Pymodbus Documentation

Release 1.0

Galen Collins

May 19, 2017

Contents

Contents:

CHAPTER 1

Pymodbus Library Examples

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

Example Library Code

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()
```

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 from 0x00 to 0xFF, a
# request to 0x100 will respond with an invalid address exception. This is
# because many devices exhibit this kind of behavior (but not all)::
#
#     block = ModbusSequentialDataBlock(0x00, [0]*0xff)
#
# Continuing, you can choose to use a sequential or a sparse DataBlock in
# your data context.  The difference is that the sequential has no gaps in
# the data while the sparse can. Once again, there are devices that exhibit
# both forms of behavior::
#
#     block = ModbusSparseDataBlock({0x00: 0, 0x05: 1})
#     block = ModbusSequentialDataBlock(0x00, [0]*5)
#
# Alternately, you can use the factory methods to initialize the DataBlocks
# or simply do not pass them to have them initialized to 0x00 on the full
# address range::
#
#     store = ModbusSlaveContext(di = ModbusSequentialDataBlock.create())
#     store = ModbusSlaveContext()
#
# Finally, you are allowed to use the same DataBlock reference for every
# table or you you may use a seperate DataBlock for each table. This depends
# if you would like functions to be able to access and modify the same data
# or not::
#
#     block = ModbusSequentialDataBlock(0x00, [0]*0xff)
#     store = ModbusSlaveContext(di=block, co=block, hr=block, ir=block)
#
```

```
# The server then makes use of a server context that allows the server to
# respond with different slave contexts for different unit ids. By default
# it will return the same context for every unit id supplied (broadcast
# mode). However, this can be overloaded by setting the single flag to False
# and then supplying a dictionary of unit id to context mapping::
#
#     slaves  = {
#         0x01: ModbusSlaveContext(...),
#         0x02: ModbusSlaveContext(...),
#         0x03: ModbusSlaveContext(...),
#     }
#     context = ModbusServerContext(slaves=slaves, single=False)
#
# The slave context can also be initialized in zero_mode which means that a
# request to address(0-7) will map to the address (0-7). The default is
# False which is based on section 4.4 of the specification, so address(0-7)
# will map to (1-8)::
#
#     store = ModbusSlaveContext(..., zero_mode=True)
#---------------------------------------------------------------------------#
store = ModbusSlaveContext(
    di = ModbusSequentialDataBlock(0, [17]*100),
    co = ModbusSequentialDataBlock(0, [17]*100),
    hr = ModbusSequentialDataBlock(0, [17]*100),
    ir = ModbusSequentialDataBlock(0, [17]*100))
context = ModbusServerContext(slaves=store, single=True)

#---------------------------------------------------------------------------#
# initialize the server information
#---------------------------------------------------------------------------#
# If you don't set this or any fields, they are defaulted to empty strings.
#---------------------------------------------------------------------------#
identity = ModbusDeviceIdentification()
identity.VendorName  = 'Pymodbus'
identity.ProductCode = 'PM'
identity.VendorUrl   = 'http://github.com/bashwork/pymodbus/'
identity.ProductName = 'Pymodbus Server'
identity.ModelName   = 'Pymodbus Server'
identity.MajorMinorRevision = '1.0'

#---------------------------------------------------------------------------#
# run the server you want
#---------------------------------------------------------------------------#
StartTcpServer(context, identity=identity, address=("localhost", 5020))
#StartUdpServer(context, identity=identity, address=("localhost", 502))
#StartSerialServer(context, identity=identity, port='/dev/pts/3',␣
↪framer=ModbusRtuFramer)
#StartSerialServer(context, identity=identity, port='/dev/pts/3',␣
↪framer=ModbusAsciiFramer)
```

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()
```

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
```

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]
```

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', port=5020)
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, unit=1)

#---------------------------------------------------------------------------#
# If you need to decode a collection of registers in a weird layout, the
# payload decoder can help you as well.
#
# Here we demonstrate decoding a random register layout, unpacked it looks
# like the following:
#
# - a 8 byte string 'abcdefgh'
# - a 32 bit float 22.34
# - a 16 bit unsigned int 0x1234
# - 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,  unit=1)
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()
```

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))
```

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=5020)
#client = ModbusClient(method='ascii', port='/dev/pts/2', timeout=1)
# client = ModbusClient(method='rtu', port='/dev/ttyp0', timeout=1)
client.connect()

#---------------------------------------------------------------------------#
# specify slave to query
#---------------------------------------------------------------------------#
# The slave to query is specified in an optional parameter for each
# individual request. This can be done by specifying the `unit` parameter
# which defaults to `0x00`
#---------------------------------------------------------------------------#
rr = client.read_coils(1, 1, unit=0x01)

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


rr = client.read_discrete_inputs(0, 8, unit=1)
assert(rq.function_code < 0x80)     # test that we are not an error

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

rr = client.read_input_registers(1, 8, unit=1)
assert(rq.function_code < 0x80)     # test that we are not an error

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

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(method='rtu', port="/dev/ttyp0")
# client = ModbusClient('127.0.0.1', port=5020)
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(unit=1)
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] == 'Pymodbus')  # test the vendor name
assert(rr.information[1] == 'PM')           # test the product code
assert(rr.information[2] == '1.0')     # test the code revision

rq = ReportSlaveIdRequest(unit=1)
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(unit=1)
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(unit=1)
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(unit=1)
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(unit=1)
rr = client.execute(rq)
# assert(rr == None)                              # not supported by reference
#assert(rr.message[0] == 0x0000)                # test the resulting message

rq = RestartCommunicationsOptionRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                              # not supported by reference
#assert(rr.message == 0x0000)                   # test the resulting message

rq = ReturnDiagnosticRegisterRequest(unit=1)
```

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

rq = ChangeAsciiInputDelimiterRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ForceListenOnlyModeRequest(unit=1)
client.execute(rq)                              # does not send a response

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

rq = ReturnBusCommunicationErrorCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnBusExceptionErrorCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnSlaveMessageCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnSlaveNoResponseCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnSlaveNAKCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnSlaveBusyCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnSlaveBusCharacterOverrunCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ReturnIopOverrunCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = ClearOverrunCountRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

rq = GetClearModbusPlusRequest(unit=1)
rr = client.execute(rq)
#assert(rr == None)                             # not supported by reference

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

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/riptideio/pymodbus/'
identity.ProductName = 'Pymodbus Server'
identity.ModelName   = 'Pymodbus Server'
identity.MajorMinorRevision = '1.0'

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

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

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

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

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
from pymodbus.client.sync import ModbusSerialClient

#---------------------------------------------------------------------------#
# choose between threads or processes
#---------------------------------------------------------------------------#
#from multiprocessing import Process as Worker
from threading import Thread as Worker
from threading import Lock
_thread_lock = Lock()
#---------------------------------------------------------------------------#
# initialize the test
#---------------------------------------------------------------------------#
# Modify the parameters below to control how we are testing the client:
#
# * workers - the number of workers to use at once
# * cycles  - the total number of requests to send
# * host    - the host to send the requests to
#---------------------------------------------------------------------------#
workers = 10
cycles  = 1000
host    = '127.0.0.1'


#---------------------------------------------------------------------------#
# perform the test
#---------------------------------------------------------------------------#
# This test is written such that it can be used by many threads of processes
# although it should be noted that there are performance penalties
# associated with each strategy.
#---------------------------------------------------------------------------#
```

```
def single_client_test(host, cycles):
    ''' Performs a single threaded test of a synchronous
    client against the specified host

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

    try:
        count  = 0
        # client = ModbusTcpClient(host, port=5020)
        client = ModbusSerialClient(method="rtu", port="/dev/ttyp0", baudrate=9600)
        while count < cycles:
            with _thread_lock:
                result = client.read_holding_registers(10, 1, unit=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))
print "time taken to complete %s cycle by %s workers is %s seconds" % (cycles,␣
↪workers, stop-start)
```

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))
```

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))
```

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()
```

Thread Safe Datastore Example

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


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

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

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

```
        self._factory = factory

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

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


class ReadWriteLock(object):
    ''' This reader writer lock gurantees write order, but not
    read order and is generally biased towards allowing writes
    if they are available to prevent starvation.

    TODO:

    * allow user to choose between read/write/random biasing
    - currently write biased
    - read biased allow N readers in queue
    - random is 50/50 choice of next
    '''

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

    def __is_pending_writer(self):
        return (self.writer                                 # if there is a current␣
↪writer
            or (self.queue                                  # or if there is a waiting␣
↪writer
           and (self.queue[0] != self.read_condition)))     # or if the queue head is␣
↪not a reader

    def acquire_reader(self):
        ''' Notifies the lock that a new reader is requesting
        the underlying resource.
        '''
        with self.lock:
            if self.__is_pending_writer():                  # if there are existing␣
↪writers waiting
                if self.read_condition not in self.queue:  # do not pollute the queue␣
↪with readers
                    self.queue.append(self.read_condition) # add the readers in line␣
↪for the queue
                while self.__is_pending_writer():           # until the current writer␣
↪is finished
                    self.read_condition.wait(1)             # wait on our condition
                if self.queue and self.read_condition == self.queue[0]: # if the read␣
↪condition is at the queue head
```

```
                    self.queue.pop(0)                           # then go ahead and remove␣
↪it
            self.readers += 1                                   # update the current␣
↪number of readers

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

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

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

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

            with rwlock.get_reader_lock():
                do_read_operation()
        '''
        try:
```

```
            self.acquire_reader()
            yield self
        finally: self.release_reader()

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

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


class ThreadSafeDataBlock(BaseModbusDataBlock):
    ''' This is a simple decorator for a data block. This allows
    a user to inject an existing data block which can then be
    safely operated on from multiple cocurrent threads.

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

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

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

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

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

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

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

    def setValues(self, address, values):
```

```
        ''' Sets the requested values of the datastore

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


if __name__ == "__main__":

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

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

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

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

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

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

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

Custom Pymodbus Code

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)
```

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
```

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"]
```

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"]
```

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
#0001000000080110f0012000801ff
#0001000000170110001200081000010001000100010001000100010001
#00010000000601060012000 1
#00010000000601050012ff00
#00010000001b0117001200080000000810000100010001000100010001000100010001
#0001000000020107
#0001000000020 0b
#0001000000020 0c
#0001000000020111
#0001000000030 11400
#0001000000030 11500
#0001000000080 1160012ffff0000
#0001000000040 1180012
#0001000000050 12b0e0100
#0001000000060 1080000000
#0001000000060 1080010000
#0001000000060 1080020000
#0001000000060 1080030000
#0001000000060 1080040000
#0001000000060 108000a0000
#0001000000060 108000b0000
#0001000000060 108000c0000
#0001000000060 108000d0000
#0001000000060 108000e0000
#0001000000060 108000f0000
#0001000000060 1080010000
#0001000000060 1080011000 0
#0001000000060 1080012000 0
#0001000000060 1080013000 0
#0001000000060 1080014000 0
#0001000000060 1080015000 0
# -----------------------------------------------------------
# Modbus RTU Messages
# -----------------------------------------------------------
# [Address ][ Function Code] [ Data ][ CRC ]
#  1b         1b               Nb      2b
#
# ./message-parser -b -p rtu -f messages
# -----------------------------------------------------------
#0103001200 8e409
#0102001200 8d9c9
#0104001200 851c9
```

```
#0101001200089dc9
#010f0012000801ff06d6
#01100012000810000100010001000100010001000100010001d551
#010600120001e80f
#01050012ff002c3f
#01170012000800000000810000100010001000100010001000100010001e6f8
#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
#:01100012000810000100010001000100010001000100010001BD
#:010600120001E6
#:01050012FF00E9
#:01170012000800000000810000100010001000100010001000100010001AE
#: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
#7b01170012000800000008100001000100010001000100010001000100010001e6f87d
#7b010741e27d
#7b010b41e77d
#7b010c00257d
#7b0111c02c7d
#7b0114002f007d
#7b0115002e907d
#7b01160012ffff00004e217d
#7b0118001201d27d
#7b012b0e010070777d
#7b0108000000000e00b7d
#7b0108000100000b1cb7d
#7b01080002000041cb7d
#7b0108000300000100b7d
#7b0108000400000a1ca7d
#7b0108000a0000c0097d
#7b0108000b000091c97d
#7b0108000c000020087d
#7b0108000d000071c87d
#7b0108000e000081c87d
#7b0108000f0000d0087d
#7b0108001000000e1ce7d
#7b0108001100000b00e7d
#7b0108001200000400e7d
#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
# --------------------------------------------------------------
```

```
#000100000013010310000100010001000100010001000100010001
#00010000000401020lff
#000100000013010410000100010001000100010001000100010001
#00010000000401010lff
#000100000000601 0f00120008
#000100000000601 1000120008
#000100000000601 0600120001
#000100000000601 050012ff00
#000100000013011710000100010001000100010001000100010001
#000100000003010700
#000100000006010b00000008
#000100000009010c06000000000000
#000100000501110300ff
#000100000003011400
#000100000003011500
#000100000801160012ffff0000
#000100000160118001200100001000100010001000100010001000100010001
#000100000008012b0e0183000000
#000100000006010800000000
#000100000006010800010000
#000100000006010800020000
#000100000006010800030000
#000100000401080004
#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
#0117100001000100010001000100010001d640
#0107002230
#010b00000008a5cd
#010c060000000000006135
#01110300ffacbc
#0114002f00
#0115002e90
```

```
#01160012ffff00004e21
#0118001200100001000100010001000100010001000100010001d74d
#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
#:010C06000000000000ED
#: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
#7b0117100001000100010001000100010001000100010001d6407d
#7b01070022307d
#7b010b00000008a5cd7d
#7b010c0600000000000061357d
#7b01110300ffacbc7d
#7b0114002f007d
#7b0115002e907d
#7b01160012ffff00004e217d
#7b0118001200100001000100010001000100010001000100010001d74d7d
#7b012b0e01830000000faf7d
#7b0108000000000e00b7d
#7b0108000100000b1cb7d
#7b0108000200000041cb7d
#7b0108000300000100b7d
#7b0108000481d97d
#7b0108000a0000c0097d
#7b0108000b000091c97d
#7b0108000c000020087d
#7b0108000d000071c87d
#7b0108000e000081c87d
#7b0108000f0000d0087d
#7b0108001000000e1ce7d
#7b0108001100000b00e7d
#7b0108001200000400e7d
#7b0108001300000011ce7d
#7b0108001400000a00f7d
#7b0108001500000f1cf7d
```

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.

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)
```

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()
```

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!"
```

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()
```

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,␣
↪WriteMultipleRegistersResponse

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

compiler = FFI()
compiler.cdef("""
```

```
    typedef struct _modbus modbus_t;

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

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

    void modbus_get_response_timeout(modbus_t *ctx, uint32_t *to_sec, uint32_t *to_
↪usec);
    void modbus_set_response_timeout(modbus_t *ctx, uint32_t to_sec, uint32_t to_
↪usec);

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

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

    int modbus_mask_write_register(modbus_t *ctx, int addr, uint16_t and_mask, uint16_
↪t or_mask);
    int modbus_send_raw_request(modbus_t *ctx, uint8_t *raw_req, int raw_req_length);

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

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

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

#---------------------------------------------------------------------------
# helper utilites
#---------------------------------------------------------------------------

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

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

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

def cast_to_int32(data):
```

```
    return int(compiler.cast('int32_t', data))

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

class LibmodbusLevel1Client(object):
    ''' A raw wrapper around the libmodbus c library. Feel free
    to use it if you want increased performance and don't mind the
    entire protocol not being implemented.
    '''

    @classmethod
    def create_tcp_client(klass, host='127.0.0.1', port=Defaults.Port):
        ''' Create a TCP modbus client for the supplied parameters.

        :param host: The host to connect to
        :param port: The port to connect to on that host
        :returns: A new level1 client
        '''
        client = LIB.modbus_new_tcp(host.encode(), port)
        return klass(client)

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

        :param port: The serial port to attach to
        :param stopbits: The number of stop bits to use
        :param bytesize: The bytesize of the serial messages
        :param parity: Which kind of parity to use
        :param baudrate: The baud rate to use for the serial device
        :returns: A new level1 client
        '''
        port     = kwargs.get('port', '/dev/ttyS0')
        baudrate = kwargs.get('baud', Defaults.Baudrate)
        parity   = kwargs.get('parity', Defaults.Parity)
        bytesize = kwargs.get('bytesize', Defaults.Bytesize)
        stopbits = kwargs.get('stopbits', Defaults.Stopbits)
        client = LIB.modbus_new_rtu(port, baudrate, parity, bytesize, stopbits)
        return klass(client)

    def __init__(self, client):
        ''' Initalize a new instance of the LibmodbusLevel1Client. This
        method should not be used, instead new instances should be created
        using the two supplied factory methods:

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

        :param client: The underlying client instance to operate with.
        '''
        self.client = client
        self.slave  = Defaults.UnitId

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

```
        :param slave: The new slave to operate against
        :returns: The resulting slave to operate against
        '''
        self.slave = self._execute(LIB.modbus_set_slave, slave)
        return self.slave

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

        :returns: True if successful, throws otherwise
        '''
        return (self.__execute(LIB.modbus_connect) == 0)

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

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

    def close(self):
        ''' Closes and frees the underlying connection
        and context structure.

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

    def __execute(self, command, *args):
        ''' Run the supplied command against the currently
        instantiated client with the supplied arguments. This
        will make sure to correctly handle resulting errors.

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

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

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

    def read_input_bits(self, address, count=1):
        '''
```

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

    def write_bit(self, address, value):
        '''

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

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

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

    def write_register(self, address, value):
        '''

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

    def write_registers(self, address, values):
        '''

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

    def read_registers(self, address, count=1):
        '''

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

```
    def read_input_registers(self, address, count=1):
        '''

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

    def read_and_write_registers(self, read_address, read_count, write_address, write_
↪registers):
        '''

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

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

class LibmodbusClient(ModbusClientMixin):
    ''' A facade around the raw level 1 libmodbus client
    that implements the pymodbus protocol on top of the lower level
    client.
    '''

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

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

```
        'ReadInputRegistersRequest'         : lambda c, r: c.read_input_registers(r.
↪address, r.count),
        'ReadWriteMultipleRegistersRequest' : lambda c, r: c.read_and_write_
↪registers(r.read_address, r.read_count, r.write_address, r.write_registers),
    }

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

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

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

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

        :param client: The underlying client instance to operate with.
        '''
        self.client = client

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

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

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

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

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

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]
```

Example Frontend Code

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_MASK | GDK_BUTTON_RELEASE_MASK</property>
    <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_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
        <child>
          <widget class="GtkHBox" id="hbox1">
            <property name="visible">True</property>
            <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_
↪MASK | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
            <child>
              <widget class="GtkLabel" id="label1">
                <property name="visible">True</property>
                <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_
↪HINT_MASK | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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 | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                  <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
          <child>
            <widget class="GtkLabel" id="label2">
              <property name="visible">True</property>
              <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_
↪HINT_MASK | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
              <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</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">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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
          <child>
            <widget class="GtkLabel" id="label3">
              <property name="visible">True</property>
```

```
                <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_
↪HINT_MASK | GDK_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
            <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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_BUTTON_PRESS_MASK | GDK_BUTTON_RELEASE_MASK</property>
                <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>
```

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()
```

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()
```

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

bit_read_message — Bit Read Modbus Messages

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

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

API Documentation

Bit Reading Request/Response messages

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

This function code is used to read from 1 to 2000(0x7d0) contiguous status of coils in a remote device. The Request PDU specifies the starting address, ie the address of the first coil specified, and the number of coils. In the PDU Coils are addressed starting at zero. Therefore coils numbered 1-16 are addressed as 0-15.

Initializes a new instance

Parameters

- **address** – The address to start reading from
- **count** – The number of bits to read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a request pdu

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes a request pdu

Returns The encoded pdu

execute(*context*)
Run a read coils request against a datastore

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

Parameters **context** – The datastore to request from

Returns The initializes response message, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + Quantity of Coils (n Bytes) :return:

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

The coils in the response message are packed as one coil per bit of the data field. Status is indicated as 1= ON and 0= OFF. The LSB of the first data byte contains the output addressed in the query. The other coils follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned output quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

Intializes a new instance

Parameters **values** – The request values to respond with

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes response pdu

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

encode()
Encodes response pdu

Returns The encoded packet message

getBit(*address*)
Helper function to get the specified bit's value

Parameters **address** – The bit to query

Returns The value of the requested bit

resetBit(*address*)
Helper function to set the specified bit to 0

Parameters **address** – The bit to reset

setBit(*address*, *value=1*)
Helper function to set the specified bit

Parameters

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

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

This function code is used to read from 1 to 2000(0x7d0) contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, ie the address of the first input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore Discrete inputs numbered 1-16 are addressed as 0-15.

Intializes a new instance

Parameters

- **address** – The address to start reading from
- **count** – The number of bits to read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a request pdu

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes a request pdu

Returns The encoded pdu

execute(*context*)
Run a read discrete input request against a datastore

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

Parameters **context** – The datastore to request from

Returns The initializes response message, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + Quantity of Coils (n Bytes) :return:

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

The discrete inputs in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The LSB of the first data byte contains the input addressed in the query. The other inputs follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

Intializes a new instance

Parameters **values** – The request values to respond with

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes response pdu

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

encode()
Encodes response pdu

Returns The encoded packet message

getBit(*address*)
Helper function to get the specified bit's value

Parameters **address** – The bit to query

Returns The value of the requested bit

resetBit(*address*)
Helper function to set the specified bit to 0

Parameters **address** – The bit to reset

setBit(*address*, *value=1*)
Helper function to set the specified bit

Parameters

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

class pymodbus.bit_read_message.**ReadBitsRequestBase**(*address*, *count*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

Base class for Messages Requesting bit values

Initializes the read request data

Parameters

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

- **count** – The number of bits after 'address' to read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a request pdu

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes a request pdu

Returns The encoded pdu

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + Quantity of Coils (n Bytes) :return:

class pymodbus.bit_read_message.**ReadBitsResponseBase**(*values*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Base class for Messages responding to bit-reading values

Initializes a new instance

Parameters **values** – The requested values to be returned

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes response pdu

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

encode()
Encodes response pdu

Returns The encoded packet message

getBit(*address*)
Helper function to get the specified bit's value

Parameters **address** – The bit to query

Returns The value of the requested bit

resetBit(*address*)
Helper function to set the specified bit to 0

Parameters **address** – The bit to reset

setBit(*address*, *value=1*)
Helper function to set the specified bit

Parameters

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

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

This function code is used to read from 1 to 2000(0x7d0) contiguous status of coils in a remote device. The Request PDU specifies the starting address, ie the address of the first coil specified, and the number of coils. In the PDU Coils are addressed starting at zero. Therefore coils numbered 1-16 are addressed as 0-15.

Initializes a new instance

Parameters

- **address** – The address to start reading from
- **count** – The number of bits to read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a request pdu

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes a request pdu

Returns The encoded pdu

execute(*context*)
Run a read coils request against a datastore

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

Parameters **context** – The datastore to request from

Returns The initializes response message, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + Quantity of Coils (n Bytes) :return:

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

The coils in the response message are packed as one coil per bit of the data field. Status is indicated as 1= ON and 0= OFF. The LSB of the first data byte contains the output addressed in the query. The other coils follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned output quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

Intializes a new instance

Parameters **values** – The request values to respond with

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes response pdu

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

encode()
Encodes response pdu

Returns The encoded packet message

getBit(*address*)
Helper function to get the specified bit's value

Parameters **address** – The bit to query

Returns The value of the requested bit

resetBit(*address*)
Helper function to set the specified bit to 0

Parameters **address** – The bit to reset

setBit(*address*, *value=1*)
Helper function to set the specified bit

Parameters

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

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

This function code is used to read from 1 to 2000(0x7d0) contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, ie the address of the first input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore Discrete inputs numbered 1-16 are addressed as 0-15.

Intializes a new instance

Parameters

- **address** – The address to start reading from
- **count** – The number of bits to read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a request pdu

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes a request pdu

Returns The encoded pdu

execute(*context*)
Run a read discrete input request against a datastore

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

Parameters **context** – The datastore to request from

Returns The initializes response message, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + Quantity of Coils (n Bytes) :return:

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

The discrete inputs in the response message are packed as one input per bit of the data field. Status is indicated as 1= ON; 0= OFF. The LSB of the first data byte contains the input addressed in the query. The other inputs follow toward the high order end of this byte, and from low order to high order in subsequent bytes.

If the returned input quantity is not a multiple of eight, the remaining bits in the final data byte will be padded with zeros (toward the high order end of the byte). The Byte Count field specifies the quantity of complete bytes of data.

Intializes a new instance

Parameters **values** – The request values to respond with

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes response pdu

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

encode()
Encodes response pdu

Returns The encoded packet message

getBit(*address*)
Helper function to get the specified bit's value

Parameters **address** – The bit to query

Returns The value of the requested bit

resetBit(*address*)
Helper function to set the specified bit to 0

Parameters **address** – The bit to reset

setBit(*address*, *value=1*)
Helper function to set the specified bit

Parameters

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

bit_write_message — Bit Write Modbus Messages

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

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

API Documentation

Bit Writing Request/Response

TODO write mask request/response

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

Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to write a single output to either ON or OFF in a remote device.

The requested ON/OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the output to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the output.

The Request PDU specifies the address of the coil to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0. The requested ON/OFF state is specified by a constant in the Coil Value field. A value of 0XFF00 requests the coil to be ON. A value of 0X0000 requests the coil to be off. All other values are illegal and will not affect the coil.

Initializes a new instance

Parameters

- **address** – The variable address to write
- **value** – The value to write at address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coil request

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes write coil request

Returns The byte encoded message

execute(*context*)
Run a write coil request against a datastore

Parameters **context** – The datastore to request from

Returns The populated response or exception message

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

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

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

Initializes a new instance

Parameters

- **address** – The variable address written to
- **value** – The value written at address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coil response

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

encode()
Encodes write coil response

Returns The byte encoded message

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

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

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

Initializes a new instance

Parameters

- **address** – The starting request address
- **values** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coils request

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes write coils request

Returns The byte encoded message

execute(*context*)
Run a write coils request against a datastore

Parameters **context** – The datastore to request from

Returns The populated response or exception message

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

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

Initializes a new instance

Parameters

- **address** – The starting variable address written to
- **count** – The number of values written

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coils response

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

encode()
Encodes write coils response

Returns The byte encoded message

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

This function code is used to write a single output to either ON or OFF in a remote device.

The requested ON/OFF state is specified by a constant in the request data field. A value of FF 00 hex requests the output to be ON. A value of 00 00 requests it to be OFF. All other values are illegal and will not affect the output.

The Request PDU specifies the address of the coil to be forced. Coils are addressed starting at zero. Therefore coil numbered 1 is addressed as 0. The requested ON/OFF state is specified by a constant in the Coil Value field. A value of 0XFF00 requests the coil to be ON. A value of 0X0000 requests the coil to be off. All other values are illegal and will not affect the coil.

Initializes a new instance

Parameters

- **address** – The variable address to write
- **value** – The value to write at address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coil request

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes write coil request

Returns The byte encoded message

execute(*context*)
Run a write coil request against a datastore

Parameters **context** – The datastore to request from

Returns The populated response or exception message

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

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

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

Initializes a new instance

Parameters

- **address** – The variable address written to
- **value** – The value written at address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coil response

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

encode()
Encodes write coil response

Returns The byte encoded message

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

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

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

Initializes a new instance

Parameters

- **address** – The starting request address
- **values** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coils request

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes write coils request

Returns The byte encoded message

execute(*context*)
Run a write coils request against a datastore

Parameters **context** – The datastore to request from

Returns The populated response or exception message

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

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

Initializes a new instance

Parameters

- **address** – The starting variable address written to
- **count** – The number of values written

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a write coils response

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

encode()
Encodes write coils response

Returns The byte encoded message

client.common — Twisted Async Modbus Client

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

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

API Documentation

Modbus Client Common

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

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

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

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

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

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

class pymodbus.client.common.**ModbusClientMixin**

Bases: object

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

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

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

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

client.sync — Twisted Synchronous Modbus Client

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

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

API Documentation

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

Bases: *pymodbus.client.sync.BaseModbusClient*

Implementation of a modbus tcp client

Initialize a client instance

Parameters

- **host** – The host to connect to (default 127.0.0.1)
- **port** – The modbus port to connect to (default 502)
- **source_address** – The source address tuple to bind to (default ('', 0))
- **framer** – The modbus framer to use (default ModbusSocketFramer)

Note: The host argument will accept ipv4 and ipv6 hosts

---

close()
Closes the underlying socket connection

connect()
Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **address** – The starting address to read from
- **count** – The number of coils to read

- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Bases: *pymodbus.client.sync.BaseModbusClient*

Implementation of a modbus udp client

Initialize a client instance

Parameters

- **host** – The host to connect to (default 127.0.0.1)
- **port** – The modbus port to connect to (default 502)
- **framer** – The modbus framer to use (default ModbusSocketFramer)
- **timeout** – The timeout to use for this socket (default None)

close()
Closes the underlying socket connection

connect()
Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

- **address** – The address of the register to write

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Bases: *pymodbus.client.sync.BaseModbusClient*

Implementation of a modbus serial client

Initialize a serial client instance

The methods to connect are:

```
- ascii
- rtu
- binary
```

Parameters

- **method** – The method to use for connection
- **port** – The serial port to attach to
- **stopbits** – The number of stop bits to use

- **bytesize** – The bytesize of the serial messages
- **parity** – Which kind of parity to use
- **baudrate** – The baud rate to use for the serial device
- **timeout** – The timeout between serial requests (default 3s)

close()
Closes the underlying socket connection

connect()
Connect to the modbus serial server

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **address** – The starting address to write to

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

Returns A deferred response handle

class pymodbus.client.sync.**BaseModbusClient**(*framer*, ***kwargs*)

Bases: *pymodbus.client.common.ModbusClientMixin*

Inteface for a modbus synchronous client. Defined here are all the methods for performing the related request methods. Derived classes simply need to implement the transport methods and set the correct framer.

Initialize a client instance

Parameters **framer** – The modbus framer implementation to use

close()

Closes the underlying socket connection

connect()

Connect to the modbus remote host

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Bases: *pymodbus.client.sync.BaseModbusClient*

Implementation of a modbus tcp client

Initialize a client instance

Parameters

- **host** – The host to connect to (default 127.0.0.1)
- **port** – The modbus port to connect to (default 502)
- **source_address** – The source address tuple to bind to (default ('', 0))
- **framer** – The modbus framer to use (default ModbusSocketFramer)

Note: The host argument will accept ipv4 and ipv6 hosts

---

close()
Closes the underlying socket connection

connect()
Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Bases: *pymodbus.client.sync.BaseModbusClient*

Implementation of a modbus udp client

Initialize a client instance

Parameters

- **host** – The host to connect to (default 127.0.0.1)
- **port** – The modbus port to connect to (default 502)
- **framer** – The modbus framer to use (default ModbusSocketFramer)
- **timeout** – The timeout to use for this socket (default None)

close()

Closes the underlying socket connection

connect()

Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Bases: *pymodbus.client.sync.BaseModbusClient*

Implementation of a modbus serial client

Initialize a serial client instance

The methods to connect are:

```
- ascii
- rtu
- binary
```

Parameters

- **method** – The method to use for connection
- **port** – The serial port to attach to

- **stopbits** – The number of stop bits to use
- **bytesize** – The bytesize of the serial messages
- **parity** – Which kind of parity to use
- **baudrate** – The baud rate to use for the serial device
- **timeout** – The timeout between serial requests (default 3s)

close()
Closes the underlying socket connection

connect()
Connect to the modbus serial server

Returns True if connection succeeded, False otherwise

execute(*request=None*)

Parameters **request** – The request to process

Returns The result of the request execution

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **address** – The starting address to read from
- **count** – The number of registers to read

- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

client.async — Twisted Async Modbus Client

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

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

API Documentation

Implementation of a Modbus Client Using Twisted

Example run:

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

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

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

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

Another example:

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

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

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

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

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

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

Initializes the framer module

Parameters **framer** – The framer to use for the protocol

connectionLost(*reason*)
Called upon a client disconnect

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

connectionMade()
Called upon a successful client connection.

dataReceived(*data*)
Get response, check for valid message, decode result

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

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

logPrefix()
Return a prefix matching the class name, to identify log messages related to this protocol instance.

makeConnection(*transport*)
Make a connection to a transport and a server.

This sets the 'transport' attribute of this Protocol, and calls the connectionMade() callback.

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

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

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

Initializes the framer module

Parameters **framer** – The framer to use for the protocol

connectionRefused()

Called due to error from write in connected mode.

Note this is a result of ICMP message generated by *previous* write.

datagramReceived(*data*, *params*)

Get response, check for valid message, decode result

Parameters

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

doStart()

Make sure startProtocol is called.

This will be called by makeConnection(), users should not call it.

doStop()

Make sure stopProtocol is called.

This will be called by the port, users should not call it.

execute(*request*)

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

logPrefix()

Return a prefix matching the class name, to identify log messages related to this protocol instance.

makeConnection(*transport*)

Make a connection to a transport and a server.

This sets the 'transport' attribute of this DatagramProtocol, and calls the doStart() callback.

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

startProtocol()

Called when a transport is connected to this protocol.

Will only be called once, even if multiple ports are connected.

stopProtocol()
Called when the transport is disconnected.

Will only be called once, after all ports are disconnected.

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

class pymodbus.client.async.**ModbusClientFactory**
Bases: twisted.internet.protocol.ReconnectingClientFactory

Simple client protocol factory

buildProtocol(*addr*)
Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

doStart()
Make sure startFactory is called.

Users should not call this function themselves!

doStop()
Make sure stopFactory is called.

Users should not call this function themselves!

forProtocol(*protocol*, **args*, ***kwargs*)
Create a factory for the given protocol.

It sets the C{protocol} attribute and returns the constructed factory instance.

@param protocol: A L{Protocol} subclass

@param args: Positional arguments for the factory.

@param kwargs: Keyword arguments for the factory.

@return: A L{Factory} instance wired up to C{protocol}.

logPrefix()
Describe this factory for log messages.

protocol
alias of *ModbusClientProtocol*

resetDelay()
Call this method after a successful connection: it resets the delay and the retry counter.

retry(*connector=None*)
Have this connector connect again, after a suitable delay.

startFactory()
This will be called before I begin listening on a Port or Connector.

It will only be called once, even if the factory is connected to multiple ports.

This can be used to perform 'unserialization' tasks that are best put off until things are actually running, such as connecting to a database, opening files, etcetera.

startedConnecting(*connector*)
Called when a connection has been started.

You can call connector.stopConnecting() to stop the connection attempt.

@param connector: a Connector object.

stopFactory()
This will be called before I stop listening on all Ports/Connectors.

This can be overridden to perform 'shutdown' tasks such as disconnecting database connections, closing files, etc.

It will be called, for example, before an application shuts down, if it was connected to a port. User code should not call this function directly.

stopTrying()
Put a stop to any attempt to reconnect in progress.

class pymodbus.client.async.**ModbusClientProtocol**(*framer=None*, ***kwargs*)
Bases: twisted.internet.protocol.Protocol, *pymodbus.client.common.ModbusClientMixin*

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

Initializes the framer module

Parameters **framer** – The framer to use for the protocol

connectionLost(*reason*)
Called upon a client disconnect

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

connectionMade()
Called upon a successful client connection.

dataReceived(*data*)
Get response, check for valid message, decode result

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

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

logPrefix()
Return a prefix matching the class name, to identify log messages related to this protocol instance.

makeConnection(*transport*)
Make a connection to a transport and a server.

This sets the 'transport' attribute of this Protocol, and calls the connectionMade() callback.

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address
- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Parameters

- **address** – The starting address to write to
- **value** – The value to write to the specified address

- **unit** – The slave unit this request is targeting

Returns A deferred response handle

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

Parameters

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

Returns A deferred response handle

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

Bases: twisted.internet.protocol.ReconnectingClientFactory

Simple client protocol factory

buildProtocol(*addr*)

Create an instance of a subclass of Protocol.

The returned instance will handle input on an incoming server connection, and an attribute "factory" pointing to the creating factory.

Alternatively, L{None} may be returned to immediately close the new connection.

Override this method to alter how Protocol instances get created.

@param addr: an object implementing L{twisted.internet.interfaces.IAddress}

doStart()

Make sure startFactory is called.

Users should not call this function themselves!

doStop()

Make sure stopFactory is called.

Users should not call this function themselves!

forProtocol(*protocol*, **args*, ***kwargs*)

Create a factory for the given protocol.

It sets the C{protocol} attribute and returns the constructed factory instance.

@param protocol: A L{Protocol} subclass

@param args: Positional arguments for the factory.

@param kwargs: Keyword arguments for the factory.

@return: A L{Factory} instance wired up to C{protocol}.

logPrefix()

Describe this factory for log messages.

protocol

alias of *ModbusClientProtocol*

resetDelay()

Call this method after a successful connection: it resets the delay and the retry counter.

retry(*connector=None*)

Have this connector connect again, after a suitable delay.

startFactory()
This will be called before I begin listening on a Port or Connector.

It will only be called once, even if the factory is connected to multiple ports.

This can be used to perform 'unserialization' tasks that are best put off until things are actually running, such as connecting to a database, opening files, etcetera.

startedConnecting(*connector*)
Called when a connection has been started.

You can call connector.stopConnecting() to stop the connection attempt.

@param connector: a Connector object.

stopFactory()
This will be called before I stop listening on all Ports/Connectors.

This can be overridden to perform 'shutdown' tasks such as disconnecting database connections, closing files, etc.

It will be called, for example, before an application shuts down, if it was connected to a port. User code should not call this function directly.

stopTrying()
Put a stop to any attempt to reconnect in progress.

constants — Modbus Default Values

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

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

API Documentation

Constants For Modbus Server/Client

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

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

A collection of modbus default values

Port
The default modbus tcp server port (502)

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

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

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

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

TransactionId
The starting transaction identifier number (0)

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

UnitId
The modbus slave addrss. Currently this is set to 0x00 which means this request should be broadcast to all the slave devices (really means that all the devices should respons).

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

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

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

This defaults to (N)one.

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

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

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

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

Create a new instance

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

These represent various status codes in the modbus protocol.

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

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

On
This indicates that the given modbus entity is on

Off
This indicates that the given modbus entity is off

SlaveOn
This indicates that the given modbus slave is running

SlaveOff
This indicates that the given modbus slave is not running

Create a new instance

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

An enumeration representing the various byte endianess.

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

Big
This indicates that the bytes are in little endian format

Little
This indicates that the bytes are in big endian format

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

Create a new instance

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

Represents the type of modbus plus request

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

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

Create a new instance

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

Represents what type of device information to read

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

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

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

Specific
Request to return a single data object.

Create a new instance

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

Represents the more follows condition

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

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

Create a new instance

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

A collection of modbus default values

Port
The default modbus tcp server port (502)

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

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

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

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

TransactionId
The starting transaction identifier number (0)

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

UnitId
The modbus slave addrss. Currently this is set to 0x00 which means this request should be broadcast to all the slave devices (really means that all the devices should respons).

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

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

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

This defaults to (N)one.

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

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

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

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

Create a new instance

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

These represent various status codes in the modbus protocol.

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

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

On
This indicates that the given modbus entity is on

Off
This indicates that the given modbus entity is off

SlaveOn
This indicates that the given modbus slave is running

SlaveOff
This indicates that the given modbus slave is not running

Create a new instance

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

An enumeration representing the various byte endianess.

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

Big
This indicates that the bytes are in little endian format

Little
This indicates that the bytes are in big endian format

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

Create a new instance

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

Represents the type of modbus plus request

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

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

Create a new instance

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

Represents what type of device information to read

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

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

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

Specific
Request to return a single data object.

Create a new instance

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

Represents the more follows condition

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

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

Create a new instance

Server Datastores and Contexts

The following are the API documentation strings taken from the sourcecode

store — Datastore for Modbus Server Context

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

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

API Documentation

Modbus Server Datastore

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

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

Datastore Implementation

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

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

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

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

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

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

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

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

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

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

Bases: object

Base class for a modbus datastore

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

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

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

Used to initialize a store to one value

Parameters

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

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

Returns the requested values from the datastore

Parameters

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

Returns The requested values from a:a+c

reset()

Resets the datastore to the initialized default value

setValues(*address*, *values*)

Returns the requested values from the datastore

Parameters

- **address** – The starting address

- **values** – The values to store

validate(*address*, *count=1*)
Checks to see if the request is in range

Parameters

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

Returns True if the request in within range, False otherwise

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

Creates a sequential modbus datastore

Initializes the datastore

Parameters

- **address** – The starting address of the datastore
- **values** – Either a list or a dictionary of values

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

Returns An initialized datastore

default(*count*, *value=False*)
Used to initialize a store to one value

Parameters

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

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

Parameters

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

Returns The requested values from a:a+c

reset()
Resets the datastore to the initialized default value

setValues(*address*, *values*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks to see if the request is in range

Parameters

- **address** – The starting address

- **count** – The number of values to test for

Returns True if the request in within range, False otherwise

class pymodbus.datastore.store.**ModbusSparseDataBlock**(*values*)
Bases: *pymodbus.datastore.store.BaseModbusDataBlock*

Creates a sparse modbus datastore

Initializes the datastore

Using the input values we create the default datastore value and the starting address

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

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

Returns An initialized datastore

default(*count*, *value=False*)
Used to initialize a store to one value

Parameters

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

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

Parameters

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

Returns The requested values from a:a+c

reset()
Resets the datastore to the initialized default value

setValues(*address*, *values*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks to see if the request is in range

Parameters

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

Returns True if the request in within range, False otherwise

class pymodbus.datastore.store.**BaseModbusDataBlock**
Bases: object

Base class for a modbus datastore

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

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

default (*count*, *value=False*)
Used to initialize a store to one value

Parameters

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

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

Parameters

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

Returns The requested values from a:a+c

reset ()
Resets the datastore to the initialized default value

setValues (*address*, *values*)
Returns the requested values from the datastore

Parameters

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

validate (*address*, *count=1*)
Checks to see if the request is in range

Parameters

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

Returns True if the request in within range, False otherwise

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

Creates a sequential modbus datastore

Initializes the datastore

Parameters

- **address** – The starting address of the datastore
- **values** – Either a list or a dictionary of values

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

Returns An initialized datastore

default(*count*, *value=False*)
Used to initialize a store to one value

Parameters

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

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

Parameters

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

Returns The requested values from a:a+c

reset()
Resets the datastore to the initialized default value

setValues(*address*, *values*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks to see if the request is in range

Parameters

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

Returns True if the request in within range, False otherwise

class pymodbus.datastore.store.**ModbusSparseDataBlock**(*values*)
Bases: *pymodbus.datastore.store.BaseModbusDataBlock*

Creates a sparse modbus datastore

Initializes the datastore

Using the input values we create the default datastore value and the starting address

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

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

Returns An initialized datastore

default(*count*, *value=False*)
Used to initialize a store to one value

Parameters

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

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

Parameters

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

Returns The requested values from a:a+c

reset()
Resets the datastore to the initialized default value

setValues(*address*, *values*)
Sets the requested values of the datastore

Parameters

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

validate(*address*, *count=1*)
Checks to see if the request is in range

Parameters

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

Returns True if the request in within range, False otherwise

context — Modbus Server Contexts

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

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

API Documentation

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

This represents a master collection of slave contexts. If single is set to true, it will be treated as a single context so every unit-id returns the same context. If single is set to false, it will be interpreted as a collection of slave contexts.

Initializes a new instance of a modbus server context.

Parameters

- **slaves** – A dictionary of client contexts
- **single** – Set to true to treat this as a single context

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

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

Initializes the datastores, defaults to fully populated sequential data blocks if none are passed in.

Parameters **kwargs** – Each element is a ModbusDataBlock

'di' - Discrete Inputs initializer 'co' - Coils initializer 'hr' - Holding Register initializer 'ir' - Input Registers iniatializer

decode(*fx*)
Converts the function code to the datastore to

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

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

getValues(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

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

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

Initializes the datastores, defaults to fully populated sequential data blocks if none are passed in.

Parameters **kwargs** – Each element is a ModbusDataBlock

'di' - Discrete Inputs initializer 'co' - Coils initializer 'hr' - Holding Register initializer 'ir' - Input Registers iniatializer

decode(*fx*)
Converts the function code to the datastore to

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

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

getValues (*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset ()
Resets all the datastores to their default values

setValues (*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate (*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

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

This represents a master collection of slave contexts. If single is set to true, it will be treated as a single context so every unit-id returns the same context. If single is set to false, it will be interpreted as a collection of slave contexts.

Initializes a new instance of a modbus server context.

Parameters

- **slaves** – A dictionary of client contexts
- **single** – Set to true to treat this as a single context

remote — Remote Slave Context

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

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

API Documentation

class pymodbus.datastore.remote.**RemoteSlaveContext**(*client*)
Bases: *pymodbus.interfaces.IModbusSlaveContext*

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

Initializes the datastores

Parameters **client** – The client to retrieve values with

decode(*fx*)
Converts the function code to the datastore to

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

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

getValues(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

class pymodbus.datastore.remote.**RemoteSlaveContext**(*client*)
Bases: *pymodbus.interfaces.IModbusSlaveContext*

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

Initializes the datastores

Parameters **client** – The client to retrieve values with

decode (*fx*)
Converts the function code to the datastore to

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

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

getValues (*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset ()
Resets all the datastores to their default values

setValues (*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate (*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

diag_message — Diagnostic Modbus Messages

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

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

API Documentation

Diagnostic Record Read/Write

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

class pymodbus.diag_message.**DiagnosticStatusRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

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

Base initializer for a diagnostic request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

class pymodbus.diag_message.**DiagnosticStatusResponse**(***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

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

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

Base initializer for a diagnostic response

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

Initializes a new instance of the request

Parameters **message** – The message to send to loopback

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

execute(**args*)
Executes the loopback request (builds the response)

Returns The populated loopback response message

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

class pymodbus.diag_message.**ReturnQueryDataResponse**(*message=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusResponse*

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

Initializes a new instance of the response

Parameters **message** – The message to loopback

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

class pymodbus.diag_message.**RestartCommunicationsOptionRequest**(*toggle=False*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusRequest*

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

Initializes a new request

Parameters **toggle** – Set to True to toggle, False otherwise

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

execute(**args*)
Clear event log and restart

Returns The initialized response message

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

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

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

Initializes a new response

Parameters **toggle** – Set to True if we toggled, False otherwise

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

Forces the addressed remote device to its Listen Only Mode for MODBUS communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

class pymodbus.diag_message.**ForceListenOnlyModeResponse**(***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusResponse*

Forces the addressed remote device to its Listen Only Mode for MODBUS communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

This does not send a response

Initializer to block a return response

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7 .

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

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

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

Clears the overrun error counter and reset the error flag

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

Clears the overrun error counter and reset the error flag

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

Returns The initialized response message

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

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

Returns a series of 54 16-bit words (108 bytes) in the data field of the response (this function differs from the usual two-byte length of the data field). The data contains the statistics for the Modbus Plus peer processor in the slave device.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

class pymodbus.diag_message.**DiagnosticStatusRequest**(**kwargs)
Bases: *pymodbus.pdu.ModbusRequest*

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

Base initializer for a diagnostic request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

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

class pymodbus.diag_message.**DiagnosticStatusResponse**(**kwargs)
Bases: *pymodbus.pdu.ModbusResponse*

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

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

Base initializer for a diagnostic response

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

A large majority of the diagnostic functions are simple status request functions. They work by sending 0x0000 as data and their function code and they are returned 2 bytes of data.

If a function inherits this, they only need to implement the execute method

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

execute(**args*)
Base function to raise if not implemented

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

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

A large majority of the diagnostic functions are simple status request functions. They work by sending 0x0000 as data and their function code and they are returned 2 bytes of data.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

Initializes a new instance of the request

Parameters **message** – The message to send to loopback

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

execute(**args*)
Executes the loopback request (builds the response)

Returns The populated loopback response message

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

class pymodbus.diag_message.**ReturnQueryDataResponse**(*message=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusResponse*

The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

Initializes a new instance of the response

Parameters **message** – The message to loopback

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

class pymodbus.diag_message.**RestartCommunicationsOptionRequest**(*toggle=False*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusRequest*

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

Initializes a new request

Parameters **toggle** – Set to True to toggle, False otherwise

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

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

Returns The encoded packet

execute(**args*)
Clear event log and restart

Returns The initialized response message

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

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

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

Initializes a new response

Parameters **toggle** – Set to True if we toggled, False otherwise

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

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

Returns The encoded packet

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

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

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

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

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

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnDiagnosticRegisterResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The contents of the remote device's 16-bit diagnostic register are returned in the response

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ChangeAsciiInputDelimiterRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The character 'CHAR' passed in the request data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases of a Line Feed is not required at the end of ASCII messages.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ChangeAsciiInputDelimiterResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The character 'CHAR' passed in the request data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases of a Line Feed is not required at the end of ASCII messages.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ForceListenOnlyModeRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

Forces the addressed remote device to its Listen Only Mode for MODBUS communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ForceListenOnlyModeResponse**(***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusResponse*

Forces the addressed remote device to its Listen Only Mode for MODBUS communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

This does not send a response

Initializer to block a return response

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ClearCountersRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The goal is to clear ll counters and the diagnostic register. Also, counters are cleared upon power-up

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ClearCountersResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The goal is to clear ll counters and the diagnostic register. Also, counters are cleared upon power-up

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnBusMessageCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of messages that the remote device has detected on the communications systems since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnBusMessageCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages that the remote device has detected on the communications systems since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnBusCommunicationErrorCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of CRC errors encountered by the remote device since its last restart, clear counter operation, or power-up

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnBusCommunicationErrorCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of CRC errors encountered by the remote device since its last restart, clear counter operation, or power-up

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnBusExceptionErrorCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of modbus exception responses returned by the remote device since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnBusExceptionErrorCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of modbus exception responses returned by the remote device since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnSlaveMessageCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnSlaveMessageCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnSlaveNoResponseCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnSlaveNoReponseCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnSlaveNAKCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7 .

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnSlaveNAKCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnSlaveBusyCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnSlaveBusyCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnSlaveBusCharacterOverrunCountRequest**(*data=0*, ***kwargs*)

Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnSlaveBusCharacterOverrunCountResponse**(*data=0*, ***kwargs*)

Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ReturnIopOverrunCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

An IOP overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction. This function is specific to the 884.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ReturnIopOverrunCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

The response data field returns the quantity of messages addressed to the slave that it could not handle due to an 884 IOP overrun condition, since its last restart, clear counters operation, or power-up.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**ClearOverrunCountRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

Clears the overrun error counter and reset the error flag

An error flag should be cleared, but nothing else in the specification mentions is, so it is ignored.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**ClearOverrunCountResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

Clears the overrun error counter and reset the error flag

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

class pymodbus.diag_message.**GetClearModbusPlusRequest**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleRequest*

In addition to the Function code (08) and Subfunction code (00 15 hex) in the query, a two-byte Operation field is used to specify either a 'Get Statistics' or a 'Clear Statistics' operation. The two operations are exclusive - the 'Get' operation cannot clear the statistics, and the 'Clear' operation does not return statistics prior to clearing them. Statistics are also cleared on power-up of the slave device.

General initializer for a simple diagnostic request

The data defaults to 0x0000 if not provided as over half of the functions require it.

Parameters **data** – The data to send along with the request

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic request

Parameters **data** – The data to decode into the function code

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

execute(**args*)
Execute the diagnostic request on the given device

Returns The initialized response message

get_response_pdu_size()
Func_code (1 byte) + Sub function code (2 byte) + Data (2 * N bytes) :return:

class pymodbus.diag_message.**GetClearModbusPlusResponse**(*data=0*, ***kwargs*)
Bases: *pymodbus.diag_message.DiagnosticStatusSimpleResponse*

Returns a series of 54 16-bit words (108 bytes) in the data field of the response (this function differs from the usual two-byte length of the data field). The data contains the statistics for the Modbus Plus peer processor in the slave device.

General initializer for a simple diagnostic response

Parameters **data** – The resulting data to return to the client

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Base decoder for a diagnostic response

Parameters **data** – The data to decode into the function code

encode()
Base encoder for a diagnostic response we encode the data set in self.message

Returns The encoded packet

device — Modbus Device Representation

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Modbus Device Controller

These are the device management handlers. They should be maintained in the server context and the various methods should be inserted in the correct locations.

class pymodbus.device.**ModbusAccessControl**
Bases: *pymodbus.interfaces.Singleton*

This is a simple implementation of a Network Management System table. Its purpose is to control access to the server (if it is used). We assume that if an entry is in the table, it is allowed accesses to resources. However, if the host does not appear in the table (all unknown hosts) its connection will simply be closed.

Since it is a singleton, only one version can possible exist and all instances pull from here.

Create a new instance

add(*host*)
Add allowed host(s) from the NMS table

Parameters **host** – The host to add

check(*host*)
Check if a host is allowed to access resources

Parameters **host** – The host to check

remove(*host*)
Remove allowed host(s) from the NMS table

Parameters **host** – The host to remove

class pymodbus.device.**ModbusPlusStatistics**
Bases: object

This is used to maintain the current modbus plus statistics count. As of right now this is simply a stub to complete the modbus implementation. For more information, see the modbus implementation guide page 87.

Initialize the modbus plus statistics with the default information.

encode()
Returns a summary of the modbus plus statistics

Returns 54 16-bit words representing the status

reset()
This clears all of the modbus plus statistics

summary()
Returns a summary of the modbus plus statistics

Returns 54 16-bit words representing the status

class pymodbus.device.**ModbusDeviceIdentification**(*info=None*)
Bases: object

This is used to supply the device identification for the readDeviceIdentification function

For more information read section 6.21 of the modbus application protocol.

Initialize the datastore with the elements you need. (note acceptable range is [0x00-0x06,0x80-0xFF] inclusive)

Parameters **information** – A dictionary of {int:string} of values

summary()
Return a summary of the main items

Returns An dictionary of the main items

update(*value*)
Update the values of this identity using another identify as the value

Parameters **value** – The value to copy values from

class pymodbus.device.**DeviceInformationFactory**
Bases: *pymodbus.interfaces.Singleton*

This is a helper factory that really just hides some of the complexity of processing the device information requests (function code 0x2b 0x0e).

Create a new instance

classmethod get(*control*, *read_code=1*, *object_id=0*)
Get the requested device data from the system

Parameters

- **control** – The control block to pull data from
- **read_code** – The read code to process
- **object_id** – The specific object_id to read

Returns The requested data (id, length, value)

class pymodbus.device.**ModbusControlBlock**
Bases: *pymodbus.interfaces.Singleton*

This is a global singleotn that controls all system information

All activity should be logged here and all diagnostic requests should come from here.

Create a new instance

addEvent(*event*)
Adds a new event to the event log

Parameters **event** – A new event to add to the log

clearEvents()
Clears the current list of events

getDiagnostic(*bit*)
This gets the value in the diagnostic register

Parameters **bit** – The bit to get

Returns The current value of the requested bit

getDiagnosticRegister()
This gets the entire diagnostic register

Returns The diagnostic register collection

getEvents()
Returns an encoded collection of the event log.

Returns The encoded events packet

reset()
This clears all of the system counters and the diagnostic register

setDiagnostic(*mapping*)
This sets the value in the diagnostic register

Parameters **mapping** – Dictionary of key:value pairs to set

class pymodbus.device.**ModbusAccessControl**
Bases: *pymodbus.interfaces.Singleton*

This is a simple implementation of a Network Management System table. Its purpose is to control access to the server (if it is used). We assume that if an entry is in the table, it is allowed accesses to resources. However, if the host does not appear in the table (all unknown hosts) its connection will simply be closed.

Since it is a singleton, only one version can possible exist and all instances pull from here.

Create a new instance

add(*host*)
Add allowed host(s) from the NMS table

Parameters **host** – The host to add

check(*host*)
Check if a host is allowed to access resources

Parameters **host** – The host to check

remove(*host*)
Remove allowed host(s) from the NMS table

Parameters **host** – The host to remove

class pymodbus.device.**ModbusPlusStatistics**
Bases: object

This is used to maintain the current modbus plus statistics count. As of right now this is simply a stub to complete the modbus implementation. For more information, see the modbus implementation guide page 87.

Initialize the modbus plus statistics with the default information.

encode()
Returns a summary of the modbus plus statistics

Returns 54 16-bit words representing the status

reset()
This clears all of the modbus plus statistics

summary()
Returns a summary of the modbus plus statistics

Returns 54 16-bit words representing the status

class pymodbus.device.**ModbusDeviceIdentification**(*info=None*)
Bases: object

This is used to supply the device identification for the readDeviceIdentification function

For more information read section 6.21 of the modbus application protocol.

Initialize the datastore with the elements you need. (note acceptable range is [0x00-0x06,0x80-0xFF] inclusive)

Parameters **information** – A dictionary of {int:string} of values

summary()
Return a summary of the main items

Returns An dictionary of the main items

update(*value*)
Update the values of this identity using another identify as the value

Parameters **value** – The value to copy values from

class pymodbus.device.**DeviceInformationFactory**
Bases: *pymodbus.interfaces.Singleton*

This is a helper factory that really just hides some of the complexity of processing the device information requests (function code 0x2b 0x0e).

Create a new instance

classmethod get(*control*, *read_code=1*, *object_id=0*)
Get the requested device data from the system

Parameters

- **control** – The control block to pull data from
- **read_code** – The read code to process
- **object_id** – The specific object_id to read

Returns The requested data (id, length, value)

class pymodbus.device.**ModbusControlBlock**
Bases: *pymodbus.interfaces.Singleton*

This is a global singleotn that controls all system information

All activity should be logged here and all diagnostic requests should come from here.

Create a new instance

addEvent(*event*)
Adds a new event to the event log

Parameters **event** – A new event to add to the log

clearEvents()
Clears the current list of events

getDiagnostic(*bit*)
This gets the value in the diagnostic register

Parameters **bit** – The bit to get

Returns The current value of the requested bit

getDiagnosticRegister()
This gets the entire diagnostic register

Returns The diagnostic register collection

getEvents()
Returns an encoded collection of the event log.

Returns The encoded events packet

reset()
This clears all of the system counters and the diagnostic register

setDiagnostic(*mapping*)
This sets the value in the diagnostic register

Parameters **mapping** – Dictionary of key:value pairs to set

factory — Request/Response Decoders

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Modbus Request/Response Decoder Factories

The following factories make it easy to decode request/response messages. To add a new request/response pair to be decodeable by the library, simply add them to the respective function lookup table (order doesn't matter, but it does help keep things organized).

Regardless of how many functions are added to the lookup, O(1) behavior is kept as a result of a pre-computed lookup dictionary.

class pymodbus.factory.**ServerDecoder**
Bases: *pymodbus.interfaces.IModbusDecoder*

Request Message Factory (Server)

To add more implemented functions, simply add them to the list

Initializes the client lookup tables

decode(*message*)
Wrapper to decode a request packet

Parameters **message** – The raw modbus request packet

Returns The decoded modbus message or None if error

lookupPduClass(*function_code*)
Use *function_code* to determine the class of the PDU.

Parameters **function_code** – The function code specified in a frame.

Returns The class of the PDU that has a matching *function_code*.

class pymodbus.factory.**ClientDecoder**
Bases: *pymodbus.interfaces.IModbusDecoder*

Response Message Factory (Client)

To add more implemented functions, simply add them to the list

Initializes the client lookup tables

decode(*message*)
Wrapper to decode a response packet

Parameters **message** – The raw packet to decode

Returns The decoded modbus message or None if error

lookupPduClass(*function_code*)
Use *function_code* to determine the class of the PDU.

Parameters **function_code** – The function code specified in a frame.

Returns The class of the PDU that has a matching *function_code*.

class pymodbus.factory.**ServerDecoder**
Bases: *pymodbus.interfaces.IModbusDecoder*

Request Message Factory (Server)

To add more implemented functions, simply add them to the list

Initializes the client lookup tables

decode(*message*)
Wrapper to decode a request packet

Parameters **message** – The raw modbus request packet

Returns The decoded modbus message or None if error

lookupPduClass(*function_code*)
Use *function_code* to determine the class of the PDU.

Parameters **function_code** – The function code specified in a frame.

Returns The class of the PDU that has a matching *function_code*.

class pymodbus.factory.**ClientDecoder**
Bases: *pymodbus.interfaces.IModbusDecoder*

Response Message Factory (Client)

To add more implemented functions, simply add them to the list

Initializes the client lookup tables

decode (*message*)
Wrapper to decode a response packet

Parameters **message** – The raw packet to decode

Returns The decoded modbus message or None if error

lookupPduClass (*function_code*)
Use *function_code* to determine the class of the PDU.

Parameters **function_code** – The function code specified in a frame.

Returns The class of the PDU that has a matching *function_code*.

interfaces — System Interfaces

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Pymodbus Interfaces

A collection of base classes that are used throughout the pymodbus library.

class pymodbus.interfaces.**Singleton**
Bases: object

Singleton base class http://mail.python.org/pipermail/python-list/2007-July/450681.html

Create a new instance

class pymodbus.interfaces.**IModbusDecoder**
Bases: object

Modbus Decoder Base Class

This interface must be implemented by a modbus message decoder factory. These factories are responsible for abstracting away converting a raw packet into a request / response message object.

decode (*message*)
Wrapper to decode a given packet

Parameters **message** – The raw modbus request packet

Returns The decoded modbus message or None if error

lookupPduClass (*function_code*)
Use *function_code* to determine the class of the PDU.

Parameters **function_code** – The function code specified in a frame.

Returns The class of the PDU that has a matching *function_code*.

class pymodbus.interfaces.**IModbusFramer**
Bases: object

A framer strategy interface. The idea is that we abstract away all the detail about how to detect if a current message frame exists, decoding it, sending it, etc so that we can plug in a new Framer object (tcp, rtu, ascii).

addToFrame(*message*)
Add the next message to the frame buffer

This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

The raw packet is built off of a fully populated modbus request / response message.

Parameters **message** – The request/response to send

Returns The built packet

checkFrame()
Check and decode the next frame

Returns True if we successful, False otherwise

getFrame()
Get the next frame from the buffer

Returns The frame data or ‘’

isFrameReady()
Check if we should continue decode logic

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result with current frame header

We basically copy the data back over from the current header to the result header. This may not be needed for serial messages.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

class pymodbus.interfaces.**IModbusSlaveContext**
Bases: object

Interface for a modbus slave data context

Derived classes must implemented the following methods: reset(self) validate(self, fx, address, count=1) getValues(self, fx, address, count=1) setValues(self, fx, address, values)

decode (*fx*)
Converts the function code to the datastore to

Parameters **fx** – The function we are working with

Returns one of [d(iscretes),i(inputs),h(oliding),c(oils)

getValues (*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

- **fx** – The function we are working with
- **address** – The starting address
- **count** – The number of values to retrieve

Returns The requested values from a:a+c

reset ()
Resets all the datastores to their default values

setValues (*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

- **fx** – The function we are working with
- **address** – The starting address
- **values** – The new values to be set

validate (*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

- **fx** – The function we are working with
- **address** – The starting address
- **count** – The number of values to test

Returns True if the request in within range, False otherwise

class pymodbus.interfaces.**IPayloadBuilder**
Bases: object

This is an interface to a class that can build a payload for a modbus register write command. It should abstract the codec for encoding data to the required format (bcd, binary, char, etc).

build ()
Return the payload buffer as a list

This list is two bytes per element and can thus be treated as a list of registers.

Returns The payload buffer as a list

class pymodbus.interfaces.**Singleton**
Bases: object

Singleton base class http://mail.python.org/pipermail/python-list/2007-July/450681.html

Create a new instance

class pymodbus.interfaces.**IModbusDecoder**
Bases: object

Modbus Decoder Base Class

This interface must be implemented by a modbus message decoder factory. These factories are responsible for abstracting away converting a raw packet into a request / response message object.

decode(*message*)
Wrapper to decode a given packet

Parameters **message** – The raw modbus request packet

Returns The decoded modbus message or None if error

lookupPduClass(*function_code*)
Use *function_code* to determine the class of the PDU.

Parameters **function_code** – The function code specified in a frame.

Returns The class of the PDU that has a matching *function_code*.

class pymodbus.interfaces.**IModbusFramer**
Bases: object

A framer strategy interface. The idea is that we abstract away all the detail about how to detect if a current message frame exists, decoding it, sending it, etc so that we can plug in a new Framer object (tcp, rtu, ascii).

addToFrame(*message*)
Add the next message to the frame buffer

This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

The raw packet is built off of a fully populated modbus request / response message.

Parameters **message** – The request/response to send

Returns The built packet

checkFrame()
Check and decode the next frame

Returns True if we successful, False otherwise

getFrame()
Get the next frame from the buffer

Returns The frame data or ''

isFrameReady()
Check if we should continue decode logic

This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result with current frame header

We basically copy the data back over from the current header to the result header. This may not be needed for serial messages.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

class pymodbus.interfaces.**IModbusSlaveContext**
Bases: object

Interface for a modbus slave data context

Derived classes must implemented the following methods: reset(self) validate(self, fx, address, count=1) getValues(self, fx, address, count=1) setValues(self, fx, address, values)

decode(*fx*)
Converts the function code to the datastore to

Parameters **fx** – The function we are working with

Returns one of [d(iscretes),i(inputs),h(oliding),c(oils)

getValues(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

- **fx** – The function we are working with
- **address** – The starting address
- **count** – The number of values to retrieve

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

- **fx** – The function we are working with
- **address** – The starting address
- **values** – The new values to be set

validate(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

- **fx** – The function we are working with
- **address** – The starting address
- **count** – The number of values to test

Returns True if the request in within range, False otherwise

class pymodbus.interfaces.**IPayloadBuilder**
Bases: object

This is an interface to a class that can build a payload for a modbus register write command. It should abstract the codec for encoding data to the required format (bcd, binary, char, etc).

build()
Return the payload buffer as a list

This list is two bytes per element and can thus be treated as a list of registers.

Returns The payload buffer as a list

exceptions — Exceptions Used in PyModbus

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Pymodbus Exceptions

Custom exceptions to be used in the Modbus code.

exception pymodbus.exceptions.**ModbusException**(*string*)
Bases: exceptions.Exception

Base modbus exception

Initialize the exception :param string: The message to append to the error

exception pymodbus.exceptions.**ModbusIOException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from data i/o

Initialize the exception :param string: The message to append to the error

exception pymodbus.exceptions.**ParameterException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from invalid parameter

Initialize the exception

Parameters **string** – The message to append to the error

exception pymodbus.exceptions.**NotImplementedException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from not implemented function

Initialize the exception :param string: The message to append to the error

exception pymodbus.exceptions.**ConnectionException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from a bad connection

Initialize the exception

Parameters **string** – The message to append to the error

exception pymodbus.exceptions.**NoSuchSlaveException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from making a request to a slave that does not exist

Initialize the exception

Parameters **string** – The message to append to the error

class pymodbus.exceptions.**ModbusException**(*string*)
Bases: exceptions.Exception

Base modbus exception

Initialize the exception :param string: The message to append to the error

class pymodbus.exceptions.**ModbusIOException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from data i/o

Initialize the exception :param string: The message to append to the error

class pymodbus.exceptions.**ParameterException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from invalid parameter

Initialize the exception

Parameters **string** – The message to append to the error

class pymodbus.exceptions.**NotImplementedException**(*string=''*)
Bases: *pymodbus.exceptions.ModbusException*

Error resulting from not implemented function

Initialize the exception :param string: The message to append to the error

other_message — Other Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Diagnostic record read/write

Currently not all implemented

class pymodbus.other_message.**ReadExceptionStatusRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to read the contents of eight Exception Status outputs in a remote device. The function provides a simple method for accessing this information, because the Exception Output references are known (no output reference is needed in the function).

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**ReadExceptionStatusResponse**(*status=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains the status of the eight Exception Status outputs. The outputs are packed into one data byte, with one bit per output. The status of the lowest output reference is contained in the least significant bit of the byte. The contents of the eight Exception Status outputs are device specific.

Initializes a new instance

Parameters **status** – The status response to report

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**GetCommEventCounterRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to get a status word and an event count from the remote device's communication event counter.

By fetching the current count before and after a series of messages, a client can determine whether the messages were handled normally by the remote device.

The device's event counter is incremented once for each successful message completion. It is not incremented for exception responses, poll commands, or fetch event counter commands.

The event counter can be reset by means of the Diagnostics function (code 08), with a subfunction of Restart Communications Option (code 00 01) or Clear Counters and Diagnostic Register (code 00 0A).

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**GetCommEventCounterResponse**(*count=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains a two-byte status word, and a two-byte event count. The status word will be all ones (FF FF hex) if a previously-issued program command is still being processed by the remote device (a busy condition exists). Otherwise, the status word will be all zeros.

Initializes a new instance

Parameters **count** – The current event counter value

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**GetCommEventLogRequest**(**kwargs)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to get a status word, event count, message count, and a field of event bytes from the remote device.

The status word and event counts are identical to that returned by the Get Communications Event Counter function (11, 0B hex).

The message counter contains the quantity of messages processed by the remote device since its last restart, clear counters operation, or power-up. This count is identical to that returned by the Diagnostic function (code 08), sub-function Return Bus Message Count (code 11, 0B hex).

The event bytes field contains 0-64 bytes, with each byte corresponding to the status of one MODBUS send or receive operation for the remote device. The remote device enters the events into the field in chronological order. Byte 0 is the most recent event. Each new byte flushes the oldest byte from the field.

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**GetCommEventLogResponse**(**kwargs)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains a two-byte status word field, a two-byte event count field, a two-byte message count field, and a field containing 0-64 bytes of events. A byte count field defines the total length of the data in these four field

Initializes a new instance

Parameters

- **status** – The status response to report
- **message_count** – The current message count

- **event_count** – The current event count
- **events** – The collection of events to send

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**ReportSlaveIdRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to read the description of the type, the current status, and other information specific to a remote device.

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**ReportSlaveIdResponse**(*identifier='x00'*, *status=True*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The format of a normal response is shown in the following example. The data contents are specific to each type of device.

Initializes a new instance

Parameters

- **identifier** – The identifier of the slave

- **status** – The status response to report

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Since the identifier is device dependent, we just return the raw value that a user can decode to whatever it should be.

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**ReadExceptionStatusRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to read the contents of eight Exception Status outputs in a remote device. The function provides a simple method for accessing this information, because the Exception Output references are known (no output reference is needed in the function).

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**ReadExceptionStatusResponse**(*status=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains the status of the eight Exception Status outputs. The outputs are packed into one data byte, with one bit per output. The status of the lowest output reference is contained in the least significant bit of the byte. The contents of the eight Exception Status outputs are device specific.

Initializes a new instance

Parameters **status** – The status response to report

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**GetCommEventCounterRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to get a status word and an event count from the remote device's communication event counter.

By fetching the current count before and after a series of messages, a client can determine whether the messages were handled normally by the remote device.

The device's event counter is incremented once for each successful message completion. It is not incremented for exception responses, poll commands, or fetch event counter commands.

The event counter can be reset by means of the Diagnostics function (code 08), with a subfunction of Restart Communications Option (code 00 01) or Clear Counters and Diagnostic Register (code 00 0A).

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**GetCommEventCounterResponse**(*count=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains a two-byte status word, and a two-byte event count. The status word will be all ones (FF FF hex) if a previously-issued program command is still being processed by the remote device (a busy condition exists). Otherwise, the status word will be all zeros.

Initializes a new instance

Parameters **count** – The current event counter value

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**ReportSlaveIdRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to read the description of the type, the current status, and other information specific to a remote device.

Initializes a new instance

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**ReportSlaveIdResponse**(*identifier='x00'*, *status=True*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The format of a normal response is shown in the following example. The data contents are specific to each type of device.

Initializes a new instance

Parameters

- **identifier** – The identifier of the slave
- **status** – The status response to report

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Since the identifier is device dependent, we just return the raw value that a user can decode to whatever it should be.

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

mei_message — MEI Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Encapsulated Interface (MEI) Transport Messages

class pymodbus.mei_message.**ReadDeviceInformationRequest**(*read_code=None*, *object_id=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code allows reading the identification and additional information relative to the physical and functional description of a remote device, only.

The Read Device Identification interface is modeled as an address space composed of a set of addressable data elements. The data elements are called objects and an object Id identifies them.

Initializes a new instance

Parameters

- **read_code** – The device information read code
- **object_id** – The object to read from

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.mei_message.**ReadDeviceInformationResponse**(*read_code=None*, *information=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Initializes a new instance

Parameters

- **read_code** – The device information read code
- **information** – The requested information request

classmethod calculateRtuFrameSize(*buffer*)
Calculates the size of the message

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the response.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.mei_message.**ReadDeviceInformationRequest**(*read_code=None*, *object_id=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code allows reading the identification and additional information relative to the physical and functional description of a remote device, only.

The Read Device Identification interface is modeled as an address space composed of a set of addressable data elements. The data elements are called objects and an object Id identifies them.

Initializes a new instance

Parameters

- **read_code** – The device information read code

- **object_id** – The object to read from

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.mei_message.**ReadDeviceInformationResponse**(*read_code=None*, *information=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Initializes a new instance

Parameters

- **read_code** – The device information read code
- **information** – The requested information request

classmethod **calculateRtuFrameSize**(*buffer*)
Calculates the size of the message

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the response.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

file_message — File Modbus Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

File Record Read/Write Messages

Currently none of these messages are implemented

class pymodbus.file_message.**FileRecord**(***kwargs*)
Bases: object

Represents a file record and its relevant data.

Initializes a new instance

Params reference_type Defaults to 0x06 (must be)

Params file_number Indicates which file number we are reading

Params record_number Indicates which record in the file

Params record_data The actual data of the record

Params record_length The length in registers of the record

Params response_length The length in bytes of the record

class pymodbus.file_message.**ReadFileRecordRequest**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to perform a file record read. All request data lengths are provided in terms of number of bytes and all record lengths are provided in terms of registers.

A file is an organization of records. Each file contains 10000 records, addressed 0000 to 9999 decimal or 0x0000 to 0x270f. For example, record 12 is addressed as 12. The function can read multiple groups of references. The groups can be separating (non-contiguous), but the references within each group must be sequential. Each group is defined in a seperate 'sub-request' field that contains seven bytes:

```
The reference type: 1 byte (must be 0x06)
The file number: 2 bytes
The starting record number within the file: 2 bytes
The length of the record to be read: 2 bytes
```

The quantity of registers to be read, combined with all other fields in the expected response, must not exceed the allowable length of the MODBUS PDU: 235 bytes.

Initializes a new instance

Parameters **records** – The file record requests to be read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**ReadFileRecordResponse**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response is a series of 'sub-responses,' one for each 'sub-request.' The byte count field is the total combined count of bytes in all 'sub-responses.' In addition, each 'sub-response' contains a field that shows its own byte count.

Initializes a new instance

Parameters **records** – The requested file records

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.file_message.**WriteFileRecordRequest**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to perform a file record write. All request data lengths are provided in terms of number of bytes and all record lengths are provided in terms of the number of 16 bit words.

Initializes a new instance

Parameters **records** – The file record requests to be read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run the write file record request against the context

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**WriteFileRecordResponse**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request.

Initializes a new instance

Parameters **records** – The file record requests to be read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.file_message.**MaskWriteRegisterRequest**(*address=0*, *and_mask=65535*, *or_mask=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to modify the contents of a specified holding register using a combination of an AND mask, an OR mask, and the register's current contents. The function can be used to set or clear individual bits in the register.

Initializes a new instance

Parameters

- **address** – The mask pointer address (0x0000 to 0xffff)
- **and_mask** – The and bitmask to apply to the register address

- **or_mask** – The or bitmask to apply to the register address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a mask write register request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**MaskWriteRegisterResponse**(*address=0*, *and_mask=65535*, *or_mask=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request. The response is returned after the register has been written.

Initializes a new instance

Parameters

- **address** – The mask pointer address (0x0000 to 0xffff)
- **and_mask** – The and bitmask applied to the register address
- **or_mask** – The or bitmask applied to the register address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.file_message.**ReadFifoQueueRequest**(*address=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code allows to read the contents of a First-In-First-Out (FIFO) queue of register in a remote device. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers.

The queue count register is returned first, followed by the queued data registers. The function reads the queue contents, but does not clear them.

Initializes a new instance

Parameters **address** – The fifo pointer address (0x0000 to 0xffff)

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**ReadFifoQueueResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

In a normal response, the byte count shows the quantity of bytes to follow, including the queue count bytes and value register bytes (but not including the error check field). The queue count is the quantity of data registers in the queue (not including the count register).

If the queue count exceeds 31, an exception response is returned with an error code of 03 (Illegal Data Value).

Initializes a new instance

Parameters **values** – The list of values of the fifo to return

classmethod **calculateRtuFrameSize**(*buffer*)
Calculates the size of the message

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the response.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.file_message.**FileRecord**(***kwargs*)
Bases: object

Represents a file record and its relevant data.

Initializes a new instance

Params reference_type Defaults to 0x06 (must be)

Params file_number Indicates which file number we are reading

Params record_number Indicates which record in the file

Params record_data The actual data of the record

Params record_length The length in registers of the record

Params response_length The length in bytes of the record

class pymodbus.file_message.**ReadFileRecordRequest**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to perform a file record read. All request data lengths are provided in terms of number of bytes and all record lengths are provided in terms of registers.

A file is an organization of records. Each file contains 10000 records, addressed 0000 to 9999 decimal or 0x0000 to 0x270f. For example, record 12 is addressed as 12. The function can read multiple groups of references. The groups can be separating (non-contiguous), but the references within each group must be sequential. Each group is defined in a seperate 'sub-request' field that contains seven bytes:

```
The reference type: 1 byte (must be 0x06)
The file number: 2 bytes
The starting record number within the file: 2 bytes
The length of the record to be read: 2 bytes
```

The quantity of registers to be read, combined with all other fields in the expected response, must not exceed the allowable length of the MODBUS PDU: 235 bytes.

Initializes a new instance

Parameters **records** – The file record requests to be read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**ReadFileRecordResponse**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response is a series of 'sub-responses,' one for each 'sub-request.' The byte count field is the total combined count of bytes in all 'sub-responses.' In addition, each 'sub-response' contains a field that shows its own byte count.

Initializes a new instance

Parameters **records** – The requested file records

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.file_message.**WriteFileRecordRequest**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to perform a file record write. All request data lengths are provided in terms of number of bytes and all record lengths are provided in terms of the number of 16 bit words.

Initializes a new instance

Parameters **records** – The file record requests to be read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run the write file record request against the context

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**WriteFileRecordResponse**(*records=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request.

Initializes a new instance

Parameters **records** – The file record requests to be read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.file_message.**ReadFifoQueueRequest**(*address=0*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code allows to read the contents of a First-In-First-Out (FIFO) queue of register in a remote device. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers.

The queue count register is returned first, followed by the queued data registers. The function reads the queue contents, but does not clear them.

Initializes a new instance

Parameters **address** – The fifo pointer address (0x0000 to 0xffff)

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes the incoming request

Parameters **data** – The data to decode into the address

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

Parameters **context** – The datastore to request from

Returns The populated response

class pymodbus.file_message.**ReadFifoQueueResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

In a normal response, the byte count shows the quantity of bytes to follow, including the queue count bytes and value register bytes (but not including the error check field). The queue count is the quantity of data registers in the queue (not including the count register).

If the queue count exceeds 31, an exception response is returned with an error code of 03 (Illegal Data Value).

Initializes a new instance

Parameters **values** – The list of values of the fifo to return

classmethod **calculateRtuFrameSize**(*buffer*)
Calculates the size of the message

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the response.

decode(*data*)
Decodes a the response

Parameters **data** – The packet data to decode

encode()
Encodes the response

Returns The byte encoded message

events — Events Used in PyModbus

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Modbus Remote Events

An event byte returned by the Get Communications Event Log function can be any one of four types. The type is defined by bit 7 (the high-order bit) in each byte. It may be further defined by bit 6.

class pymodbus.events.**CommunicationRestartEvent**
Bases: *pymodbus.events.ModbusEvent*

Remote device Initiated Communication Restart

The remote device stores this type of event byte when its communications port is restarted. The remote device can be restarted by the Diagnostics function (code 08), with sub-function Restart Communications Option (code 00 01).

That function also places the remote device into a 'Continue on Error' or 'Stop on Error' mode. If the remote device is placed into 'Continue on Error' mode, the event byte is added to the existing event log. If the remote device is placed into 'Stop on Error' mode, the byte is added to the log and the rest of the log is cleared to zeros.

The event is defined by a content of zero.

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**EnteredListenModeEvent**
Bases: *pymodbus.events.ModbusEvent*

Remote device Entered Listen Only Mode

The remote device stores this type of event byte when it enters the Listen Only Mode. The event is defined by a content of 04 hex.

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteReceiveEvent**(***kwargs*)
Bases: *pymodbus.events.ModbusEvent*

Remote device MODBUS Receive Event

The remote device stores this type of event byte when a query message is received. It is stored before the remote device processes the message. This event is defined by bit 7 set to logic '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
---------------------------------
0   Not Used
2   Not Used
3   Not Used
4   Character Overrun
5   Currently in Listen Only Mode
6   Broadcast Receive
7   1
```

Initialize a new event instance

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteSendEvent**(**kwargs*)
Bases: *pymodbus.events.ModbusEvent*

Remote device MODBUS Send Event

The remote device stores this type of event byte when it finishes processing a request message. It is stored if the remote device returned a normal or exception response, or no response.

This event is defined by bit 7 set to a logic '0', with bit 6 set to a '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
-------------------------------------------------------------
0   Read Exception Sent (Exception Codes 1-3)
1   Slave Abort Exception Sent (Exception Code 4)
2   Slave Busy Exception Sent (Exception Codes 5-6)
3   Slave Program NAK Exception Sent (Exception Code 7)
4   Write Timeout Error Occurred
5   Currently in Listen Only Mode
6   1
7   0
```

Initialize a new event instance

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**ModbusEvent**
Bases: object

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteReceiveEvent**(**kwargs*)
Bases: *pymodbus.events.ModbusEvent*

Remote device MODBUS Receive Event

The remote device stores this type of event byte when a query message is received. It is stored before the remote device processes the message. This event is defined by bit 7 set to logic '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
---------------------------------
0   Not Used
2   Not Used
3   Not Used
4   Character Overrun
5   Currently in Listen Only Mode
6   Broadcast Receive
7   1
```

Initialize a new event instance

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteSendEvent**(**kwargs*)
Bases: *pymodbus.events.ModbusEvent*

Remote device MODBUS Send Event

The remote device stores this type of event byte when it finishes processing a request message. It is stored if the remote device returned a normal or exception response, or no response.

This event is defined by bit 7 set to a logic '0', with bit 6 set to a '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
-------------------------------------------------------------
0   Read Exception Sent (Exception Codes 1-3)
1   Slave Abort Exception Sent (Exception Code 4)
2   Slave Busy Exception Sent (Exception Codes 5-6)
3   Slave Program NAK Exception Sent (Exception Code 7)
4   Write Timeout Error Occurred
5   Currently in Listen Only Mode
6   1
7   0
```

Initialize a new event instance

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**EnteredListenModeEvent**
Bases: *pymodbus.events.ModbusEvent*

Remote device Entered Listen Only Mode

The remote device stores this type of event byte when it enters the Listen Only Mode. The event is defined by a content of 04 hex.

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**CommunicationRestartEvent**
Bases: *pymodbus.events.ModbusEvent*

Remote device Initiated Communication Restart

The remote device stores this type of event byte when its communications port is restarted. The remote device can be restarted by the Diagnostics function (code 08), with sub-function Restart Communications Option (code 00 01).

That function also places the remote device into a 'Continue on Error' or 'Stop on Error' mode. If the remote device is placed into 'Continue on Error' mode, the event byte is added to the existing event log. If the remote device is placed into 'Stop on Error' mode, the byte is added to the log and the rest of the log is cleared to zeros.

The event is defined by a content of zero.

decode(*event*)
Decodes the event message to its status bits

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

payload — Modbus Payload Utilities

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Modbus Payload Builders

A collection of utilities for building and decoding modbus messages payloads.

class pymodbus.payload.**BinaryPayloadBuilder**(*payload=None*, *endian='<'*)
Bases: *pymodbus.interfaces.IPayloadBuilder*

A utility that helps build payload messages to be written with the various modbus messages. It really is just a simple wrapper around the struct module, however it saves time looking up the format strings. What follows is a simple example:

```
builder = BinaryPayloadBuilder(endian=Endian.Little)
builder.add_8bit_uint(1)
builder.add_16bit_uint(2)
payload = builder.build()
```

Initialize a new instance of the payload builder

Parameters

- **payload** – Raw payload data to initialize with
- **endian** – The endianess of the payload

add_16bit_int(*value*)
Adds a 16 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_16bit_uint(*value*)
Adds a 16 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_32bit_float(*value*)
Adds a 32 bit float to the buffer

Parameters **value** – The value to add to the buffer

add_32bit_int(*value*)
Adds a 32 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_32bit_uint(*value*)
Adds a 32 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_64bit_float(*value*)
Adds a 64 bit float(double) to the buffer

Parameters **value** – The value to add to the buffer

add_64bit_int(*value*)
Adds a 64 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_64bit_uint(*value*)
Adds a 64 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_8bit_int(*value*)
Adds a 8 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_8bit_uint(*value*)
Adds a 8 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_bits(*values*)
Adds a collection of bits to be encoded

If these are less than a multiple of eight, they will be left padded with 0 bits to make it so.

Parameters **value** – The value to add to the buffer

add_string(*value*)
Adds a string to the buffer

Parameters **value** – The value to add to the buffer

build()
Return the payload buffer as a list

This list is two bytes per element and can thus be treated as a list of registers.

Returns The payload buffer as a list

reset()
Reset the payload buffer

to_registers()
Convert the payload buffer into a register layout that can be used as a context block.

Returns The register layout to use as a block

to_string()
Return the payload buffer as a string

Returns The payload buffer as a string

class pymodbus.payload.**BinaryPayloadDecoder**(*payload*, *endian='<'*)
Bases: object

A utility that helps decode payload messages from a modbus reponse message. It really is just a simple wrapper around the struct module, however it saves time looking up the format strings. What follows is a simple example:

```
decoder = BinaryPayloadDecoder(payload)
first   = decoder.decode_8bit_uint()
second  = decoder.decode_16bit_uint()
```

Initialize a new payload decoder

Parameters

- **payload** – The payload to decode with
- **endian** – The endianess of the payload

decode_16bit_int()
Decodes a 16 bit signed int from the buffer

decode_16bit_uint()
Decodes a 16 bit unsigned int from the buffer

decode_32bit_float()
Decodes a 32 bit float from the buffer

decode_32bit_int()
Decodes a 32 bit signed int from the buffer

decode_32bit_uint()
Decodes a 32 bit unsigned int from the buffer

decode_64bit_float()
Decodes a 64 bit float(double) from the buffer

decode_64bit_int()
Decodes a 64 bit signed int from the buffer

decode_64bit_uint()
Decodes a 64 bit unsigned int from the buffer

decode_8bit_int()
Decodes a 8 bit signed int from the buffer

decode_8bit_uint()
Decodes a 8 bit unsigned int from the buffer

decode_bits()
Decodes a byte worth of bits from the buffer

decode_string(*size=1*)
Decodes a string from the buffer

Parameters **size** – The size of the string to decode

classmethod fromCoils(*klass*, *coils*, *endian='<'*)
Initialize a payload decoder with the result of reading a collection of coils from a modbus device.

The coils are treated as a list of bit(boolean) values.

Parameters

- **coils** – The coil results to initialize with
- **endian** – The endianess of the payload

Returns An initialized PayloadDecoder

classmethod fromRegisters(*klass*, *registers*, *endian='<'*)
Initialize a payload decoder with the result of reading a collection of registers from a modbus device.

The registers are treated as a list of 2 byte values. We have to do this because of how the data has already been decoded by the rest of the library.

Parameters

- **registers** – The register results to initialize with
- **endian** – The endianess of the payload

Returns An initialized PayloadDecoder

reset()
Reset the decoder pointer back to the start

class pymodbus.payload.**BinaryPayloadBuilder**(*payload=None*, *endian='<'*)
Bases: *pymodbus.interfaces.IPayloadBuilder*

A utility that helps build payload messages to be written with the various modbus messages. It really is just a simple wrapper around the struct module, however it saves time looking up the format strings. What follows is a simple example:

```
builder = BinaryPayloadBuilder(endian=Endian.Little)
builder.add_8bit_uint(1)
builder.add_16bit_uint(2)
payload = builder.build()
```

Initialize a new instance of the payload builder

Parameters

- **payload** – Raw payload data to initialize with
- **endian** – The endianess of the payload

add_16bit_int(*value*)
Adds a 16 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_16bit_uint(*value*)
Adds a 16 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_32bit_float(*value*)
Adds a 32 bit float to the buffer

Parameters **value** – The value to add to the buffer

add_32bit_int(*value*)
Adds a 32 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_32bit_uint(*value*)
Adds a 32 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_64bit_float(*value*)
Adds a 64 bit float(double) to the buffer

Parameters **value** – The value to add to the buffer

add_64bit_int(*value*)
Adds a 64 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_64bit_uint(*value*)
Adds a 64 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_8bit_int(*value*)
Adds a 8 bit signed int to the buffer

Parameters **value** – The value to add to the buffer

add_8bit_uint(*value*)
Adds a 8 bit unsigned int to the buffer

Parameters **value** – The value to add to the buffer

add_bits(*values*)
Adds a collection of bits to be encoded

If these are less than a multiple of eight, they will be left padded with 0 bits to make it so.

Parameters **value** – The value to add to the buffer

add_string(*value*)
Adds a string to the buffer

Parameters **value** – The value to add to the buffer

build()
Return the payload buffer as a list

This list is two bytes per element and can thus be treated as a list of registers.

Returns The payload buffer as a list

reset()
Reset the payload buffer

to_registers()
Convert the payload buffer into a register layout that can be used as a context block.

Returns The register layout to use as a block

to_string()
Return the payload buffer as a string

Returns The payload buffer as a string

class pymodbus.payload.**BinaryPayloadDecoder**(*payload*, *endian='<'*)
Bases: object

A utility that helps decode payload messages from a modbus reponse message. It really is just a simple wrapper around the struct module, however it saves time looking up the format strings. What follows is a simple example:

```
decoder = BinaryPayloadDecoder(payload)
first   = decoder.decode_8bit_uint()
second  = decoder.decode_16bit_uint()
```

Initialize a new payload decoder

Parameters

- **payload** – The payload to decode with
- **endian** – The endianess of the payload

decode_16bit_int()
Decodes a 16 bit signed int from the buffer

decode_16bit_uint()
Decodes a 16 bit unsigned int from the buffer

decode_32bit_float()
Decodes a 32 bit float from the buffer

decode_32bit_int()
Decodes a 32 bit signed int from the buffer

decode_32bit_uint()
Decodes a 32 bit unsigned int from the buffer

decode_64bit_float()
Decodes a 64 bit float(double) from the buffer

decode_64bit_int()
Decodes a 64 bit signed int from the buffer

decode_64bit_uint()
Decodes a 64 bit unsigned int from the buffer

decode_8bit_int()
Decodes a 8 bit signed int from the buffer

decode_8bit_uint()
Decodes a 8 bit unsigned int from the buffer

decode_bits()
Decodes a byte worth of bits from the buffer

decode_string(*size=1*)
Decodes a string from the buffer

Parameters **size** – The size of the string to decode

classmethod fromCoils(*klass*, *coils*, *endian='<'*)
Initialize a payload decoder with the result of reading a collection of coils from a modbus device.

The coils are treated as a list of bit(boolean) values.

Parameters

- **coils** – The coil results to initialize with
- **endian** – The endianess of the payload

Returns An initialized PayloadDecoder

classmethod fromRegisters(*klass*, *registers*, *endian='<'*)
Initialize a payload decoder with the result of reading a collection of registers from a modbus device.

The registers are treated as a list of 2 byte values. We have to do this because of how the data has already been decoded by the rest of the library.

Parameters

- **registers** – The register results to initialize with
- **endian** – The endianess of the payload

Returns An initialized PayloadDecoder

reset()
Reset the decoder pointer back to the start

pdu — Base Structures

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Contains base classes for modbus request/response/error packets

class pymodbus.pdu.**ModbusRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusPDU*

Base class for a modbus request PDU

Proxy to the lower level initializer

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – is a string object

Raises A not implemented exception

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

Raises A not implemented exception

class pymodbus.pdu.**ModbusResponse**(***kwargs*)
Bases: *pymodbus.pdu.ModbusPDU*

Base class for a modbus response PDU

should_respond
A flag that indicates if this response returns a result back to the client issuing the request

_rtu_frame_size
Indicates the size of the modbus rtu response used for calculating how much to read.

Proxy to the lower level initializer

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – is a string object

Raises A not implemented exception

encode()
Encodes the message

Raises A not implemented exception

class pymodbus.pdu.**ModbusExceptions**
Bases: *pymodbus.interfaces.Singleton*

An enumeration of the valid modbus exceptions

Create a new instance

classmethod decode(*code*)
Given an error code, translate it to a string error name.

Parameters **code** – The code number to translate

class pymodbus.pdu.**ExceptionResponse**(*function_code*, *exception_code=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Base class for a modbus exception PDU

Initializes the modbus exception response

Parameters

- **function_code** – The function to build an exception response for
- **exception_code** – The specific modbus exception to return

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a modbus exception response

Parameters **data** – The packet data to decode

encode()
Encodes a modbus exception response

Returns The encoded exception packet

class pymodbus.pdu.**IllegalFunctionRequest**(*function_code*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

Defines the Modbus slave exception type 'Illegal Function' This exception code is returned if the slave:

```
- does not implement the function code **or**
- is not in a state that allows it to process the function
```

Initializes a IllegalFunctionRequest

Parameters **function_code** – The function we are erroring on

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
This is here so this failure will run correctly

Parameters **data** – Not used

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

Raises A not implemented exception

execute(*context*)
Builds an illegal function request error response

Parameters **context** – The current context for the message

Returns The error response packet

class pymodbus.pdu.**ModbusPDU**(***kwargs*)
Bases: object

Base class for all Modbus mesages

transaction_id
This value is used to uniquely identify a request response pair. It can be implemented as a simple counter

protocol_id
This is a constant set at 0 to indicate Modbus. It is put here for ease of expansion.

unit_id
This is used to route the request to the correct child. In the TCP modbus, it is used for routing (or not used at all. However, for the serial versions, it is used to specify which child to perform the requests against. The value 0x00 represents the broadcast address (also 0xff).

check
This is used for LRC/CRC in the serial modbus protocols

skip_encode
This is used when the message payload has already been encoded. Generally this will occur when the PayloadBuilder is being used to create a complicated message. By setting this to True, the request will pass the currently encoded message through instead of encoding it again.

Initializes the base data for a modbus request

classmethod **calculateRtuFrameSize**(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – is a string object

Raises A not implemented exception

encode()
Encodes the message

Raises A not implemented exception

class pymodbus.pdu.**ModbusRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusPDU*

Base class for a modbus request PDU

Proxy to the lower level initializer

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – is a string object

Raises A not implemented exception

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

Raises A not implemented exception

class pymodbus.pdu.**ModbusResponse**(***kwargs*)
Bases: *pymodbus.pdu.ModbusPDU*

Base class for a modbus response PDU

should_respond
A flag that indicates if this response returns a result back to the client issuing the request

_rtu_frame_size
Indicates the size of the modbus rtu response used for calculating how much to read.

Proxy to the lower level initializer

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes data part of the message.

Parameters **data** – is a string object

Raises A not implemented exception

encode()
Encodes the message

Raises A not implemented exception

class pymodbus.pdu.**ModbusExceptions**
Bases: *pymodbus.interfaces.Singleton*

An enumeration of the valid modbus exceptions

Create a new instance

classmethod decode(*code*)
Given an error code, translate it to a string error name.

Parameters **code** – The code number to translate

class pymodbus.pdu.**ExceptionResponse**(*function_code*, *exception_code=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Base class for a modbus exception PDU

Initializes the modbus exception response

Parameters

- **function_code** – The function to build an exception response for
- **exception_code** – The specific modbus exception to return

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decodes a modbus exception response

Parameters **data** – The packet data to decode

encode()
Encodes a modbus exception response

Returns The encoded exception packet

class pymodbus.pdu.**IllegalFunctionRequest**(*function_code*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

Defines the Modbus slave exception type 'Illegal Function' This exception code is returned if the slave:

```
- does not implement the function code **or**
- is not in a state that allows it to process the function
```

Initializes a IllegalFunctionRequest

Parameters **function_code** – The function we are erroring on

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
This is here so this failure will run correctly

Parameters **data** – Not used

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the message

Raises A not implemented exception

execute(*context*)
Builds an illegal function request error response

Parameters **context** – The current context for the message

Returns The error response packet

pymodbus — Pymodbus Library

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

Pymodbus: Modbus Protocol Implementation

TwistedModbus is built on top of the code developed by:

> Copyright (c) 2001-2005 S.W.A.C. GmbH, Germany. Copyright (c) 2001-2005 S.W.A.C. Bohemia s.r.o., Czech Republic. Hynek Petrak <hynek@swac.cz>

Released under the the BSD license

register_read_message — Register Read Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Register Reading Request/Response

class pymodbus.register_read_message.**ReadHoldingRegistersRequest**(*address=None*, *count=None*, ***kwargs*)

Bases: *pymodbus.register_read_message.ReadRegistersRequestBase*

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

Initializes a new instance of the request

Parameters

- **address** – The starting address to read from
- **count** – The number of registers to read from address

calculateRtuFrameSize(*buffer*)

Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)

Decode a register request packet

Parameters **data** – The request to decode

doException(*exception*)

Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a read holding request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + 2 * Quantity of Coils (n Bytes) :return:

class pymodbus.register_read_message.**ReadHoldingRegistersResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersResponseBase*

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

Initializes a new response instance

Parameters **values** – The resulting register values

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register response packet

Parameters **data** – The request to decode

encode()
Encodes the response packet

Returns The encoded packet

getRegister(*index*)
Get the requested register

Parameters **index** – The indexed register to retrieve

Returns The request register

class pymodbus.register_read_message.**ReadInputRegistersRequest**(*address=None*, *count=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersRequestBase*

This function code is used to read from 1 to approx. 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

Initializes a new instance of the request

Parameters

- **address** – The starting address to read from

- **count** – The number of registers to read from address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register request packet

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a read input request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + 2 * Quantity of Coils (n Bytes) :return:

class pymodbus.register_read_message.**ReadInputRegistersResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersResponseBase*

This function code is used to read from 1 to approx. 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

Initializes a new response instance

Parameters **values** – The resulting register values

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register response packet

Parameters **data** – The request to decode

encode()
Encodes the response packet

Returns The encoded packet

getRegister(*index*)
Get the requested register

Parameters **index** – The indexed register to retrieve

Returns The request register

class pymodbus.register_read_message.**ReadWriteMultipleRegistersRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction. The write operation is performed before the read.

Holding registers are addressed starting at zero. Therefore holding registers 1-16 are addressed in the PDU as 0-15.

The request specifies the starting address and number of holding registers to be read as well as the starting address, number of holding registers, and the data to be written. The byte count specifies the number of bytes to follow in the write data field."

Initializes a new request message

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode the register request packet

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

class pymodbus.register_read_message.**ReadWriteMultipleRegistersResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains the data from the group of registers that were read. The byte count field specifies the quantity of bytes to follow in the read data field.

Initializes a new instance

Parameters **values** – The register values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode the register response packet

Parameters **data** – The response to decode

encode()
Encodes the response packet

Returns The encoded packet

class pymodbus.register_read_message.**ReadRegistersRequestBase**(*address*, *count*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

Base class for reading a modbus register

Initializes a new instance

Parameters

- **address** – The address to start the read from
- **count** – The number of registers to read

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register request packet

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + 2 * Quantity of Coils (n Bytes) :return:

class pymodbus.register_read_message.**ReadRegistersResponseBase**(*values*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

Base class for responsing to a modbus register read

Initializes a new instance

Parameters **values** – The values to write to

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register response packet

Parameters **data** – The request to decode

encode()
Encodes the response packet

Returns The encoded packet

getRegister(*index*)
Get the requested register

Parameters **index** – The indexed register to retrieve

Returns The request register

class pymodbus.register_read_message.**ReadHoldingRegistersRequest**(*address=None*, *count=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersRequestBase*

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

Initializes a new instance of the request

Parameters

- **address** – The starting address to read from
- **count** – The number of registers to read from address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register request packet

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a read holding request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + 2 * Quantity of Coils (n Bytes) :return:

class pymodbus.register_read_message.**ReadHoldingRegistersResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersResponseBase*

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

Initializes a new response instance

Parameters **values** – The resulting register values

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register response packet

Parameters **data** – The request to decode

encode()
Encodes the response packet

Returns The encoded packet

getRegister(*index*)
Get the requested register

Parameters **index** – The indexed register to retrieve

Returns The request register

class pymodbus.register_read_message.**ReadInputRegistersRequest**(*address=None*, *count=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersRequestBase*

This function code is used to read from 1 to approx. 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

Initializes a new instance of the request

Parameters

- **address** – The starting address to read from
- **count** – The number of registers to read from address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register request packet

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a read input request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Byte Count(1 byte) + 2 * Quantity of Coils (n Bytes) :return:

class pymodbus.register_read_message.**ReadInputRegistersResponse**(*values=None*, ***kwargs*)
Bases: *pymodbus.register_read_message.ReadRegistersResponseBase*

This function code is used to read from 1 to approx. 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

Initializes a new response instance

Parameters **values** – The resulting register values

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a register response packet

Parameters **data** – The request to decode

encode()
Encodes the response packet

Returns The encoded packet

getRegister(*index*)
Get the requested register

Parameters **index** – The indexed register to retrieve

Returns The request register

class pymodbus.register_read_message.**ReadWriteMultipleRegistersRequest**(***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction. The write operation is performed before the read.

Holding registers are addressed starting at zero. Therefore holding registers 1-16 are addressed in the PDU as 0-15.

The request specifies the starting address and number of holding registers to be read as well as the starting address, number of holding registers, and the data to be written. The byte count specifies the number of bytes to follow in the write data field."

Initializes a new request message

Parameters

- **read_address** – The address to start reading from
- **read_count** – The number of registers to read from address
- **write_address** – The address to start writing to
- **write_registers** – The registers to write to the specified address

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode the register request packet

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

class pymodbus.register_read_message.**ReadWriteMultipleRegistersResponse**(*values=None*, ***kwargs*)

Bases: *pymodbus.pdu.ModbusResponse*

The normal response contains the data from the group of registers that were read. The byte count field specifies the quantity of bytes to follow in the read data field.

Initializes a new instance

Parameters **values** – The register values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode the register response packet

Parameters **data** – The response to decode

encode()
Encodes the response packet

Returns The encoded packet

register_write_message — Register Write Messages

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Register Writing Request/Response Messages

class pymodbus.register_write_message.**WriteSingleRegisterRequest**(*address=None*, *value=None*, ***kwargs*)

Bases: pymodbus.pdu.ModbusRequest

This function code is used to write a single holding register in a remote device.

The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

Initializes a new instance

Parameters

- **address** – The address to start writing add
- **value** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encode a write single register packet packet request

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Register Address(2 byte) + Register Value (2 bytes) :return:

class pymodbus.register_write_message.**WriteSingleRegisterResponse**(*address=None*, *value=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request, returned after the register contents have been written.

Initializes a new instance

Parameters

- **address** – The address to start writing add
- **value** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

class pymodbus.register_write_message.**WriteMultipleRegistersRequest**(*address=None*, *values=None*, ***kwargs*)
Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to write a block of contiguous registers (1 to approx. 120 registers) in a remote device.

The requested written values are specified in the request data field. Data is packed as two bytes per register.

Initializes a new instance

Parameters

- **address** – The address to start writing to
- **values** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encode a write single register packet packet request

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

class pymodbus.register_write_message.**WriteMultipleRegistersResponse**(*address=None*, *count=None*, ***kwargs*)

Bases: *pymodbus.pdu.ModbusResponse*

"The normal response returns the function code, starting address, and quantity of registers written.

Initializes a new instance

Parameters

- **address** – The address to start writing to
- **count** – The number of registers to write to

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

class pymodbus.register_write_message.**WriteSingleRegisterRequest**(*address=None*, *value=None*, ***kwargs*)

Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to write a single holding register in a remote device.

The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

Initializes a new instance

Parameters

- **address** – The address to start writing add
- **value** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encode a write single register packet packet request

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

get_response_pdu_size()
Func_code (1 byte) + Register Address(2 byte) + Register Value (2 bytes) :return:

class pymodbus.register_write_message.**WriteSingleRegisterResponse**(*address=None*, *value=None*, ***kwargs*)

Bases: *pymodbus.pdu.ModbusResponse*

The normal response is an echo of the request, returned after the register contents have been written.

Initializes a new instance

Parameters

- **address** – The address to start writing add
- **value** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

class pymodbus.register_write_message.**WriteMultipleRegistersRequest**(*address=None*, *values=None*, ***kwargs*)

Bases: *pymodbus.pdu.ModbusRequest*

This function code is used to write a block of contiguous registers (1 to approx. 120 registers) in a remote device.

The requested written values are specified in the request data field. Data is packed as two bytes per register.

Initializes a new instance

Parameters

- **address** – The address to start writing to
- **values** – The values to write

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

doException(*exception*)
Builds an error response based on the function

Parameters **exception** – The exception to return

Raises An exception response

encode()
Encode a write single register packet packet request

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

Parameters **context** – The datastore to request from

Returns An initialized response, exception message otherwise

class pymodbus.register_write_message.**WriteMultipleRegistersResponse**(*address=None*, *count=None*, ***kwargs*)

Bases: *pymodbus.pdu.ModbusResponse*

"The normal response returns the function code, starting address, and quantity of registers written.

Initializes a new instance

Parameters

- **address** – The address to start writing to

- **count** – The number of registers to write to

calculateRtuFrameSize(*buffer*)
Calculates the size of a PDU.

Parameters **buffer** – A buffer containing the data that have been received.

Returns The number of bytes in the PDU.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

server.sync — Twisted Synchronous Modbus Server

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Implementation of a Threaded Modbus Server

pymodbus.server.sync.**StartTcpServer**(*context=None*, *identity=None*, *address=None*, ***kwargs*)
A factory to start and run a tcp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.sync.**StartUdpServer**(*context=None*, *identity=None*, *address=None*, ***kwargs*)
A factory to start and run a udp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **framer** – The framer to operate with (default ModbusSocketFramer)
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.sync.**StartSerialServer**(*context=None*, *identity=None*, ***kwargs*)
A factory to start and run a serial modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **framer** – The framer to operate with (default ModbusAsciiFramer)
- **port** – The serial port to attach to
- **stopbits** – The number of stop bits to use
- **bytesize** – The bytesize of the serial messages
- **parity** – Which kind of parity to use
- **baudrate** – The baud rate to use for the serial device
- **timeout** – The timeout to use for the serial device
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

class pymodbus.server.sync.**ModbusBaseRequestHandler**(*request*, *client_address*, *server*)
Bases: SocketServer.BaseRequestHandler

Implements the modbus server protocol

This uses the socketserver.BaseRequestHandler to implement the client handler.

execute(*request*)
The callback to call with the resulting message

Parameters **request** – The decoded request message

finish()
Callback for when a client disconnects

handle()
Callback when we receive any data

send(*message*)
Send a request (string) to the network

Parameters **message** – The unencoded modbus response

setup()
Callback for when a client connects

class pymodbus.server.sync.**ModbusSingleRequestHandler**(*request*, *client_address*, *server*)
Bases: *pymodbus.server.sync.ModbusBaseRequestHandler*

Implements the modbus server protocol

This uses the socketserver.BaseRequestHandler to implement the client handler for a single client(serial clients)

execute(*request*)
The callback to call with the resulting message

Parameters **request** – The decoded request message

finish()
Callback for when a client disconnects

handle()
Callback when we receive any data

send(*message*)
Send a request (string) to the network

Parameters **message** – The unencoded modbus response

setup()
Callback for when a client connects

class pymodbus.server.sync.**ModbusConnectedRequestHandler**(*request*, *client_address*, *server*)
Bases: *pymodbus.server.sync.ModbusBaseRequestHandler*

Implements the modbus server protocol

This uses the socketserver.BaseRequestHandler to implement the client handler for a connected protocol (TCP).

execute(*request*)
The callback to call with the resulting message

Parameters **request** – The decoded request message

finish()
Callback for when a client disconnects

handle()
Callback when we receive any data, until self.running becomes not True. Blocks indefinitely awaiting data. If shutdown is required, then the global socket.settimeout(<seconds>) may be used, to allow timely checking of self.running. However, since this also affects socket connects, if there are outgoing socket connections used in the same program, then these will be prevented, if the specfied timeout is too short. Hence, this is unreliable.

To respond to Modbus...Server.server_close() (which clears each handler's self.running), derive from this class to provide an alternative handler that awakens from time to time when no input is available and checks self.running. Use Modbus...Server(handler=...) keyword to supply the alternative request handler class.

send(*message*)
Send a request (string) to the network

Parameters **message** – The unencoded modbus response

setup()
Callback for when a client connects

class pymodbus.server.sync.**ModbusDisconnectedRequestHandler**(*request*, *client_address*, *server*)
Bases: *pymodbus.server.sync.ModbusBaseRequestHandler*

Implements the modbus server protocol

This uses the socketserver.BaseRequestHandler to implement the client handler for a disconnected protocol (UDP). The only difference is that we have to specify who to send the resulting packet data to.

execute(*request*)
The callback to call with the resulting message

Parameters **request** – The decoded request message

finish()
Callback for when a client disconnects

handle()
Callback when we receive any data

send(*message*)
Send a request (string) to the network

Parameters **message** – The unencoded modbus response

setup()
Callback for when a client connects

class pymodbus.server.sync.**ModbusTcpServer**(*context*, *framer=None*, *identity=None*, *address=None*, *handler=None*, ***kwargs*)
Bases: SocketServer.ThreadingTCPServer

A modbus threaded tcp socket server

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

Overloaded initializer for the socket server

If the identify structure is not passed in, the ModbusControlBlock uses its own empty structure.

Parameters

- **context** – The ModbusServerContext datastore
- **framer** – The framer strategy to use
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **handler** – A handler for each client session; default is ModbusConnectedRequestHandler
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

close_request(*request*)
Called to clean up an individual request.

fileno()
Return socket file number.

Interface required by select().

finish_request(*request*, *client_address*)
Finish one request by instantiating RequestHandlerClass.

get_request()
Get the request and client address from the socket.

May be overridden.

handle_error(*request*, *client_address*)
Handle an error gracefully. May be overridden.

The default is to print a traceback and continue.

handle_request()
Handle one request, possibly blocking.

Respects self.timeout.

handle_timeout()
Called if no new request arrives within self.timeout.

Overridden by ForkingMixIn.

process_request(*request*, *client*)
Callback for connecting a new client thread

Parameters

- **request** – The request to handle
- **client** – The address of the client

process_request_thread(*request*, *client_address*)
Same as in BaseServer but as a thread.

In addition, exception handling is done here.

serve_forever(*poll_interval=0.5*)
Handle one request at a time until shutdown.

Polls for shutdown every poll_interval seconds. Ignores self.timeout. If you need to do periodic tasks, do them in another thread.

server_activate()
Called by constructor to activate the server.

May be overridden.

server_bind()
Called by constructor to bind the socket.

May be overridden.

server_close()
Callback for stopping the running server

shutdown()
Stops the serve_forever loop.

Overridden to signal handlers to stop.

shutdown_request(*request*)
Called to shutdown and close an individual request.

verify_request(*request*, *client_address*)
Verify the request. May be overridden.

Return True if we should proceed with this request.

class pymodbus.server.sync.**ModbusUdpServer**(*context*, *framer=None*, *identity=None*, *address=None*, *handler=None*, ***kwargs*)
Bases: SocketServer.ThreadingUDPServer

A modbus threaded udp socket server

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

Overloaded initializer for the socket server

If the identify structure is not passed in, the ModbusControlBlock uses its own empty structure.

Parameters

- **context** – The ModbusServerContext datastore
- **framer** – The framer strategy to use
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **handler** – A handler for each client session; default is ModbusDisonnectedRequestHandler
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

fileno()
Return socket file number.

Interface required by select().

finish_request(*request*, *client_address*)
Finish one request by instantiating RequestHandlerClass.

handle_error(*request*, *client_address*)
Handle an error gracefully. May be overridden.

The default is to print a traceback and continue.

handle_request()
Handle one request, possibly blocking.

Respects self.timeout.

handle_timeout()
Called if no new request arrives within self.timeout.

Overridden by ForkingMixIn.

process_request(*request*, *client*)
Callback for connecting a new client thread

Parameters

- **request** – The request to handle
- **client** – The address of the client

process_request_thread(*request*, *client_address*)
Same as in BaseServer but as a thread.

In addition, exception handling is done here.

serve_forever(*poll_interval=0.5*)
Handle one request at a time until shutdown.

Polls for shutdown every poll_interval seconds. Ignores self.timeout. If you need to do periodic tasks, do them in another thread.

server_bind()
Called by constructor to bind the socket.

May be overridden.

server_close()
Callback for stopping the running server

shutdown()
Stops the serve_forever loop.

Blocks until the loop has finished. This must be called while serve_forever() is running in another thread, or it will deadlock.

verify_request(*request*, *client_address*)
Verify the request. May be overridden.

Return True if we should proceed with this request.

class pymodbus.server.sync.**ModbusSerialServer**(*context*, *framer=None*, *identity=None*, ***kwargs*)

Bases: object

A modbus threaded serial socket server

We inherit and overload the socket server so that we can control the client threads as well as have a single server context instance.

Overloaded initializer for the socket server

If the identify structure is not passed in, the ModbusControlBlock uses its own empty structure.

Parameters

- **context** – The ModbusServerContext datastore
- **framer** – The framer strategy to use
- **identity** – An optional identify structure
- **port** – The serial port to attach to
- **stopbits** – The number of stop bits to use
- **bytesize** – The bytesize of the serial messages
- **parity** – Which kind of parity to use
- **baudrate** – The baud rate to use for the serial device
- **timeout** – The timeout to use for the serial device
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

serve_forever()
Callback for connecting a new client thread

Parameters

- **request** – The request to handle
- **client** – The address of the client

server_close()
Callback for stopping the running server

pymodbus.server.sync.**StartTcpServer**(*context=None*, *identity=None*, *address=None*, ***kwargs*)
A factory to start and run a tcp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.sync.**StartUdpServer**(*context=None*, *identity=None*, *address=None*, ***kwargs*)
A factory to start and run a udp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **address** – An optional (interface, port) to bind to.
- **framer** – The framer to operate with (default ModbusSocketFramer)
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

pymodbus.server.sync.**StartSerialServer**(*context=None*, *identity=None*, ***kwargs*)
A factory to start and run a serial modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure
- **framer** – The framer to operate with (default ModbusAsciiFramer)
- **port** – The serial port to attach to
- **stopbits** – The number of stop bits to use
- **bytesize** – The bytesize of the serial messages
- **parity** – Which kind of parity to use
- **baudrate** – The baud rate to use for the serial device
- **timeout** – The timeout to use for the serial device
- **ignore_missing_slaves** – True to not send errors on a request to a missing slave

server.async — Twisted Asynchronous Modbus Server

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

transaction — Transaction Controllers for Pymodbus

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Collection of transaction based abstractions

class pymodbus.transaction.**FifoTransactionManager**(*client*, ***kwargs*)
Bases: pymodbus.transaction.ModbusTransactionManager

Impelements a transaction for a manager where the results are returned in a FIFO manner.

Initializes an instance of the ModbusTransactionManager

Parameters **client** – The client socket wrapper

addTransaction(*request*, *tid=None*)
Adds a transaction to the handler

This holds the requets in case it needs to be resent. After being sent, the request is removed.

Parameters

- **request** – The request to hold on to

- **tid** – The overloaded transaction id to use

delTransaction(*tid*)
Removes a transaction matching the referenced tid

Parameters **tid** – The transaction to remove

execute(*request*)
Starts the producer to send the next request to consumer.write(Frame(request))

getNextTID()
Retrieve the next unique transaction identifier

This handles incrementing the identifier after retrieval

Returns The next unique transaction identifier

getTransaction(*tid*)
Returns a transaction matching the referenced tid

If the transaction does not exist, None is returned

Parameters **tid** – The transaction to retrieve

reset()
Resets the transaction identifier

class pymodbus.transaction.**DictTransactionManager**(*client*, ***kwargs*)
Bases: pymodbus.transaction.ModbusTransactionManager

Impelements a transaction for a manager where the results are keyed based on the supplied transaction id.

Initializes an instance of the ModbusTransactionManager

Parameters **client** – The client socket wrapper

addTransaction(*request*, *tid=None*)
Adds a transaction to the handler

This holds the requets in case it needs to be resent. After being sent, the request is removed.

Parameters

- **request** – The request to hold on to
- **tid** – The overloaded transaction id to use

delTransaction(*tid*)
Removes a transaction matching the referenced tid

Parameters **tid** – The transaction to remove

execute(*request*)
Starts the producer to send the next request to consumer.write(Frame(request))

getNextTID()
Retrieve the next unique transaction identifier

This handles incrementing the identifier after retrieval

Returns The next unique transaction identifier

getTransaction(*tid*)
Returns a transaction matching the referenced tid

If the transaction does not exist, None is returned

Parameters **tid** – The transaction to retrieve

reset()
Resets the transaction identifier

class pymodbus.transaction.**ModbusSocketFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus Socket Frame controller

Before each modbus TCP message is an MBAP header which is used as a message frame. It allows us to easily separate messages as follows:

```
[         MBAP Header         ] [ Function Code] [ Data ]
[ tid ][ pid ][ length ][ uid ]
  2b     2b     2b        1b           1b           Nb

while len(message) > 0:
    tid, pid, length`, uid = struct.unpack(">HHHB", message)
    request = message[0:7 + length - 1`]
    message = [7 + length - 1:]

* length = uid + function code + data
* The -1 is to account for the uid byte
```

Initializes a new instance of the framer

Parameters **decoder** – The decoder factory implementation to use

addToFrame(*message*)
Adds new packet data to the current frame buffer

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

Parameters **message** – The populated request/response to send

checkFrame()
Check and decode the next frame Return true if we were successful

getFrame()
Return the next frame from the buffered data

Returns The next full frame buffer

getRawFrame()
Returns the complete buffer

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder factory know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result with the transport specific header information (pid, tid, uid, checksum, etc)

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

resetFrame()
Reset the entire message frame. This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

class pymodbus.transaction.**ModbusRtuFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus RTU Frame controller:

```
[ Start Wait ] [Address ][ Function Code] [ Data ][ CRC ][  End Wait  ]
  3.5 chars     1b         1b               Nb      2b      3.5 chars
```

Wait refers to the amount of time required to transmist at least x many characters. In this case it is 3.5 characters. Also, if we recieve a wait of 1.5 characters at any point, we must trigger an error message. Also, it appears as though this message is little endian. The logic is simplified as the following:

```
block-on-read:
    read until 3.5 delay
    check for errors
    decode
```

The following table is a listing of the baud wait times for the specified baud rates:

```
------------------------------------------------------------------
 Baud  1.5c (18 bits)   3.5c (38 bits)
------------------------------------------------------------------
 1200   13333.3 us       31666.7 us
 4800    3333.3 us        7916.7 us
 9600    1666.7 us        3958.3 us
19200     833.3 us        1979.2 us
38400     416.7 us         989.6 us
------------------------------------------------------------------
1 Byte = start + 8 bits + parity + stop = 11 bits
(1/Baud)(bits) = delay seconds
```

Initializes a new instance of the framer

Parameters **decoder** – The decoder factory implementation to use

addToFrame(*message*)
This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

Parameters **message** – The populated request/response to send

checkFrame()
Check if the next frame is available. Return True if we were successful.

getFrame()
Get the next frame from the buffer

Returns The frame data or ''

getRawFrame()
Returns the complete buffer

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateHeader()
Try to set the headers *uid*, *len* and *crc*.

This method examines *self.__buffer* and writes meta information into *self.__header*. It calculates only the values for headers that are not already in the dictionary.

Beware that this method will raise an IndexError if *self.__buffer* is not yet long enough.

populateResult(*result*)
Populates the modbus result header

The serial packets do not have any header information that is copied.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

resetFrame()
Reset the entire message frame. This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

class pymodbus.transaction.**ModbusAsciiFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus ASCII Frame Controller:

```
[ Start ][Address ][ Function ][ Data ][ LRC ][ End ]
  1c        2c          2c          Nc     2c      2c

* data can be 0 - 2x252 chars
* end is '\r\n' (Carriage return line feed), however the line feed
  character can be changed via a special command
* start is ':'
```

This framer is used for serial transmission. Unlike the RTU protocol, the data in this framer is transferred in plain text ascii.

Initializes a new instance of the framer

Parameters **decoder** – The decoder implementation to use

addToFrame(*message*)
Add the next message to the frame buffer This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet Built off of a modbus request/response

Parameters **message** – The request/response to send

Returns The encoded packet

checkFrame()
Check and decode the next frame

Returns True if we successful, False otherwise

getFrame()
Get the next frame from the buffer

Returns The frame data or ''

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result header

The serial packets do not have any header information that is copied.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

class pymodbus.transaction.**ModbusBinaryFramer**(*decoder*)

Bases: *pymodbus.interfaces.IModbusFramer*

Modbus Binary Frame Controller:

```
[ Start ][Address ][ Function ][ Data ][ CRC ][ End ]
  1b        1b          1b          Nb     2b     1b

* data can be 0 - 2x252 chars
* end is   '}'
* start is '{'
```

The idea here is that we implement the RTU protocol, however, instead of using timing for message delimiting, we use start and end of message characters (in this case { and }). Basically, this is a binary framer.

The only case we have to watch out for is when a message contains the { or } characters. If we encounter these characters, we simply duplicate them. Hopefully we will not encounter those characters that often and will save a little bit of bandwitch without a real-time system.

Protocol defined by jamod.sourceforge.net.

Initializes a new instance of the framer

Parameters **decoder** – The decoder implementation to use

addToFrame(*message*)

Add the next message to the frame buffer This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()

Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)

Creates a ready to send modbus packet

Parameters **message** – The request/response to send

Returns The encoded packet

checkFrame()

Check and decode the next frame

Returns True if we are successful, False otherwise

getFrame()

Get the next frame from the buffer

Returns The frame data or ''

isFrameReady()

Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result header

The serial packets do not have any header information that is copied.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

class pymodbus.transaction.**DictTransactionManager**(*client*, ***kwargs*)
Bases: pymodbus.transaction.ModbusTransactionManager

Impelements a transaction for a manager where the results are keyed based on the supplied transaction id.

Initializes an instance of the ModbusTransactionManager

Parameters **client** – The client socket wrapper

addTransaction(*request*, *tid=None*)
Adds a transaction to the handler

This holds the requets in case it needs to be resent. After being sent, the request is removed.

Parameters

- **request** – The request to hold on to
- **tid** – The overloaded transaction id to use

delTransaction(*tid*)
Removes a transaction matching the referenced tid

Parameters **tid** – The transaction to remove

execute(*request*)
Starts the producer to send the next request to consumer.write(Frame(request))

getNextTID()
Retrieve the next unique transaction identifier

This handles incrementing the identifier after retrieval

Returns The next unique transaction identifier

getTransaction(*tid*)
Returns a transaction matching the referenced tid

If the transaction does not exist, None is returned

Parameters **tid** – The transaction to retrieve

reset()
Resets the transaction identifier

class pymodbus.transaction.**FifoTransactionManager**(*client*, ***kwargs*)
Bases: pymodbus.transaction.ModbusTransactionManager

Impelements a transaction for a manager where the results are returned in a FIFO manner.

Initializes an instance of the ModbusTransactionManager

Parameters **client** – The client socket wrapper

addTransaction(*request*, *tid=None*)
Adds a transaction to the handler

This holds the requets in case it needs to be resent. After being sent, the request is removed.

Parameters

- **request** – The request to hold on to
- **tid** – The overloaded transaction id to use

delTransaction(*tid*)
Removes a transaction matching the referenced tid

Parameters **tid** – The transaction to remove

execute(*request*)
Starts the producer to send the next request to consumer.write(Frame(request))

getNextTID()
Retrieve the next unique transaction identifier

This handles incrementing the identifier after retrieval

Returns The next unique transaction identifier

getTransaction(*tid*)
Returns a transaction matching the referenced tid

If the transaction does not exist, None is returned

Parameters **tid** – The transaction to retrieve

reset()
Resets the transaction identifier

class pymodbus.transaction.**ModbusSocketFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus Socket Frame controller

Before each modbus TCP message is an MBAP header which is used as a message frame. It allows us to easily separate messages as follows:

```
[         MBAP Header         ] [ Function Code] [ Data ]
[ tid ][ pid ][ length ][ uid ]
  2b     2b     2b        1b           1b           Nb

while len(message) > 0:
    tid, pid, length`, uid = struct.unpack(">HHHB", message)
    request = message[0:7 + length - 1`]
    message = [7 + length - 1:]

* length = uid + function code + data
* The -1 is to account for the uid byte
```

Initializes a new instance of the framer

Parameters **decoder** – The decoder factory implementation to use

addToFrame(*message*)
Adds new packet data to the current frame buffer

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

Parameters **message** – The populated request/response to send

checkFrame()
Check and decode the next frame Return true if we were successful

getFrame()
Return the next frame from the buffered data

Returns The next full frame buffer

getRawFrame()
Returns the complete buffer

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder factory know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result with the transport specific header information (pid, tid, uid, checksum, etc)

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

resetFrame()
Reset the entire message frame. This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

class pymodbus.transaction.**ModbusRtuFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus RTU Frame controller:

```
[ Start Wait ] [Address ][ Function Code] [ Data ][ CRC ][  End Wait  ]
  3.5 chars     1b         1b               Nb      2b      3.5 chars
```

Wait refers to the amount of time required to transmist at least x many characters. In this case it is 3.5 characters. Also, if we recieve a wait of 1.5 characters at any point, we must trigger an error message. Also, it appears as though this message is little endian. The logic is simplified as the following:

```
block-on-read:
    read until 3.5 delay
    check for errors
    decode
```

The following table is a listing of the baud wait times for the specified baud rates:

```
------------------------------------------------------------------
 Baud  1.5c (18 bits)   3.5c (38 bits)
------------------------------------------------------------------
 1200   13333.3 us       31666.7 us
 4800    3333.3 us        7916.7 us
 9600    1666.7 us        3958.3 us
19200     833.3 us        1979.2 us
38400     416.7 us         989.6 us
------------------------------------------------------------------
1 Byte = start + 8 bits + parity + stop = 11 bits
(1/Baud)(bits) = delay seconds
```

Initializes a new instance of the framer

Parameters **decoder** – The decoder factory implementation to use

addToFrame(*message*)
This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

Parameters **message** – The populated request/response to send

checkFrame()
Check if the next frame is available. Return True if we were successful.

getFrame()
Get the next frame from the buffer

Returns The frame data or ‘’

getRawFrame()
Returns the complete buffer

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateHeader()
Try to set the headers *uid*, *len* and *crc*.

This method examines *self.__buffer* and writes meta information into *self.__header*. It calculates only the values for headers that are not already in the dictionary.

Beware that this method will raise an IndexError if *self.__buffer* is not yet long enough.

populateResult(*result*)
Populates the modbus result header

The serial packets do not have any header information that is copied.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

resetFrame()
Reset the entire message frame. This allows us to skip ovver errors that may be in the stream. It is hard to know if we are simply out of sync or if there is an error in the stream as we have no way to check the start or end of the message (python just doesn't have the resolution to check for millisecond delays).

class pymodbus.transaction.**ModbusAsciiFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus ASCII Frame Controller:

```
[ Start ][Address ][ Function ][ Data ][ LRC ][ End ]
  1c        2c          2c         Nc     2c      2c

* data can be 0 - 2x252 chars
* end is '\r\n' (Carriage return line feed), however the line feed
  character can be changed via a special command
* start is ':'
```

This framer is used for serial transmission. Unlike the RTU protocol, the data in this framer is transferred in plain text ascii.

Initializes a new instance of the framer

Parameters **decoder** – The decoder implementation to use

addToFrame(*message*)
Add the next message to the frame buffer This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet Built off of a modbus request/response

Parameters **message** – The request/response to send

Returns The encoded packet

checkFrame()
Check and decode the next frame

Returns True if we successful, False otherwise

getFrame()
Get the next frame from the buffer

Returns The frame data or ''

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result header

The serial packets do not have any header information that is copied.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

class pymodbus.transaction.**ModbusBinaryFramer**(*decoder*)
Bases: *pymodbus.interfaces.IModbusFramer*

Modbus Binary Frame Controller:

```
[ Start ][Address ][ Function ][ Data ][ CRC ][ End ]
  1b        1b          1b          Nb     2b     1b

* data can be 0 - 2x252 chars
* end is   '}'
* start is '{'
```

The idea here is that we implement the RTU protocol, however, instead of using timing for message delimiting, we use start and end of message characters (in this case { and }). Basically, this is a binary framer.

The only case we have to watch out for is when a message contains the { or } characters. If we encounter these characters, we simply duplicate them. Hopefully we will not encounter those characters that often and will save a little bit of bandwitch without a real-time system.

Protocol defined by jamod.sourceforge.net.

Initializes a new instance of the framer

Parameters **decoder** – The decoder implementation to use

addToFrame(*message*)
Add the next message to the frame buffer This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

advanceFrame()
Skip over the current framed message This allows us to skip over the current message after we have processed it or determined that it contains an error. It also has to reset the current frame header handle

buildPacket(*message*)
Creates a ready to send modbus packet

Parameters **message** – The request/response to send

Returns The encoded packet

checkFrame()
Check and decode the next frame

Returns True if we are successful, False otherwise

getFrame()
Get the next frame from the buffer

Returns The frame data or ''

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result header

The serial packets do not have any header information that is copied.

Parameters **result** – The response packet

processIncomingPacket(*data*, *callback*)
The new packet processing pattern

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

- **data** – The new packet data
- **callback** – The function to send results to

utilities — Extra Modbus Helpers

Module author: Galen Collins <bashwork@gmail.com>

Section author: Galen Collins <bashwork@gmail.com>

API Documentation

Modbus Utilities

A collection of utilities for packing data, unpacking data computing checksums, and decode checksums.

pymodbus.utilities.**pack_bitstring**(*bits*)
Creates a string out of an array of bits

Parameters **bits** – A bit array

example:

```
bits   = [False, True, False, True]
result = pack_bitstring(bits)
```

pymodbus.utilities.**unpack_bitstring**(*string*)
Creates bit array out of a string

Parameters **string** – The modbus data packet to decode

example:

```
bytes  = 'bytes to decode'
result = unpack_bitstring(bytes)
```

pymodbus.utilities.**default**(*value*)
Given a python object, return the default value of that object.

Parameters **value** – The value to get the default of

Returns The default value

pymodbus.utilities.**computeCRC**(*data*)
Computes a crc16 on the passed in string. For modbus, this is only used on the binary serial protocols (in this case RTU).

The difference between modbus's crc16 and a normal crc16 is that modbus starts the crc value out at 0xffff.

Parameters **data** – The data to create a crc16 of

Returns The calculated CRC

pymodbus.utilities.**checkCRC**(*data*, *check*)
Checks if the data matches the passed in CRC

Parameters

- **data** – The data to create a crc16 of
- **check** – The CRC to validate

Returns True if matched, False otherwise

pymodbus.utilities.**computeLRC**(*data*)
Used to compute the longitudinal redundancy check against a string. This is only used on the serial ASCII modbus protocol. A full description of this implementation can be found in appendex B of the serial line modbus description.

Parameters **data** – The data to apply a lrc to

Returns The calculated LRC

pymodbus.utilities.**checkLRC**(*data*, *check*)
Checks if the passed in data matches the LRC

Parameters

- **data** – The data to calculate
- **check** – The LRC to validate

Returns True if matched, False otherwise

pymodbus.utilities.**rtuFrameSize**(*data*, *byte_count_pos*)
Calculates the size of the frame based on the byte count.

Parameters

- **data** – The buffer containing the frame.
- **byte_count_pos** – The index of the byte count in the buffer.

Returns The size of the frame.

The structure of frames with a byte count field is always the same:

•first, there are some header fields

•then the byte count field

•then as many data bytes as indicated by the byte count,

•finally the CRC (two bytes).

To calculate the frame size, it is therefore sufficient to extract the contents of the byte count field, add the position of this field, and finally increment the sum by three (one byte for the byte count field, two for the CRC).

pymodbus.utilities.**default**(*value*)
Given a python object, return the default value of that object.

Parameters **value** – The value to get the default of

Returns The default value

pymodbus.utilities.**dict_property**(*store*, *index*)
Helper to create class properties from a dictionary. Basically this allows you to remove a lot of possible boilerplate code.

Parameters

- **store** – The store store to pull from
- **index** – The index into the store to close over

Returns An initialized property set

pymodbus.utilities.**pack_bitstring**(*bits*)
Creates a string out of an array of bits

Parameters **bits** – A bit array

example:

```
bits   = [False, True, False, True]
result = pack_bitstring(bits)
```

pymodbus.utilities.**unpack_bitstring**(*string*)
Creates bit array out of a string

Parameters **string** – The modbus data packet to decode

example:

```
bytes  = 'bytes to decode'
result = unpack_bitstring(bytes)
```

pymodbus.utilities.**__generate_crc16_table**()
Generates a crc16 lookup table

Note: This will only be generated once

pymodbus.utilities.**computeCRC**(*data*)
Computes a crc16 on the passed in string. For modbus, this is only used on the binary serial protocols (in this case RTU).

The difference between modbus's crc16 and a normal crc16 is that modbus starts the crc value out at 0xffff.

Parameters **data** – The data to create a crc16 of

Returns The calculated CRC

pymodbus.utilities.**checkCRC**(*data*, *check*)
Checks if the data matches the passed in CRC

Parameters

- **data** – The data to create a crc16 of
- **check** – The CRC to validate

Returns True if matched, False otherwise

pymodbus.utilities.**computeLRC**(*data*)
Used to compute the longitudinal redundancy check against a string. This is only used on the serial ASCII modbus protocol. A full description of this implementation can be found in appendex B of the serial line modbus description.

Parameters **data** – The data to apply a lrc to

Returns The calculated LRC

pymodbus.utilities.**checkLRC**(*data*, *check*)
Checks if the passed in data matches the LRC

Parameters

- **data** – The data to calculate
- **check** – The LRC to validate

Returns True if matched, False otherwise

pymodbus.utilities.**rtuFrameSize**(*data*, *byte_count_pos*)
Calculates the size of the frame based on the byte count.

Parameters

- **data** – The buffer containing the frame.
- **byte_count_pos** – The index of the byte count in the buffer.

Returns The size of the frame.

The structure of frames with a byte count field is always the same:

•first, there are some header fields

- then the byte count field
- then as many data bytes as indicated by the byte count,
- finally the CRC (two bytes).

To calculate the frame size, it is therefore sufficient to extract the contents of the byte count field, add the position of this field, and finally increment the sum by three (one byte for the byte count field, two for the CRC).

CHAPTER 3

Indices and tables

- genindex
- modindex
- search

Python Module Index

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

Z