OmuxNET Optomux Software

Copyright:

Description:

The Optomux communications specification is contained in a document currently available at:

https://www.opto22.com/support/resources-tools/documents/1572_optomux_protocol_guide-pdf.

The OmuxNET software provides implements the Optomux communications specification in Python3.4. The OmuxNET software requires 3 files:

- OmuxNET.py – Optomux command construction and Optomux response processing
- OmuxTTL.py – Linux serial port configuration and control
- OmuxUTL.py – various functions such as logging function calls for debugging

OmuxNET.py:

This file implements an Optomux interface similar to the Opto 22 Optoware driver. That is, it maintains the concept of Optomux device addresses, Optomux module positions, Optomux command modifiers, and Optomux Information data exchange similar to the Optoware Send_Receive_Optomux driver function.

OmuxNET actually wraps the 80 or so Optoware driver command numbers described in the Optomux Protocol document section 'List of Driver Commands by Number' on pp 38-40 in Python functions of a

similar name. The wrappers try as much as possible to simplify passing of positions and info and to avoid passing modifiers when the purpose of the modifier was to create a sub command.

For example, the 'Set Time Delay' function takes a character modifier from the string 'HIJK' to change the output point action between pulse and delay, low or hi. Thus 'Set Time Delay'' has been split into 4 functions which avoid the need to pass a modifier as a function argument.

def pulse_on(self,address,positions,delay):
def delay_on(self,address,positions,delay):
def pulse_off(self,address,positions,delay):
def delay_off(self,address,positions,delay):

As much as is possible, the OmuxNET implementation of Optoware commands allows positions to be passed as either a tuple or bitmask, that is (0,1,2,3) or 15 should act the same.

As luck would have it, there was one Optomux command that required the module position to be passed as a hex character rather than a mask or tuple, the 'Average and Read Input' command on page 103. For some reason my B3000s return a N01\r for that command anyway as if they don't support it.

Example:

See the 'Set Output Waveform' example on 'p132 Optomux Protocol Guide' as a comparison of this software to Optoware parameter passing.

Optoware driver example to Start a continuous square wave at output points 4, 13, and 14 using the High limit is 170 (66.4% of full scale at 8.74 minutes for full-scale change x 2).

```
OMUX_Pos(0) = 4
OMUX_Pos(1) = 13
OMUX_Pos(2) = 14
OMUX_Pos(3) = -1

OMUX_Mod(0) = 6 'period is 8.74 mins
OMUX_Mod(1) = 4 'square wave
OMUX_Info(0) = 170 'high limit
OMUX_Info(1) = 0 'low limit

result = Send_Receive_Optomux(OMUX_Handle, 43, OMUX_Pos(0), _
   OMUX_Mod(0), OMUX_Info(0), OMUX_TimeOut)
```

OmuxNET would handle this using:

```
def set_output_waveform(self,address,positions,rate,shape, hi_limit,lo_limit):
```

It could be called with either a positions tuple or a positions bitmask:

- set_output_waveform(43,(4,13,14),6,4,170,0)
- set_output_waveform(43,0x6010,6,4,170,0)

Known issues:

- My B3000s don't seem to generate an analog square wave in response to either the 'Enhanced Output Waveform' or 'Set Output Waveform' command even though they ACK it as being a valid command. All of the other waveforms work.
- The 'Average and Read Input' command is NAKd as being unsupported. This is true even though I send the exact ASCII Optomux message in the example on p103 of the users guide.
- If I connect an Analog 10V output to an analog +/-10 V input, I can emulate temperature inputs and compute a temperature using the equations from the section 'Converting Temperature Readings' starting on page 155 if I use the 'Read Analog Inputs' or 'Read Averaged Inputs' commands.
- The B3000 unexpectedly changes the value returned by 'Read Analog Inputs' from an analog module when the module is set as a temperature input using the command 'Set Temperature Probe Type'. I have no analog temperature modules for testing the 'Read Temperature Inputs' and 'Read Average Temperature Inputs' so it would be a surprise if I am converting the Optomux data to a correct temperature reading.

OmuxTTY:

OmuxTTY contains declarations and functions to handle configuring, opening, closing, reading, and writing the serial ports in a Linux environment. It is likely that OmuxNET would work on other platforms if OmuxTTY was ported to the target platform. I wanted the Linux Python experience.

OmuxUTL:

Predominately involved in wrapping functions to allow debugging calls.

- utl.start_logging() - turns on logging
- utl.stop_logging() - turns off logging

There is also a RangeList class that is used in the functions that implement the temperature linearizion tables in the Optomux Protocol Guide.

Simple example of use:

In a bash terminal, run by typing:
python3.4 -i OmuxNET.py

At the bottom of the file OmuxNET.py there is a __main__ which shows how to list serial ports, choose a baud rate, open a port, and senn commands such as power_up_clear or get_optomux_type.

In the example, the Optomux network is opened, and all 256 addresses are pinged with a power_up_clear. Any that return a response are added to a list of devices, and they are queried with get_optomux_type.