# C:\data\python\oneWire.py

```
#!/usr/bin/python

'''
  Maxim Integrated (formerly Dallas Semiconductor)
  OneWire Communications library...
    oneWireDevice
      This class provides a "wire device" abstraction. Instances represent
      oneWire devices, which can perform functions such as temp() and
      port(), e.g.
        hW = oneWire("SERIAL, COM7")
        tank = oneWireDevice(hW,'28 FA B2 FF 00 00 00 84')
        print tank.temp()
    oneWire
      This class provides a "wire" abstraction. It creates a wire object
      based on the DS2480B serial interface. This represents the general
      library for completing oneWire calls. Operations apply to the bus
      so most functions require passing the serial number of the device
      with which you communicate. e.g.
        hW = oneWire("SERIAL, COM7")
        devices = hW.search('28')
        tank = devices[0]
        hW.temp(tank)
    oneWireIO
      This class performs "low-level" oneWire I/O operations assuming a
      DS2480 interface. It translates 3 rudimentary functions into data
      and commands for the DS2480B. Users normally don't need to make
      calls to this class. Rudimentary functions include:
        cmd()  - commands the DS2480B
        rst()  - resets the oneWire bus, i.e. generates a bus reset pulse
        send() - sends (and receives) oneWire data
    oneWirePhysical
      This class performs the actual serial communications to the DS2480B.
  High-level routines pass hex strings for readability/printability.
  Low-level routines automatically convert hex strings to ASCII strings
  for communications and vice versa for return data.
  oneWire device serial number (sn) format used (matches data sheets) ...
    1. defined as hex string: 64 bits, 16 hex characters
    2. with or with space delimiting; spaces preferred for readability
    3. least significant byte first (left), which is the family code
    4. most significant byte last (right), which is the CRC8
    5. note: each byte (hex pair) written MS nibble first
    e.g.
      '28 FA B2 FF 00 00 00 84' --> "28" family code, DS18B20, crc: 84
      or '28FAB2FF00000084'
      True msb to lsg orderedSN --> "84 00 00 00 FF B2 FA 28"
'''


  # external references
  import re                       # regular expressions
  from time import sleep, ctime   # timing functions
  from root import root, eq       # root lib provides debug/logging support
```

```
from crc import crc, hexsplit   # independent library for crc functions
import serial                   # serial port support
print('Serial VERSION: ', serial.VERSION)

# supported 1-wire device families ...
(DS18B20,DS2413,DS2408,DS2438,DS28EA00,DS28E04) = ('28','3A','29','26','42','1C')
wDEVICES = {
    '28':('DS18B20','temp'),
    '3A':('DS2413','port'),
    '12':('DS2406','port*'),
    '29':('DS2408','port'),
    '26':('DS2438','gauge'),
    '42':('DS28EA00','temp','indexed'),
    '1C':('DS28E04','port','indexed'),
    '09':('DS2502','id'),
    '10':('DS1920','temp*'),
    '1D':('DS2423','RAM+CNTR')
    }
# high-level oneWire port operation mappings to low-level calls...
(wIORESET,wIOCLEAR,wIOSET,wIOIN,wIOREG,wIOOUT,wIOTOGGLE,wIOPULSE) =
('RESET','CLEAR','SET','IN','REG','OUT','TOGGLE','PULSE')

# class defining the physical layer for the oneWire communications
# using RS232 COM port interface to DS2480B
class oneWirePhysical(root):

    VERSION = 'oneWirePhysical:1.10'
    hDEVICE = None  # serial port object

    # constructor for physical (serial) device
    # params string defines the serial port, e.g. "SERIAL COM11"
    # default port parameters 9600 N 8 1 can be overridden with params string
    def __init__(self, params, log=None):
        if log == None: log = 'oneWire.log'
        root.__init__(self,log,root.VERBOSE) # verbose mode until known good.
        self.debug("# call:oneWirePhysical.__init__(%s,%s)" % (str(params),log))
        mode,portname,baud,bits,parity,stop = (re.split('[, ]+',params)+[None,None,None,None])
[:6]
        if mode.upper() == 'SERIAL':
            self.hDEVICE = serial.Serial(port=portname,baudrate=baud or 9600,timeout=5)
            self.hDEVICE.parity = parity or 'N'
            self.hDEVICE.bytesize = int(bits or '8')
            self.hDEVICE.stopbits = int(stop or '1')

    # performs lowest level io, pass binary 'bytes' in and out
    # waits for return data by default
    def io(self, byteData, wait=True):
#        print('OUT[' + str(len(byteData)) + ']: "' + byteData.hex(' ').upper() + '"')
        self.hDEVICE.write(byteData)
        if wait:
          rxd = self.hDEVICE.read(len(byteData))
#          print('IN[' + str(len(rxd)) + ']: "' + rxd.hex(' ').upper() + '"')
        else:
          rxd = b''
        return rxd

    # empties device buffers
```

```
    def flush(self):
        if self.hDEVICE.inWaiting(): self.hDEVICE.read(self.hDEVICE.inWaiting())


# class defining low-level oneWire operations based on DS2480B protocol
class oneWireIO(oneWirePhysical):

    VERSION = 'oneWireIO:1.00'
    presence = 0
    BUS = ('SHORTED','OK','ALARM','EMPTY')

    # constructor method
    def __init__(self, params, log=None):
        oneWirePhysical.__init__(self, params, log)
        self.debug("# call:oneWireIO.__init__(%s,%s)" % (str(params),log))
        self.scribe('# One Wire configured with ' + params)
        self.wireState = None
        self.hCRC = crc()

    # 1-wire CRC8 wrapper method returns CRC for hex string input, with or without spaces
    def crc(self, hexStr):
        return self.hCRC.crc8(hexStr)

    # converts a hex string, with or without spaces, into a (binary) string
    def hex2str(self, hexStr, escape=False):
        hexStr = hexStr.replace(' ','')
        hexPairs = re.findall('..',hexStr)
        hx = []
        for x in hexPairs:
          hx.append(chr(int(x,16)))
          if escape and ord(hx[-1])==0xE3:
            hx.append(hx[-1])
        return ''.join(hx)

    # converts a (binary string into a space delimitted readable hex string
    def str2hex(self, bstr):
        hx = []
        for x in list(bstr):
          hx.append("%02X" % (ord(x)))
        return ' '.join(hx)

    # converts a hex string, with or without spaces, into 'bytes' type
    def hex2bytes(self, hexStr, escape=False):
        if escape:
            hexStr = hexStr.replace('E3','E3 E3')
        return bytes.fromhex(hexStr)

    # converts 'bytes' type into a space delimitted readable hex string
    def bytes2hex(self, theBytes):
        return theBytes.hex(' ').upper()

    # io helper function wrapper to send/receive data as hexstrings
    def ioHex(self,hexStr, wait=True, escape=False):
        #print('ioHex:', hexStr)
        return self.bytes2hex(self.io(self.hex2bytes(hexStr,escape),wait))
        #return self.str2hex(self.io(self.hex2str(hexStr,escape),wait))
```

```
    # basic 1-wire operations...

    # commands the DS2480 to send a 1-wire reset
    # success returns wire presence code, 0xC9 for DS2480, 0xCD for DS2480B
    def rst(self):
        #DS2480 must be in command mode to send reset command.
        if not self.wireState == oneWire.COMMAND:
            self.ioHex(oneWire.CMDMODE, False)
            self.debug("#>Command Mode")
            self.wireState = oneWire.COMMAND
        self.flush()
        wire = self.ioHex(oneWire.RESETCMD) or '0x03'
        # check for presence pulse, 2 lsb's
        # 0 - wire shorted, 1 - presence, 2 - alarm presence, 3 - none
        self.presence = int(wire,16) & 0x03
        self.debug("#>1-wire Reset: %s [%s]" %(wire,self.BUS[self.presence]))
        return self.presence

    # commands the wire (DS2480B), every byte sent does not always return a byte
    def cmd(self, hexStr, wait=True):
        if not self.wireState == oneWire.COMMAND:
            self.ioHex(oneWire.CMDMODE, False)
            self.debug("#>Command Mode")
            self.wireState = oneWire.COMMAND
        self.debug(">>"+hexStr)
        result = self.ioHex(hexStr,wait,False)
        if result: self.debug("<<"+result)
        return result

    # sends wire data, every byte out returns a byte, data needs to escape 0xE3 code
    def send(self, hexStr):
        if not self.wireState == oneWire.DATA:
            self.ioHex(oneWire.DATAMODE, False)
            self.debug("#>Data Mode")
            self.wireState = oneWire.DATA
        self.debug("->"+hexStr)
        result = self.ioHex(hexStr,True,True)
        self.debug("<-"+result)
        return result

    # wrapper for decimal data byte...
    def sendByte(self, b):
        self.send("%02X" % (ord(b)))

#DEFAULT_1WIRE_CONFIGURATION = '19 45 55'
DEFAULT_1WIRE_CONFIGURATION = '19 45 5B 65'

# class defining high level 1-wire functions
class oneWire(oneWireIO):

    VERSION = 'oneWire:1.11'

# constants
    # scribe modes...
```

```
        (QUIET, NORMAL, VERBOSE) = (0,1,2)
        # internal DS2480 state machine...
        (UNKNOWN,DATA,COMMAND) = (0,1,2)
        # DS2480 commands and parameters...
    #    (SINGLEBIT0,SINGLEBIT1) = ('81','91')
    #    (ACCELOFF,ACCELON) = ('A1','B1')
    #    (DATAMODE,CMDMODE) = ('E1','E3')
    #    (RESETCMD,PULSECMD) = ('C1','E1')
        # DS2480 commands and parameters for flex speed...
        (SINGLEBIT0,SINGLEBIT1,STRONGBIT0,STRONGBIT1) = ('85','95','87','97')
        (ACCELOFF,ACCELON) = ('A5','B5')
        (DATAMODE,CMDMODE) = ('E1','E3')
        (RESETCMD,PULSE5DISARM,PULSE5ARM,PROG12DISARM,PROG12ARM) = ('C5','ED','EF','FD','FF')
        (REGSPEED,FLEXSPEED,OVERDRIVE) = ('00','04','08')
        (PRGMFLAG,ARMSTRONGPULLUP,DISARMSTRONGPULLUP,TERMINATEPULSE) = ('10','EF','ED','F1')
        RESET = ('C8','C9','CD')  # valid reset return codes
        # 1-wire device ROM commands and other operational commands...
        (MATCHROM,SEARCHROM,SKIPROM,READROM) = ('55','F0','CC','33')
        (CONVERT,RDSCRATCH,READ,WRITE) = ('44','BE','F5','5A')
        # DS18B20 codes
        (DURATIONCODE,CONVERTTEMP,READSCRATCHPAD) = ('3A','44','BE')
        # DS2408, DS2413, and DS28EA00 port operations
        (CAR,CAW,RPR,RAL,PAW) = ('F5','5A','F0','C3','A5')

        def __init__(self, params, log=None, config=DEFAULT_1WIRE_CONFIGURATION):
            oneWireIO.__init__(self, params, log)
            print('VERSION:', oneWire.VERSION + "/" + oneWireIO.VERSION + "/" +
    oneWirePhysical.VERSION)
            self.debug("# call:oneWire.__init__(%s,%s,%s)" % (str(params),log,config))
            self.scribe("##1-wire configuration initialization sequence: " + config)
            self.cmd(config)
            # check for presence pulse...
                # 0 - wire shorted, 1 - presence, 2 - alarm presence, 3 - none
            busDetect = self.rst()
            if busDetect == 0:
                print("ERROR: Bus short detected! Halted!")
                self.scribe("\n##!!! Initial 1-wire reset failure... !!!\n")
                self.dumpLog()
                exit(101)
            elif busDetect == 3:
                print("WARNING: No bus/devices detected!")
                self.scribe("\n##!!! Initial 1-wire reset failure... !!!\n")
                self.dumpLog()
            else:
                self.scribe('## OneWire initialized!')
                self.scribeMode = self.NORMAL   # default to normal

        # return all attributes for a sn
        def getAttributes(self, sn):
            f = self.getFamily(sn)
            wt = wDEVICES[f][1] if f in wDEVICES else None
            return (self.getDevice(sn), f, self.getIndex(sn),
              self.orderedSN(sn),self.piSN(sn),wt)

        # lookup device based on family code from serial number
        def getDevice(self, sn):
```

```
        family = sn[:2]
        if family in wDEVICES:
          return wDEVICES[family][0]
        else:
          return "???"

      # strip family code from serial number, least significant byte (first)
      def getFamily(self, sn):
        return sn[:2]

      # decimal index of programmable address, byte after family code.
      def getIndex(self, sn, bits=8):
          family = self.getFamily(sn)
          if family in wDEVICES and 'indexed' in wDEVICES[family]:
              return int(sn.replace(' ','')[2:4],16) & (1<<bits)-1
          return ''

      # mask bits of sn before calculating checksum for addressable devices.
      def maskSN(self, sn):
        if sn[:2]==DS28E04: # replace second byte of sn with 7F
          snHex = hexsplit(sn,2)
          snHex[1] = '7F'
          sn = ' '.join(snHex)
        return sn

      # split by hex pairs, reverse array order, join as space delimited string
      def orderedSN(self, sn):
        return ' '.join(hexsplit(sn,2)[::-1]) # sn msb to lsb in hex

      # sn in RPi format: <family> - <6 byte sn msb to lsb in hex> no crc, lower case
      def piSN(self, sn):
        pairs = hexsplit(sn,2)
        return '-'.join([pairs[0],''.join(pairs[6:0:-1])]).lower()

      # implement a binary search of the wire to detect devices matching the family
      #   a family of 00 matches all families, i.e. all devices.
      # Creates sn(s) as ordered hex pairs (byte, MS nibble first), LSB first, left to right,
      #   algorithm always takes the 0 branch first at any conflict.
      #   i.e. it discovers devices from left to right on the "binary tree".
      # The DS2480 supports an accelerator mode for search that automatically
      # reads the true state, complement state, and then writes the seed for each bit
      # DS2480B interleaves seed bits and returns sn interleaved with conflicts.
      #   sends:   64 bit seed interleaved with zeros as 128 bits
      #            translates to 192 OneWire bits (64 * true/complement/seed)
      #   returns: 64 bit serial number interleaved with 64 conflicts
      # this means a lot of bit twiddling in the search algorithm to encode/decode bitstreams
      def search(self, family='00'):
          self.debug("# call:search(%s)" % family)
          found = []  # return array of serial numbers
          # the following arrays serve as lookup tables for fast/simple encoding and decoding
search data.
          # seed2search expands a single hex char into 8 bits of one wire search data...
          seed2search =
('00','02','08','0A','20','22','28','2A','80','82','88','8A','A0','A2','A8','AA')
          # seach2data deconvolves one hex char of search response to 2 bits of sn
          search2data = (0,0,1,1,0,0,1,1,2,2,3,3,2,2,3,3)
```

```
        # search2conflict deconvolves one hex character of search response to 2 bits of conflict data
        search2conflict = ((0,0),(0,1),(0,0),(0,1),(1,0),(1,1),(1,0),(1,1),(0,0),(0,1),(0,0),(0,1),(1,0),(1,1),(1,0),(1,1))
        # seedhex builds the starting seed for the search
        # define starting seed, reverse order of family hex code followed by zeros -- reverse hex order
        seedHex = list(family[::-1] + '0'*14)
        # offsetHex flags algorithm that no conflicts before the first 2 characters matter.
        #  i.e. limits conflicts to second byte of sn or later, so only devices of specified family searched.
        offsetHex = 2 if family != '00' else 0

#        print("family: ", family)
#        print("seed: ", seedHex)
        while True:
            # The 64-bit (16 char) hex string seed must be formatted into
            # 16 bytes of hex data with interleaved 0's per DS2480B operation
            # data is least significant byte first, but MSB first per byte!
            searchHex = []
            for n in range(16):
                # process each hex character, least significant nibble to most significant
                searchHex.append(seed2search[int(seedHex[n],16)])

            # search pass using DS2480B acceleration mode; DS2480B reads an address bit and complement
            # DS2480B sends seed data when it detects a conflict, thus seed decides branch taken.
            self.rst()                                  # reset oneWire bus
            self.send(oneWire.SEARCHROM)                # put devices in search ROM mode
            self.cmd(oneWire.ACCELON, False)            # turn DS2480B acceleration mode on
            response = self.send(' '.join(searchHex))   # perform search
            self.cmd(oneWire.ACCELOFF, False)           # return DS2480B to normal mode
            self.rst()                                  # reset oneWire bus
            # deconvolve the SN and conflict data of the response...
            responseHex = list(response.replace(' ',''))
            snHex = []
            conflicts = ()
            for n in range(8):
                # each two bits consists of the SN bit and conflict data
                snHex.append("%02X" % (
                  (search2data[int(responseHex[4*n+2],16)]<<6) +
                  (search2data[int(responseHex[4*n+3],16)]<<4) +
                  (search2data[int(responseHex[4*n],16)]<<2) +
                  (search2data[int(responseHex[4*n+1],16)]) ))
                # 64 conflicts stored in reverse order, i.e. index 0 == MSB
                conflicts = (
                  search2conflict[int(responseHex[4*n+2],16)] +
                  search2conflict[int(responseHex[4*n+3],16)] +
                  search2conflict[int(responseHex[4*n+0],16)] +
                  search2conflict[int(responseHex[4*n+1],16)] +
                  conflicts )
            sn = ' '.join(snHex)
#            print("sn: ", sn, self.crc(sn), self.crc(self.maskSN(sn)))
#            print("conflicts: ", ''.join(map(str,conflicts)))
            # add valid devices (i.e. good CRC and "in-family") to found list
            # note: if no "in-family" devices exist, but other devices exist on oneWire network,
            #   then valid sn may pass CRC, but not included as out of family and search stops.
```

```
            if self.crc(self.maskSN(sn))==0 and (family=='00' or family==self.getFamily(sn)):
                found.append(sn)
            else:
                break # sn not -in-family or bad wire communications, so exit search.
            # evaluate conflict data msb to lsb to find the last conflict
            # don't need to check 8 least significant bits if family search
            (conflictHex,conflictBit) = (-1,-1)
            for n in range(64-(4*offsetHex)):
                if conflicts[n]!=0:
                    (conflictHex,conflictBit) = ((63-n) // 4,1<<((63-n) & 3))   # bit position of
last conflict
                    currentBranch = int(seedHex[conflictHex],16) & conflictBit  # get state of
the seed at this position
#                    print "CONFLICT: ", (conflictHex,conflictBit),currentBranch
                    # toggle the seed bit at this conflict:
                    #   0 means branch hasn't been taken, so take it.
                    #   1 means branch already taken, so reset it.
                    seedHex[conflictHex] = "%1X" % (int(seedHex[conflictHex],16) ^ conflictBit)
                    # if branch not yet tried then exit and try it, otherwise continue looking
for earlier conflicts.
                    if currentBranch==0:
                        break   # follow this branch
                    else:
                        (conflictHex,conflictBit) = (-1,-1) # ignore branch and continue looking
#            print("seed: ", seedHex, conflictHex)
            looking = (conflictHex > (offsetHex-1))
            if conflictHex < offsetHex: break
#        print("found: ", found)
        return found

    # convenience wrapper functions ...
    # converts raw temperature reading to Celsius
    def tempC(self, t):
        if t == None: return None
        return (t/16.0)

    # converts raw temperature reading to Fahrenheit
    def tempF(self, t):
        if t == None: return None
        return (9*t/80.0)+32

    # returns raw temperature of DS18B20 and DS28EA00 devices
    def temp(self, sn, units=None):
        self.debug("# call:temp(%s,%s)" % (sn,units))
        if self.getFamily(sn) in(DS18B20,DS28EA00):
            # setup conversion...
            self.rst()
            self.send(' '.join((oneWire.MATCHROM,sn)))
            self.cmd(' '.join((oneWire.DURATIONCODE,oneWire.ARMSTRONGPULLUP)))
            self.cmd(oneWire.TERMINATEPULSE)    # if not sent by itself, response delayed???
            self.send(oneWire.CONVERTTEMP)
            sleep(1.0)                          # wait for conversion to occur (with strong
pullup)
            self.cmd(' '.join((oneWire.DISARMSTRONGPULLUP,oneWire.TERMINATEPULSE)))
            # read the result...
            self.rst()
            self.send(' '.join((oneWire.MATCHROM,sn,oneWire.READSCRATCHPAD)))
```

```
            x = self.send(' '.join(['FF']*9))   # 9 bytes to return whole scratchpad with CRC
            self.debug("# temp[family:%s] scratchpad: %s" % (self.getFamily(sn), x))
            if self.crc(x)==0 and x!="00 00 00 00 00 00 00 00 00":
                # for valid result, strip temperature as first two bytes (4 hex chr), LSB first
                hv = '0x' + x[3:5] + x[0:2]
                v = int(hv,16)
                if v>=32768: v -= 65536         # correct for negative temperatures
                if units == 'F': return self.tempF(v)
                if units == 'C': return self.tempC(v)
                if units == 'X': return hv
                return (x, hv, v, self.tempC(v), self.tempF(v))
            return None
        else:
            self.scribe("# call:temp => UNKNOWN temperature device: " + sn)
            return None

    # low level port I/O functions...
    # retrieves the RAW (i.e. no inversion handling) port pin states
    def portIn(self, sn):
        self.scribe("# call:portIn(%s)" % (sn))
        family = self.getFamily(sn)
        mask = 0xFF if family == DS2408 else 0x03
        sequence = oneWire.CAR + ' FF'
        self.rst()
        response = hexsplit(self.send(' '.join((oneWire.MATCHROM,sn,sequence))),2)
        self.rst()
        checkOK = self.crc(self.maskSN(''.join(response[1:9])))==0
        data = mask # default
        if checkOK:
            data = int(response[-1],16)
            if family in (DS2413,DS28EA00):
                # DS2413 and DS28EA00 latch and port pin bits interleaved
                data = ((data & 0x04)>>1) + ((data & 0x01))
        else:
            self.scribe("# error:portIn")
            return 0xFFFF
        return data & mask
    # retrieves RAW (i.e. no inversion handling) latched port state data
    def portReadReg(self, sn):
        self.scribe("# call:portReadReg(%s)" % (sn))
        family = self.getFamily(sn)
        mask = 0xFF if family == DS2408 else 0x03
        if family == DS2408:
            sequence = oneWire.RPR + ' 89 00 FF'
        elif family == DS28E04:
            sequence = oneWire.RPR + ' 21 02 FF'
        elif family in (DS2413,DS28EA00):
            sequence = oneWire.CAR + ' FF'
        else:
            return None
        self.rst()
        response = hexsplit(self.send(' '.join((oneWire.MATCHROM,sn,sequence))),2)
        self.rst()
        checkOK = self.crc(self.maskSN(''.join(response[1:9])))==0
        data = mask # default
        if checkOK:
```

```
            data = int(response[-1],16)
            if family in (DS2413,DS28EA00):
                # DS2413 and DS28EA00 latch and port pin bits interleaved
                data = ((data & 0x08)>>2) + ((data & 0x02)>>1)
        else:
            self.scribe("# error:portReadReg")
            return 0xFFFF
        return data & mask

    # sets the RAW (i.e. no inversion handling) port latch state data
    def portWriteReg(self, sn, data):
        self.scribe("# call:portWriteReg(%s,0x%02X)" % (sn,data))
        family = self.getFamily(sn)
        mask = 0xFF if family == DS2408 else 0x03
        if type(data)=='str': data = int(data,16) # hex string to decimal
        # mask data and fill bits
        data = (data & mask) | (~mask & 0xFF)
        # write instruction different for DS28EA00
        sequence = oneWire.PAW if family == DS28EA00 else oneWire.CAW
        sequence += ' %02X %02X FF FF' % (data,data^0xFF)
        self.rst()
        response = hexsplit(self.send(' '.join((oneWire.MATCHROM,sn,sequence))),2)
        self.rst()
        checkOK = self.crc(self.maskSN(''.join(response[1:9])))==0
        if checkOK and (response[-2] == 'AA'):    # valid response.
            data = int(response[-1],16)
            if family in (DS2413,DS28EA00):
                # DS2413 and DS28EA00 latch and port pin bits interleaved
                data = ((data & 0x08)>>2) + ((data & 0x02)>>1)
        else:
            self.scribe("# error:portWriteReg")
            return 0xFFFF
        return data & mask

    # high-level DS2408, DS2413, or DS28EA00 port operations with error checking...
    # gets or sets port latch or pin data per operation. Assumes...
    #   !!! resolves inverting logic operation so that a 1 turns the output ON
    #   wIOIN and wIOREG do not change the port
    #   wIOSET, wIOCLEAR, and wIOTOGGLE operations preserve bits with 0 value and only change 1
bits
    #   wIORESET, wIOOUT do not preserve bits and write all bits of the port, both 0's and 1's.
    def port(self, sn, op, data=0xFF):
        self.scribe("# call:port(%s,%s,0x%02X)" % (sn,op,data))
        family = self.getFamily(sn)
        if family in (DS2408, DS2413, DS28EA00, DS28E04):
            REGMASK = 0xFF if family == DS2408 else 0x03
            if type(data)=='str': data = int(data,16) # hex string to decimal
            if op == wIORESET:
                data = self.portWriteReg(sn,REGMASK)
            elif op == wIOCLEAR:
                data = self.portWriteReg(sn, data | self.portReadReg(sn))
            elif op == wIOSET:
                data = self.portWriteReg(sn, ~data & self.portReadReg(sn))
            elif op == wIOIN:
                data = self.portIn(sn)
            elif op == wIOREG:
```

```
                data = self.portReadReg(sn)
            elif op == wIOOUT:
                data = self.portWriteReg(sn, ~data)  ##inverts?
            elif op == wIOTOGGLE:
                data = self.portWriteReg(sn, data  ^ self.portReadReg(sn))
            elif op == wIOPULSE:
                data = self.portWriteReg(sn, data ^ self.portWriteReg(sn, data  ^
self.portReadReg(sn)))
            else:
                self.scribe("# call:port => UNKNOWN port operation[" + sn + "]: " + str(op))
                return None
            return REGMASK & ~data
        else:
            self.scribe("# call:port => UNKNOWN port device: " + sn)
            return None

# wrapper class for access to wire device operations that do not require passing sn for calls...
# NOT ALL DEVICES CAPABLE OF ALL OPERATIONS! Error handling in base call.
class oneWireDevice(root):

    VERSION = "oneWireDevice:1.00"
    # for each object created define references for wire handle, sn, and family
    def __init__(self,hWIRE,sn,log=None):
        if log == None: log = 'oneWire.log'
        root.__init__(self,log,root.VERBOSE) # verbose mode until known good.
        self.debug("# call:oneWireDevice.__init__(%s,%s,%s)" % (str(hWIRE),sn,log))
        self.sn = sn
        self.hWIRE = hWIRE
        self.family = hWIRE.getfamily(sn)
        self.scribeMode = root.NORMAL

# wrapper functions to oneWire class, assume error handling in based call...
    # temperature measurement: units None, 'F', or 'C'
    def temp(self, units=None):
        self.debug("# call:oneWireDevice.temp(%s)" % (units,))
        return self.hWIRE.temp(self.sn,units)

    # port I/O
    def port(self, op, data=0xFF):
        self.debug("# call:oneWireDevice.port(%s)" % (op,data))
        return self.hWIRE.port(self.sn, op, data)
```