#!/usr/bin/env python import logging import itertools import struct import platform import serial import time import sys #from Arduino import m328 from Arduino import m328p as uK from serial.tools import list_ports from pygments.styles import arduino from libxml2mod import parent if platform.system() == 'Windows': import _winreg as winreg else: import glob

#logging.basicConfig(level=logging.DEBUG) logging.basicConfig(level=logging.ERROR) log = logging.getLogger(**name**)

""" Arduino UNO board pinout """ pin_name = [0,1,2,3,4,5,6,7, #portD 0,1,2,3,4,5, #portB 0,1,2,3,4,5] #portC pin_port = [uK.PORTD,uK.PORTD,uK.PORTD,uK.PORTD,uK.PORTD,uK.PORTD,uK.PORTD,uK.PORTD, uK.PORTB,uK.PORTB,uK.PORTB,uK.PORTB,uK.PORTB,uK.PORTB, #portB uK.PORTC,uK.PORTC,uK.PORTC,uK.PORTC,uK.PORTC,uK.PORTC] #portC """ Proces commands for Arduino UART API """ PROCES_RESET = 0x00 READ_REGISTER = 0x10 SET_REGISTER = 0x20 SET_REGISTER_BIT = 0x30 CLR_REGISTER_BIT = 0x40 READ_REGISTER_BIT = 0x50 WAIT_UNTIL_BIT_IS_SET = 0x60 WAIT_UNTIL_BIT_IS_CLEARED = 0x70 READ_16_BIT_REGISTER_INCR_ADDR = 0x80 READ_16_BIT_REGISTER_DECR_ADDR = 0x90 REPEAT_CMD_BUFFER = 0xA0 SET_DATA = 0xB0

def enumerate_serial_ports(): """ Uses the Win32 registry to return a iterator of serial (COM) ports existing on this computer. """ path = 'HARDWARE\DEVICEMAP\SERIALCOMM' try: key = winreg.OpenKey(winreg.HKEY_LOCAL_MACHINE, path) except WindowsError: raise Exception

```
for i in itertools.count():
    try:
        val = winreg.EnumValue(key, i)
        yield (str(val[1]))  # , str(val[0]))
    except EnvironmentError:
        break
```

def build_cmd_str(cmd, args=None): """ Build a command string that can be sent to the arduino.

```
Input:
    cmd (str): the command to send to the arduino, must not
        contain a % character
    args (iterable): the arguments to send to the command

@TODO: a strategy is needed to escape % characters in the args
"""
if args:
    args = '%'.join(map(str, args))
else:
    args = ''
return "@{cmd}%{args}$!".format(cmd=cmd, args=args)
```

def find_port(baud, timeout): """ Find the first port that is connected to an arduino with a compatible sketch installed. """ if platform.system() == 'Windows': ports = enumerate_serial_ports() elif platform.system() == 'Darwin': ports = [i[0] for i in list_ports.comports()] else: ports = glob.glob("/dev/ttyUSB*") + glob.glob("/dev/ttyACM*") for p in ports: log.debug('Found {0}, testing...'.format(p)) try: sr = serial.Serial(p, baud, timeout=timeout) except (serial.serialutil.SerialException, OSError) as e: log.debug(str(e)) continue time.sleep(2) version = get_version(sr) if version != 6 : #'version': # Davidtle moram dat hex 6 log.debug('Bad version {0}. This is not a Shrimp/Arduino!'.format(version)) sr.close() continue log.info('Using port {0}.'.format(p)) if sr: return sr return None

def get_version(sr): cmd_str = build_cmd_str("version") try: packet = bytearray() packet.append(PROCES_RESET) """ packet.append(PROCES_RESET) Although the communication is designe to send two bytes in a packet we send youst one in get_version... The uK proces detect one missing byte and resets the firmware... """ sr.write(packet) sr.flush() except Exception: return None ver = sr.read() return int(int.from_bytes(ver,byteorder='big'))

class Arduino:

```
def __init__(self, baud=115200, port=None, timeout=2, sr=None):
    """
    Initializes serial communication with Arduino if no connection is
    given. Attempts to self-select COM port, if not specified.
    """
    if not sr:
        if not port:
            sr = find_port(baud, timeout)
            if not sr:
                raise ValueError("Could not find port.")
        else:
            sr = serial.Serial(port, baud, timeout=timeout)
    print('Arduino on: ',sr.port,'@',sr.baudrate,'bps')
    time.sleep(0.10)
    sr.flush()
    self.sr = sr
    self.cmd_buffer_num = 0
    self.cmd_do_buffer_num = 0
    self.cmd_loop_buffer_num = 0
    self.F_CPU=16
    self.TimerOne=Timer(self)

"""
#####################################################################
##       Misc FUNCTIONS
#####################################################################
```

```
"""

def sendAPICmd(self, cmd_str):
    cmd_api_str = bytearray()
    for cmd_byte in cmd_str:
        cmd_api_str.append(cmd_byte)
    try:
        self.sr.write(cmd_api_str)
        self.sr.flush()
        self.cmd_buffer_num += len(cmd_str)
        if self.cmd_buffer_num > 255:
            self.cmd_buffer_num -= 256
    except:
        print("Unexpected error:", sys.exc_info()[0])
        pass

def version(self):
    return get_version(self.sr)

def softwareReset(self):
    cmd_string = bytearray()
    cmd_string.append(PROCES_RESET)
    try:
        self.sendAPICmd(cmd_string)
        if self.sr.inWaiting()>0 :
            self.sr.flushInput()
        rd = self.sr.read()
        x = int.from_bytes(rd,byteorder='big', signed=False)
        return 0
    except:
        return -1

def cmdDo(self):
    self.cmd_do_buffer_num = self.cmd_buffer_num

def cmdLoop(self):
    if self.cmd_do_buffer_num <= self.cmd_buffer_num:
        Repeat_last_cmd_buffer_num = self.cmd_buffer_num - self.cmd_do_buffer_nu
    else:
        Repeat_last_cmd_buffer_num = self.cmd_buffer_num + (256 - self.cmd_do_bu
    try:
        cmd_string = bytearray()
        cmd_string.append(REPEAT_CMD_BUFFER)
        cmd_string.append(Repeat_last_cmd_buffer_num)
        self.sendAPICmd(cmd_string)
    except:
```

```
            pass

    def registerWrite(self, reg_name, reg_value):
        log.debug('registerWrite: ' +str(reg_name) +'=' +str(reg_value) )
        try:
            cmd_string = bytearray()
            cmd_string.append(SET_DATA)
            cmd_string.append(reg_value)
            cmd_string.append(SET_REGISTER)
            cmd_string.append(reg_name)
            self.sendAPICmd(cmd_string)
        except:
            pass

    def registerRead(self, reg_name):
        log.debug('registerRead: ' + str(reg_name))
        try:
            cmd_string = bytearray()
            cmd_string.append(READ_REGISTER)
            cmd_string.append(reg_name)
            self.sendAPICmd(cmd_string)
            if self.sr.inWaiting()>0 :
                self.sr.flushInput()
            rd = self.sr.read()
            x = int.from_bytes(rd,byteorder='big', signed=False)
            return int(x)
        except:
            pass

    def registerRead16b(self, reg_name):
        log.debug('registerRead: ' + str(reg_name))
        try:
            cmd_string = bytearray()
            cmd_string.append(READ_16_BIT_REGISTER_INCR_ADDR)
            cmd_string.append(reg_name)
            self.sendAPICmd(cmd_string)
            if self.sr.inWaiting()>0 :
                self.sr.flushInput()
            rd = self.sr.read(2)
            x = int.from_bytes(rd,byteorder='little', signed=False)
            return int(x)
        except:
            pass

    def writeRegisterBit(self, bit_name, reg_name, bit_value):
        log.debug('Write Register.Bit: ' +str(reg_name) +'.' +str(bit_name) +'=' + s
```

```
        cmd_string = bytearray()
        if bit_value==1 or bit_value =='HIGH':
            cmd_string.append(SET_REGISTER_BIT +bit_name)
        else:
            cmd_string.append(CLR_REGISTER_BIT +bit_name)
        try:
            cmd_string.append(reg_name)
            self.sendAPICmd(cmd_string)
        except:
            pass

    def readADC(self):
        """
        Execute the ANALOG READ of pre-set channel
        and returns the VALUE of ADC conversion.
        Warning:
            ADC channel MUST be preset eather with:
                - analogRead function or
                - setting the ADMUX register
        """
        log.debug('readADC: ' )
        if self.sr.inWaiting()>0 :
            self.sr.flushInput()
        cmd_str = bytearray()
        cmd_str.append(SET_REGISTER_BIT+6)          # start ADC conversion
        cmd_str.append(uK.ADCSRA)
        cmd_str.append(WAIT_UNTIL_BIT_IS_CLEARED+6) # wait ADC to complete conwersio
        cmd_str.append(uK.ADCSRA)
        cmd_str.append(READ_16_BIT_REGISTER_INCR_ADDR)
        cmd_str.append(uK.ADCL)
        try:
            self.sendAPICmd(cmd_str)
            rd= self.sr.readline(2)
            x = int.from_bytes(rd,byteorder='little', signed=False)
            return int(x)
        except:
            log.error('readADC not executed.')

    def waitUntilBitIsSet(self, bit_num, *reg_name):
        """
        Arduino will not procede with execution of instrructions UNTIL
        the corespondet bit will be SET to 1.
        Meanwhile all other instructions send by computer will be saved
        into the hardware buffer.
        """
        cmd_str = bytearray()
```

```
        if len(reg_name) > 0:
            #ok it is more advances ... reg and bit
            log.debug('hardware bit wait to be set: bit=' +str(bit_num)+' reg=' +str
            cmd_str.append(WAIT_UNTIL_BIT_IS_SET+bit_num)
# start ADC conversion
            cmd_str.append(reg_name[0])
        else:
            # Arduino pinout
            log.debug('hardware bit wait to be set: Arduino pinout=' +str(bit_num))
            log.debug('hardware bit wait to be set: bit=' +str(pin_name[bit_num])+'
            cmd_str.append(WAIT_UNTIL_BIT_IS_SET+pin_name[bit_num])
            cmd_str.append(pin_port[bit_num]-2)
        try:
            self.sendAPICmd(cmd_str)
        except:
            log.error('waitUntilBitIsSet not executed.')

    def waitUntilBitIsCleared(self, bit_num, *reg_name):
        """
        Arduino will not procede with execution of instrructions UNTIL
        the corespondet bit will be SET to 0 (cleared).
        Meanwhile all other instructions send by computer will be saved
        into the hardware buffer.
        """
        cmd_str = bytearray()
        if len(reg_name) > 0:
            #ok it is more advances ... reg and bit
            log.debug('hardware bit wait to be set: bit=' +str(bit_num)+' reg=' +str
            cmd_str.append(WAIT_UNTIL_BIT_IS_CLEARED+bit_num)
# start ADC conversion
            cmd_str.append(reg_name[0])
        else:
            # Arduino pinout
            log.debug('hardware bit wait to be set: Arduino pinout=' +str(bit_num))
            log.debug('hardware bit wait to be set: bit=' +str(pin_name[bit_num])+'
            cmd_str.append(WAIT_UNTIL_BIT_IS_CLEARED+pin_name[bit_num])
            cmd_str.append(pin_port[bit_num]-2)
        try:
            self.sendAPICmd(cmd_str)
        except:
            log.error('waitUntilBitIsSet not executed.')

    """
    ##########################################################################
    ##      ArduinoIDE-like FUNCTIONS
    ##########################################################################
```

```
"""

def pinMode(self, pin, val):
    """
    Sets I/O mode of pin
    inputs:
       pin: pin number to toggle
       val: "INPUT" or "OUTPUT"
    """
    cmd_pin = pin_name[pin]
    cmd_port= pin_port[pin]-1 #ddrX = portx - 1

    if val == "OUTPUT":
        cmd_pin += SET_REGISTER_BIT
    elif val == "INPUT_PULLUP":
        cmd_pin += CLR_REGISTER_BIT
    else: #INPUT - default option
        cmd_pin += CLR_REGISTER_BIT
    try:
        cmd_string = bytearray()
        cmd_string.append(cmd_pin)
        cmd_string.append(cmd_port)
        if val == "INPUT_PULLUP":
            cmd_string.append(pin_name[pin]+SET_REGISTER_BIT)
            cmd_string.append(pin_port[pin])
        self.sendAPICmd(cmd_string)
    except:
        pass

def digitalWrite(self, pin, val):
    """
    Sends digitalWrite command
    to digital pin on Arduino
    ----------
    inputs:
       pin : digital pin number
       val : either "HIGH" or "LOW"
    """
    cmd_pin = pin_name[pin]
    cmd_port= pin_port[pin]
    cmd_string = bytearray()

    if val == "LOW" or val == 0:
        cmd_pin += CLR_REGISTER_BIT
    else:
        cmd_pin += SET_REGISTER_BIT
```

```
        try:
            cmd_string = bytearray()
            cmd_string.append(cmd_pin)
            cmd_string.append(cmd_port)
            self.sendAPICmd(cmd_string)
            self.sr.flush()
        except:
            pass

    def digitalRead(self, pin):
        """
        Returns the value of a specified
        digital pin.
        inputs:
           pin : digital pin number for measurement
        returns:
           value: 0 for "LOW", 1 for "HIGH"
        """
        cmd_string = bytearray()
        cmd_string.append(pin_name[pin] + READ_REGISTER_BIT)
        cmd_string.append(pin_port[pin] - 2)
        try:
            if self.sr.inWaiting()>0 :
                self.sr.flushInput()
            self.sendAPICmd(cmd_string)
            rd = self.sr.read()
            x = int.from_bytes(rd,byteorder='big')
            return int(x)
        except:
            return -10

    def analogRead(self, pin):
        """
        Returns 10-bit ADC value of
        a specified analog pin.
        ----------
        inputs:
           pin : analog pin number for measurement
        returns:
           value: integer from 0 to 1023
        """
        if self.sr.inWaiting()>0 :
            self.sr.flushInput()
        cmd_str = bytearray()
        cmd_str.append(SET_DATA)
        cmd_str.append(0x40+pin)
```

```
        cmd_str.append(SET_REGISTER)
        cmd_str.append(uK.ADMUX)

        cmd_str.append(SET_DATA)
        cmd_str.append(0xC7)
        cmd_str.append(SET_REGISTER)
        cmd_str.append(uK.ADCSRA)

        cmd_str.append(WAIT_UNTIL_BIT_IS_CLEARED+6)
        cmd_str.append(uK.ADCSRA)
        cmd_str.append(READ_16_BIT_REGISTER_INCR_ADDR)
        cmd_str.append(uK.ADCL)
        try:
            self.sendAPICmd(cmd_str)
            self.sr.flush()
        except:
            pass
        rd= self.sr.readline(2)
        x = int.from_bytes(rd,byteorder='little', signed=False)
        try:
            return int(x)
        except:
            return 0

    def analogWrite(self, pin, val):
        """
        Sends analogWrite pwm command
        to pin on Arduino#https://github.com/PaulStoffregen/TimerOne/blob/master/Tim
        ------------
        inputs:
           pin : pin number
           val : integer 0 (off) to 255 (always on)
        """
        if val > 255:
            val = 255
        elif val < 0:
            val = 0
        cmd_str = build_cmd_str("aw", (pin, val))
        try:
            self.sr.write(cmd_str)
            self.sr.flush()
        except:
            pass

    def test():
        def init():
```

```
        log.debug('initialize..')
        #self.softwareReset()
        self.digitalWrite(13, 1)
```

class Timer: #https://github.com/PaulStoffregen/TimerOne/blob/master/TimerOne.h
def **init**(self, parent): log.debug('TimerOne added to ARduino...') self.parent
= parent self.TIMER_RESOLUTION = 65536
self.prescaler_clock_select_bits = 0 self.tccr1a=0 self.tccr1b=0

```
def initialize(self, microseconds=1000000):
    log.debug('initialize..')
    self.tccr1b = uK.WGM13
    self.parent.registerWrite(uK.TCCR1B, self.tccr1b) #set WGM13 -> set mode as p
    self.tccr1a = 0x00
    self.parent.registerWrite(uK.TCCR1A, self.tccr1a) # clear
    self.setPeriod(microseconds)

def setPeriod(self,microseconds):
    cycles = (self.parent.F_CPU / 2) * microseconds
    pwmPeriod = int(cycles)
    cmd_str = bytearray()
    if cycles < self.TIMER_RESOLUTION:
        self.prescaler_clock_select_bits = 1<<uK.CS10
        pwmPeriod = cycles
    elif cycles < (self.TIMER_RESOLUTION * 8):
        self.prescaler_clock_select_bits = 1<<uK.CS11
        pwmPeriod = cycles / 8;
    elif cycles < (self.TIMER_RESOLUTION * 64):
        self.prescaler_clock_select_bits = 1<<uK.CS11 | 1<<uK.CS10
        pwmPeriod = cycles / 64;
    elif cycles < (self.TIMER_RESOLUTION * 256):
        self.prescaler_clock_select_bits = 1<<uK.CS12
        pwmPeriod = cycles / 256
    elif cycles < (self.TIMER_RESOLUTION * 1024):
        self.prescaler_clock_select_bits = 1<<uK.CS12 | 1<<uK.CS10
        pwmPeriod = cycles / 1024
    else:
        self.prescaler_clock_select_bits = 1<<uK.CS12 | 1<<uK.CS10
        log.error("Timer One period time is to long. Max value is 8.39s = 838860
        pwmPeriod = self.TIMER_RESOLUTION - 1

    #ICR1 = pwmPeriod;
    pwmPeriod_in_bytes = int(pwmPeriod).to_bytes(16, "little", signed=False)
    self.parent.registerWrite(uK.ICR1H, pwmPeriod_in_bytes[1])
    log.debug('wrifying ICR1H ... = ' + str(self.parent.registerRead(uK.ICR1H)))
    self.parent.registerWrite(uK.ICR1L, pwmPeriod_in_bytes[0])
```

```
        log.debug('wrifying ICR1L ... = ' + str(self.parent.registerRead(uK.ICR1L)))



        #timer1 Start
        self.tccr1b = 1<<uK.WGM13 | self.prescaler_clock_select_bits
        self.parent.registerWrite(uK.TCCR1B, self.tccr1b)

        #ICR1 = pwmPeriod;
        #pwmPeriod_in_bytes = int(pwmPeriod).to_bytes(16, "little", signed=False)
        #cmd_str.append(SET_DATA)
        #cmd_str.append(pwmPeriod_in_bytes[0])
        #cmd_str.append(SET_REGISTER)
        #cmd_str.append(uK.ICR1L)
        #cmd_str.append(SET_DATA)
        #cmd_str.append(pwmPeriod_in_bytes[1])
        #cmd_str.append(SET_REGISTER)
        #cmd_str.append(uK.ICR1H)
        #self.parent.sendAPICmd(cmd_str)
        log.debug('wrifying ICR1L ... = ' + str(self.parent.registerRead(uK.ICR1L)))
        log.debug('wrifying ICR1H ... = ' + str(self.parent.registerRead(uK.ICR1H)))
        log.debug('Check TIMER1 ... = ' + str(self.parent.registerRead16b(uK.TCNT1L)
        log.debug('Check TIMER1 ... = ' + str(self.parent.registerRead16b(uK.TCNT1L)

    def start(self):
        self.parent.registerWrite(uK.TCCR1B,self.tccr1b)

    def stop(self):
        self.parent.registerWrite(uK.TCCR1B,0x00)
```