</property>
              <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
              <property name="label" translatable="yes">Starting Address</property>
            </widget>
          </child>
          <child>
            <widget class="GtkEntry" id="addressTxt">
              <property name="width_request">230</property>
              <property name="visible">True</property>
              <property name="can_focus">True</property>
              <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
            </widget>
            <packing>
              <property name="expand">False</property>
              <property name="padding">20</property>
              <property name="position">1</property>
            </packing>
          </child>
        </widget>
        <packing>
          <property name="position">1</property>
        </packing>
      </child>
      <child>
        <widget class="GtkHBox" id="hbox3">
          <property name="visible">True</property>
          <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTT
          <child>
            <widget class="GtkLabel" id="label3">
              <property name="visible">True</property>
              <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
```

```
            <property name="label" translatable="yes">Number of Devices</property>
          </widget>
        </child>
        <child>
          <widget class="GtkSpinButton" id="deviceTxt">
            <property name="width_request">230</property>
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
            <property name="adjustment">1 0 2000 1 10 0</property>
          </widget>
          <packing>
            <property name="expand">False</property>
            <property name="padding">20</property>
            <property name="position">1</property>
          </packing>
        </child>
      </widget>
      <packing>
        <property name="position">2</property>
      </packing>
    </child>
    <child>
      <widget class="GtkHButtonBox" id="hbuttonbox1">
        <property name="visible">True</property>
        <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUT
        <property name="layout_style">GTK_BUTTONBOX_SPREAD</property>
        <child>
          <widget class="GtkButton" id="helpBtn">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
            <property name="label" translatable="yes">gtk-help</property>
            <property name="use_stock">True</property>
            <property name="response_id">0</property>
            <signal name="clicked" handler="on_helpBtn_clicked"/>
          </widget>
        </child>
        <child>
          <widget class="GtkButton" id="startBtn">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
            <property name="receives_default">True</property>
            <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
            <property name="label" translatable="yes">gtk-apply</property>
            <property name="use_stock">True</property>
            <property name="response_id">0</property>
            <signal name="clicked" handler="on_startBtn_clicked"/>
          </widget>
          <packing>
            <property name="position">1</property>
          </packing>
        </child>
        <child>
          <widget class="GtkButton" id="quitBtn">
            <property name="visible">True</property>
            <property name="can_focus">True</property>
```

```
                <property name="receives_default">True</property>
                <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
                <property name="label" translatable="yes">gtk-stop</property>
                <property name="use_stock">True</property>
                <property name="response_id">0</property>
                <signal name="clicked" handler="on_quitBtn_clicked"/>
              </widget>
              <packing>
                <property name="position">2</property>
              </packing>
            </child>
          </widget>
          <packing>
            <property name="position">3</property>
          </packing>
        </child>
      </widget>
    </child>
  </widget>
</glade-interface>
```

1.3.2 TK Frontend Example

Main Program

This is an example simulator that is written using the native tk toolkit. Although it currently does not have a frontend for modifying the context values, it does allow one to expose N virtual modbus devices to a network which is useful for testing data center monitoring tools.

Note: The virtual networking code will only work on linux

```
#!/usr/bin/env python
'''
Note that this is not finished
'''
#---------------------------------------------------------------------------#
# System
#---------------------------------------------------------------------------#
import os
import getpass
import pickle
from threading import Thread

#---------------------------------------------------------------------------#
# For Gui
#---------------------------------------------------------------------------#
from Tkinter import *
from tkFileDialog import askopenfilename as OpenFilename
from twisted.internet import tksupport
root = Tk()
tksupport.install(root)

#---------------------------------------------------------------------------#
# 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(__name__)

#---------------------------------------------------------------------------#
# 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.info('listening on port %d' % port)
                return port
            except twisted_error.CannotListenError:
                pass

    def run(self):
        ''' Used to run the simulator '''
        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):
        Thread.__init__(self)
        self.setDaemon(True)

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

#---------------------------------------------------------------------------#
# Main Gui Class
#---------------------------------------------------------------------------#
class SimulatorFrame(Frame):
    '''
    This class implements the GUI for the flasher application
    '''
    subnet  = 205
    number  = 1
    restart = 0

    def __init__(self, master, font):
        ''' Sets up the gui, callback, and widget handles '''
        Frame.__init__(self, master)
        self._widgets = []
```

```
        #---------------------------------------------------------------------------#
        # Initialize Buttons Handles
        #---------------------------------------------------------------------------#
        frame = Frame(self)
        frame.pack(side=BOTTOM, pady=5)

        button = Button(frame, text="Apply", command=self.start_clicked, font=font)
        button.pack(side=LEFT, padx=15)
        self._widgets.append(button)

        button = Button(frame, text="Help",  command=self.help_clicked, font=font)
        button.pack(side=LEFT, padx=15)
        self._widgets.append(button)

        button = Button(frame, text="Close", command=self.close_clicked, font=font)
        button.pack(side=LEFT, padx=15)
        #self._widgets.append(button) # we don't want to grey this out

        #---------------------------------------------------------------------------#
        # Initialize Input Fields
        #---------------------------------------------------------------------------#
        frame = Frame(self)
        frame.pack(side=TOP, padx=10, pady=5)

        self.tsubnet_value = StringVar()
        label = Label(frame, text="Starting Address", font=font)
        label.grid(row=0, column=0, pady=10)
        entry = Entry(frame, textvariable=self.tsubnet_value, font=font)
        entry.grid(row=0, column=1, pady=10)
        self._widgets.append(entry)

        self.tdevice_value = StringVar()
        label = Label(frame, text="Device to Simulate", font=font)
        label.grid(row=1, column=0, pady=10)
        entry = Entry(frame, textvariable=self.tdevice_value, font=font)
        entry.grid(row=1, column=1, pady=10)
        self._widgets.append(entry)

        image = PhotoImage(file='fileopen.gif')
        button = Button(frame, image=image, command=self.file_clicked)
        button.image = image
        button.grid(row=1, column=2, pady=10)
        self._widgets.append(button)

        self.tnumber_value = StringVar()
        label = Label(frame, text="Number of Devices", font=font)
        label.grid(row=2, column=0, pady=10)
        entry = Entry(frame, textvariable=self.tnumber_value, font=font)
        entry.grid(row=2, column=1, pady=10)
        self._widgets.append(entry)

        #if not root_test():
        #    self.error_dialog("This program must be run with root permissions!", True)

#---------------------------------------------------------------------------#
# Gui helpers
#---------------------------------------------------------------------------#
# Not callbacks, but used by them
```

```
#---------------------------------------------------------------------------#
    def show_buttons(self, state=False):
        ''' Greys out the buttons '''
        state = 'active' if state else 'disabled'
        for widget in self._widgets:
            widget.configure(state=state)

    def destroy_interfaces(self):
        ''' This is used to reset the virtual interfaces '''
        if self.restart:
            n = NetworkReset()
            n.start()

    def error_dialog(self, message, quit=False):
        ''' Quick pop-up for error messages '''
        dialog = gtk.MessageDialog(
            parent          = self.window,
            flags           = gtk.DIALOG_DESTROY_WITH_PARENT | gtk.DIALOG_MODAL,
            type            = gtk.MESSAGE_ERROR,
            buttons         = gtk.BUTTONS_CLOSE,
            message_format  = message)
        dialog.set_title('Error')
        if quit:
            dialog.connect("response", lambda w, r: gtk.main_quit())
        else: dialog.connect("response", lambda w, r: w.destroy())
        dialog.show()

#---------------------------------------------------------------------------#
# Button Actions
#---------------------------------------------------------------------------#
# These are all callbacks for the various buttons
#---------------------------------------------------------------------------#
    def start_clicked(self):
        ''' Starts the simulator '''
        start = 1
        base = "172.16"

        # check starting network
        net = self.tsubnet_value.get()
        octets = net.split('.')
        if len(octets) == 4:
            base = "%s.%s" % (octets[0], octets[1])
            net = int(octets[2]) % 255
            start = int(octets[3]) % 255
        else:
            self.error_dialog("Invalid starting address!");
            return False

        # check interface size
        size = int(self.tnumber_value.get())
        if (size >= 1):
            for i in range(start, (size + start)):
                j = i % 255
                cmd = "/sbin/ifconfig eth0:%d %s.%d.%d" % (i, base, net, j)
                os.system(cmd)
                if j == 254: net = net + 1
            self.restart = 1
        else:
```

```
            self.error_dialog("Invalid number of devices!");
            return False

        # check input file
        filename = self.tdevice_value.get()
        if os.path.exists(filename):
            self.show_buttons(state=False)
            try:
                handle = Simulator(config=filename)
                handle.run()
            except ConfigurationException, ex:
                self.error_dialog("Error %s" % ex)
                self.show_buttons(state=True)
        else:
            self.error_dialog("Device to emulate does not exist!");
            return False

    def help_clicked(self):
        ''' Quick pop-up for about page '''
        data = gtk.AboutDialog()
        data.set_version("0.1")
        data.set_name(('Modbus Simulator'))
        data.set_authors(["Galen Collins"])
        data.set_comments(('First Select a device to simulate,\n'
            + 'then select the starting subnet of the new devices\n'
            + 'then select the number of device to simulate and click start'))
        data.set_website("http://code.google.com/p/pymodbus/")
        data.connect("response", lambda w,r: w.hide())
        data.run()

    def close_clicked(self):
        ''' Callback for close button '''
        #self.destroy_interfaces()
        reactor.stop()

    def file_clicked(self):
        ''' Callback for the filename change '''
        file = OpenFilename()
        self.tdevice_value.set(file)

class SimulatorApp(object):
    ''' The main wx application handle for our simulator
    '''

    def __init__(self, master):
        '''
        Called by wxWindows to initialize our application

        :param master: The master window to connect to
        '''
        font  = ('Helvetica', 12, 'normal')
        frame = SimulatorFrame(master, font)
        frame.pack()

#---------------------------------------------------------------------------#
# Main handle function
#---------------------------------------------------------------------------#
# This is called when the application is run from a console
```

```
# We simply start the gui and start the twisted event loop
#---------------------------------------------------------------------------#
def main():
    '''
    Main control function
    This either launches the gui or runs the command line application
    '''
    debug = True
    if debug:
        try:
            log.setLevel(logging.DEBUG)
                logging.basicConfig()
        except Exception, e:
                print "Logging is not supported on this system"
    simulator = SimulatorApp(root)
    root.title("Modbus Simulator")
    reactor.run()

#---------------------------------------------------------------------------#
# Library/Console Test
#---------------------------------------------------------------------------#
# If this is called from console, we start main
#---------------------------------------------------------------------------#
if __name__ == "__main__":
    main()
```

1.3.3 WX Frontend Example

Main Program

This is an example simulator that is written using the python wx bindings. Although it currently does not have a frontend for modifying the context values, it does allow one to expose N virtual modbus devices to a network which is useful for testing data center monitoring tools.

Note: The virtual networking code will only work on linux

```
#!/usr/bin/env python
'''
Note that this is not finished
'''
#---------------------------------------------------------------------------#
# System
#---------------------------------------------------------------------------#
import os
import getpass
import pickle
from threading import Thread

#---------------------------------------------------------------------------#
# For Gui
#---------------------------------------------------------------------------#
import wx
from twisted.internet import wxreactor
wxreactor.install()
```

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

#---------------------------------------------------------------------------#
# 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()))
                print 'listening on port', port
                return port
            except twisted_error.CannotListenError:
                pass

    def run(self):
        ''' Used to run the simulator '''
        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):
        ''' Initializes a new instance of the network reset thread '''
        Thread.__init__(self)
        self.setDaemon(True)

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

#---------------------------------------------------------------------------#
# Main Gui Class
#---------------------------------------------------------------------------#
class SimulatorFrame(wx.Frame):
    '''
    This class implements the GUI for the flasher application
    '''
    subnet = 205
    number = 1
    restart = 0

    def __init__(self, parent, id, title):
```

```
        '''
        Sets up the gui, callback, and widget handles
        '''
        wx.Frame.__init__(self, parent, id, title)
        wx.EVT_CLOSE(self, self.close_clicked)

        #---------------------------------------------------------------------------#
        # Add button row
        #---------------------------------------------------------------------------#
        panel = wx.Panel(self, -1)
        box = wx.BoxSizer(wx.HORIZONTAL)
        box.Add(wx.Button(panel, 1, 'Apply'), 1)
        box.Add(wx.Button(panel, 2, 'Help'),  1)
        box.Add(wx.Button(panel, 3, 'Close'), 1)
        panel.SetSizer(box)

        #---------------------------------------------------------------------------#
        # Add input boxes
        #---------------------------------------------------------------------------#
        #self.tdevice    = self.tree.get_widget("fileTxt")
        #self.tsubnet    = self.tree.get_widget("addressTxt")
        #self.tnumber    = self.tree.get_widget("deviceTxt")

        #---------------------------------------------------------------------------#
        # Tie callbacks
        #---------------------------------------------------------------------------#
        self.Bind(wx.EVT_BUTTON, self.start_clicked, id=1)
        self.Bind(wx.EVT_BUTTON, self.help_clicked,  id=2)
        self.Bind(wx.EVT_BUTTON, self.close_clicked, id=3)

        #if not root_test():
        #    self.error_dialog("This program must be run with root permissions!", True)

#---------------------------------------------------------------------------#
# Gui helpers
#---------------------------------------------------------------------------#
# Not callbacks, but used by them
#---------------------------------------------------------------------------#
    def show_buttons(self, state=False, all=0):
        ''' Greys out the buttons '''
        if all:
            self.window.set_sensitive(state)
        self.bstart.set_sensitive(state)
        self.tdevice.set_sensitive(state)
        self.tsubnet.set_sensitive(state)
        self.tnumber.set_sensitive(state)

    def destroy_interfaces(self):
        ''' This is used to reset the virtual interfaces '''
        if self.restart:
            n = NetworkReset()
            n.start()

    def error_dialog(self, message, quit=False):
        ''' Quick pop-up for error messages '''
        log.debug("error event called")
        dialog = wx.MessageDialog(self, message, 'Error',
            wx.OK | wx.ICON_ERROR)
```

```
        dialog.ShowModel()
        if quit: self.Destroy()
        dialog.Destroy()

#---------------------------------------------------------------------------#
# Button Actions
#---------------------------------------------------------------------------#
# These are all callbacks for the various buttons
#---------------------------------------------------------------------------#
    def start_clicked(self, widget):
        ''' Starts the simulator '''
        start = 1
        base = "172.16"

        # check starting network
        net = self.tsubnet.get_text()
        octets = net.split('.')
        if len(octets) == 4:
            base = "%s.%s" % (octets[0], octets[1])
            net = int(octets[2]) % 255
            start = int(octets[3]) % 255
        else:
            self.error_dialog("Invalid starting address!");
            return False

        # check interface size
        size = int(self.tnumber.get_text())
        if (size >= 1):
            for i in range(start, (size + start)):
                j = i % 255
                cmd = "/sbin/ifconfig eth0:%d %s.%d.%d" % (i, base, net, j)
                os.system(cmd)
                if j == 254: net = net + 1
            self.restart = 1
        else:
            self.error_dialog("Invalid number of devices!");
            return False

        # check input file
        if os.path.exists(self.file):
            self.show_buttons(state=False)
            try:
                handle = Simulator(config=self.file)
                handle.run()
            except ConfigurationException, ex:
                self.error_dialog("Error %s" % ex)
                self.show_buttons(state=True)
        else:
            self.error_dialog("Device to emulate does not exist!");
            return False

    def help_clicked(self, widget):
        ''' Quick pop-up for about page '''
        data = gtk.AboutDialog()
        data.set_version("0.1")
        data.set_name(('Modbus Simulator'))
        data.set_authors(["Galen Collins"])
        data.set_comments(('First Select a device to simulate,\n'
```

```
            + 'then select the starting subnet of the new devices\n'
            + 'then select the number of device to simulate and click start'))
        data.set_website("http://code.google.com/p/pymodbus/")
        data.connect("response", lambda w,r: w.hide())
        data.run()

    def close_clicked(self, event):
        ''' Callback for close button '''
        log.debug("close event called")
        reactor.stop()

    def file_changed(self, event):
        ''' Callback for the filename change '''
        self.file = widget.get_filename()

class SimulatorApp(wx.App):
    ''' The main wx application handle for our simulator
    '''

    def OnInit(self):
        ''' Called by wxWindows to initialize our application

        :returns: Always True
        '''
        log.debug("application initialize event called")
        reactor.registerWxApp(self)
        frame = SimulatorFrame(None, -1, "Pymodbus Simulator")
        frame.CenterOnScreen()
        frame.Show(True)
        self.SetTopWindow(frame)
        return True

#---------------------------------------------------------------------------#
# Main handle function
#---------------------------------------------------------------------------#
# This is called when the application is run from a console
# We simply start the gui and start the twisted event loop
#---------------------------------------------------------------------------#
def main():
    '''
    Main control function
    This either launches the gui or runs the command line application
    '''
    debug = True
    if debug:
        try:
            log.setLevel(logging.DEBUG)
                logging.basicConfig()
        except Exception, e:
                print "Logging is not supported on this system"
    simulator = SimulatorApp(0)
    reactor.run()

#---------------------------------------------------------------------------#
# Library/Console Test
#---------------------------------------------------------------------------#
# If this is called from console, we start main
#---------------------------------------------------------------------------#
```

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

1.3.4 Bottle Web Frontend Example

Summary

This is a simple example of adding a live REST api on top of a running pymodbus server. This uses the bottle microframework to achieve this.

The example can be hosted under twisted as well as the bottle internal server and can furthermore be run behind gunicorn, cherrypi, etc wsgi containers.

Main Program

CHAPTER 2

Pymodbus Library API Documentation

The following are the API documentation strings taken from the sourcecode

2.1 bit_read_message — Bit Read Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.1.1 API Documentation

2.2 bit_write_message — Bit Write Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.2.1 API Documentation

2.3 client.common — Twisted Async Modbus Client

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.3.1 API Documentation

2.4 client.sync — Twisted Synchronous Modbus Client

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.4.1 API Documentation

2.5 `client.async` — Twisted Async Modbus Client

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.5.1 API Documentation

2.6 `constants` — Modbus Default Values

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.6.1 API Documentation

2.7 Server Datastores and Contexts

The following are the API documentation strings taken from the sourcecode

2.7.1 `store` — Datastore for Modbus Server Context

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

2.7.2 `context` — Modbus Server Contexts

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

2.7.3 `remote` — Remote Slave Context

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

2.8 diag_message — Diagnostic Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.8.1 API Documentation

2.9 device — Modbus Device Representation

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.9.1 API Documentation

2.10 factory — Request/Response Decoders

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.10.1 API Documentation

2.11 interfaces — System Interfaces

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.11.1 API Documentation

2.12 exceptions — Exceptions Used in PyModbus

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.12.1 API Documentation

2.13 other_message — Other Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.13.1 API Documentation

2.14 mei_message — MEI Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.14.1 API Documentation

2.15 file_message — File Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.15.1 API Documentation

2.16 events — Events Used in PyModbus

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.16.1 API Documentation

2.17 payload — Modbus Payload Utilities

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.17.1 API Documentation

2.18 pdu — Base Structures

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.18.1 API Documentation

2.19 pymodbus — Pymodbus Library

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.20 register_read_message — Register Read Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.20.1 API Documentation

2.21 register_write_message — Register Write Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.21.1 API Documentation

2.22 server.sync — Twisted Synchronous Modbus Server

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.22.1 API Documentation

2.23 server.async — Twisted Asynchronous Modbus Server

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.23.1 API Documentation

2.24 transaction — Transaction Controllers for Pymodbus

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.24.1 API Documentation

2.25 utilities — Extra Modbus Helpers

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

2.25.1 API Documentation

CHAPTER 3

Indices and tables

- genindex
- modindex
- search