lib\onewire.py

```
# MIT License
# Derived from Adafruit CircuitPython OneWire library
"""
`onewire`
====================================================
Implements a full 1-Wire bus protocol supporting an extendible set of devices
OneWireBus includes direct interface to low level reset, bit read, and bit
write operations of the core onewireio module, as well as higher level-byte
and block (i.e. bytearray) read/write operations and device discovery (scan).
See readme for details

* Author(s): CanyonCasa
"""

#__version__ = "0.0.0-auto.0"
#__repo__ = "https://github.com/canyoncasa/OneWire.git"

from microcontroller import Pin
import onewireio
import digitalio
from time import monotonic as mark, sleep


class OneWireBus:
    """A class to represent a 1-Wire bus/pin."""
    # OneWire bus commands...
    SEARCH_ROM = 0xF0
    REGISTERED = {}     # Holds defined device types used to auto assign found devices

    def __init__(self, pin: Pin) -> None:
        self.pin = pin
        self.io = onewireio.OneWire(self.pin)
        self.timex = None

    # reset, read_bit, and write_bit functions wrapped because testing bus reassigns io
    def reset(self, test: bool=False) -> bool:
        """Perform a bus reset"""
        # patch until reset stuck low fix to onewireio reset
```

```
            if test:
                self.io.deinit()
                x = digitalio.DigitalInOut(self.pin)
                state = x.value
                x.deinit()
                self.io = onewireio.OneWire(self.pin)
                if not state:
                    return True # bus stuck low failure
            return self.io.reset()

        def readbit(self) -> bool:
            """Reads a single bit from the bus"""
            return self.io.read_bit()

        def writebit(self,bit) -> None:
            """Writes a single bit to bus"""
            self.io.write_bit(bit)

        @property
        def busy(self):
            """Reports whether or not bus is available or waiting on a conversion"""
            return not self.timex==None

        @property
        def ready(self):
            """Reports whether or not a pending conversion has completed and clears busy flag if done"""
            if mark() > self.timex:
                self.timex = None
            return self.timex==None

        def hold(self,wait=None):
            """Defines a hold time for initializing busy state"""
            if wait:
                self.timex = mark() + wait

        @staticmethod
        def crc8(data: bytearray) -> int:
            """Perform the 1-Wire CRC check on the provided data. Returns 0 for successful 8 byte crc"""
            if data==None:
                return None
            crc = 0
            for byte in data:
```

```
            crc ^= byte
            for _ in range(8):
                if crc & 0x01:
                    crc = (crc >> 1) ^ 0x8C
                else:
                    crc >>= 1
                crc &= 0xFF
        return crc

    @staticmethod
    def crc8snx(data: bytearray) -> int:
        """Perform a 1-wire CRC check on a SN with masking."""
        if OneWireBus.REGISTERED[data[0]]:
            if hasattr(OneWireBus.REGISTERED[data[0]],'MASK'):
                data[1] = OneWireBus.REGISTERED[data[0]].MASK
        return OneWireBus.crc8(data)

    @staticmethod
    def crc16i(data: bytearray, seed: int = 0, invert: bool=True, as_bytes: bool=True):
        """Generates CRC16; inverted, as bytearray, by default for appending to data block"""
        oddparity = [0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0]
        crc = seed
        for i in range(len(data)):
            cdata = data[i]
            cdata = (cdata ^ crc) & 0xFF
            crc = crc >> 8
            if oddparity[cdata & 0x0F] ^ oddparity[cdata >> 4]:
                crc = crc ^ 0xC001
            cdata = cdata << 6
            crc = crc ^ cdata
            cdata = cdata << 1
            crc = crc ^ cdata
        if invert:
            crc = crc ^ 0xFFFF
        return crc if not as_bytes else bytearray([crc&0xFF,(crc>>8)&0xFF])

    @staticmethod
    def crc16check(data: bytearray):
        """Checks a data block with an inverted crc16 and return zero for valid data"""
        crc = OneWireBus.crc16i(data, 0, False, False)
        return crc ^ 0xB001
```

```
    @staticmethod
    def bytes2hex(b: bytearray) -> str:
        """Converts a bytearray to defacto human readable hex string format"""
        return '' if not b else " ".join(list(map(lambda x: "{:02x}".format(x).upper(), b)))

    @staticmethod
    def bytes4addr(address: any) -> bytearray:
        """Converts an address of various forms to rom bytearray for I/O calls"""
        if type(address)==dict:    # assume direct from scan function
            return address['rom']
        elif type(address)==bytearray:
            return address
        elif type(address)==str:    # hex string with or w/o spaces or node-red format
            astr = address.replace(' ','').replace('-','')
            a = bytearray([int(astr[i:i+2],16) for i in range(0,len(astr),2)])
            if len(a)==7:
                a.append(OneWireBus.crc8snx(a))
            return a
        else:
            print('WARN[OneWireBus.bytes4addr]: unknown address type =>', address, type(address))
            return None

    @staticmethod
    def frmt_addr(rom) -> dict:
        """Converts a rom bytearray into multiple address forms for display, etc"""
        family = rom[0]
        sn = OneWireBus.bytes2hex(rom)    # defacto hex string
        hxx = sn.replace(' ','')    # hexstring, no spaces
        nr = (hxx[:2] + '-' + hxx[2:14]).lower()    # node-red format
        rev = ' '.join(sn.split(' ')[::-1]) # reverse order
        return {'family': family,'rom': rom, 'sn': sn, 'hex':hxx,'nodered': nr,'reverse': rev}

    def define_device(self, address, params: dict={}, dev_class=None):
        """Associates a specific device with its bus"""
        if not address:
            return Device(self,None,params)
        abytes = self.bytes4addr(address)
        if not abytes: return None
        family = abytes[0]
        dclass = dev_class if not dev_class==None else OneWireBus.REGISTERED.get(family,Device)
        return dclass(self, abytes, params)
```

```
    def read(self, n: int) -> bytearray:
        """Reads n number of bytes from the bus and returns as a bytearray."""
        buf = bytearray(n) if type(n)==int else n   # accomodate a buffer or buffer size
        for i in range(n):
            buf[i] = self.readbyte()
        return buf

    def readbyte(self) -> int:
        """Read a single byte from the data bus"""
        val = 0
        for i in range(8):
            val |= self.readbit() << i
        return val

    @staticmethod
    def register(family: int, device_class):
        """Adds a device class to the list of registered device classes"""
        OneWireBus.REGISTERED[family] = device_class

    def scan(self,family=None) -> list:
        """Scan bus for devices present and return a list of valid multi-format addresses for each."""
           # if family defined, searches only for devices matching that family.
           # searches "zero branch" first; i.e. left-to-right binary tree
           # walks back conflicts from MSB to LSB
        addresses = []
        seed = bytearray([0]*8) # start with all zeros
        if family:  # override first byte with family if defined
            seed[0] = int(family,16)
        while True: # loop until no conflicts remain...
            rom, conflicts = self._search_rom(seed) # perform a single pass to discover a single device
            if rom==None:
                break
            if conflicts==None: # conflicts==None for errors!
                print(f"ERROR[OneWireBus.scan]: NO devices present of stuck bus!")
                break
            if not self.crc8snx(rom): # rom as bytearray; zero crc for a valid address
                dev = self.frmt_addr(rom)
                addresses.append(dev)
            else:
                print(f"ERROR[OneWireBus.scan]: failed CRC! Device {dev['sn']} ignored")
                break
            if not len(conflicts):  # no more conflicts, done!
```

```
                break
            else:
                while len(conflicts):
                    (byte,bit) = conflicts[-1]  # last conflict (i.e. closest to MSB)
                    if family and byte==0: # no more devices within family
                        return addresses
                    sbit = seed[byte] & 1<<bit and 1
                    if sbit==0: # 1 path not yet taken, toggle seed bit and try again
                        seed[byte] ^= 1<<bit
                        break   # conflicts loop
                    else:
                        conflicts.pop() # 1 branch taken already, try an earlier conflict
                if len(conflicts)==0:   # taking 1 branch if len>0, else exhausted conflicts
                    break
        return addresses

    def status(self,dump=False):
        """Reports on bus health; optionally dumps to console"""
        r = self.reset(True)
        if dump:
            print(f"Reset: {('Bus OK', 'Bus fault/no devices')[r]}")
            print('Registered devices...')
            for f in sorted(OneWireBus.REGISTERED.keys()):
                print(f'  {f:02}:   {OneWireBus.REGISTERED[f].DESC}')
        return r

    def write(self, buf: bytearray) -> None:
        """Write the bytes from ``buf`` to the bus."""
        for i in range(len(buf)):
            self.writebyte(buf[i])

    def writebyte(self, value: int) -> None:
        """Writes a single byte of data to the bus"""
        for i in range(8):
            bit = (value >> i) & 0x1
            self.writebit(bit)

    def _search_rom(self, seed: bytearray) -> tuple: # (bytearray, array)
        """Internal routine used by scan for device discovery; works like DS2480B w/o interleaving"""
        if self.reset(True):    # False when any devices present; tests for stuck bus
            return None, None
        # set search mode: read true bit; read complement bit; write true bit or seed bit for a conflict
```

```
            self.writebyte(OneWireBus.SEARCH_ROM)
            rom = bytearray(8)
            conflicts = []
            for byte in range(8):
                r_b = 0
                for bit in range(8):
                    n = byte*8 + bit
                    sb = seed[byte] & 1<<bit and 1  # seed bit
                    b = self.readbit()  # 1st address bit read (true)
                    if self.readbit():  # 2nd address bit read (complement)
                        if b:  # 11: there are no devices or there is an error on the bus
                            return None, None
                    else:
                        if not b:  # 00: collision, two devices with different bit states
                            conflicts += [(byte,bit)]    # mark conflict
                            b = sb  # replace true bit with seed
                    self.writebit(b)
                    r_b |= b << bit
                rom[byte] = r_b
            return rom, conflicts


    class Device:
        """A base class to represent any single device on the 1-Wire bus, generally overridden by specific class."""
        TYPE = 'bus'
        DESC = 'Generic device, supports bus search'
        FAMILY = None
        MATCH_ROM = 0x55
        SKIP_ROM = 0xCC

        def __init__(self, bus: OneWireBus, address: any, params: dict={}):
            self.bus = bus  # resident of bus
            if address:
                self.address = OneWireBus.bytes4addr(address)   # save address as bytearray
                self.family = self.address[0]  # extract family code
                self.sn = self.bus.bytes2hex(self.address)  # de facto hex string format
            else:
                self.address = None
                self.family = Device.FAMILY
                self.sn = None
            self.desc = params.get('desc',Device.DESC)
```

```
    # select a single device on the bus
    def select(self, skip=False) -> None:
        if not self.address: return
        if not self.bus.reset():
            if skip:
                self.bus.write([Device.SKIP_ROM])
            else:
                self.bus.write([Device.MATCH_ROM])
                self.bus.write(self.address)

    def search(self,family=None):
        found = self.bus.scan(family if family else self.family)
        return found

    def info(self):
        return { 'address': self.address, 'family':self.family, 'sn': self.sn, 'desc': self.desc }

    @staticmethod
    def register(family: int, device_class):
        OneWireBus.register(family, device_class)
```