OmmpNET Python OptoMMP Software

Copyright:

Description:

This software is a Python3.4 implementation of the OptoMMP protocol as described in the document, **OPTOMMP PROTOCOL GUIDE, Form 1465-180319—March 2018,** available at https://www.opto22.com/support/resources-tools/documents/1465-optommp-protocol-guide.

Testing has been limited to a small number of digital commands to verify the communications packet structures. However, the commands OrderedDict is relatively complete and most functions to implement the remaining commands would tend to be 1 or 2 lines of code per command.

The source consists of 3 files:

- OmmpNET.py – the memory map declarations and functions
- OmmpUTL.py – logging functionality
- OmmpTST.py – example code showing how to call the routines in OmmpNET.py

OmmpNET.py:

Commands:

Commands is an OrderedDict containing entries such as:

('Status R/W Operation Code',CmdFmt(prng(0xf0380000),4,'UI','I')),

- The key is a command name text string
- The value is a CmdFmt namedtuple

CmdFmt:

CmdFmt is defined as:

CmdFmt = namedtuple('CmdFmt',['ofs','len','typ','pak'])

The fields are used as follows:

- ofs – a prng which can be used to compute the lower part of the memory map offset
- len – the OptoMMP command data length in bytes
- typ – the OptoMMP data type as defined in the users guide section, **A: Opto 22 Hardware Memory Map**
- pak – this is a conversion from the OptoMMP type to the closest Python pack/unpack format type

CmdFmt.ofs:

- The function prng is used to create a Python range by setting up the beginning, ending, and range step. The beginning range is specified in the user's guide column.

- Many of the memory parameters exist as an array of values which are located on specified boundaries.
    - As an example, older racks could support up to 16 digital 4 point modules. So there could be digital 64 points.
    - Each digital point has a number of parameters that can be read and/or written.
    - The manual section DIGITAL POINT WRITE—READ/WRITE indicates that “only the first three points are shown in the table. Each successive point starts on an even 40 hex boundary and follows the same pattern.
    - The dictionary entry as: ('Digital Point State',CmdFmt(prng(0xF0800000,64,64),4,'B','I')) is used to create a range of 64 Digital Point State' offset values on 64 byte boundaries. This means the digital state of the first point is found at 0xF0800000, and the remaining 63 point states are addressed thereafter every 64 bytes. This is done as: range(ofs,cnt*siz+ofs,siz).
    - Memory mapped values that are not an array element have a default range(ofs,ofs+1) which means the range only contain the CmdFmt.ofs[0] element.

Index argument:

- The index argument is used with the prng value CmdFmt.ofs[index] to select an array element when necessary.
- If the memory mapped value to be read or written is not an array, then prng only contains the beginning offset.

Reading and writing individual memory mapped values:

Functions are provided for reading and writing values in the memory map. They are:

- def write_mmap_value(self,aa,cmd,value,index=0)
- def read_mmap_value(self,aa,cmd,index=0).

These functions call the appropriate block or quadlet read or write depending on the length returned by calcsize(CmdFmt.pak).

When an array of values exists, index specifies the array element. For example, index would be a point, a timer, an event, etc.

Reading and writing a block of contiguous memory mapped values:

Functions are provided for reading and writing blocks of values in the memory map. They are:

- def write_n_mmap_values(self,aa,cmd,n,values,idx=0):
- def read_n_mmap_values(self,aa,cmd,n,idx=0):

These functions call only the read/write block request functions.

When an array of values exists, the index parameter specifies the array element, ie: a point, a timer, an event, etc.

Both of these functions modify the memory map entry, ('Read/Write Data Block',CmdFmt(0,0,'?',")), filling in the CmdFmt values. The prng value is set to access the base address for the block to be read/written.

The CmdFmt pak field is build by concatenating the pak fields of the individual elements to be read/written. This is done to allow reusing the read_block_request/write_block_request functions which obtain pack/unpack instructions from the CmdFmt.pak member.

Some mmap parameters contain multiple values, ie: an IP address, a firmware version, etc. read_n_mmap_values repacks the tuple it received from read_block_request, then unpacks it a parameter at a time so that values such as IP addresses are contained in a tuple within the data tuple.

For example, om.read_map_values(aa,'Status Read Memory Map Version',9) converts read_block_request's response data as follows:

- RspFmt(rcode=0, data=(1, 0, 0, 57349, 2, 0, 4027580416, 33554951, 2, 0, 2, 0)) becomes
- RspFmt(rcode=0, data=(1, 0, 0, 57349, 2, 0, 4027580416, 33554951, (2, 0, 2, 0))).

OmmpUTL.py

Contains decorators for logging function calls and routines to log info and error messages as required. Consider the function write_quadlet_request. It contains logging function calls and a decorator for debugging purposes.

Note that there is a great amount of detail in the logging at present, primarily due to the @utl.logger decorator. Obviously the logging could be tuned, but during development, it is a great help, especially seeing the requests and responses exchanged with the B3000-ENET brain.

Call utl.start_logging() or utl_stop_logging().

```
@utl.logger
def write_quadlet_request(self,aa,mmap,index,value):
    """
    aa    -  is the device's UDP address as a tuple, ie:(ip,port)
    mmap  - commands dictionary value
    index - index for addressing array of say tomers, events, points, etc
    value - value to be written
    """
    tl = self.get_next_transaction_label()
    s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    # create a tuple for passing to pack
    txargs = self.WriteQuadletRequest(tl<<2,0<<4,0xffff,mmap.ofs[index],value)
    utl.log_info_message(str(txargs))
    # if value is a sequence, replace it in txargs with it's contents
    txargs = self.flatten_nested_sequences(txargs,True)
    utl.log_info_message(str(txargs))
    # append the value's pack format string
    fmt = self.WriteQuadletRequestHeaderFormatString + mmap.pak
    # finally can create the packet
    pkt = pack(fmt,*txargs)
    utl.log_info_message('Request pkt: {}'.format(binascii.hexlify(pkt)))
    s.sendto(pkt,aa)
    d = s.recvfrom(1024)
    utl.log_info_message('Response pkt: {}'.format(binascii.hexlify(d[0])))
    rsp = unpack(self.WriteResponseHeaderFormatString,d[0])
    rsp = self.WriteQuadletResponse(*(rsp[0]>>2,rsp[1]>>4,rsp[2]>>4))
    return (rsp.rcode,self.ecodes[rsp.rcode])
```

A sample of logging output for power_up_clear follows.

```
>>> om.power_up_clear(aa)
OmmpNET.power_up_clear(aa=('192.168.10.225', 2001))
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Status R/W Operation
Code,value=1)
OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4030201856, 4030201857), len=4, typ='UI',
```

pak='I'),index=0,value=1)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=64, tcode=0, dofs_hi=65535, dofs_lo=4030201856, value=1)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=64, tcode=0, dofs_hi=65535, dofs_lo=4030201856, value=1),start=True)
(64, 0, 65535, 4030201856, 1)
Request pkt: b'000040000000ffffff038000000000001'
Response pkt: b'000040200000000000000000'
RspFmt(rcode=0, data='No Error')
>>>

In this logging output, the sequence is as follows:

- om.power_up_clear(aa) is called which calls
- om.write_mmap_value(aa,' Status R/W Operation Code',1)
- om.write_quadlet_request is called since a power_up_clear will fit in a quadlet
- om.get_next_transaction_label() gets then increments tl
- om.write_quadlet_request is called, sends a request and receives a response
  - Request pkt: b'000040000000ffffff038000000000001'
  - Response pkt: b'000040200000000000000000'
- RspFmt(rcode=0, data='No Error') shows the command succeeded

OmmpTST.py:

Contains examples of how to call the OmmpNET functions.

Example routine to generate a 10Hz squarewave using timer events:

```
def generate_10_hz_sqw_on_point_0(om,aa):
"""
cfg p0.0 as a digital output
deactivate p0.0
set timer 0 delay to 50ms
set timer 0 start event on activation of P0.0
set timer 0 reaction event to turn off P0.0
set timer 1 delay to 50ms
set timer 1 start event to P0.0 clear
set timer 1 reaction event to set P0.0
activate p0.0 to start squarewave
"""
# send a power up clear
print('send a power up clear',om.power_up_clear(aa))
# Set first point to an Output
print('make digital point 0 an output',om.set_point_type(aa,0,0x180))
# turn Off point P0.0
print('turn off digital point 0',om.deactivate_digital_point(aa,0))
# Set up a 50ms timer
print('set t0 delay',om.set_timer_delay(aa,0,50))
print('get t0 delay',om.get_timer_delay(aa,0))
# start it when P0.0 Sets
print('set t0 trigger on mask',om.set_timer_trigger_on_mask(aa,0,1))
print('get t0 trigger on mask',om.get_timer_trigger_on_mask(aa,0))
# use Event to clear P0.0
print('set t0 reaction off mask',om.set_timer_reaction_off_mask(aa,0,1))
print('get t0 reaction off mask',om.get_timer_reaction_off_mask(aa,0))
# Set up a 50ms timer
print('set t1 delay',om.set_timer_delay(aa,1,50))
print('get t1 delay',om.get_timer_delay(aa,1))
# start it when P0.0 clears
print('set t1 trigger off mask',om.set_timer_trigger_off_mask(aa,1,1))
print('get t1 trigger off mask',om.get_timer_trigger_off_mask(aa,1))
# use Event to Set P0.0
print('set t1 reaction on mask',om.set_timer_reaction_on_mask(aa,1,1))
print('get t1 reaction on mask',om.get_timer_reaction_on_mask(aa,1))
# turn On P0.0 to start timer 0
print('turn on digital point 0 to start sqw',om.activate_digital_point(aa,0))
   # use Event to clear P0.0
   print('set t0 reaction off mask',self.set_timer_reaction_off_mask(aa,0,1))
   print('get t0 reaction off mask',self.get_timer_reaction_off_mask(aa,0))
```

```
    # Set up a 50ms timer
    print('set t1 delay',self.set_timer_delay(aa,1,50))
    print('get t1 delay',self.get_timer_delay(aa,1))
    # start it when P0.0 clears
    print('set t1 trigger off mask',self.set_timer_trigger_off_mask(aa,1,1))
    print('get t1 trigger off mask',self.get_timer_trigger_off_mask(aa,1))
    # use Event to Set P0.0
    print('set t1 reaction on mask',self.set_timer_reaction_on_mask(aa,1,1))
    print('get t1 reaction on mask',self.get_timer_reaction_on_mask(aa,1))
    # turn On P0.0 to start timer 0
    print('turn on digital point 0 to start sqw',self.activate_digital_point(aa,0))
```

Logging output for the above:

```
>>> utl.start_logging()
logging started
>>> generate_10_hz_sqw_on_point_0(om,aa)
OmmpNET.power_up_clear(aa=('192.168.10.225', 2001))
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Status R/W Operation
Code,value=1)
OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4030201856, 4030201857), len=4, typ='UI',
pak='I'),index=0,value=1)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=0, tcode=0, dofs_hi=65535, dofs_lo=4030201856, value=1)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=0, tcode=0, dofs_hi=65535,
dofs_lo=4030201856, value=1),start=True)
(0, 0, 65535, 4030201856, 1)
Request pkt: b'000000000000ffffff038000000000001'
Response pkt: b'000000200000000000000000'
send a power up clear RspFmt(rcode=0, data='No Error')
OmmpNET.set_point_type(aa=('192.168.10.225', 2001),point=0,typ=384)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Point Type,value=384,index=0)
OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4039114756, 4039118852, 64), len=4, typ='UI',
pak='I'),index=0,value=384)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=4, tcode=0, dofs_hi=65535, dofs_lo=4039114756, value=384)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=4, tcode=0, dofs_hi=65535,
dofs_lo=4039114756, value=384),start=True)
(4, 0, 65535, 4039114756, 384)
Request pkt: b'000004000000ffffff0c0000400000180'
Response pkt: b'000004200000000000000000'
make digital point 0 an output RspFmt(rcode=0, data='No Error')
OmmpNET.deactivate_digital_point(aa=('192.168.10.225', 2001),point=0)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Digital Point
Deactivate,value=1,index=0)
```

OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4035969028, 4035973124, 64), len=4, typ='B',
pak='I'),index=0,value=1)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=8, tcode=0, dofs_hi=65535, dofs_lo=4035969028, value=1)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=8, tcode=0, dofs_hi=65535,
dofs_lo=4035969028, value=1),start=True)
(8, 0, 65535, 4035969028, 1)
Request pkt: b'000008000000fffff090000400000001'
Response pkt: b'000008200000000000000000'
turn off digital point 0 RspFmt(rcode=0, data='No Error')
OmmpNET.set_timer_delay(aa=('192.168.10.225', 2001),timer=0,period=50)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Delay,value=50,index=0)
OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4040425536, 4040491072, 128), len=4, typ='UI',
pak='I'),index=0,value=50)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=12, tcode=0, dofs_hi=65535, dofs_lo=4040425536, value=50)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=12, tcode=0, dofs_hi=65535,
dofs_lo=4040425536, value=50),start=True)
(12, 0, 65535, 4040425536, 50)
Request pkt: b'00000c000000fffff0d4004000000032'
Response pkt: b'00000c200000000000000000'
set t0 delay RspFmt(rcode=0, data='No Error')
OmmpNET.get_timer_delay(aa=('192.168.10.225', 2001),timer=0)
OmmpNET.read_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Delay,index=0)
OmmpNET.read_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4040425536, 4040491072, 128), len=4, typ='UI', pak='I'),index=0)
OmmpNET.get_next_transaction_label()
ReadQuadletRequest(tl=16, tcode=64, dofs_hi=65535, dofs_lo=4040425536)
Request: b'000010400000fffff0d40040'
Response: b'000010600000000000000000000000032'
get t0 delay RspFmt(rcode=0, data=(50,))
OmmpNET.set_timer_trigger_on_mask(aa=('192.168.10.225', 2001),timer=0,mask=1)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Start Input On
Mask,value=1,index=0)
OmmpNET.write_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425472,
4040491008, 128), len=8, typ='M', pak='Q'),index=0,value=1)
OmmpNET.get_next_transaction_label()
WriteBlockRequest(tl=20, tcode=16, dofs_hi=65535, dofs_lo=4040425472, length=8, value=1)
OmmpNET.flatten_nested_sequences(t=WriteBlockRequest(tl=20, tcode=16, dofs_hi=65535,
dofs_lo=4040425472, length=8, value=1),start=True)
(20, 16, 65535, 4040425472, 8, 1)
Request: b'000014100000fffff0d400000008000000000000000000001'
Response: b'000014200000000000000000'
set t0 trigger on mask RspFmt(rcode=0, data='No Error')
OmmpNET.get_timer_trigger_on_mask(aa=('192.168.10.225', 2001),timer=0)
OmmpNET.read_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Start Input On Mask,index=0)
OmmpNET.read_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425472,

4040491008, 128), len=8, typ='M', pak='Q'),index=0)
OmmpNET.get_next_transaction_label()
ReadBlockRequest(tl=24, tcode=80, dofs_hi=65535, dofs_lo=4040425472, length=8)
Request: b'000018500000fffff0d4000000080000'
Response: b'000018700000000000000000000800320000000000000001'
get t0 trigger on mask RspFmt(rcode=0, data=(1,))
OmmpNET.set_timer_reaction_off_mask(aa=('192.168.10.225', 2001),timer=0,mask=1)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Expired Output Off
Mask,value=1,index=0)
OmmpNET.write_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425512,
4040491048, 128), len=8, typ='M', pak='Q'),index=0,value=1)
OmmpNET.get_next_transaction_label()
WriteBlockRequest(tl=28, tcode=16, dofs_hi=65535, dofs_lo=4040425512, length=8, value=1)
OmmpNET.flatten_nested_sequences(t=WriteBlockRequest(tl=28, tcode=16, dofs_hi=65535,
dofs_lo=4040425512, length=8, value=1),start=True)
(28, 16, 65535, 4040425512, 8, 1)
Request: b'00001c100000fffff0d400280008000000000000000000001'
Response: b'00001c200000000000000000'
set t0 reaction off mask RspFmt(rcode=0, data='No Error')
OmmpNET.get_timer_reaction_off_mask(aa=('192.168.10.225', 2001),timer=0)
OmmpNET.read_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Expired Output Off
Mask,index=0)
OmmpNET.read_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425512,
4040491048, 128), len=8, typ='M', pak='Q'),index=0)
OmmpNET.get_next_transaction_label()
ReadBlockRequest(tl=32, tcode=80, dofs_hi=65535, dofs_lo=4040425512, length=8)
Request: b'000020500000fffff0d4002800080000'
Response: b'000020700000000000000000000800320000000000000001'
get t0 reaction off mask RspFmt(rcode=0, data=(1,))
OmmpNET.set_timer_delay(aa=('192.168.10.225', 2001),timer=1,period=50)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Delay,value=50,index=1)
OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4040425536, 4040491072, 128), len=4, typ='UI',
pak='I'),index=1,value=50)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=36, tcode=0, dofs_hi=65535, dofs_lo=4040425664, value=50)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=36, tcode=0, dofs_hi=65535,
dofs_lo=4040425664, value=50),start=True)
(36, 0, 65535, 4040425664, 50)
Request pkt: b'000024000000fffff0d400c000000032'
Response pkt: b'00002420000000000000000'
set t1 delay RspFmt(rcode=0, data='No Error')
OmmpNET.get_timer_delay(aa=('192.168.10.225', 2001),timer=1)
OmmpNET.read_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Delay,index=1)
OmmpNET.read_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4040425536, 4040491072, 128), len=4, typ='UI', pak='I'),index=1)
OmmpNET.get_next_transaction_label()
ReadQuadletRequest(tl=40, tcode=64, dofs_hi=65535, dofs_lo=4040425664)
Request: b'000028400000fffff0d400c0'

Response: b'000028600000000000000000000032'
get t1 delay RspFmt(rcode=0, data=(50,))
OmmpNET.set_timer_trigger_off_mask(aa=('192.168.10.225', 2001),timer=1,mask=1)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Start Input Off Mask,value=1,index=1)
OmmpNET.write_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425480, 4040491016, 128), len=8, typ='M', pak='Q'),index=1,value=1)
OmmpNET.get_next_transaction_label()
WriteBlockRequest(tl=44, tcode=16, dofs_hi=65535, dofs_lo=4040425608, length=8, value=1)
OmmpNET.flatten_nested_sequences(t=WriteBlockRequest(tl=44, tcode=16, dofs_hi=65535, dofs_lo=4040425608, length=8, value=1),start=True)
(44, 16, 65535, 4040425608, 8, 1)
Request: b'00002c100000fffff0d400880008000000000000000001'
Response: b'00002c200000000000000000'
set t1 trigger off mask RspFmt(rcode=0, data='No Error')
OmmpNET.get_timer_trigger_off_mask(aa=('192.168.10.225', 2001),timer=1)
OmmpNET.read_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Start Input Off Mask,index=1)
OmmpNET.read_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425480, 4040491016, 128), len=8, typ='M', pak='Q'),index=1)
OmmpNET.get_next_transaction_label()
ReadBlockRequest(tl=48, tcode=80, dofs_hi=65535, dofs_lo=4040425608, length=8)
Request: b'000030500000fffff0d4008800080000'
Response: b'000030700000000000000000000800320000000000000001'
get t1 trigger off mask RspFmt(rcode=0, data=(1,))
OmmpNET.set_timer_reaction_on_mask(aa=('192.168.10.225', 2001),timer=1,mask=1)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Expired Output On Mask,value=1,index=1)
OmmpNET.write_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425504, 4040491040, 128), len=8, typ='M', pak='Q'),index=1,value=1)
OmmpNET.get_next_transaction_label()
WriteBlockRequest(tl=52, tcode=16, dofs_hi=65535, dofs_lo=4040425632, length=8, value=1)
OmmpNET.flatten_nested_sequences(t=WriteBlockRequest(tl=52, tcode=16, dofs_hi=65535, dofs_lo=4040425632, length=8, value=1),start=True)
(52, 16, 65535, 4040425632, 8, 1)
Request: b'000034100000fffff0d400a000080000000000000000001'
Response: b'00003420000000000000000'
set t1 reaction on mask RspFmt(rcode=0, data='No Error')
OmmpNET.get_timer_reaction_on_mask(aa=('192.168.10.225', 2001),timer=1)
OmmpNET.read_mmap_value(aa=('192.168.10.225', 2001),cmd=Timer Expired Output On Mask,index=1)
OmmpNET.read_block_request(aa=('192.168.10.225', 2001),mmap=CmdFmt(ofs=range(4040425504, 4040491040, 128), len=8, typ='M', pak='Q'),index=1)
OmmpNET.get_next_transaction_label()
ReadBlockRequest(tl=56, tcode=80, dofs_hi=65535, dofs_lo=4040425632, length=8)
Request: b'000038500000fffff0d400a000080000'
Response: b'000038700000000000000000000800320000000000000001'
get t1 reaction on mask RspFmt(rcode=0, data=(1,))
OmmpNET.activate_digital_point(aa=('192.168.10.225', 2001),point=0)
OmmpNET.write_mmap_value(aa=('192.168.10.225', 2001),cmd=Digital Point

Activate,value=1,index=0)
OmmpNET.write_quadlet_request(aa=('192.168.10.225',
2001),mmap=CmdFmt(ofs=range(4035969024, 4035973120, 64), len=4, typ='B',
pak='I'),index=0,value=1)
OmmpNET.get_next_transaction_label()
WriteQuadletRequest(tl=60, tcode=0, dofs_hi=65535, dofs_lo=4035969024, value=1)
OmmpNET.flatten_nested_sequences(t=WriteQuadletRequest(tl=60, tcode=0, dofs_hi=65535,
dofs_lo=4035969024, value=1),start=True)
(60, 0, 65535, 4035969024, 1)
Request pkt: b'00003c000000fffff090000000000001'
Response pkt: b'00003c200000000000000000'
turn on digital point 0 to start sqw RspFmt(rcode=0, data='No Error')
>>>