Pymodbus Documentation

Release 1.0

Galen Collins

May 19, 2011

CONTENTS

Contents:

CHAPTER

ONE

PYMODBUS LIBRARY EXAMPLES

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

1.1 Example Library Code

1.1.1 Asynchronous Client Example

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

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

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

The following is an example of how to use the asynchronous modbus
client implementation from pymodbus.
'''
#---------------------------------------------------------------------------#
# import the various server implementations
#---------------------------------------------------------------------------#
from pymodbus.client.async import ModbusTcpClient as ModbusClient
#from pymodbus.client.async import ModbusUdpClient as ModbusClient
#from pymodbus.client.async 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.
#---------------------------------------------------------------------------#
client = ModbusClient('127.0.0.1')

#---------------------------------------------------------------------------#
# helper method to test deferred callbacks
#---------------------------------------------------------------------------#
def dassert(deferred, callback):
    def _tester():
        assert(callback())
    deferred.callback(_tester)
    deferred.errback(lambda _: assert(False))

#---------------------------------------------------------------------------#
# example requests
#---------------------------------------------------------------------------#
# simply call the methods that you would like to use. An example session
# is displayed below along with some assert checks.
#---------------------------------------------------------------------------#
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 == [False]*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 == [10]*8)      # test the expected value

rq = client.readwrite_registers(1, [20]*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 == [20]*8)      # test the expected value

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

1.1.2 Asynchronous Server Example

```
#!/usr/bin/env python
#---------------------------------------------------------------------------#
# 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.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext

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

#---------------------------------------------------------------------------#
# 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)

#---------------------------------------------------------------------------#
# run the server you want
#---------------------------------------------------------------------------#
StartTcpServer(context)
#StartUdpServer(context)
#StartSerialServer(context, port='/tmp/tty1')
```

1.1.3 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 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):
        ModbusResponse.__init__(self)
        self.address = address
        self.count = 16

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

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

    def execute(self, context):
        if not (1 <= self.count <= 0x7d0):
            return self.doException(merror.IllegalValue)
        if not context.validate(self.function_code, self.address, self.count):
            return self.doException(merror.IllegalAddress)
        values = context.getValues(self.function_code, self.address, self.count)
        return CustomModbusResponse(values)

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

class Read16CoilsRequest(ReadCoilsRequest):

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

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

```
#---------------------------------------------------------------------------#
# execute the request with your client
#---------------------------------------------------------------------------#
# using the with context, the client will automatically be connected
# and closed when it leaves the current scope.
#---------------------------------------------------------------------------#
with ModbusClient('127.0.0.1') as client:
    request = CustomModbusRequest(0)
    result = client.execute(request)
    print result
```

1.1.4 Modbus Logging Example

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

#---------------------------------------------------------------------------#
# This will simply send everything logged to console
#---------------------------------------------------------------------------#
logging.basicConfig()
log = logging.getLogger()
log.setLevel(logging.DEBUG)

#---------------------------------------------------------------------------#
# This will send the error messages in the specified namespace to a file.
# The available namespaces in pymodbus are as follows:
#---------------------------------------------------------------------------#
# * pymodbus.*          - The root namespace
# * pymodbus.server.*   - all logging messages involving the modbus server
# * pymodbus.client.*   - all logging messages involving the client
# * pymodbus.protocol.* - all logging messages inside the protocol layer
#---------------------------------------------------------------------------#
logging.basicConfig()
log = logging.getLogger('pymodbus.server')
log.setLevel(logging.ERROR)

#---------------------------------------------------------------------------#
# This will send the error messages to the specified handlers:
# * docs.python.org/library/logging.html
#---------------------------------------------------------------------------#
log = logging.getLogger('pymodbus')
log.setLevel(logging.ERROR)
handlers = [
    Handlers.RotatingFileHandler("logfile", maxBytes=1024*1024),
    Handlers.SMTPHandler("mx.host.com", "pymodbus@host.com", ["support@host.com"], "Pymodbus"),
    Handlers.SysLogHandler(facility="daemon"),
    Handlers.DatagramHandler('localhost', 12345),
]
[log.addHandler(h) for h in handlers]
```

1.1.5 Modbus Scraper Example

```
#!/usr/bin/env python
'''
This is a little out of date, give me a second to redo it.
This utility can be used to fully scrape a modbus device
and store its data as a Mobus Context for use with the
simulator.
'''
import pickle
from optparse import OptionParser
from pymodbus.client.sync import ModbusTcpClient

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

#---------------------------------------------------------------------------#
# Helper Classes
#---------------------------------------------------------------------------#
class ClientException(Exception):
    ''' Exception for configuration error '''

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

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

class ClientScraper(object):
    ''' Exception for configuration error '''

    def __init__(self, host, port, address):
        '''
        Initializes the connection paramaters and requests
        @param host The host to connect to
        @param port The port the server resides on
        @param address The range to read to:from
        '''
        self.client = ModbusTcpClient(host=host, port=port)
        self.requests = range(*[int(j) for j in address.split(':')])

    def process(self, data):
        '''
        Starts the device scrape
        '''
        if (self.client.connect())
        f = ModbusClientFactory(self.requests)
        self.p = reactor.connectTCP(self.host, self.port, f)

class ContextBuilder(object):
    '''
    This class is used to build our server datastore
    for use with the modbus simulator.
    '''
```

```
    def __init__(self, output):
        '''
        Initializes the ContextBuilder and checks data values
        @param file The output file for the server context
        '''
        try:
            self.file = open(output, "w")
        except Exception:
            raise ClientException("Unable to open file [%s]" % output)

    def build(self):
        ''' Builds the final output store file '''
        try:
            result = self.makeContext()
            pickle.dump(result, self.file)
            print "Device successfully scraped!"
        except Exception:
            raise ClientException("Invalid data")
        self.file.close()
        reactor.stop()

    def makeContext(self):
        ''' Builds the server context based on the passed in data '''
        # ModbusServerContext(d=sd, c=sc, h=sh, i=si)
        return "string"

#---------------------------------------------------------------------------#
# Main start point
#---------------------------------------------------------------------------#
def main():
    ''' Server launcher '''
    parser = OptionParser()
    parser.add_option("-o", "--output",
        help="The resulting output file for the scrape",
        dest="file", default="output.store")
    parser.add_option("-p", "--port",
        help="The port to connect to",
        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="range", default="0:500")
    parser.add_option("-D", "--debug",
        help="Enable debug tracing",
        action="store_true", dest="debug", default=False)
    (opt, arg) = parser.parse_args()

    # enable debugging information
    if opt.debug:
        try:
            client_log.setLevel(logging.DEBUG)
                logging.basicConfig()
        except Exception, e:
                print "Logging is not supported on this system"

    # Begin scraping
```

```
    try:
        #ctx = ContextBuilder(opt.file)
        s = ClientScraper(opt.host, opt.port, opt.range)
        reactor.callWhenRunning(s.start)
        reactor.run()
    except ClientException, err:
        print err
        parser.print_help()

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

1.1.6 Modbus Simulator Example

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

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

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

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

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

#---------------------------------------------------------------------------#
# Extra Global Functions
#---------------------------------------------------------------------------#
# These are extra helper functions that don't belong in a class
#---------------------------------------------------------------------------#
import getpass
def root_test():
    ''' Simple test to see if we are running as root '''
    return True # removed for the time being as it isn't portable
    #return getpass.getuser() == "root"

#---------------------------------------------------------------------------#
# Helper Classes
#---------------------------------------------------------------------------#
class ConfigurationException(Exception):
    ''' Exception for configuration error '''

    def __init__(self, string):
```

```
        ''' Initializes the ConfigurationException instance

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

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

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

class Configuration:
    '''
    Class used to parse configuration file and create and modbus
    datastore.

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

    def __init__(self, config):
        '''
        Trys to load a configuration file, lets the file not
        found exception fall through

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

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

#---------------------------------------------------------------------------#
# Main start point
#---------------------------------------------------------------------------#
def main():
    ''' Server launcher '''
    parser = OptionParser()
    parser.add_option("-c", "--conf",
                      help="The configuration file to load",
```

```
                    dest="file")
    parser.add_option("-D", "--debug",
                    help="Turn on to enable tracing",
                    action="store_true", dest="debug", default=False)
    (opt, arg) = parser.parse_args()

    # enable debugging information
    if opt.debug:
        try:
            server_log.setLevel(logging.DEBUG)
            protocol_log.setLevel(logging.DEBUG)
        except Exception, e:
                print "Logging is not supported on this system"

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

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

1.1.7 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.
#---------------------------------------------------------------------------#
client = ModbusClient('127.0.0.1')

#---------------------------------------------------------------------------#
# example requests
#---------------------------------------------------------------------------#
# simply call the methods that you would like to use. An example session
# is displayed below along with some assert checks.
#---------------------------------------------------------------------------#
rq = client.write_coil(1, True)
rr = client.read_coils(1,1)
assert(rq.function_code < 0x80)     # test that we are not an error
assert(rr.bits[0] == True)          # test the expected value

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

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

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

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

rq = client.readwrite_registers(1, [20]*8)
rr = client.read_input_registers(1,8)
assert(rq.function_code < 0x80)     # test that we are not an error
```

```
assert(rr.registers == [20]*8)      # test the expected value

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

1.1.8 Synchronous Server Example

```
#!/usr/bin/env python
#---------------------------------------------------------------------------#
# 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.datastore import ModbusSequentialDataBlock
from pymodbus.datastore import ModbusSlaveContext, ModbusServerContext

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

#---------------------------------------------------------------------------#
# 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)

#---------------------------------------------------------------------------#
# run the server you want
#---------------------------------------------------------------------------#
StartTcpServer(context)
#StartUdpServer(context)
#StartSerialServer(context, port='/tmp/tty1')
```

1.1.9 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
#---------------------------------------------------------------------------#
from pymodbus.client.sync import ModbusTcpClient
from time import time

#---------------------------------------------------------------------------#
# initialize the test
#---------------------------------------------------------------------------#
client = ModbusTcpClient('127.0.0.1')
count  = 0
start  = time()
iterations = 10000

#---------------------------------------------------------------------------#
# perform the test
#---------------------------------------------------------------------------#
while count < iterations:
    result = client.read_holding_registers(10, 1, 0).getRegister(0)
    count += 1

#---------------------------------------------------------------------------#
# check our results
#---------------------------------------------------------------------------#
stop = time()
print "%d requests/second" % ((1.0 * count) / (stop - start))
```

1.2 Example Frontend Code

1.2.1 Glade/GTK Frontend Example

Main Program

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

Note: The virtual networking code will only work on linux

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

#---------------------------------------------------------------------------#
# For Gui
#---------------------------------------------------------------------------#
from twisted.internet import gtk2reactor
```

```
gtk2reactor.install()
import gtk
from gtk import glade

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

#---------------------------------------------------------------------------#
# Logging
#---------------------------------------------------------------------------#
import logging
log = logging.getLogger(__name__)

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

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

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

#---------------------------------------------------------------------------#
# Extra Global Functions
#---------------------------------------------------------------------------#
# These are extra helper functions that don't belong in a class
#---------------------------------------------------------------------------#
def root_test():
    ''' Simple test to see if we are running as root '''
    return getpass.getuser() == "root"

#---------------------------------------------------------------------------#
# Simulator Class
#---------------------------------------------------------------------------#
class Simulator(object):
    '''
    Class used to parse configuration file and create and modbus
    datastore.

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

    def __init__(self, config):
        '''
        Trys to load a configuration file, lets the file not
        found exception fall through
```

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

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

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

    def run(self):
        ''' Used to run the simulator '''
        reactor.callWhenRunning(self._simulator)

#---------------------------------------------------------------------------#
# Network reset thread
#---------------------------------------------------------------------------#
# This is linux only, maybe I should make a base class that can be filled
# in for linux(debian/redhat)/windows/nix
#---------------------------------------------------------------------------#
class NetworkReset(Thread):
    '''
    This class is simply a daemon that is spun off at the end of the
    program to call the network restart function (an easy way to
    remove all the virtual interfaces)
    '''
    def __init__(self):
        Thread.__init__(self)
        self.setDaemon(True)

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

#---------------------------------------------------------------------------#
# Main Gui Class
#---------------------------------------------------------------------------#
```

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

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

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

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

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

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

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

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

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

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

```
            self.error_dialog("Invalid starting address!");
            return False

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

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

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

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

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

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

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

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

Glade Layout File

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

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE glade-interface SYSTEM "glade-2.0.dtd">
<!--Generated with glade3 3.4.0 on Thu Nov 20 10:51:52 2008 -->
<glade-interface>
  <widget class="GtkWindow" id="window">
    <property name="visible">True</property>
    <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_PRESS
    <property name="title" translatable="yes">Modbus Simulator</property>
    <property name="resizable">False</property>
    <property name="window_position">GTK_WIN_POS_CENTER</property>
    <signal name="destroy" handler="on_window_destroy"/>
    <child>
      <widget class="GtkVBox" id="vbox1">
        <property name="width_request">400</property>
        <property name="height_request">200</property>
        <property name="visible">True</property>
        <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTTON_P
        <child>
          <widget class="GtkHBox" id="hbox1">
            <property name="visible">True</property>
            <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_BUTT
            <child>
              <widget class="GtkLabel" id="label1">
                <property name="visible">True</property>
                <property name="events">GDK_POINTER_MOTION_MASK | GDK_POINTER_MOTION_HINT_MASK | GDK_
                <property name="label" translatable="yes">Device to Simulate</property>
              </widget>
            </child>
            <child>
              <widget class="GtkHButtonBox" id="hbuttonbox2">
                <property name="visible">True</property>
```

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

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

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

1.2.2 TK Frontend Example

Main Program

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

Note: The virtual networking code will only work on linux

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

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

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

```
#---------------------------------------------------------------------------#
# Logging
#---------------------------------------------------------------------------#
import logging
log = logging.getLogger(__name__)

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

#---------------------------------------------------------------------------#
# Extra Global Functions
#---------------------------------------------------------------------------#
# These are extra helper functions that don't belong in a class
#---------------------------------------------------------------------------#
def root_test():
    ''' Simple test to see if we are running as root '''
    return getpass.getuser() == "root"

#---------------------------------------------------------------------------#
# Simulator Class
#---------------------------------------------------------------------------#
class Simulator(object):
    '''
    Class used to parse configuration file and create and modbus
    datastore.

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

    def __init__(self, config):
        '''
        Trys to load a configuration file, lets the file not
        found exception fall through

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

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

```
        return ModbusServerContext(slaves=slave)

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

    def run(self):
        ''' Used to run the simulator '''
        reactor.callWhenRunning(self._simulator)

#---------------------------------------------------------------------------#
# Network reset thread
#---------------------------------------------------------------------------#
# This is linux only, maybe I should make a base class that can be filled
# in for linux(debian/redhat)/windows/nix
#---------------------------------------------------------------------------#
class NetworkReset(Thread):
    '''
    This class is simply a daemon that is spun off at the end of the
    program to call the network restart function (an easy way to
    remove all the virtual interfaces)
    '''
    def __init__(self):
        Thread.__init__(self)
        self.setDaemon(True)

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

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

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

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

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

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

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

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

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

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

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

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

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

#------------------------------------------------------------------------#
# Gui helpers
#------------------------------------------------------------------------#
# Not callbacks, but used by them
#------------------------------------------------------------------------#
    def show_buttons(self, state=False):
        ''' Greys out the buttons '''
        state = 'active' if state else 'disabled'
        for widget in self._widgets:
            widget.configure(state=state)
```

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

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

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

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

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

        # check input file
        filename = self.tdevice_value.get()
        if os.path.exists(filename):
            self.show_buttons(state=False)
```

```
            try:
                handle = Simulator(config=filename)
                handle.run()
            except ConfigurationException, ex:
                self.error_dialog("Error %s" % ex)
                self.show_buttons(state=True)
        else:
            self.error_dialog("Device to emulate does not exist!");
            return False

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

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

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

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

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

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

#---------------------------------------------------------------------------#
# Main handle function
#---------------------------------------------------------------------------#
# This is called when the application is run from a console
# We simply start the gui and start the twisted event loop
#---------------------------------------------------------------------------#
def main():
    '''
    Main control function
    This either launches the gui or runs the command line application
    '''
```

```
    debug = True
    if debug:
        try:
            log.setLevel(logging.DEBUG)
                logging.basicConfig()
        except Exception, e:
                print "Logging is not supported on this system"
    simulator = SimulatorApp(root)
    root.title("Modbus Simulator")
    reactor.run()

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

1.2.3 WX Frontend Example

Main Program

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

Note: The virtual networking code will only work on linux

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

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

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

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

```
# Logging
#---------------------------------------------------------------------------#
import logging
log = logging.getLogger(__name__)

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

#---------------------------------------------------------------------------#
# Extra Global Functions
#---------------------------------------------------------------------------#
# These are extra helper functions that don't belong in a class
#---------------------------------------------------------------------------#
def root_test():
    ''' Simple test to see if we are running as root '''
    return getpass.getuser() == "root"

#---------------------------------------------------------------------------#
# Simulator Class
#---------------------------------------------------------------------------#
class Simulator(object):
    '''
    Class used to parse configuration file and create and modbus
    datastore.

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

    def __init__(self, config):
        '''
        Trys to load a configuration file, lets the file not
        found exception fall through

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

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

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

    def run(self):
        ''' Used to run the simulator '''
        reactor.callWhenRunning(self._simulator)

#---------------------------------------------------------------------------#
# Network reset thread
#---------------------------------------------------------------------------#
# This is linux only, maybe I should make a base class that can be filled
# in for linux(debian/redhat)/windows/nix
#---------------------------------------------------------------------------#
class NetworkReset(Thread):
    '''
    This class is simply a daemon that is spun off at the end of the
    program to call the network restart function (an easy way to
    remove all the virtual interfaces)
    '''
    def __init__(self):
        ''' Initializes a new instance of the network reset thread '''
        Thread.__init__(self)
        self.setDaemon(True)

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

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

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

        #-----------------------------------------------------------------------#
        # Add button row
        #-----------------------------------------------------------------------#
        panel = wx.Panel(self, -1)
```

```
        box = wx.BoxSizer(wx.HORIZONTAL)
        box.Add(wx.Button(panel, 1, 'Apply'), 1)
        box.Add(wx.Button(panel, 2, 'Help'),  1)
        box.Add(wx.Button(panel, 3, 'Close'), 1)
        panel.SetSizer(box)

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

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

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

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

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

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

#---------------------------------------------------------------------------#
# Button Actions
#---------------------------------------------------------------------------#
# These are all callbacks for the various buttons
#---------------------------------------------------------------------------#
    def start_clicked(self, widget):
```

```
        ''' Starts the simulator '''
        start = 1
        base = "172.16"

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

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

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

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

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

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

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

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

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

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

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

1.2.4 Web Frontend Example

```
'''
Pymodbus Web Frontend
======================================
```

```
This is a simple web frontend using bottle as the web framework.
This can be hosted using any wsgi adapter.
'''
from bottle import route, request, Bottle
from bottle import jinja2_template as template
from pymodbus.device import ModbusAccessControl
from pymodbus.device import ModbusControlBlock

#---------------------------------------------------------------------------#
# REST API
#---------------------------------------------------------------------------#
class Response(object):
    '''
    A collection of common responses for the frontend api
    '''
    successful = { 'status' : 200 }
    failure    = { 'status' : 500 }

class ModbusApiWebApp(object):
    '''
    This is the web REST api interace into the pymodbus
    service.  It can be consumed by any utility that can
    make web requests (javascript).
    '''
    _namespace = '/api/v1'

    def __init__(self, server):
        ''' Initialize a new instance of the ModbusApi

        :param server: The current server instance
        '''
        self._server = server

    #---------------------------------------------------------------------------#
    # Device API
    #---------------------------------------------------------------------------#
    def get_device(self):
        return {
            'mode'        : self._server.control.Mode,
            'delimiter'   : self._server.control.Delimiter,
            'listen-only' : self._server.control.ListenOnly,
            'identity'    : self._server.control.Identity.summary(),
            'counters'    : dict(self._server.control.Counter),
            'diagnostic'  : self._server.control.getDiagnosticRegister(),
        }

    def get_device_identity(self):
        return {
            'identity' : dict(self._server.control.Identity)
        }

    def get_device_events(self):
        return {
            'events' : self._server.control.Events
        }

    def delete_device_events(self):
        self._server.control.clearEvents()
```

```
        return Response.successful

    def get_device_host(self):
        return {
            'hosts' : list(self._server.access)
        }

    def post_device_host(self):
        value = request.forms.get('host')
        if value:
            self._server.access.add(value)
        return Response.successful

    def delete_device_host(self):
        value = request.forms.get('host')
        if value:
            self._server.access.remove(value)
        return Response.successful

    def post_device_delimiter(self):
        value = request.forms.get('delimiter')
        if value:
            self._server.control.Delimiter = value
        return Response.successful

    def post_device_mode(self):
        value = request.forms.get('mode')
        if value:
            self._server.control.Mode = value
        return Response.successful

    def post_device_reset(self):
        self._server.control.reset()
        return Response.successful

    #---------------------------------------------------------------------#
    # Datastore API
    #---------------------------------------------------------------------#

#---------------------------------------------------------------------------#
# Configurations
#---------------------------------------------------------------------------#
def register_routes(application, register):
    ''' A helper method to register the routes of an application
    based on convention.

    :param application: The application instance to register
    :param register: The bottle instance to register the application with
    '''
    from bottle import route

    methods = dir(application)
    methods = filter(lambda n: not n.startswith('_'), methods)
    for method in methods:
        pieces = method.split('_')
        verb, path = pieces[0], pieces[1:]
        path.insert(0, application._namespace)
        path = '/'.join(path)
```

```
            func = getattr(application, method)
            register.route(path, method=verb, name=method)(func)

def build_application(server):
    ''' Helper method to create and initiailze a bottle application

    :param server: The modbus server to pull instance data from
    :returns: An initialied bottle application
    '''
    api = ModbusApiWebApp(server)
    register = Bottle()
    register_routes(api, register)
    return register

#---------------------------------------------------------------------------#
# Start Methods
#---------------------------------------------------------------------------#
def RunModbusFrontend(server, port=503):
    ''' Helper method to host bottle in twisted

    :param server: The modbus server to pull instance data from
    :param port: The port to host the service on
    '''
    from bottle import TwistedServer, run
    application = build_application(server)
    run(app=application, server=TwistedServer, port=port)

def RunDebugModbusFrontend(server, port=503):
    ''' Helper method to start the bottle server

    :param server: The modbus server to pull instance data from
    :param port: The port to host the service on
    '''
    from bottle import run

    application = build_application(server)
    run(app=application, port=port)

if __name__ == '__main__':
    from pymodbus.server.async import ModbusServerFactory

    RunDebugModbusFrontend(ModbusServerFactory)
```

CHAPTER

TWO

PYMODBUS LIBRARY API DOCUMENTATION

The following are the API documentation strings taken from the sourcecode

2.1 `bit_read_message` — Bit Read Modbus Messages

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

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

2.1.1 API Documentation

Bit Reading Request/Response messages

class `pymodbus.bit_read_message.`**`ReadBitsRequestBase`**(*address*, *count*, ***kwargs*)

Base class for Messages Requesting bit values

`decode`(*data*)

Decodes a request pdu

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

`encode`()

Encodes a request pdu

Returns The encoded pdu

class `pymodbus.bit_read_message.`**`ReadBitsResponseBase`**(*values*, ***kwargs*)

Base class for Messages responding to bit-reading values

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

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

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

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

Returns The initializes response message, exception message otherwise

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

class pymodbus.bit_read_message.**ReadDiscreteInputsRequest**(*address=None*, *count=None*, ***kwargs*)
This function code is used to read from 1 to 2000(0x7d0) contiguous status of discrete inputs in a remote device. The Request PDU specifies the starting address, ie the address of the first input specified, and the number of inputs. In the PDU Discrete Inputs are addressed starting at zero. Therefore Discrete inputs numbered 1-16 are addressed as 0-15.

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

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

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

Returns The initializes response message, exception message otherwise

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

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.

2.2 bit_write_message — Bit Write Modbus Messages

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

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

2.2.1 API Documentation

Bit Writing Request/Response

TODO write mask request/response

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

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

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

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

decode(*data*)

Decodes a write coil request

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

encode()

Encodes write coil request

Returns The byte encoded message

execute(*context*)

Run a write coil request against a datastore

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

Returns The populated response or exception message

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

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

decode(*data*)

Decodes a write coil response

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

encode()

Encodes write coil response

Returns The byte encoded message

class pymodbus.bit_write_message.**WriteMultipleCoilsRequest**(*address=None*, *values=None*, ***kwargs*)

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

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

decode(*data*)

Decodes a write coils request

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

encode()

Encodes write coils request

Returns The byte encoded message

execute(*context*)

Run a write coils request against a datastore

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

Returns The populated response or exception message

class pymodbus.bit_write_message.**WriteMultipleCoilsResponse**(*address=None*, *count=None*, ***kwargs*)

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

decode(*data*)

Decodes a write coils response

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

encode()

Encodes write coils response

Returns The byte encoded message

2.3 client.common — Twisted Async Modbus Client

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

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

2.3.1 API Documentation

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

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)
```

read_coils(*address*, *count=1*, *unit=0*)

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*, *unit=0*)

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*, *unit=0*)

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*, *unit=0*)

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

Returns A deferred response handle

write_coil(*address*, *value*, *unit=0*)

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*, *unit=0*)

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*, *unit=0*)

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*, *unit=0*)

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

2.4 client.sync — Twisted Synchronous Modbus Client

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

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

2.4.1 API Documentation

class pymodbus.client.sync.**ModbusTransactionManager**(*client*)
This is a simply pull producer that feeds requests to the modbus client

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

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

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

class pymodbus.client.sync.**ModbusTcpClient**(*host='127.0.0.1'*, *port=502*)
Implementation of a modbus tcp client

close()
Closes the underlying socket connection

connect()
Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

class pymodbus.client.sync.**ModbusUdpClient**(*host='127.0.0.1'*, *port=502*)
Implementation of a modbus udp client

close()
Closes the underlying socket connection

connect()
Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

class pymodbus.client.sync.**ModbusSerialClient**(*method='ascii'*, ***kwargs*)
Implementation of a modbus udp client

close()
Closes the underlying socket connection

connect()
Connect to the modbus tcp server

Returns True if connection succeeded, False otherwise

2.5 client.async — Twisted Async Modbus Client

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

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

2.5.1 API Documentation

Implementation of a Modbus Client Using Twisted

Example Run:

```
from pymodbus.client.async import ModbusClientFactory
from pymodbus.bit_read_message import ReadCoilsRequest

def clientTest():
    requests = [ ReadCoilsRequest(0,99) ]
    p = reactor.connectTCP("localhost", 502, ModbusClientFactory(requests))

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

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

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

connectionLost(*reason*)

Called upon a client disconnect

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

connectionMade()

Called upon a successful client connection.

dataReceived(*data*)

Get response, check for valid message, decode result

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

execute(*request*)

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

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

Simple client protocol factory

protocol

alias of ModbusClientProtocol

2.6 constants — Modbus Default Values

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

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

2.6.1 API Documentation

Constants For Modbus Server/Client

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

class pymodbus.constants.**Defaults**
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)

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.

class pymodbus.constants.**ModbusStatus**
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

2.7 Server Datastores and Contexts

The following are the API documentation strings taken from the sourcecode

2.7.1 `store` — Datastore for Modbus Server Context

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

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

API Documentation

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**
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*)
Creates a sequential modbus datastore

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

Parameters

- **address** – The starting address

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

Returns The requested values from a:a+c

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

Parameters

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

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

Parameters

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

Returns True if the request in within range, False otherwise

class pymodbus.datastore.store.**ModbusSparseDataBlock**(*values*)
Creates a sparse modbus datastore

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

Parameters

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

Returns The requested values from a:a+c

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

Parameters

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

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

Parameters

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

Returns True if the request in within range, False otherwise

2.7.2 `context` — Modbus Server Contexts

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

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

API Documentation

class pymodbus.datastore.context.**ModbusSlaveContext**(**args*, ***kwargs*)
This creates a modbus data model with each data access stored in its own personal block

getValues(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

class pymodbus.datastore.context.**ModbusServerContext**(*slaves=None*, *single=True*)
This represents a master collection of slave contexts. If single is set to true, it will be treated as a single context so every unit-id returns the same context. If single is set to false, it will be interpreted as a collection of slave contexts.

2.7.3 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*)

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

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

Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset()

Resets all the datastores to their default values

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

Sets the datastore with the supplied values

Parameters

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

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

Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

2.7.4 database — Database Slave Context

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

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

API Documentation

class pymodbus.datastore.database.**DatabaseSlaveContext**(**args*, ***kwargs*)
This creates a modbus data model with each data access stored in its own personal block

getValues(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

2.7.5 modredis — Redis Slave Context

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

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

API Documentation

class pymodbus.datastore.modredis.**RedisSlaveContext**(***kwargs*)
This is a modbus slave context using redis as a backing store.

getValues(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns The requested values from a:a+c

reset()
Resets all the datastores to their default values

setValues(*fx*, *address*, *values*)
Sets the datastore with the supplied values

Parameters

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

validate(*fx*, *address*, *count=1*)
Validates the request to make sure it is in range

Parameters

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

Returns True if the request in within range, False otherwise

2.8 diag_message — Diagnostic Modbus Messages

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

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

2.8.1 API Documentation

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

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

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

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

Returns The encoded packet

class pymodbus.diag_message.**DiagnosticStatusResponse**
This is a base class for all of the diagnostic response functions

It works by performing all of the encoding and decoding of variable data and lets the higher classes define what extra data to append and how to execute a request

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

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

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

Returns The encoded packet

class pymodbus.diag_message.**DiagnosticStatusSimpleRequest**(*data=0*)
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

execute(**args*)
Base function to raise if not implemented

class pymodbus.diag_message.**DiagnosticStatusSimpleResponse**(*data*)
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.

class pymodbus.diag_message.**ReturnQueryDataRequest**(*message=0*)
The data passed in the request data field is to be returned (looped back) in the response. The entire response message should be identical to the request.

execute(**args*)
Executes the loopback request (builds the response)

Returns The populated loopback response message

class pymodbus.diag_message.**ReturnQueryDataResponse**(*message=0*)
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.

class pymodbus.diag_message.**RestartCommunicationsOptionRequest**(*toggle=False*)
The remote device serial line port must be initialized and restarted, and all of its communications event counters are cleared. If the port is currently in Listen Only Mode, no response is returned. This function is the only one that brings the port out of Listen Only Mode. If the port is not currently in Listen Only Mode, a normal response is returned. This occurs before the restart is executed.

execute(**args*)
Clear event log and restart

Returns The initialized response message

class pymodbus.diag_message.**RestartCommunicationsOptionResponse**(*toggle=False*)
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.

class pymodbus.diag_message.**ReturnDiagnosticRegisterRequest**(*data=0*)
The contents of the remote device's 16-bit diagnostic register are returned in the response

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnDiagnosticRegisterResponse**(*data*)
The contents of the remote device's 16-bit diagnostic register are returned in the response

class pymodbus.diag_message.**ChangeAsciiInputDelimiterRequest**(*data=0*)
The character 'CHAR' passed in the request data field becomes the end of message delimiter for future messages (replacing the default LF character). This function is useful in cases of a Line Feed is not required at the end of ASCII messages.

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

Returns The initialized response message

class pymodbus.diag_message.**ChangeAsciiInputDelimiterResponse**(*data*)
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.

class pymodbus.diag_message.**ForceListenOnlyModeRequest**(*data=0*)
Forces the addressed remote device to its Listen Only Mode for MODBUS communications. This isolates it from the other devices on the network, allowing them to continue communicating without interruption from the addressed remote device. No response is returned.

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

Returns The initialized response message

class pymodbus.diag_message.**ForceListenOnlyModeResponse**
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

class pymodbus.diag_message.**ClearCountersRequest**(*data=0*)
The goal is to clear ll counters and the diagnostic register. Also, counters are cleared upon power-up

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

Returns The initialized response message

class pymodbus.diag_message.**ClearCountersResponse**(*data*)
The goal is to clear ll counters and the diagnostic register. Also, counters are cleared upon power-up

class pymodbus.diag_message.**ReturnBusMessageCountRequest**(*data=0*)
The response data field returns the quantity of messages that the remote device has detected on the communications systems since its last restart, clear counters operation, or power-up

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnBusMessageCountResponse**(*data*)
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

class pymodbus.diag_message.**ReturnBusCommunicationErrorCountRequest**(*data=0*)
The response data field returns the quantity of CRC errors encountered by the remote device since its last restart, clear counter operation, or power-up

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnBusCommunicationErrorCountResponse**(*data*)
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

class pymodbus.diag_message.**ReturnBusExceptionErrorCountRequest**(*data=0*)
The response data field returns the quantity of modbus exception responses returned by the remote device since its last restart, clear counters operation, or power-up

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnBusExceptionErrorCountResponse**(*data*)
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

class pymodbus.diag_message.**ReturnSlaveMessageCountRequest**(*data=0*)
The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnSlaveMessageCountResponse**(*data*)
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

class pymodbus.diag_message.**ReturnSlaveNoResponseCountRequest**(*data=0*)
The response data field returns the quantity of messages addressed to the remote device, or broadcast, that the remote device has processed since its last restart, clear counters operation, or power-up

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnSlaveNoReponseCountResponse**(*data*)
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

class pymodbus.diag_message.**ReturnSlaveNAKCountRequest**(*data=0*)
The response data field returns the quantity of messages addressed to the remote device for which it returned a Negative Acknowledge (NAK) exception response, since its last restart, clear counters operation, or power-up. Exception responses are described and listed in section 7 .

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnSlaveNAKCountResponse**(*data*)
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.

class pymodbus.diag_message.**ReturnSlaveBusyCountRequest**(*data=0*)
The response data field returns the quantity of messages addressed to the remote device for which it returned a Slave Device Busy exception response, since its last restart, clear counters operation, or power-up.

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnSlaveBusyCountResponse**(*data*)
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.

class pymodbus.diag_message.**ReturnSlaveBusCharacterOverrunCountRequest**(*data=0*)
The response data field returns the quantity of messages addressed to the remote device that it could not handle due to a character overrun condition, since its last restart, clear counters operation, or power-up. A character overrun is caused by data characters arriving at the port faster than they can be stored, or by the loss of a character due to a hardware malfunction.

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

Returns The initialized response message

class pymodbus.diag_message.**ReturnSlaveBusCharacterOverrunCountResponse**(*data*)
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.

class pymodbus.diag_message.**ClearOverrunCountRequest**(*data=0*)
Clears the overrun error counter and reset the error flag

An error flag should be cleared, but nothing else in the specification mentions is, so it is ignored.

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

Returns The initialized response message

class pymodbus.diag_message.**ClearOverrunCountResponse**(*data*)
Clears the overrun error counter and reset the error flag

2.9 device — Modbus Device Representation

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

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

2.9.1 API Documentation

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

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

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

Parameters **host** – The host to add

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

Parameters **host** – The host to check

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

Parameters **host** – The host to remove

class pymodbus.device.**ModbusDeviceIdentification**(*info=None*)
This is used to supply the device identification for the readDeviceIdentification function

For more information read section 6.21 of the modbus application protocol.

summary()
Return a summary of the main items

Returns An dictionary of the main items

update(*input*)
Update the values of this identity using another identify as the value

Parameters **input** – The value to copy values from

class pymodbus.device.**ModbusControlBlock**
This is a global singleotn that controls all system information

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

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

2.10 factory — Request/Response Decoders

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

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

2.10.1 API Documentation

Modbus Request/Response Decoder Factories

class pymodbus.factory.**ServerDecoder**
Request Message Factory (Server)

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

decode(*message*)
Wrapper to decode a request packet

Parameters **message** – The raw modbus request packet

Returns The decoded modbus message or None if error

f
alias of `ReadWriteMultipleRegistersRequest`

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**
Response Message Factory (Client)

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

decode(*message*)
Wrapper to decode a response packet

Parameters **message** – The raw packet to decode

Returns The decoded modbus message or None if error

f
alias of `ReadWriteMultipleRegistersResponse`

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*.

2.11 `interfaces` — System Interfaces

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

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

2.11.1 API Documentation

Pymodbus Interfaces

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

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

class `pymodbus.interfaces.`**IModbusDecoder**
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**
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**
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

2.12 exceptions — Exceptions Used in PyModbus

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

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

2.12.1 API Documentation

Pymodbus Exceptions

Custom exceptions to be used in the Modbus code.

class pymodbus.exceptions.**ModbusException** (*string*)
Base modbus exception

class pymodbus.exceptions.**ModbusIOException** (*string=''*)
Error resulting from data i/o

class pymodbus.exceptions.**ParameterException** (*string=''*)
Error resulting from invalid paramater

class pymodbus.exceptions.**NotImplementedException** (*string=''*)
Error resulting from not implemented function

2.13 other_message — Other Modbus Messages

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

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

2.13.1 API Documentation

Diagnostic record read/write

Currently not all implemented

class pymodbus.other_message.**ReadExceptionStatusRequest**
This function code is used to read the contents of eight Exception Status outputs in a remote device. The function provides a simple method for accessing this information, because the Exception Output references are known (no output reference is needed in the function).

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**ReadExceptionStatusResponse**(*status*)
The normal response contains the status of the eight Exception Status outputs. The outputs are packed into one data byte, with one bit per output. The status of the lowest output reference is contained in the least significant bit of the byte. The contents of the eight Exception Status outputs are device specific.

decode(*data*)
Decodes a the response

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

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**GetCommEventCounterRequest**
This function code is used to get a status word and an event count from the remote device's communication event counter.

By fetching the current count before and after a series of messages, a client can determine whether the messages were handled normally by the remote device.

The device's event counter is incremented once for each successful message completion. It is not incremented for exception responses, poll commands, or fetch event counter commands.

The event counter can be reset by means of the Diagnostics function (code 08), with a subfunction of Restart Communications Option (code 00 01) or Clear Counters and Diagnostic Register (code 00 0A).

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**GetCommEventCounterResponse**(*count*)
The normal response contains a two-byte status word, and a two-byte event count. The status word will be all

ones (FF FF hex) if a previously-issued program command is still being processed by the remote device (a busy condition exists). Otherwise, the status word will be all zeros.

decode(*data*)
Decodes a the response

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

encode()
Encodes the response

Returns The byte encoded message

class pymodbus.other_message.**ReportSlaveIdRequest**
This function code is used to read the description of the type, the current status, and other information specific to a remote device.

decode(*data*)
Decodes data part of the message.

Parameters **data** – The incoming data

encode()
Encodes the message

execute()
Run a read exeception status request against the store

Returns The populated response

class pymodbus.other_message.**ReportSlaveIdResponse**(*identifier*, *status=True*)
The format of a normal response is shown in the following example. The data contents are specific to each type of device.

decode(*data*)
Decodes a the response

Since the identifier is device dependent, we just return the raw value that a user can decode to whatever it should be.

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

encode()
Encodes the response

Returns The byte encoded message

2.14 file_message — File Modbus Messages

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

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

2.14.1 API Documentation

File Record Read/Write Messages

Currently none of these messages are implemented

class pymodbus.file_message.**ReadFifoQueueRequest**(*address*)
This function code allows to read the contents of a First-In-First-Out (FIFO) queue of register in a remote device. The function returns a count of the registers in the queue, followed by the queued data. Up to 32 registers can be read: the count, plus up to 31 queued data registers.

The queue count register is returned first, followed by the queued data registers. The function reads the queue contents, but does not clear them.

decode(*data*)
Decodes the incoming request

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

encode()
Encodes the request packet

Returns The byte encoded packet

execute(*context*)
Run a read exeception status request against the store

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

Returns The populated response

class pymodbus.file_message.**ReadFifoQueueResponse**(*values*)
In a normal response, the byte count shows the quantity of bytes to follow, including the queue count bytes and value register bytes (but not including the error check field). The queue count is the quantity of data registers in the queue (not including the count register).

If the queue count exceeds 31, an exception response is returned with an error code of 03 (Illegal Data Value).

classmethod **calculateRtuFrameSize**(*buffer*)
Calculates the size of a response containing a FIFO queue.

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

2.15 events — Events Used in PyModbus

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

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

2.15.1 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.**ModbusEvent**

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*)
Remote device MODBUS Receive Event

The remote device stores this type of event byte when a query message is received. It is stored before the remote device processes the message. This event is defined by bit 7 set to logic '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
---------------------------------
0   Not Used
2   Not Used
3   Not Used
4   Character Overrun
5   Currently in Listen Only Mode
6   Broadcast Receive
7   1
```

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

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**RemoteSendEvent**(**kwargs)
Remote device MODBUS Send Event

The remote device stores this type of event byte when it finishes processing a request message. It is stored if the remote device returned a normal or exception response, or no response.

This event is defined by bit 7 set to a logic '0', with bit 6 set to a '1'. The other bits will be set to a logic '1' if the corresponding condition is TRUE. The bit layout is:

```
Bit Contents
--------------------------------------------------------
0   Read Exception Sent (Exception Codes 1-3)
1   Slave Abort Exception Sent (Exception Code 4)
2   Slave Busy Exception Sent (Exception Codes 5-6)
3   Slave Program NAK Exception Sent (Exception Code 7)
4   Write Timeout Error Occurred
5   Currently in Listen Only Mode
6   1
7   0
```

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

Parameters **event** – The event to decode

encode()
Encodes the status bits to an event message

Returns The encoded event message

class pymodbus.events.**EnteredListenModeEvent**
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**
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

2.16 pdu — Base Structures

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

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

2.16.1 API Documentation

Contains base classes for modbus request/response/error packets

class pymodbus.pdu.**ModbusPDU**(***kwargs*)
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

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*)
Base class for a modbus request PDU

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

Parameters **exception** – The exception to return

Raises An exception response

class pymodbus.pdu.**ModbusResponse**(***kwargs*)
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.

class pymodbus.pdu.**ModbusExceptions**
An enumeration of the valid modbus exceptions

class pymodbus.pdu.**ExceptionResponse**(*function_code*, *exception_code=None*, ***kwargs*)
Base class for a modbus exception 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*)
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
```

decode(*data*)
This is here so this failure will run correctly

Parameters **data** – Not used

execute(*context*)
Builds an illegal function request error response

Parameters **context** – The current context for the message

Returns The error response packet

2.17 pymodbus — Pymodbus Library

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

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

2.17.1 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

2.18 `register_read_message` — Register Read Messages

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

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

2.18.1 API Documentation

Register Reading Request/Response

class `pymodbus.register_read_message.`**`ReadRegistersRequestBase`**(*address*, *count*, ***kwargs*)

Base class for reading a modbus register

`decode`(*data*)

Decode a register request packet

Parameters **data** – The request to decode

`encode`()

Encodes the request packet

Returns The encoded packet

class `pymodbus.register_read_message.`**`ReadRegistersResponseBase`**(*values*, ***kwargs*)

Base class for responsing to a modbus register read

`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*)

This function code is used to read the contents of a contiguous block of holding registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore registers numbered 1-16 are addressed as 0-15.

execute(*context*)

Run a read holding request against a datastore

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

Returns An initialized response, exception message otherwise

class pymodbus.register_read_message.**ReadHoldingRegistersResponse**(*values=None*, ***kwargs*)

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.

class pymodbus.register_read_message.**ReadInputRegistersRequest**(*address=None*, *count=None*, ***kwargs*)

This function code is used to read from 1 to approx. 125 contiguous input registers in a remote device. The Request PDU specifies the starting register address and the number of registers. In the PDU Registers are addressed starting at zero. Therefore input registers numbered 1-16 are addressed as 0-15.

execute(*context*)

Run a read input request against a datastore

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

Returns An initialized response, exception message otherwise

class pymodbus.register_read_message.**ReadInputRegistersResponse**(*values=None*, ***kwargs*)

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.

class pymodbus.register_read_message.**ReadWriteMultipleRegistersRequest**(*read_address*, *read_count*, *write_address*, *write_registers*, ***kwargs*)

This function code performs a combination of one read operation and one write operation in a single MODBUS transaction. The write operation is performed before the read.

Holding registers are addressed starting at zero. Therefore holding registers 1-16 are addressed in the PDU as 0-15.

The request specifies the starting address and number of holding registers to be read as well as the starting address, number of holding registers, and the data to be written. The byte count specifies the number of bytes to follow in the write data field."

decode(*data*)

Decode the register request packet

Parameters **data** – The request to decode

encode()
Encodes the request packet

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

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

Returns An initialized response, exception message otherwise

class pymodbus.register_read_message.**ReadWriteMultipleRegistersResponse**(*values=None*, ***kwargs*)
The normal response contains the data from the group of registers that were read. The byte count field specifies the quantity of bytes to follow in the read data field.

decode(*data*)
Decode the register response packet

Parameters **data** – The response to decode

encode()
Encodes the response packet

Returns The encoded packet

2.19 register_write_message — Register Write Messages

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

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

2.19.1 API Documentation

Register Writing Request/Response Messages

class pymodbus.register_write_message.**WriteSingleRegisterRequest**(*address=None*, *value=None*, ***kwargs*)
This function code is used to write a single holding register in a remote device.

The Request PDU specifies the address of the register to be written. Registers are addressed starting at zero. Therefore register numbered 1 is addressed as 0.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

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

Returns An initialized response, exception message otherwise

class pymodbus.register_write_message.**WriteSingleRegisterResponse**(*address=None*, *value=None*, ***kwargs*)
The normal response is an echo of the request, returned after the register contents have been written.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

class pymodbus.register_write_message.**WriteMultipleRegistersRequest**(*address=None*, *values=None*, ***kwargs*)
This function code is used to write a block of contiguous registers (1 to approx. 120 registers) in a remote device.

The requested written values are specified in the request data field. Data is packed as two bytes per register.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

execute(*context*)
Run a write single register request against a datastore

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

Returns An initialized response, exception message otherwise

class pymodbus.register_write_message.**WriteMultipleRegistersResponse**(*address=None*, *count=None*, ***kwargs*)
"The normal response returns the function code, starting address, and quantity of registers written.

decode(*data*)
Decode a write single register packet packet request

Parameters **data** – The request to decode

encode()
Encode a write single register packet packet request

Returns The encoded packet

2.20 server.sync — Twisted Synchronous Modbus Server

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

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

2.20.1 API Documentation

Implementation of a Threaded Modbus Server

class pymodbus.server.sync.**ModbusRequestHandler**(*request*, *client_address*, *server*)
Implements the modbus server protocol

This uses the socketserver.BaseRequestHandler to implement the client handler.

decode(*message*)
Decodes a request packet

Parameters **message** – The raw modbus request packet

Returns The decoded modbus message or None if error

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*)
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.

process_request(*request*, *client*)
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

class pymodbus.server.sync.**ModbusUdpServer**(*context*, *framer=None*, *identity=None*)
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.

process_request(*request*, *client*)
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

class pymodbus.server.sync.**ModbusSerialServer**(*context*, *framer=None*, *identity=None*, ***kwargs*)
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.

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*)
A factory to start and run a tcp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure

pymodbus.server.sync.**StartUdpServer**(*context=None*, *identity=None*)
A factory to start and run a udp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure

pymodbus.server.sync.**StartSerialServer**(*context=None*, *identity=None*, ***kwargs*)
A factory to start and run a udp modbus server

Parameters

- **context** – The ModbusServerContext datastore
- **identity** – An optional identify structure

2.21 server.async — Twisted Asynchronous Modbus Server

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

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

2.21.1 API Documentation

Implementation of a Twisted Modbus Server

class pymodbus.server.async.**ModbusTcpProtocol**
Implements a modbus server in twisted

connectionLost(*reason*)
Callback for when a client disconnects

Parameters **reason** – The client's reason for disconnecting

connectionMade()
Callback for when a client connects

Note, since the protocol factory cannot be accessed from the protocol __init__, the client connection made is essentially our __init__ method.

dataReceived(*data*)
Callback when we receive any data

Parameters **data** – The data sent by the client

class pymodbus.server.async.**ModbusUdpProtocol**(*store*, *framer=None*, *identity=None*)
Implements a modbus udp server in twisted

datagramReceived(*data*, *addr*)
Callback when we receive any data

Parameters **data** – The data sent by the client

class pymodbus.server.async.**ModbusServerFactory**(*store*, *framer=None*, *identity=None*)
Builder class for a modbus server

This also holds the server datastore so that it is persisted between connections

protocol
alias of ModbusTcpProtocol

pymodbus.server.async.**StartTcpServer**(*context*, *identity=None*)
Helper method to start the Modbus Async TCP server

Parameters

- **context** – The server data context
- **identify** – The server identity to use (default empty)

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

Helper method to start the Modbus Async Udp server

Parameters

- **context** – The server data context
- **identify** – The server identity to use (default empty)

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

Helper method to start the Modbus Async Serial server :param context: The server data context :param identify: The server identity to use (default empty) :param framer: The framer to use (default ModbusAsciiFramer)

2.22 transaction — Transaction Controllers for Pymodbus

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

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

2.22.1 API Documentation

Collection of transaction based abstractions

class pymodbus.transaction.**ModbusSocketFramer**(*decoder*)

Modbus Socket Frame controller

Before each modbus TCP message is an MBAP header which is used as a message frame. It allows us to easily separate messages as follows:

```
[         MBAP Header         ] [ Function Code] [ Data ]
[ tid ][ pid ][ length ][ uid ]
  2b     2b     2b        1b           1b           Nb

while len(message) > 0:
    tid, pid, length`, uid = struct.unpack(">HHHB", message)
    request = message[0:7 + length - 1`]
    message = [7 + length - 1:]

* length = uid + function code + data
* The -1 is to account for the uid byte
```

addToFrame(*message*)

Adds new packet data to the current frame buffer

Parameters **message** – The most recent packet

advanceFrame()

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

buildPacket(*message*)
Creates a ready to send modbus packet

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

checkFrame()
Check and decode the next frame Return true if we were successful

getFrame()
Return the next frame from the buffered data

Returns The next full frame buffer

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

class pymodbus.transaction.**ModbusRtuFramer**(*decoder*)
Modbus RTU Frame controller:

```
[ Start Wait ] [Address ][ Function Code] [ Data ][ CRC ][  End Wait  ]
  3.5 chars     1b         1b               Nb      2b      3.5 chars
```

Wait refers to the amount of time required to transmist at least x many characters. In this case it is 3.5 characters. Also, if we recieve a wait of 1.5 characters at any point, we must trigger an error message. Also, it appears as though this message is little endian. The logic is simplified as the following:

```
block-on-read:
    read until 3.5 delay
    check for errors
    decode
```

The following table is a listing of the baud wait times for the specified baud rates:

```
------------------------------------------------------------------
 Baud  1.5c (18 bits)   3.5c (38 bits)
------------------------------------------------------------------
 1200   13333.3 us       31666.7 us
 4800    3333.3 us        7916.7 us
 9600    1666.7 us        3958.3 us
19200     833.3 us        1979.2 us
38400     416.7 us         989.6 us
------------------------------------------------------------------
1 Byte = start + 8 bits + parity + stop = 11 bits
(1/Baud)(bits) = delay seconds
```

addToFrame(*message*)
This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

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

buildPacket(*message*)
Creates a ready to send modbus packet

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

checkFrame()
Check if the next frame is available. Return True if we were successful.

getFrame()
Get the next frame from the buffer

Returns The frame data or ''

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

class pymodbus.transaction.**ModbusAsciiFramer**(*decoder*)

Modbus ASCII Frame Controller:

```
[ Start ][Address ][ Function ][ Data ][ LRC ][ End ]
  1c        2c         2c         Nc     2c      2c

* data can be 0 - 2x252 chars
* end is '\r\n' (Carriage return line feed), however the line feed
  character can be changed via a special command
* start is ':'
```

This framer is used for serial transmission. Unlike the RTU protocol, the data in this framer is transferred in plain text ascii.

addToFrame(*message*)

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

Parameters **message** – The most recent packet

advanceFrame()

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

buildPacket(*message*)

Creates a ready to send modbus packet Built off of a modbus request/response

Parameters **message** – The request/response to send

Returns The encoded packet

checkFrame()

Check and decode the next frame

Returns True if we successful, False otherwise

getFrame()

Get the next frame from the buffer

Returns The frame data or ''

isFrameReady()

Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result header

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

Parameters **result** – The response packet

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

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

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

class pymodbus.transaction.**ModbusBinaryFramer**(*decoder*)
Modbus Binary Frame Controller:

```
[ Start ][Address ][ Function ][ Data ][ CRC ][ End ]
  1b        1b          1b          Nb     2b     1b

* data can be 0 - 2x252 chars
* end is   '}'
* start is '{'
```

The idea here is that we implement the RTU protocol, however, instead of using timing for message delimiting, we use start and end of message characters (in this case { and }). Basically, this is a binary framer.

The only case we have to watch out for is when a message contains the { or } characters. If we encounter these characters, we simply duplicate them. Hopefully we will not encounter those characters that often and will save a little bit of bandwitch without a real-time system.

Protocol defined by jamod.sourceforge.net.

addToFrame(*message*)
Add the next message to the frame buffer This should be used before the decoding while loop to add the received data to the buffer handle.

Parameters **message** – The most recent packet

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

buildPacket(*message*)
Creates a ready to send modbus packet

Parameters **message** – The request/response to send

Returns The encoded packet

checkFrame()
Check and decode the next frame

Returns True if we are successful, False otherwise

getFrame()
Get the next frame from the buffer

Returns The frame data or ‘’

isFrameReady()
Check if we should continue decode logic This is meant to be used in a while loop in the decoding phase to let the decoder know that there is still data in the buffer.

Returns True if ready, False otherwise

populateResult(*result*)
Populates the modbus result header

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

Parameters **result** – The response packet

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

This takes in a new request packet, adds it to the current packet stream, and performs framing on it. That is, checks for complete messages, and once found, will process all that exist. This handles the case when we read N + 1 or 1 / N messages at a time instead of 1.

The processed and decoded messages are pushed to the callback function to process and send.

Parameters

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

2.23 `utilities` — Extra Modbus Helpers

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

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

2.23.1 API Documentation

Modbus Utilities

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

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**(*buffer*, *byte_count_pos*)

Calculates the size of the frame based on the byte count.

Parameters

- **buffer** – 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

THREE

INDICES AND TABLES

- *genindex*
- *modindex*
- *search*

PYTHON MODULE INDEX

INDEX

Symbols

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W