

OmstNET Mystic Software

Copyright:

Copyright (c) 2018 Douglas E. Moore
dougmo52@gmail.com

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Description:

This software is a Python3.4 implementation of the Mystic ASCII protocol as described in the document, **MISTIC PROTOCOL USER'S GUIDE, Form 270-100823 — August, 2010**, available at http://documents.opto22.com/0270_Mistic_Protocol_Manual.pdf.

There are 4 files at this time:

- OmstNET.py – the protocol implementation
- OmstTTY.py – the Linux TTY interface
- OmstUTL.py – various utilities, in particular logging
- OmstTST.py – a sample app showing how to use OmstNET functions to send commands to and receive responses from the Mystic network.

OmstNET.py:

This file contains the bulk of the code required to implement the Mystic protocol as described in the Mystic users guide.

Of particular interest is the commands dictionary. Each entry in this dictionary is keyed using a command name, and the dictionary data is a MsgFmt namedtuple. The MsgFmt namedtuple is used to hold information related to the construction of each command, as well as any response data expected in return. The command and response format strings were taken from the command descriptions in the user's guide. There is a function defined in the code for each of these commands dictionary entries.

Consider the following commands dictionary entry, "READ AND CLEAR 32 BIT COUNTER GROUP':MsgFmt('U MMMM','DDDDDDDD')". The corresponding function definition is, "def read_and_clear_32_bit_counter_group(self,aa,MMMM)".

This command takes one data field, 'MMMM'. 'MMMM' is a bit mask which along with a command character tells the device that a group command is being sent. The 'DDDDDDDD' response field is to be a 32 bit quantity, and because the Mystic device will recognize this as a group command, it will respond with an array of 32 bit quantities, one for each bit set in MMMM.

Consider the command, "'READ PID LOOP PARAMETER':MsgFmt('t LL PP','DDDD')". This command takes two fields and returns 1 field. It's corresponding command would be expected to be, "def read_pid_loop_parameter(self,aa,LL,PP):". However, since this command can return 'DDDD' or 'DDDDDDDD', and because in some cases 'DDDDDDDD' is interpreted as 4 separate bytes, this command has been split into multiple subcommands based on the value passed in 'PP', and 'PP' is passed to the subcommands as a default value.

As an example, the subcommands:

- 'READ PID LOOP CONTROL WORD':MsgFmt('t LL PP','DDDD'),
- 'READ PID LOOP RATE WORD':MsgFmt('t LL PP','DDDD'),
- 'READ PID LOOP OUTPUT COUNTS':MsgFmt('t LL PP','DDDDDDDD'),
- 'READ PID LOOP INPUT, SETPOINT, OUTPUT CHANNELS':MsgFmt('t LL PP','II SS ZZ OO'),

are handled by 4 separate function definitions.

A Mystic command is constructed using these passed parameters by name, in the passed order, and the name is also used to set the field width in the constructed packet. So a 'DD' is a two ASCII characters or one binary byte, a 'DDDD' is four ASCII characters or two binary bytes, etc. Note that the functions do not pass the command character as it can be obtained by the send_receive_2 function by passing it the commands key command string. A short example is:

```
def repeat_last_response(self,aa):  
    return self.send_receive_2(aa,'REPEAT LAST RESPONSE',inspect.currentframe()) .
```

Thus in each command handler, the commands dictionary key name is passed to send_receive_2.

Python has some unique capabilities to **inspect** the function call stack at runtime for the passed parameters. Rather than checking each handler function, **inspect** can pass the calling frame, and the MsgFmt in the commands dictionary can tell us what arguments are needed to construct each command. Then when the response comes in, the MsgFmt can be used to tell the parser the names and widths of each returned field.

At present, only an 8 bit checksum and the default crc16r modes are supported. For the B3000, if the 'SET SYSTEM OPTIONS' command is sent, bits 4,5,6 should remain clear to prevent switching to an unsupported crc mode. In the event these bits get changed and saved in EEPROM, it will be necessary to boot in checksum mode to clear them.

Referring to `__main__` at bottom of file:

Request a list of serial ports:

Code:

```
# create the OmuxNET object
mn = OmstNET()

# list the available ttys
ttys = mn.tty.list_ttys()

# print a menu
print('List of available ttys')

for tty in ttys:
    print(tty,ttys[tty])

# ask user to select a port
ttychoice = int(input('Select tty: '),10)

print("\n")
```

Terminal output:

List of available ttys

0 /dev/ttyS0

1 /dev/ttyUSB0

Select tty:

Request a list of baudrates:

Code:

```
print('Check baudrate jumpers')

baudrates = mn.tty.list_baudrates()

for baudrate in baudrates:
    print(baudrate,baudrates[baudrate])
```

```
baudratechoice = int(input('Select baudrate: '),10)

if baudratechoice in mn.tty.baudrates:

    baudrate = int(mn.tty.baudrates[baudratechoice])

print("\n")
```

Terminal output:

Check baudrate jumpers

```
0 300
1 600
2 1200
3 2400
4 4800
5 9600
6 19200
7 38400
8 57600
9 76800
10 115200
```

Select baudrate:

Open the port and choose character data verification modes:

Code:

```
# open the port

if mn.tty.open(ttychoice,int(mn.tty.baudrates[baudratechoice]]):

    print('Check binary mode jumper')

    print(0,'ASCII mode')

    print(1,'BINARY mode')

    chmode = int(input('Select character mode: '),10)

    mn.binary_mode = chmode == 1

    print("\n")
```

```
print('Check verification mode jumper')
print(0,'Use CHECKSUM')
print(1,'use CRC16')
dvfmode = int(input('Select verification mode: '),10)
mn.crc_mode = dvfmode == 1
print('\n')
```

Terminal output:

```
opened '/dev/ttyUSB0:115200,N,8,1'
Check binary mode jumper
0 ASCII mode
1 BINARY mode
Select character mode:
Check verification mode jumper
0 Use CHECKSUM
1 use CRC16
Select verification mode:
```

Scan the Mystic network for devices:

Code:

```
mn.devices = mn.list_mistic_devices()
for device in mn.devices:
    rsp = mn.what_am_i(device)
    if rsp.ack == 'A':
        print('{:s} found @ {:02X}'.format(rsp.data.BrainBoardType,device))
```

Terminal:

```
B3000 (digital address) Multifunction I/O Unit found @ 00
B3000 (digital address) Multifunction I/O Unit found @ 01
B3000 (analog address) Multifunction I/O Unit found @ 02
B3000 (analog address) Multifunction I/O Unit found @ 03
```

OmstTST.py:

OmstTST.py is mainly provided to show how to use OmstNET.py to talk to a Mystic device. It does not test all commands but could in fact be modified to do so.

OmstTST.py will currently scan all 256 Mystic device addresses and build a list of devices. It then picks the first device in the list and attempts to run `verify_commands(OmstNET instance, address, configuration)` on it. The assumption in the default `cfg` parameter is that the address will have 4, 4 channel modules in the arrangement of ODC, IDC, ODC, IDC and that the outputs of each module are wired to the inputs of the next module.

At present, the `verify_commands` function checks general and various digital commands for operation, and attempts to read back data when it seems reasonable. So it is possible to configure modules and read back the configuration, or to configure counters, generate output pulses, and read counters, or toggle latches, read periods, frequencies, etc. But again, success all depends on loopback wiring. So there are really two forms of success, sending a command which is correct and therefore ACKed by the Mystic device, and comparing actual and expected data.

This is example of launching OmstTST.py in a shell and corresponding `verify_commands()` session output. The function `list_mistic_devices` scans all 256 Mystic addresses with a 'POWER UP CLEAR' but since the line is replaced with each address checked, only the “checking address FF” is visible.

Referring to `__main__` at bottom of file:

Terminal session:

Launch OmstTST.py:

```
python3.4 -i OmstTST.py
```

List available TTYs and choose one:

List of available ttys

```
0 /dev/ttyS0
```

```
1 /dev/ttyUSB2
```

choose a tty by number from the above list: 1

List available baudrates and choose one:

Check baudrate jumpers

0 300

1 600

2 1200

3 2400

4 4800

5 9600

6 19200

7 38400

8 57600

9 76800

10 115200

Select baudrate: 10

Open the selected TTY at the selected baudrate:

opened '/dev/ttyUSB2:115200,N,8,1'

Choose ASCII or BINARY character mode:

Check binary mode jumper

0 ASCII mode

1 BINARY mode

Select character mode: 1

Choose CHECKSUM or CRC16 data verification mode:

Check verification mode jumper

0 Use CHECKSUM

1 use CRC16

Select verification mode: 1

Scan the Mystic serial network for connected devices:

checking address FF

Found 4 Mystic devices

B3000 (digital address) Multifunction I/O Unit found @ 00

B3000 (digital address) Multifunction I/O Unit found @ 01

B3000 (analog address) Multifunction I/O Unit found @ 02

B3000 (analog address) Multifunction I/O Unit found @ 03

Run a sample command verification on the first Mystic device found:

POWER UP CLEAR acked

IDENTIFY TYPE acked

***** Request a 'REPEAT LAST RESPONSE' to 'IDENTIFY TYPE' command *****

IDENTIFY TYPE acked

***** Start with a clean slate *****

RESET ALL PARAMETERS TO DEFAULT acked

***** Set module configuration and verify with readback *****

SET I/O CONFIGURATION-GROUP acked

READ MODULE CONFIGURATION acked

READ MODULE CONFIGURATION readback data compare passed

***** Clear all latches *****

READ AND OPTIONALLY CLEAR LATCHES GROUP acked

***** Verify latches cleared *****

READ AND OPTIONALLY CLEAR LATCHES GROUP acked

READ AND OPTIONALLY CLEAR LATCHES GROUP readback data compare passed

***** Activate outputs verify with readback *****

SET OUTPUT MODULE STATE-GROUP acked

READ MODULE STATUS acked

READ MODULE STATUS readback data compare passed

***** Read, clear, verify positive latches *****

READ AND OPTIONALLY CLEAR LATCHES GROUP acked

READ AND OPTIONALLY CLEAR LATCHES GROUP readback mismatches,
{'PPPP': 'exp 61680 != act 0'}

***** Deactivate outputs and verify with readback *****

SET OUTPUT MODULE STATE-GROUP acked

READ MODULE STATUS acked

READ MODULE STATUS readback data compare passed

***** Read, clear, verify negative latches *****

READ AND OPTIONALLY CLEAR LATCHES GROUP acked

READ AND OPTIONALLY CLEAR LATCHES GROUP readback mismatches,
{'NNNN': 'exp 61680 != act 4095'}

***** Verify positive can be set and cleared *****

SET OUTPUT (ACTIVATE OUTPUT) acked

READ AND OPTIONALLY CLEAR LATCH acked

READ AND OPTIONALLY CLEAR LATCH readback data compare passed

READ AND OPTIONALLY CLEAR LATCH acked

READ AND OPTIONALLY CLEAR LATCH readback data compare passed

***** Verify negative latch can be set and cleared *****

CLEAR OUTPUT (DEACTIVATE OUTPUT) acked

READ AND OPTIONALLY CLEAR LATCH acked

READ AND OPTIONALLY CLEAR LATCH readback data compare passed

READ AND OPTIONALLY CLEAR LATCH acked

READ AND OPTIONALLY CLEAR LATCH readback data compare passed

***** Verify digital watchdog activates specified outputs *****

SET WATCHDOG MOMO AND DELAY acked

READ MODULE STATUS acked

READ MODULE STATUS readback data compare passed

***** Verify digital watchdog deactivates specified outputs *****

SET WATCHDOG MOMO AND DELAY acked

READ MODULE STATUS acked

READ MODULE STATUS readback data compare passed

***** sleep(1) and try to send a command, should be acked *****

SET WATCHDOG MOMO AND DELAY acked

***** If identify type is acked, wd was disabled successfully *****

IDENTIFY TYPE acked

***** Enable inputs as counters *****

ENABLE/DISABLE COUNTER GROUP acked

***** Make sure counters can be cleared *****

READ AND CLEAR 32 BIT COUNTER GROUP acked

READ 32 BIT COUNTER GROUP acked

READ 32 BIT COUNTER GROUP readback data compare passed

***** Generate 10 pulses on output channels *****

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

***** Verify counts, (assuming outputs looped back to inputs) *****

READ 32 BIT COUNTER GROUP acked

READ 32 BIT COUNTER GROUP readback miscompares,

{'DDDDDDDD': 'exp (10, 10, 10, 10, 10, 10, 10, 10) != act (10, 10, 10, 10, 0, 0, 0, 0)'}

***** Verify 16 bit counters can be read and cleared *****

READ AND CLEAR 16 BIT COUNTER acked

READ AND CLEAR 16 BIT COUNTER readback data compare passed

READ 16 BIT COUNTER acked

***** Reconfigure inputs to measure period *****

SET I/O CONFIGURATION-GROUP acked

READ AND RESTART 32 BIT PULSE/PERIOD GROUP acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

GENERATE N PULSES acked

READ PULSE/PERIOD COMPLETE STATUS acked

READ PULSE/PERIOD COMPLETE STATUS readback mismatches,

{'DDDD': 'exp 61680 != act (240,)')}

READ 32 BIT PULSE/PERIOD GROUP acked

READ 32 BIT PULSE/PERIOD GROUP readback data compare passed

***** Reconfigure inputs to measure frequency *****

SET I/O CONFIGURATION-GROUP acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

START CONTINUOUS SQUARE WAVE acked

READ FREQUENCY MEASUREMENT GROUP acked

READ FREQUENCY MEASUREMENT GROUP readback mismatches,

{'DDDD': 'exp (5, 5, 5, 5, 5, 5, 5, 5) != act (5, 5, 5, 5, 0, 0, 0, 0)'}

SET OUTPUT MODULE STATE-GROUP acked

***** Reconfigure inputs to measure positive pulse duration *****

SET I/O CONFIGURATION-GROUP acked

SET OUTPUT MODULE STATE-GROUP acked

READ AND RESTART 32 BIT PULSE/PERIOD GROUP acked

START OFF PULSE acked

START OFF PULSE acked

START OFF PULSE acked

START OFF PULSE acked

START OFF PULSE acked

START OFF PULSE acked

START OFF PULSE acked

START OFF PULSE acked

READ PULSE/PERIOD COMPLETE STATUS acked

READ PULSE/PERIOD COMPLETE STATUS readback mismatches,

{'DDDD': 'exp 61680 != act (240,)')}

READ 32 BIT PULSE/PERIOD GROUP acked

READ 32 BIT PULSE/PERIOD GROUP readback mismatches,

{'DDDDDDDD': 'exp (5000, 5000, 5000, 5000) != act (4917, 4915, 4919, 4911)'}

***** Reconfigure inputs to measure positive pulse duration *****

SET I/O CONFIGURATION-GROUP acked

SET OUTPUT MODULE STATE-GROUP acked

READ AND RESTART 32 BIT PULSE/PERIOD GROUP acked

START ON PULSE acked

START ON PULSE acked

START ON PULSE acked

START ON PULSE acked

START ON PULSE acked

START ON PULSE acked

START ON PULSE acked

START ON PULSE acked

READ PULSE/PERIOD COMPLETE STATUS acked

READ PULSE/PERIOD COMPLETE STATUS readback mismatches,

```
{'DDDD': 'exp 61680 != act (240,)')}
```

READ 32 BIT PULSE/PERIOD GROUP acked

READ 32 BIT PULSE/PERIOD GROUP readback mismatches,

```
{'DDDDDDDD': 'exp (5000, 5000, 5000, 5000) != act (5083, 5084, 5080, 5087)'}
```

>>>

OmstTTY.py

This is the serial interface to open, configure, write, read, and close serial ports. It has been tested on Linux Mint 17.3 only, and has functioned successfully on a hardware serial port, and two USB serial devices, an FTDI, and a dual port device using a non-FTDI chip. Operation has been successful at 115,200 baud running OmstTST verify_commands at full speed. In fact, there are no intentional delays placed in the list_mistic_devices function and it is typically able to run through all 256 addresses without missing an attached device, at least for the test system, a single B3000 with its 4 consecutive addresses.

Both binary and ASCII mode are supported. However, the Mistic binary mode requires a 9 bit serial word. The address byte should be sent with a mark bit, and the remaining command message bytes should have a space bit.

Linux Mint 17.3 does not support a mark/space parity mode. Thus in the write routine, when a binary message is to be sent, parity is enabled and the parity of each byte is computed such that terms can be switched between odd or even parity to make the binary address byte parity bit a mark, and the remaining command message bytes parity a space.

Binary receive has no such limitation because terms can be set to ignore the parity bits of incoming data.

OmstUTL.py

Of particular importance are the start_logging and stop_logging functions.

- From OmstNET, they can be run as:
 - mn.start_logging()
 - mn.stop_logging()
- From OmstTST, they can be run as:
 - OmstNET.utl.start_logging()

- OmstNET.utl.stop_logging()

An example for logging in a gnome terminal for a call to 'POWER UP CLEAR' with many calls to OmstTTY.rx_bytes_available() removed to decrease the printout :

```
>> OmstNET.utl.start_logging()
logging started
>>> mn.power_up_clear(0)
OmstNET.power_up_clear(aa=0)
OmstNET.send_receive_2(aa=0,cmd=POWER UP CLEAR,frame=<frame object at
0x1cee618>)
OmstNET.command(aa=0,cmd=POWER UP CLEAR,frame=<frame object at 0x1cee618>)
OmstNET.build_ascii_command(aa=0,cmd=POWER UP CLEAR,frame=<frame object at
0x1cee618>)
OmstNET.chksum(data=00A)
OmstNET.chksum(data=100)
>00AA1
OmstTTY.flush_input_buffer()
OmstTTY.rx_bytes_available()
)mstTTY.write(pkt=>00AA1
OmstNET.get_response(aa=0)
OmstNET.get_ascii_response(aa=0)
OmstNET.get_response_timeout(aa=0)
Response timeout is 0.033261 secs
OmstTTY.rx_bytes_available()
...
OmstTTY.read(n=1)
OmstTTY.rx_bytes_available()
OmstTTY.read(n=2)
OmstTTY.rx_bytes_available()
...
OmstTTY.read(n=1)
)mstNET.verify_dvf(aa=0,data=A41
OmstNET.chksum(data=A)
```

Response ('A', ") time took 0.008302 secs

RspMsg(ack='A', data="")

OmstNET.parse_response_data(aa=0,cmd=POWER UP CLEAR,pkt=RspMsg(ack='A', data=""))

ntrsp(ack='A', data=(None,))

>>> OmstNET.utl.stop_logging()

logging stopped

>>>