from machine import Pin #**File Name: SMTP.py**

button = Pin(27, Pin.IN, Pin.PULL\_UP)

import ConnectWiFi

ConnectWiFi.connect()

import umail

while True:

if not button.value(): #We press the button

smtp = umail.SMTP('smtp.gmail.com', 587, username='esp77524@gmail.com', password='jidpwiebanyadfyu')

smtp.to('esp77524@gmail.com')

smtp.send("This is a ESP32 Generated SMTP message!")

smtp.quit()

def connect(): #**ConnectWiFi.py**

import network

ssid = "KGM"

password = "nalini\_raghavendra\_195863"

station = network.WLAN(network.STA\_IF)

if station.isconnected() == True:

print("Already connected")

return

station.active(True)

station.connect(ssid, password)

while station.isconnected() == False:

pass

print("Connection successful")

print(station.ifconfig())

# uMail (MicroMail) for MicroPython **#umail.py**

# Copyright (c) 2018 Shawwwn <shawwwn1@gmai.com>

# License: MIT

import usocket

DEFAULT\_TIMEOUT = 10 # sec

LOCAL\_DOMAIN = '127.0.0.1'

CMD\_EHLO = 'EHLO'

CMD\_STARTTLS = 'STARTTLS'

CMD\_AUTH = 'AUTH'

CMD\_MAIL = 'MAIL'

AUTH\_PLAIN = 'PLAIN'

AUTH\_LOGIN = 'LOGIN'

class SMTP:

def cmd(self, cmd\_str):

sock = self.\_sock;

sock.write('%s\r\n' % cmd\_str)

resp = []

next = True

while next:

code = sock.read(3)

next = sock.read(1) == b'-'

resp.append(sock.readline().strip().decode())

return int(code), resp

def \_\_init\_\_(self, host, port, ssl=False, username=None, password=None):

import ussl

self.username = username

addr = usocket.getaddrinfo(host, port)[0][-1]

sock = usocket.socket(usocket.AF\_INET, usocket.SOCK\_STREAM)

sock.settimeout(DEFAULT\_TIMEOUT)

print('%s refused, %s' % (addr, resp))

count += 1

assert count!=len(addrs), 'recipient refused, %d, %s' % (code, resp)

code, resp = self.cmd('DATA')

assert code==354, 'data refused, %d, %s' % (code, resp)

return code, resp

def write(self, content):

self.\_sock.write(content)

def send(self, content=''):

if content:

self.write(content)

self.\_sock.write('\r\n.\r\n') # the five letter sequence marked for ending

line = self.\_sock.readline()

return (int(line[:3]), line[4:].strip().decode())

def quit(self):

self.cmd("QUIT")

self.\_sock.close()

sock.connect(addr)

if ssl:

sock = ussl.wrap\_socket(sock)

code = int(sock.read(3))

sock.readline()

assert code==220, 'cant connect to server %d, %s' % (code, resp)

self.\_sock = sock

code, resp = self.cmd(CMD\_EHLO + ' ' + LOCAL\_DOMAIN)

assert code==250, '%d' % code

if CMD\_STARTTLS in resp:

code, resp = self.cmd(CMD\_STARTTLS)

assert code==220, 'start tls failed %d, %s' % (code, resp)

self.\_sock = ussl.wrap\_socket(sock)

if username and password:

self.login(username, password)

def login(self, username, password):

self.username = username

code, resp = self.cmd(CMD\_EHLO + ' ' + LOCAL\_DOMAIN)

assert code==250, '%d, %s' % (code, resp)

auths = None

for feature in resp:

if feature[:4].upper() == CMD\_AUTH:

auths = feature[4:].strip('=').upper().split()

assert auths!=None, "no auth method"

from ubinascii import b2a\_base64 as b64

if AUTH\_PLAIN in auths:

cren = b64("\0%s\0%s" % (username, password))[:-1].decode()

code, resp = self.cmd('%s %s %s' % (CMD\_AUTH, AUTH\_PLAIN, cren))

elif AUTH\_LOGIN in auths:

code, resp = self.cmd("%s %s %s" % (CMD\_AUTH, AUTH\_LOGIN, b64(username)[:-1].decode()))

assert code==334, 'wrong username %d, %s' % (code, resp)

code, resp = self.cmd(b64(password)[:-1].decode())

else:

raise Exception("auth(%s) not supported " % ', '.join(auths))

assert code==235 or code==503, 'auth error %d, %s' % (code, resp)

return code, resp

def to(self, addrs, mail\_from=None):

mail\_from = self.username if mail\_from==None else mail\_from

code, resp = self.cmd(CMD\_EHLO + ' ' + LOCAL\_DOMAIN)

assert code==250, '%d' % code

code, resp = self.cmd('MAIL FROM: <%s>' % mail\_from)

assert code==250, 'sender refused %d, %s' % (code, resp)

if isinstance(addrs, str):

addrs = [addrs]

count = 0

for addr in addrs:

code, resp = self.cmd('RCPT TO: <%s>' % addr)

if code!=250 and code!=251:

# Helpers for generating BLE advertising payloads.

from micropython import const

import struct

import bluetooth

# Advertising payloads are repeated packets of the following form:

# 1 byte data length (N + 1)

# 1 byte type (see constants below)

# N bytes type-specific data

\_ADV\_TYPE\_FLAGS = const(0x01)

\_ADV\_TYPE\_NAME = const(0x09)

\_ADV\_TYPE\_UUID16\_COMPLETE = const(0x3)

\_ADV\_TYPE\_UUID32\_COMPLETE = const(0x5)

\_ADV\_TYPE\_UUID128\_COMPLETE = const(0x7)

\_ADV\_TYPE\_UUID16\_MORE = const(0x2)

\_ADV\_TYPE\_UUID32\_MORE = const(0x4)

\_ADV\_TYPE\_UUID128\_MORE = const(0x6)

\_ADV\_TYPE\_APPEARANCE = const(0x19)

# Generate a payload to be passed to gap\_advertise(adv\_data=...).

def advertising\_payload(limited\_disc=False, br\_edr=False, name=None, services=None, appearance=0):

payload = bytearray()

def \_append(adv\_type, value):

nonlocal payload

payload += struct.pack("BB", len(value) + 1, adv\_type) + value

\_append(

\_ADV\_TYPE\_FLAGS,

struct.pack("B", (0x01 if limited\_disc else 0x02) + (0x18 if br\_edr else 0x04)),

)

if name:

\_append(\_ADV\_TYPE\_NAME, name)

if services:

for uuid in services:

b = bytes(uuid)

if len(b) == 2:

\_append(\_ADV\_TYPE\_UUID16\_COMPLETE, b)

elif len(b) == 4:

\_append(\_ADV\_TYPE\_UUID32\_COMPLETE, b)

elif len(b) == 16:

\_append(\_ADV\_TYPE\_UUID128\_COMPLETE, b)

# See org.bluetooth.characteristic.gap.appearance.xml

if appearance:

\_append(\_ADV\_TYPE\_APPEARANCE, struct.pack("<h", appearance))

return payload

def decode\_field(payload, adv\_type):

i = 0

result = []

while i + 1 < len(payload):

if payload[i + 1] == adv\_type:

result.append(payload[i + 2 : i + payload[i] + 1])

i += 1 + payload[i]

return result

def decode\_name(payload):

n = decode\_field(payload, \_ADV\_TYPE\_NAME)

return str(n[0], "utf-8") if n else ""

def decode\_services(payload):

services = []

for u in decode\_field(payload, \_ADV\_TYPE\_UUID16\_COMPLETE):

services.append(bluetooth.UUID(struct.unpack("<h", u)[0]))

for u in decode\_field(payload, \_ADV\_TYPE\_UUID32\_COMPLETE):

services.append(bluetooth.UUID(struct.unpack("<d", u)[0]))

for u in decode\_field(payload, \_ADV\_TYPE\_UUID128\_COMPLETE):

services.append(bluetooth.UUID(u))

return services

def demo():

payload = advertising\_payload(

name="micropython",

services=[bluetooth.UUID(0x181A), bluetooth.UUID("6E400001-B5A3-F393-E0A9-E50E24DCCA9E")],

)

print(payload)

print(decode\_name(payload))

print(decode\_services(payload))

if \_\_name\_\_ == "\_\_main\_\_":

demo()

#MicroPython SSD1306 OLED driver, I2C and SPI interfaces created by Adafruit **ssd1306.py**

**Main\_OLED.py**

# Complete project details at https://RandomNerdTutorials.com

from machine import Pin, SoftI2C

import ssd1306

from time import sleep

# ESP32 Pin assignment

i2c = SoftI2C(scl=Pin(22), sda=Pin(21))

# ESP8266 Pin assignment

#i2c = SoftI2C(scl=Pin(5), sda=Pin(4))

oled\_width = 128

oled\_height = 64

oled = ssd1306.SSD1306\_I2C(oled\_width, oled\_height, i2c)

oled.text('Hello, World 1!', 0, 0)

oled.text('Hello, World 2!', 0, 10)

oled.text('Hello, World 3!', 0, 20)

oled.show()

import time

import framebuf

# register definitions

SET\_CONTRAST = const(0x81)

SET\_ENTIRE\_ON = const(0xa4)

SET\_NORM\_INV = const(0xa6)

SET\_DISP = const(0xae)

SET\_MEM\_ADDR = const(0x20)

SET\_COL\_ADDR = const(0x21)

SET\_PAGE\_ADDR = const(0x22)

SET\_DISP\_START\_LINE = const(0x40)

SET\_SEG\_REMAP = const(0xa0)

SET\_MUX\_RATIO = const(0xa8)

SET\_COM\_OUT\_DIR = const(0xc0)

SET\_DISP\_OFFSET = const(0xd3)

SET\_COM\_PIN\_CFG = const(0xda)

SET\_DISP\_CLK\_DIV = const(0xd5)

SET\_PRECHARGE = const(0xd9)

SET\_VCOM\_DESEL = const(0xdb)

SET\_CHARGE\_PUMP = const(0x8d)

class SSD1306:

def \_\_init\_\_(self, width, height, external\_vcc):

self.width = width

self.height = height

self.external\_vcc = external\_vcc

self.pages = self.height // 8

# Note the subclass must initialize self.framebuf to a framebuffer.

# This is necessary because the underlying data buffer is different

# between I2C and SPI implementations (I2C needs an extra byte).

self.poweron()

self.init\_display()

def init\_display(self):

for cmd in (

SET\_DISP | 0x00, # off

# address setting

SET\_MEM\_ADDR, 0x00, # horizontal

# resolution and layout

SET\_DISP\_START\_LINE | 0x00,

SET\_SEG\_REMAP | 0x01, # column addr 127 mapped to SEG0

SET\_MUX\_RATIO, self.height - 1,

SET\_COM\_OUT\_DIR | 0x08, # scan from COM[N] to COM0

SET\_DISP\_OFFSET, 0x00,

SET\_COM\_PIN\_CFG, 0x02 if self.height == 32 else 0x12,

# timing and driving scheme

SET\_DISP\_CLK\_DIV, 0x80,

SET\_PRECHARGE, 0x22 if self.external\_vcc else 0xf1,

SET\_VCOM\_DESEL, 0x30, # 0.83\*Vcc

# display

SET\_CONTRAST, 0xff, # maximum

SET\_ENTIRE\_ON, # output follows RAM contents

SET\_NORM\_INV, # not inverted

# charge pump

SET\_CHARGE\_PUMP, 0x10 if self.external\_vcc else 0x14,

SET\_DISP | 0x01): # on

self.write\_cmd(cmd)

self.fill(0)

self.show()

# hardware I2C interfaces.

self.i2c.writeto(self.addr, self.buffer)

def poweron(self):

pass

class SSD1306\_SPI(SSD1306):

def \_\_init\_\_(self, width, height, spi, dc, res, cs, external\_vcc=False):

self.rate = 10 \* 1024 \* 1024

dc.init(dc.OUT, value=0)

res.init(res.OUT, value=0)

cs.init(cs.OUT, value=1)

self.spi = spi

self.dc = dc

self.res = res

self.cs = cs

self.buffer = bytearray((height // 8) \* width)

self.framebuf = framebuf.FrameBuffer1(self.buffer, width, height)

super().\_\_init\_\_(width, height, external\_vcc)

def write\_cmd(self, cmd):

self.spi.init(baudrate=self.rate, polarity=0, phase=0)

self.cs.high()

self.dc.low()

self.cs.low()

self.spi.write(bytearray([cmd]))

self.cs.high()

def write\_framebuf(self):

self.spi.init(baudrate=self.rate, polarity=0, phase=0)

self.cs.high()

self.dc.high()

self.cs.low()

self.spi.write(self.buffer)

self.cs.high()

def poweron(self):

self.res.high()

time.sleep\_ms(1)

self.res.low()

time.sleep\_ms(10)

self.res.high()

def poweroff(self):

self.write\_cmd(SET\_DISP | 0x00)

def contrast(self, contrast):

self.write\_cmd(SET\_CONTRAST)

self.write\_cmd(contrast)

def invert(self, invert):

self.write\_cmd(SET\_NORM\_INV | (invert & 1))

def show(self):

x0 = 0

x1 = self.width - 1

if self.width == 64:

# displays with width of 64 pixels are shifted by 32

x0 += 32

x1 += 32

self.write\_cmd(SET\_COL\_ADDR)

self.write\_cmd(x0)

self.write\_cmd(x1)

self.write\_cmd(SET\_PAGE\_ADDR)

self.write\_cmd(0)

self.write\_cmd(self.pages - 1)

self.write\_framebuf()

def fill(self, col):

self.framebuf.fill(col)

def pixel(self, x, y, col):

self.framebuf.pixel(x, y, col)

def scroll(self, dx, dy):

self.framebuf.scroll(dx, dy)

def text(self, string, x, y, col=1):

self.framebuf.text(string, x, y, col)

class SSD1306\_I2C(SSD1306):

def \_\_init\_\_(self, width, height, i2c, addr=0x3c, external\_vcc=False):

self.i2c = i2c

self.addr = addr

self.temp = bytearray(2)

# Add an extra byte to the data buffer to hold an I2C data/command byte

# to use hardware-compatible I2C transactions. A memoryview of the

# buffer is used to mask this byte from the framebuffer operations

# (without a major memory hit as memoryview doesn't copy to a separate

# buffer).

self.buffer = bytearray(((height // 8) \* width) + 1)

self.buffer[0] = 0x40 # Set first byte of data buffer to Co=0, D/C=1

self.framebuf = framebuf.FrameBuffer1(memoryview(self.buffer)[1:], width, height)

super().\_\_init\_\_(width, height, external\_vcc)

def write\_cmd(self, cmd):

self.temp[0] = 0x80 # Co=1, D/C#=0

self.temp[1] = cmd

self.i2c.writeto(self.addr, self.temp)

def write\_framebuf(self):

# Blast out the frame buffer using a single I2C transaction to support

**microcontro\_lab\_LCD.py**

import machine

from machine import Pin, SoftI2C

from lcd\_api import LcdApi

from i2c\_lcd import I2cLcd

from time import sleep

I2C\_ADDR = 0x27

totalRows = 2

totalColumns = 16

i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=10000) #initializing the I2C method for ESP32

#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000) #initializing the I2C method for ESP8266

lcd = I2cLcd(i2c, I2C\_ADDR, totalRows, totalColumns)

while True:

lcd.putstr("I2C LCD Tutorial")

sleep(2)

lcd.clear()

lcd.move\_to(0,1)

lcd.putstr("Lets Count 0-10!")

sleep(2)

lcd.clear()

for i in range(11):

lcd.move\_to(0,1)

lcd.putstr(str(i))

sleep(1)

lcd.clear()

---------------------------------------------------------------------------------------------------------------

**main\_LCD\_16X2.py**

import machine

from machine import SoftI2C, Pin

from lcd\_api import LcdApi

from i2c\_lcd import I2cLcd

I2C\_ADDR = 0x27

totalRows = 2

totalColumns = 16

i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=10000) #initializing the I2C method for ESP32

#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000) #initializing the I2C method for ESP8266

lcd = I2cLcd(i2c, I2C\_ADDR, totalRows, totalColumns)

heart = bytearray([0x00,0x00,0x1B,0x1F,0x1F,0x0E,0x04,0x00])

face = bytearray([0x00,0x00,0x0A,0x00,0x11,0x0E,0x00,0x00])

lcd.custom\_char(0, heart)

lcd.custom\_char(1, face)

lcd.putstr(chr(0)+" ESP32 with I2C LCD "+chr(1))

----------------------------------------------------------------------------------------------------------------

import machine

from machine import Pin, SoftI2C

while True:

lcd.putstr("Om Namo Venkatesaya!")

sleep(2)

lcd.clear()

lcd.putstr("Hello World!")

sleep(2)

lcd.clear()

from lcd\_api import LcdApi

from i2c\_lcd import I2cLcd

from time import sleep

I2C\_ADDR = 0x27

totalRows = 2

totalColumns = 16

i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=10000) #I2C for ESP32

#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000) #I2C for ESP8266

lcd = I2cLcd(i2c, I2C\_ADDR, totalRows, totalColumns)

**microcontro\_lab\_LCD\_1.py**

import machine

from machine import SoftI2C, Pin

from lcd\_api import LcdApi

from i2c\_lcd import I2cLcd

I2C\_ADDR = 0x27

totalRows = 2

totalColumns = 16

i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=10000) #initializing the I2C method for ESP32

#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000) #initializing the I2C method for ESP8266

lcd = I2cLcd(i2c, I2C\_ADDR, totalRows, totalColumns)

heart = bytearray([0x00,0x00,0x1B,0x1F,0x1F,0x0E,0x04,0x00])

face = bytearray([0x00,0x00,0x0A,0x00,0x11,0x0E,0x00,0x00])

lcd.custom\_char(0, heart)

lcd.custom\_char(1, face)

lcd.putstr(chr(0)+" ESP32 with I2C LCD "+chr(1))

-----------------------------------------------------------------------------------------------------------------

"""Provides an API for talking to HD44780 compatible character LCDs."""

**‘’’lcd\_api.py’’’**

import time

class LcdApi:

"""Implements the API for talking with HD44780 compatible character LCDs.

This class only knows what commands to send to the LCD, and not how to get

them to the LCD.

It is expected that a derived class will implement the hal\_xxx functions.

"""

# The following constant names were lifted from the avrlib lcd.h

# header file, however, I changed the definitions from bit numbers

# to bit masks.

#

# HD44780 LCD controller command set

LCD\_CLR = 0x01 # DB0: clear display

LCD\_HOME = 0x02 # DB1: return to home position

LCD\_ENTRY\_MODE = 0x04 # DB2: set entry mode

LCD\_ENTRY\_INC = 0x02 # --DB1: increment

LCD\_ENTRY\_SHIFT = 0x01 # --DB0: shift

LCD\_ON\_CTRL = 0x08 # DB3: turn lcd/cursor on

LCD\_ON\_DISPLAY = 0x04 # --DB2: turn display on

LCD\_ON\_CURSOR = 0x02 # --DB1: turn cursor on

LCD\_ON\_BLINK = 0x01 # --DB0: blinking cursor

LCD\_MOVE = 0x10 # DB4: move cursor/display

LCD\_MOVE\_DISP = 0x08 # --DB3: move display (0-> move cursor)

LCD\_MOVE\_RIGHT = 0x04 # --DB2: move right (0-> left)

LCD\_FUNCTION = 0x20 # DB5: function set

LCD\_FUNCTION\_8BIT = 0x10 # --DB4: set 8BIT mode (0->4BIT mode)

LCD\_FUNCTION\_2LINES = 0x08 # --DB3: two lines (0->one line)

LCD\_FUNCTION\_10DOTS = 0x04 # --DB2: 5x10 font (0->5x7 font)

LCD\_FUNCTION\_RESET = 0x30 # See "Initializing by Instruction" section

LCD\_CGRAM = 0x40 # DB6: set CG RAM address

LCD\_DDRAM = 0x80 # DB7: set DD RAM address

LCD\_RS\_CMD = 0

LCD\_RS\_DATA = 1

LCD\_RW\_WRITE = 0

LCD\_RW\_READ = 1

def \_\_init\_\_(self, num\_lines, num\_columns):

self.num\_lines = num\_lines

if self.num\_lines > 4:

self.num\_lines = 4

self.num\_columns = num\_columns

if self.num\_columns > 40:

self.num\_columns = 40

self.cursor\_x = 0

self.cursor\_y = 0

self.implied\_newline = False

self.backlight = True

self.display\_off()

self.backlight\_on()

self.clear()

self.hal\_write\_command(self.LCD\_ENTRY\_MODE | self.LCD\_ENTRY\_INC)

self.hide\_cursor()

self.display\_on()

def clear(self):

"""Clears the LCD display and moves the cursor to the top left

corner.

"""

self.hal\_write\_command(self.LCD\_CLR)

self.hal\_write\_command(self.LCD\_HOME)

self.cursor\_x = 0

self.cursor\_y = 0

def show\_cursor(self):

"""Causes the cursor to be made visible."""

self.hal\_write\_command(self.LCD\_ON\_CTRL | self.LCD\_ON\_DISPLAY |

self.LCD\_ON\_CURSOR)

def hide\_cursor(self):

"""Causes the cursor to be hidden."""

self.hal\_write\_command(self.LCD\_ON\_CTRL | self.LCD\_ON\_DISPLAY)

def blink\_cursor\_on(self):

"""Turns on the cursor, and makes it blink."""

self.hal\_write\_command(self.LCD\_ON\_CTRL | self.LCD\_ON\_DISPLAY |

self.LCD\_ON\_CURSOR | self.LCD\_ON\_BLINK)

def blink\_cursor\_off(self):

"""Turns on the cursor, and makes it no blink (i.e. be solid)."""

self.hal\_write\_command(self.LCD\_ON\_CTRL | self.LCD\_ON\_DISPLAY |

self.LCD\_ON\_CURSOR)

def display\_on(self):

"""Turns on (i.e. unblanks) the LCD."""

self.hal\_write\_command(self.LCD\_ON\_CTRL | self.LCD\_ON\_DISPLAY)

def display\_off(self):

"""Turns off (i.e. blanks) the LCD."""

self.hal\_write\_command(self.LCD\_ON\_CTRL)

def backlight\_on(self):

as chr(0) through chr(7).

"""

location &= 0x7

self.hal\_write\_command(self.LCD\_CGRAM | (location << 3))

self.hal\_sleep\_us(40)

for i in range(8):

self.hal\_write\_data(charmap[i])

self.hal\_sleep\_us(40)

self.move\_to(self.cursor\_x, self.cursor\_y)

def hal\_backlight\_on(self):

"""Allows the hal layer to turn the backlight on.

If desired, a derived HAL class will implement this function.

"""

pass

def hal\_backlight\_off(self):

"""Allows the hal layer to turn the backlight off.

If desired, a derived HAL class will implement this function.

"""

pass

def hal\_write\_command(self, cmd):

"""Write a command to the LCD.

It is expected that a derived HAL class will implement this

function.

"""

raise NotImplementedError

def hal\_write\_data(self, data):

"""Write data to the LCD.

It is expected that a derived HAL class will implement this

function.

"""

raise NotImplementedError

def hal\_sleep\_us(self, usecs):

"""Sleep for some time (given in microseconds)."""

time.sleep\_us(usecs)

"""Turns the backlight on.

This isn't really an LCD command, but some modules have backlight

controls, so this allows the hal to pass through the command.

"""

self.backlight = True

self.hal\_backlight\_on()

def backlight\_off(self):

"""Turns the backlight off.

This isn't really an LCD command, but some modules have backlight

controls, so this allows the hal to pass through the command.

"""

self.backlight = False

self.hal\_backlight\_off()

def move\_to(self, cursor\_x, cursor\_y):

"""Moves the cursor position to the indicated position. The cursor

position is zero based (i.e. cursor\_x == 0 indicates first column).

"""

self.cursor\_x = cursor\_x

self.cursor\_y = cursor\_y

addr = cursor\_x & 0x3f

if cursor\_y & 1:

addr += 0x40 # Lines 1 & 3 add 0x40

if cursor\_y & 2: # Lines 2 & 3 add number of columns

addr += self.num\_columns

self.hal\_write\_command(self.LCD\_DDRAM | addr)

def putchar(self, char):

"""Writes the indicated character to the LCD at the current cursor

position, and advances the cursor by one position.

"""

if char == '\n':

if self.implied\_newline:

# self.implied\_newline means we advanced due to a wraparound,

# so if we get a newline right after that we ignore it.

pass

else:

self.cursor\_x = self.num\_columns

else:

self.hal\_write\_data(ord(char))

self.cursor\_x += 1

if self.cursor\_x >= self.num\_columns:

self.cursor\_x = 0

self.cursor\_y += 1

self.implied\_newline = (char != '\n')

if self.cursor\_y >= self.num\_lines:

self.cursor\_y = 0

self.move\_to(self.cursor\_x, self.cursor\_y)

def putstr(self, string):

"""Write the indicated string to the LCD at the current cursor

position and advances the cursor position appropriately.

"""

for char in string:

self.putchar(char)

def custom\_char(self, location, charmap):

"""Write a character to one of the 8 CGRAM locations, available

**i2c\_lcd.py**

import utime

import gc

from lcd\_api import LcdApi

from machine import I2C

# PCF8574 pin definitions

MASK\_RS = 0x01 # P0

MASK\_RW = 0x02 # P1

MASK\_E = 0x04 # P2

SHIFT\_BACKLIGHT = 3 # P3

SHIFT\_DATA = 4 # P4-P7

class I2cLcd(LcdApi):

#Implements a HD44780 character LCD connected via PCF8574 on I2C

def \_\_init\_\_(self, i2c, i2c\_addr, num\_lines, num\_columns):

(((cmd >> 4) & 0x0f) << SHIFT\_DATA))

self.i2c.writeto(self.i2c\_addr, bytes([byte | MASK\_E]))

self.i2c.writeto(self.i2c\_addr, bytes([byte]))

byte = ((self.backlight << SHIFT\_BACKLIGHT) |

((cmd & 0x0f) << SHIFT\_DATA))

self.i2c.writeto(self.i2c\_addr, bytes([byte | MASK\_E]))

self.i2c.writeto(self.i2c\_addr, bytes([byte]))

if cmd <= 3:

# The home and clear commands require a worst case delay of 4.1 msec

utime.sleep\_ms(5)

gc.collect()

def hal\_write\_data(self, data):

# Write data to the LCD. Data is latched on the falling edge of E.

byte = (MASK\_RS |

(self.backlight << SHIFT\_BACKLIGHT) |

(((data >> 4) & 0x0f) << SHIFT\_DATA))

self.i2c.writeto(self.i2c\_addr, bytes([byte | MASK\_E]))

self.i2c.writeto(self.i2c\_addr, bytes([byte]))

byte = (MASK\_RS |

(self.backlight << SHIFT\_BACKLIGHT) |

((data & 0x0f) << SHIFT\_DATA))

self.i2c.writeto(self.i2c\_addr, bytes([byte | MASK\_E]))

self.i2c.writeto(self.i2c\_addr, bytes([byte]))

gc.collect()

self.i2c = i2c

self.i2c\_addr = i2c\_addr

self.i2c.writeto(self.i2c\_addr, bytes([0]))

utime.sleep\_ms(20) # Allow LCD time to powerup

# Send reset 3 times

self.hal\_write\_init\_nibble(self.LCD\_FUNCTION\_RESET)

utime.sleep\_ms(5) # Need to delay at least 4.1 msec

self.hal\_write\_init\_nibble(self.LCD\_FUNCTION\_RESET)

utime.sleep\_ms(1)

self.hal\_write\_init\_nibble(self.LCD\_FUNCTION\_RESET)

utime.sleep\_ms(1)

# Put LCD into 4-bit mode

self.hal\_write\_init\_nibble(self.LCD\_FUNCTION)

utime.sleep\_ms(1)

LcdApi.\_\_init\_\_(self, num\_lines, num\_columns)

cmd = self.LCD\_FUNCTION

if num\_lines > 1:

cmd |= self.LCD\_FUNCTION\_2LINES

self.hal\_write\_command(cmd)

gc.collect()

def hal\_write\_init\_nibble(self, nibble):

# Writes an initialization nibble to the LCD.

# This particular function is only used during initialization.

byte = ((nibble >> 4) & 0x0f) << SHIFT\_DATA

self.i2c.writeto(self.i2c\_addr, bytes([byte | MASK\_E]))

self.i2c.writeto(self.i2c\_addr, bytes([byte]))

gc.collect()

def hal\_backlight\_on(self):

# Allows the hal layer to turn the backlight on

self.i2c.writeto(self.i2c\_addr, bytes([1 << SHIFT\_BACKLIGHT]))

gc.collect()

def hal\_backlight\_off(self):

#Allows the hal layer to turn the backlight off

self.i2c.writeto(self.i2c\_addr, bytes([0]))

gc.collect()

def hal\_write\_command(self, cmd):

# Write a command to the LCD. Data is latched on the falling edge of E.

byte = ((self.backlight << SHIFT\_BACKLIGHT) |

import machine

import time

import ujson as json

from machine import I2C

# Pin connections

i2c = I2C(scl=machine.Pin(22), sda=machine.Pin(21), freq=100000)

address = 0x27

# Initialize the LCD

def lcd\_init():

i2c.writeto(address, b'\x38\x39\x14\x70\x56\x6c')

time.sleep\_ms(200)

i2c.writeto(address, b'\x38\x0c\x01')

# Send string to LCD

def lcd\_string(string, row, col):

cmd = b'\x80' if row == 0 else b'\xc0' if row == 1 else b'\x94' if row == 2 else b'\xd4'

cmd += bytes([col])

i2c.writeto(address, cmd)

def lcd\_init():

i2c = I2C(scl=machine.Pin(22), sda=machine.Pin(21))

i2c.writeto(ADDRESS, b'\x38\x39\x14\x70\x56\x6c', stop=False)

time.sleep\_ms(200)

i2c.writeto(ADDRESS, b'\x38\x0c\x01', stop=False)

time.sleep\_ms(2)

# Send command to LCD

def lcd\_command(command):

i2c = I2C(scl=machine.Pin(22), sda=machine.Pin(21))

i2c.writeto(ADDRESS, ustruct.pack('B', command))

# Send string to LCD

def lcd\_string(string):

i2c = I2C(scl=machine.Pin(22), sda=machine.Pin(21))

i2c.writeto(ADDRESS, string.encode('ascii'))

# Set cursor position

def lcd\_set\_cursor(row, col):

row\_offsets = [0x00, 0x40, 0x14, 0x54]

lcd\_command(LCD\_SET\_DDRAM\_ADDRESS | (col + row\_offsets[row]))

lcd\_init()

lcd\_string("Hello, World!")

lcd\_set\_cursor(1, 0)

lcd\_string("This is line 2")

for char in string:

i2c.writeto(address, bytes([ord(char)]))

import machine

import time

import ustruct

from machine import I2C

# I2C address of the LCD

ADDRESS = 0x27

# LCD commands

LCD\_CLEAR\_DISPLAY = 0x01

LCD\_RETURN\_HOME = 0x02

LCD\_ENTRY\_MODE\_SET = 0x04

LCD\_DISPLAY\_CONTROL = 0x08

LCD\_CURSOR\_SHIFT = 0x10

LCD\_FUNCTION\_SET = 0x20

LCD\_SET\_CGRAM\_ADDRESS = 0x40

LCD\_SET\_DDRAM\_ADDRESS = 0x80

# LCD flags for display entry mode

LCD\_ENTRY\_RIGHT = 0x00

LCD\_ENTRY\_LEFT = 0x02

LCD\_ENTRY\_SHIFT\_INCREMENT = 0x01

LCD\_ENTRY\_SHIFT\_DECREMENT = 0x00

# LCD flags for display on/off control

LCD\_DISPLAY\_ON = 0x04

LCD\_DISPLAY\_OFF = 0x00

LCD\_CURSOR\_ON = 0x02

LCD\_CURSOR\_OFF = 0x00

LCD\_BLINK\_ON = 0x01

LCD\_BLINK\_OFF = 0x00

# LCD flags for display/cursor shift

LCD\_DISPLAY\_MOVE = 0x08

LCD\_CURSOR\_MOVE = 0x00

LCD\_MOVE\_RIGHT = 0x04

LCD\_MOVE\_LEFT = 0x00

# Initialize the LCD

**LCD\_chtGPT\_2004.py**

**20X4.py This is a Program working with ESP32 – 20X4 LCD**

**Working Program**

import machine **Date: 09-04-2023**

from machine import Pin, SoftI2C Files on the Device (ESP32)

from lcd\_api import LcdApi 1. lcd\_api.py

from i2c\_lcd import I2cLcd 2. I2c\_lcd.py

from time import sleep

I2C\_ADDR = 0x27

totalRows = 4

totalColumns = 20

i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=100000) #I2C for ESP32

#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000) #I2C for ESP8266

lcd = I2cLcd(i2c, I2C\_ADDR, totalRows, totalColumns)

while True:

lcd.putstr("Linuxhint.com")

sleep(5)

lcd.clear()

#sleep(10)

lcd.move\_to(0,1)

lcd.putstr("Om Namo Venkatesaya!")

sleep(5)

lcd.clear()

lcd.move\_to(0,2)

lcd.putstr("OmBhagavatheVasudev!")

sleep(5)

lcd.clear()

lcd.move\_to(0,3)

lcd.putstr("Jai Sreemannarayana!")

sleep(5)

import machine

from time import sleep\_ms

from machine import I2C

# Define LCD I2C address and other constants

LCD\_ADDR = 0x27

LCD\_WIDTH = 20

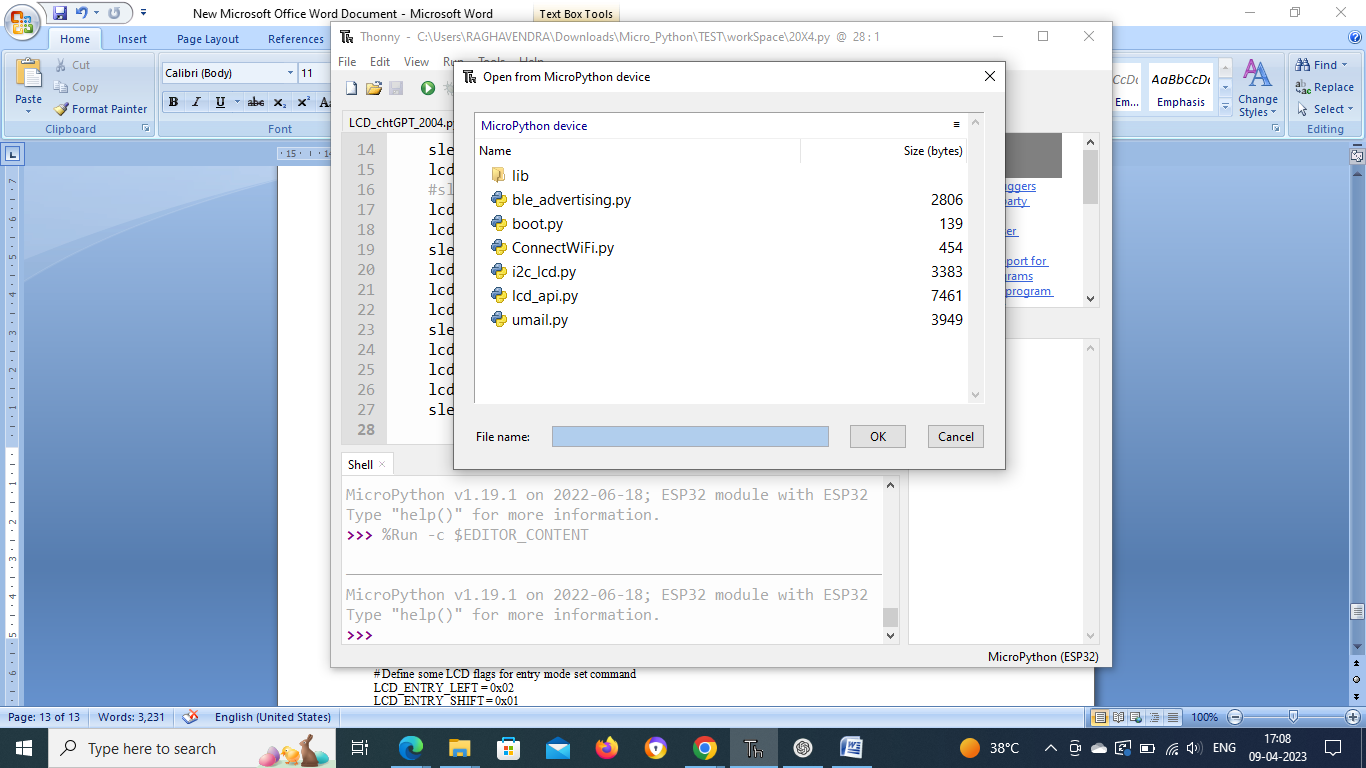
LCD\_HEIGHT = 4

# Define some LCD commands

LCD\_CLEAR\_DISPLAY = 0x01

LCD\_RETURN\_HOME = 0x02

LCD\_ENTRY\_MODE\_SET = 0x04



LCD\_DISPLAY\_CONTROL = 0x08

LCD\_CURSOR\_SHIFT = 0x10

LCD\_FUNCTION\_SET = 0x20

LCD\_SET\_CGRAM\_ADDR = 0x40

LCD\_SET\_DDRAM\_ADDR = 0x80

# Define some LCD flags for function set command

LCD\_8BIT\_MODE = 0x10

LCD\_2LINE\_MODE = 0x08

LCD\_FONT\_5X8 = 0x00

# Define some LCD flags for display control command

LCD\_DISPLAY\_ON = 0x04

LCD\_CURSOR\_ON = 0x02

LCD\_BLINK\_ON = 0x01

# Define some LCD flags for entry mode set command

LCD\_ENTRY\_LEFT = 0x02

LCD\_ENTRY\_SHIFT = 0x01

# Define I2C bus and LCD object

i2c = I2C(0, scl=machine.Pin(22), sda=machine.Pin(21), freq=100000)

lcd = machine.I2C\_LCD(i2c, LCD\_ADDR, LCD\_WIDTH, LCD\_HEIGHT)

# Initialize LCD

lcd.command(LCD\_FUNCTION\_SET | LCD\_2LINE\_MODE | LCD\_FONT\_5X8)

lcd.command(LCD\_DISPLAY\_CONTROL | LCD\_DISPLAY\_ON)

lcd.command(LCD\_ENTRY\_MODE\_SET | LCD\_ENTRY\_LEFT)

lcd.command(LCD\_CLEAR\_DISPLAY)

# Write some text to LCD

lcd.puts('Hello, world!')

sleep\_ms(1000)

lcd.puts('MicroPython +')

lcd.gotoxy(0, 1)

lcd.puts('20x4 I2C LCD')

sleep\_ms(2000)

# Clear LCD and exit

lcd.command(LCD\_CLEAR\_DISPLAY)

lcd.command(LCD\_RETURN\_HOME)

**I2C\_ADDR\_16X2\_20X4**

import machine

**20X4.py**

import machine

from machine import Pin, SoftI2C

from lcd\_api import LcdApi

from i2c\_lcd import I2cLcd

from time import sleep

I2C\_ADDR = 0x27

totalRows = 4

totalColumns = 20

i2c = SoftI2C(scl=Pin(22), sda=Pin(21), freq=100000) #I2C for ESP32

#i2c = I2C(scl=Pin(5), sda=Pin(4), freq=10000) #I2C for ESP8266

lcd = I2cLcd(i2c, I2C\_ADDR, totalRows, totalColumns)

while True:

lcd.putstr("Om Namo Venkatesaya!")

sleep(2)

lcd.clear()

#sleep(10)

lcd.move\_to(0,1)

lcd.putstr("BhagvatheVasudevaya!")

sleep(2)

lcd.clear()

lcd.move\_to(0,2)

lcd.putstr("Hare Sreenivasa!")

sleep(2)

lcd.clear()

lcd.move\_to(0,3)

lcd.putstr("Lakshmi Venkatesaya!")

sleep(2)

lcd.clear()

sdaPIN=machine.Pin(21) #for ESP32

sclPIN=machine.Pin(22)

i2c=machine.I2C(sda=sdaPIN, scl=sclPIN, freq=10000)

devices = i2c.scan()

if len(devices) == 0:

print("No i2c device!")

else:

print('i2c devices found:',len(devices))

for device in devices:

print("At address: ",hex(device))

**WiFi.py**

import network

ssid = 'KGM'

password = 'nalini\_raghavendra\_195863'

sta\_if = network.WLAN(network.STA\_IF)

sta\_if.active(True)

sta\_if.connect(ssid, password)

while not sta\_if.isconnected():

pass

print('Connected to Wi-Fi')

print('IP address:', sta\_if.ifconfig()[0])

**Blinky.py**

import machine

import time

led = machine.Pin(2, machine.Pin.OUT)

while True:

led.on()

time.sleep(1)

led.off()

time.sleep(1)

**PWM\_SERVO.py**

import machine

import time

servo = machine.PWM(machine.Pin(2), freq=50)

while True:

servo.duty(40)

time.sleep(1)

servo.duty(90)

time.sleep(1)

servo.duty(140)

time.sleep(1)

**Sensor.py**

import machine

import time

sensor = machine.ADC(machine.Pin(36))

while True:

value = sensor.read()

print(value)

time.sleep(1)

**#MicroPython SSD1306 OLED driver, I2C and SPI interfaces created by Adafruit**

import time

import framebuf

# register definitions

SET\_CONTRAST = const(0x81)

SET\_ENTIRE\_ON = const(0xa4)

SET\_NORM\_INV = const(0xa6)

SET\_DISP = const(0xae)

SET\_MEM\_ADDR = const(0x20)

SET\_COL\_ADDR = const(0x21)

SET\_PAGE\_ADDR = const(0x22)

SET\_DISP\_START\_LINE = const(0x40)

SET\_SEG\_REMAP = const(0xa0)

SET\_MUX\_RATIO = const(0xa8)

SET\_COM\_OUT\_DIR = const(0xc0)

SET\_DISP\_OFFSET = const(0xd3)

SET\_COM\_PIN\_CFG = const(0xda)

SET\_DISP\_CLK\_DIV = const(0xd5)

SET\_PRECHARGE = const(0xd9)

SET\_VCOM\_DESEL = const(0xdb)

SET\_CHARGE\_PUMP = const(0x8d)

class SSD1306:

def \_\_init\_\_(self, width, height, external\_vcc):

self.width = width

self.height = height

self.external\_vcc = external\_vcc

self.pages = self.height // 8

# Note the subclass must initialize self.framebuf to a framebuffer.

# This is necessary because the underlying data buffer is different

# between I2C and SPI implementations (I2C needs an extra byte).

self.poweron()

self.init\_display()

def init\_display(self):

for cmd in (

SET\_DISP | 0x00, # off

# address setting

SET\_MEM\_ADDR, 0x00, # horizontal

# resolution and layout

SET\_DISP\_START\_LINE | 0x00,

SET\_SEG\_REMAP | 0x01, # column addr 127 mapped to SEG0

SET\_MUX\_RATIO, self.height - 1,

SET\_COM\_OUT\_DIR | 0x08, # scan from COM[N] to COM0

SET\_DISP\_OFFSET, 0x00,

SET\_COM\_PIN\_CFG, 0x02 if self.height == 32 else 0x12,

# timing and driving scheme

SET\_DISP\_CLK\_DIV, 0x80,

SET\_PRECHARGE, 0x22 if self.external\_vcc else 0xf1,

SET\_VCOM\_DESEL, 0x30, # 0.83\*Vcc

# display

SET\_CONTRAST, 0xff, # maximum

SET\_ENTIRE\_ON, # output follows RAM contents

SET\_NORM\_INV, # not inverted

# charge pump

SET\_CHARGE\_PUMP, 0x10 if self.external\_vcc else 0x14,

SET\_DISP | 0x01): # on

self.write\_cmd(cmd)

self.fill(0)

self.show()

# hardware I2C interfaces.

self.i2c.writeto(self.addr, self.buffer)

def poweron(self):

pass

class SSD1306\_SPI(SSD1306):

def \_\_init\_\_(self, width, height, spi, dc, res, cs, external\_vcc=False):

self.rate = 10 \* 1024 \* 1024

dc.init(dc.OUT, value=0)

res.init(res.OUT, value=0)

cs.init(cs.OUT, value=1)

self.spi = spi

self.dc = dc

self.res = res

self.cs = cs

self.buffer = bytearray((height // 8) \* width)

self.framebuf = framebuf.FrameBuffer1(self.buffer, width, height)

super().\_\_init\_\_(width, height, external\_vcc)

def write\_cmd(self, cmd):

self.spi.init(baudrate=self.rate, polarity=0, phase=0)

self.cs.high()

self.dc.low()

self.cs.low()

self.spi.write(bytearray([cmd]))

self.cs.high()

def write\_framebuf(self):

self.spi.init(baudrate=self.rate, polarity=0, phase=0)

self.cs.high()

self.dc.high()

self.cs.low()

self.spi.write(self.buffer)

self.cs.high()

def poweron(self):

self.res.high()

time.sleep\_ms(1)

self.res.low()

time.sleep\_ms(10)

self.res.high()

def poweroff(self):

self.write\_cmd(SET\_DISP | 0x00)

def contrast(self, contrast):

self.write\_cmd(SET\_CONTRAST)

self.write\_cmd(contrast)

def invert(self, invert):

self.write\_cmd(SET\_NORM\_INV | (invert & 1))

def show(self):

x0 = 0

x1 = self.width - 1

if self.width == 64:

# displays with width of 64 pixels are shifted by 32

x0 += 32

x1 += 32

self.write\_cmd(SET\_COL\_ADDR)

self.write\_cmd(x0)

self.write\_cmd(x1)

self.write\_cmd(SET\_PAGE\_ADDR)

self.write\_cmd(0)

self.write\_cmd(self.pages - 1)

self.write\_framebuf()

def fill(self, col):

self.framebuf.fill(col)

def pixel(self, x, y, col):

self.framebuf.pixel(x, y, col)

def scroll(self, dx, dy):

self.framebuf.scroll(dx, dy)

def text(self, string, x, y, col=1):

self.framebuf.text(string, x, y, col)

class SSD1306\_I2C(SSD1306):

def \_\_init\_\_(self, width, height, i2c, addr=0x3c, external\_vcc=False):

self.i2c = i2c

self.addr = addr

self.temp = bytearray(2)

# Add an extra byte to the data buffer to hold an I2C data/command

# byte

# to use hardware-compatible I2C transactions. A memoryview of the

# buffer is used to mask this byte from the framebuffer operations

# (without a major memory hit as memoryview doesn't copy to a separate

# buffer).

self.buffer = bytearray(((height // 8) \* width) + 1)

self.buffer[0] = 0x40 # Set first byte of data buffer to Co=0, D/C=1

self.framebuf = framebuf.FrameBuffer1(memoryview(self.buffer)[1:], width, height)

super().\_\_init\_\_(width, height, external\_vcc)

def write\_cmd(self, cmd):

self.temp[0] = 0x80 # Co=1, D/C#=0

self.temp[1] = cmd

self.i2c.writeto(self.addr, self.temp)

def write\_framebuf(self):

# Blast out the frame buffer using a single I2C transaction to support

# SSD1306 Dial with potentiometer control

0x08, 0x04, 0x7E, 0x04, 0x08, # UP

0x10, 0x20, 0x7E, 0x20, 0x10, # Down

0x08, 0x08, 0x2A, 0x1C, 0x08, # Right

0x08, 0x1C, 0x2A, 0x08, 0x08, # Left

0x1E, 0x10, 0x10, 0x10, 0x10,

0x0C, 0x1E, 0x0C, 0x1E, 0x0C,

0x30, 0x38, 0x3E, 0x38, 0x30,

0x06, 0x0E, 0x3E, 0x0E, 0x06,

0x00, 0x00, 0x00, 0x00, 0x00,

0x00, 0x00, 0x5F, 0x00, 0x00,

0x00, 0x07, 0x00, 0x07, 0x00,

0x14, 0x7F, 0x14, 0x7F, 0x14,

0x24, 0x2A, 0x7F, 0x2A, 0x12,

0x23, 0x13, 0x08, 0x64, 0x62,

0x36, 0x49, 0x56, 0x20, 0x50,

0x00, 0x08, 0x07, 0x03, 0x00,

0x00, 0x1C, 0x22, 0x41, 0x00,

0x00, 0x41, 0x22, 0x1C, 0x00,

0x2A, 0x1C, 0x7F, 0x1C, 0x2A,

0x08, 0x08, 0x3E, 0x08, 0x08,

0x00, 0x80, 0x70, 0x30, 0x00,

0x08, 0x08, 0x08, 0x08, 0x08,

0x00, 0x00, 0x60, 0x60, 0x00,

0x20, 0x10, 0x08, 0x04, 0x02,

0x3E, 0x51, 0x49, 0x45, 0x3E,

0x00, 0x42, 0x7F, 0x40, 0x00,

0x72, 0x49, 0x49, 0x49, 0x46,

0x21, 0x41, 0x49, 0x4D, 0x33,

0x18, 0x14, 0x12, 0x7F, 0x10,

0x27, 0x45, 0x45, 0x45, 0x39,

0x3C, 0x4A, 0x49, 0x49, 0x31,

0x41, 0x21, 0x11, 0x09, 0x07,

0x36, 0x49, 0x49, 0x49, 0x36,

0x46, 0x49, 0x49, 0x29, 0x1E,

0x00, 0x00, 0x14, 0x00, 0x00,

0x00, 0x40, 0x34, 0x00, 0x00,

0x00, 0x08, 0x14, 0x22, 0x41,

0x14, 0x14, 0x14, 0x14, 0x14,

0x00, 0x41, 0x22, 0x14, 0x08,

0x02, 0x01, 0x59, 0x09, 0x06,

0x3E, 0x41, 0x5D, 0x59, 0x4E,

0x7C, 0x12, 0x11, 0x12, 0x7C, # A

0x7F, 0x49, 0x49, 0x49, 0x36,

0x3E, 0x41, 0x41, 0x41, 0x22,

0x7F, 0x41, 0x41, 0x41, 0x3E,

0x7F, 0x49, 0x49, 0x49, 0x41,

0x7F, 0x09, 0x09, 0x09, 0x01,

0x3E, 0x41, 0x41, 0x51, 0x73,

0x7F, 0x08, 0x08, 0x08, 0x7F,

0x00, 0x41, 0x7F, 0x41, 0x00,

0x20, 0x40, 0x41, 0x3F, 0x01,

0x7F, 0x08, 0x14, 0x22, 0x41,

0x7F, 0x40, 0x40, 0x40, 0x40,

0x7F, 0x02, 0x1C, 0x02, 0x7F,

0x7F, 0x04, 0x08, 0x10, 0x7F,

0x3E, 0x41, 0x41, 0x41, 0x3E,

0x7F, 0x09, 0x09, 0x09, 0x06,

0x3E, 0x41, 0x51, 0x21, 0x5E,

0x7F, 0x09, 0x19, 0x29, 0x46,

0x26, 0x49, 0x49, 0x49, 0x32,

0x03, 0x01, 0x7F, 0x01, 0x03,

0x3F, 0x40, 0x40, 0x40, 0x3F,

# \*\*\*\* Font Sizes and extra characters included \*\*\*

# \*\*\* GFX Routines included \*\*\*

# Tony Goodhew for Instructables.com

# Hardware: Button, 10K Potentiometer and SSD1306 128x64 pixel

display

from machine import Pin, I2C, ADC

from ssd1306 import SSD1306\_I2C

import math # Needed for square root and trigonometery

import time

# === Setup hardware section ===

i2c =I2C(0,sda=Pin(21),scl=Pin(22),freq=100000)

# Set up SSD1306 display

# Common SSD1306 with i2c address = 0x3C

oled = SSD1306\_I2C(128,64,i2c)

# For Adafruit display

# STEMMA/QT wires yellow to GP9 and blue to GP8

#oled = SSD1306\_I2C(128,64,i2c,addr=0x3d) # For Adafruit display

# Set up the potentiometer on ADC pin 26, ADC0

potentiometer = ADC(Pin(36))

# Set up button on GP15 as INPUT with PULL\_DOWN

# button = Pin(15, Pin.IN, Pin.PULL\_DOWN)

button = Pin(27, Pin.IN, Pin.PULL\_UP)

# === Procedures and functions section ===

pot\_min = 800 # Normal minimum raw value - adjust as necessary

pot\_range = 65535 - pot\_min

# Standard ASCII 5x8 font

# https://gist.github.com/tdicola/229b3eeddc12d58fb0bc724a9062aa05

FONT\_HEIGHT = 8

FONT\_WIDTH = 5

FONT = bytes([

0x00, 0x00, 0x00, 0x00, 0x00, # <space>

0x3E, 0x5B, 0x4F, 0x5B, 0x3E,

0x3E, 0x6B, 0x4F, 0x6B, 0x3E,

0x1C, 0x3E, 0x7C, 0x3E, 0x1C,

0x18, 0x3C, 0x7E, 0x3C, 0x18,

0x1C, 0x57, 0x7D, 0x57, 0x1C,

0x1C, 0x5E, 0x7F, 0x5E, 0x1C,

0x00, 0x18, 0x3C, 0x18, 0x00,

0xFF, 0xE7, 0xC3, 0xE7, 0xFF,

0x00, 0x18, 0x24, 0x18, 0x00,

0xFF, 0xE7, 0xDB, 0xE7, 0xFF,

0x30, 0x48, 0x3A, 0x06, 0x0E,

0x26, 0x29, 0x79, 0x29, 0x26,

0x40, 0x7F, 0x05, 0x05, 0x07,

0x40, 0x7F, 0x05, 0x25, 0x3F,

0x5A, 0x3C, 0xE7, 0x3C, 0x5A,

0x7F, 0x3E, 0x1C, 0x1C, 0x08,

0x08, 0x1C, 0x1C, 0x3E, 0x7F,

0x14, 0x22, 0x7F, 0x22, 0x14,

0x5F, 0x5F, 0x00, 0x5F, 0x5F,

0x06, 0x09, 0x7F, 0x01, 0x7F,

0x00, 0x66, 0x89, 0x95, 0x6A,

0x60, 0x60, 0x60, 0x60, 0x60,

0x94, 0xA2, 0xFF, 0xA2, 0x94,

0x1F, 0x20, 0x40, 0x20, 0x1F,

0x3A, 0x44, 0x44, 0x44, 0x3A, # o-umlaut

0x32, 0x4A, 0x48, 0x48, 0x30,

0x3A, 0x41, 0x41, 0x21, 0x7A,

0x3A, 0x42, 0x40, 0x20, 0x78,

0x00, 0x9D, 0xA0, 0xA0, 0x7D,

0x3D, 0x42, 0x42, 0x42, 0x3D, # O-umlaut

0x3D, 0x40, 0x40, 0x40, 0x3D,

0x3C, 0x24, 0xFF, 0x24, 0x24,

0x48, 0x7E, 0x49, 0x43, 0x66,

0x2B, 0x2F, 0xFC, 0x2F, 0x2B,

0xFF, 0x09, 0x29, 0xF6, 0x20,

0xC0, 0x88, 0x7E, 0x09, 0x03,

0x20, 0x54, 0x54, 0x79, 0x41,

0x00, 0x00, 0x44, 0x7D, 0x41,

0x30, 0x48, 0x48, 0x4A, 0x32,

0x38, 0x40, 0x40, 0x22, 0x7A,

0x00, 0x7A, 0x0A, 0x0A, 0x72,

0x7D, 0x0D, 0x19, 0x31, 0x7D,

0x26, 0x29, 0x29, 0x2F, 0x28,

0x26, 0x29, 0x29, 0x29, 0x26,

0x30, 0x48, 0x4D, 0x40, 0x20,

0x38, 0x08, 0x08, 0x08, 0x08,

0x08, 0x08, 0x08, 0x08, 0x38,

0x2F, 0x10, 0xC8, 0xAC, 0xBA,

0x2F, 0x10, 0x28, 0x34, 0xFA,

0x00, 0x00, 0x7B, 0x00, 0x00,

0x08, 0x14, 0x2A, 0x14, 0x22,

0x22, 0x14, 0x2A, 0x14, 0x08,

0x55, 0x00, 0x55, 0x00, 0x55, # 176 (25% block) missing in old code

0xAA, 0x55, 0xAA, 0x55, 0xAA, # 50% block

0xFF, 0x55, 0xFF, 0x55, 0xFF, # 75% block

0x00, 0x00, 0x00, 0xFF, 0x00,

0x10, 0x10, 0x10, 0xFF, 0x00,

0x14, 0x14, 0x14, 0xFF, 0x00,

0x10, 0x10, 0xFF, 0x00, 0xFF,

0x10, 0x10, 0xF0, 0x10, 0xF0,

0x14, 0x14, 0x14, 0xFC, 0x00,

0x14, 0x14, 0xF7, 0x00, 0xFF,

0x00, 0x00, 0xFF, 0x00, 0xFF,

0x14, 0x14, 0xF4, 0x04, 0xFC,

0x14, 0x14, 0x17, 0x10, 0x1F,

0x10, 0x10, 0x1F, 0x10, 0x1F,

0x14, 0x14, 0x14, 0x1F, 0x00,

0x10, 0x10, 0x10, 0xF0, 0x00,

0x00, 0x00, 0x00, 0x1F, 0x10,

0x10, 0x10, 0x10, 0x1F, 0x10,

0x10, 0x10, 0x10, 0xF0, 0x10,

0x00, 0x00, 0x00, 0xFF, 0x10,

0x10, 0x10, 0x10, 0x10, 0x10,

0x10, 0x10, 0x10, 0xFF, 0x10,

0x00, 0x00, 0x00, 0xFF, 0x14,

0x00, 0x00, 0xFF, 0x00, 0xFF,

0x00, 0x00, 0x1F, 0x10, 0x17,

0x00, 0x00, 0xFC, 0x04, 0xF4,

0x14, 0x14, 0x17, 0x10, 0x17,

0x14, 0x14, 0xF4, 0x04, 0xF4,

0x00, 0x00, 0xFF, 0x00, 0xF7,

0x14, 0x14, 0x14, 0x14, 0x14,

0x14, 0x14, 0xF7, 0x00, 0xF7,

0x14, 0x14, 0x14, 0x17, 0x14,

0x10, 0x10, 0x1F, 0x10, 0x1F,

0x14, 0x14, 0x14, 0xF4, 0x14,

0x3F, 0x40, 0x38, 0x40, 0x3F,

0x63, 0x14, 0x08, 0x14, 0x63,

0x03, 0x04, 0x78, 0x04, 0x03,

0x61, 0x59, 0x49, 0x4D, 0x43,

0x00, 0x7F, 0x41, 0x41, 0x41,

0x02, 0x04, 0x08, 0x10, 0x20,

0x00, 0x41, 0x41, 0x41, 0x7F,

0x04, 0x02, 0x01, 0x02, 0x04,

0x40, 0x40, 0x40, 0x40, 0x40,

0x00, 0x03, 0x07, 0x08, 0x00,

0x20, 0x54, 0x54, 0x78, 0x40,

0x7F, 0x28, 0x44, 0x44, 0x38,

0x38, 0x44, 0x44, 0x44, 0x28,

0x38, 0x44, 0x44, 0x28, 0x7F,

0x38, 0x54, 0x54, 0x54, 0x18,

0x00, 0x08, 0x7E, 0x09, 0x02,

0x18, 0xA4, 0xA4, 0x9C, 0x78,

0x7F, 0x08, 0x04, 0x04, 0x78,

0x00, 0x44, 0x7D, 0x40, 0x00,

0x20, 0x40, 0x40, 0x3D, 0x00,

0x7F, 0x10, 0x28, 0x44, 0x00,

0x00, 0x41, 0x7F, 0x40, 0x00,

0x7C, 0x04, 0x78, 0x04, 0x78,

0x7C, 0x08, 0x04, 0x04, 0x78,

0x38, 0x44, 0x44, 0x44, 0x38,

0xFC, 0x18, 0x24, 0x24, 0x18,

0x18, 0x24, 0x24, 0x18, 0xFC,

0x7C, 0x08, 0x04, 0x04, 0x08,

0x48, 0x54, 0x54, 0x54, 0x24,

0x04, 0x04, 0x3F, 0x44, 0x24,

0x3C, 0x40, 0x40, 0x20, 0x7C,

0x1C, 0x20, 0x40, 0x20, 0x1C,

0x3C, 0x40, 0x30, 0x40, 0x3C,

0x44, 0x28, 0x10, 0x28, 0x44,

0x4C, 0x90, 0x90, 0x90, 0x7C,

0x44, 0x64, 0x54, 0x4C, 0x44,

0x00, 0x08, 0x36, 0x41, 0x00,

0x00, 0x00, 0x77, 0x00, 0x00,

0x00, 0x41, 0x36, 0x08, 0x00,

0x02, 0x01, 0x02, 0x04, 0x02,

0x3C, 0x26, 0x23, 0x26, 0x3C,

0x1E, 0xA1, 0xA1, 0x61, 0x12, # Extension starts here

0x3A, 0x40, 0x40, 0x20, 0x7A,

0x38, 0x54, 0x54, 0x55, 0x59,

0x21, 0x55, 0x55, 0x79, 0x41,

0x22, 0x54, 0x54, 0x78, 0x42, # a-umlaut

0x21, 0x55, 0x54, 0x78, 0x40,

0x20, 0x54, 0x55, 0x79, 0x40,

0x0C, 0x1E, 0x52, 0x72, 0x12,

0x39, 0x55, 0x55, 0x55, 0x59,

0x39, 0x54, 0x54, 0x54, 0x59,

0x39, 0x55, 0x54, 0x54, 0x58,

0x00, 0x00, 0x45, 0x7C, 0x41,

0x00, 0x02, 0x45, 0x7D, 0x42,

0x00, 0x01, 0x45, 0x7C, 0x40,

0x7D, 0x12, 0x11, 0x12, 0x7D, # A-umlaut

0xF0, 0x28, 0x25, 0x28, 0xF0,

0x7C, 0x54, 0x55, 0x45, 0x00,

0x20, 0x54, 0x54, 0x7C, 0x54,

0x7C, 0x0A, 0x09, 0x7F, 0x49,

0x32, 0x49, 0x49, 0x49, 0x32,

0x10, 0x10, 0xF0, 0x10, 0xF0,

oled.pixel(ii\*sz+xt+2,yy\*sz+yt+2,c)

oled.pixel(ii\*sz+xt+2,yy\*sz+yt,c)

oled.pixel(ii\*sz+xt+2,yy\*sz+yt+1,c)

def prnt\_st(asci,xx,yy,sz,c): # Text string

if sz == 1: move = 6

if sz == 2: move = 11

if sz == 3: move = 17

for letter in(asci):

asci = ord(letter)

character(asci,xx,yy,sz,c)

xx = xx + move

def cntr\_st(s,y,sz,c): # Centres text on line y

if sz == 1: w = 6

if sz == 2: w = 11

if sz == 3: w = 17

gap = int((width - len(s) \* w)/2)

prnt\_st(s,gap,y,sz,c)

# =========== End of font support routines ===========

# Define a function to read the potentiometer

def get\_ADC(max\_required):

# Read the potentiometer value

pot = potentiometer.read\_u16()

# rescale: new = adjusted\_raw \* max\_required / original\_range

result = int((pot - pot\_min) \* max\_required / pot\_range) # Integer – whole number

# Adjust end points as necessary

if result < 0: # Bottom end – -ve value set to zero

result = 0

if result > max\_required: # Top end – Over sized set to max\_required

result = max\_required

return result

# Define procedures to draw shapes

def block(x,y,w,h,c): # Filled rectangle

for yy in range(h):

oled.hline(x,y+yy,w,c)

def circle(x,y,r,c): # Filled circle

oled.hline(x-r,y,r\*2,c)

for i in range(1,r):

a = int(math.sqrt(r\*r-i\*i)) # Pythagoras!

oled.hline(x-a,y+i,a\*2,c) # Lower half

oled.hline(x-a,y-i,a\*2,c) # Upper half

def ring(x,y,r,c): # Edge of circle

oled.pixel(x-r,y,c)

oled.pixel(x+r,y,c)

oled.pixel(x,y-r,c)

oled.pixel(x,y+r,c)

for i in range(1,r):

a = int(math.sqrt(r\*r-i\*i)) # Pythagoras!

oled.pixel(x-a,y-i,c)

oled.pixel(x+a,y-i,c)

oled.pixel(x-a,y+i,c)

oled.pixel(x+a,y+i,c)

0x00, 0x00, 0x1F, 0x10, 0x1F,

0x00, 0x00, 0x00, 0x1F, 0x14,

0x00, 0x00, 0x00, 0xFC, 0x14,

0x00, 0x00, 0xF0, 0x10, 0xF0,

0x10, 0x10, 0xFF, 0x10, 0xFF,

0x14, 0x14, 0x14, 0xFF, 0x14,

0x10, 0x10, 0x10, 0x1F, 0x00,

0x00, 0x00, 0x00, 0xF0, 0x10,

0xFF, 0xFF, 0xFF, 0xFF, 0xFF,

0xF0, 0xF0, 0xF0, 0xF0, 0xF0,

0xFF, 0xFF, 0xFF, 0x00, 0x00,

0x00, 0x00, 0x00, 0xFF, 0xFF,

0x0F, 0x0F, 0x0F, 0x0F, 0x0F,

0x38, 0x44, 0x44, 0x38, 0x44, # alpha - Greek characters

#start here at 224

0xFC, 0x4A, 0x4A, 0x4A, 0x34, # sharp-s or beta

0x7E, 0x02, 0x02, 0x06, 0x06,

0x02, 0x7E, 0x02, 0x7E, 0x02, # pi

0x63, 0x55, 0x49, 0x41, 0x63,

0x38, 0x44, 0x44, 0x3C, 0x04,

0x40, 0x7E, 0x20, 0x1E, 0x20, # mu

0x06, 0x02, 0x7E, 0x02, 0x02,

0x99, 0xA5, 0xE7, 0xA5, 0x99,

0x1C, 0x2A, 0x49, 0x2A, 0x1C,

0x4C, 0x72, 0x01, 0x72, 0x4C, # omega

0x30, 0x4A, 0x4D, 0x4D, 0x30,

0x30, 0x48, 0x78, 0x48, 0x30,

0xBC, 0x62, 0x5A, 0x46, 0x3D,

0x3E, 0x49, 0x49, 0x49, 0x00,

0x7E, 0x01, 0x01, 0x01, 0x7E, # End of Greek chars

0x2A, 0x2A, 0x2A, 0x2A, 0x2A, # equivalent to 240

0x44, 0x44, 0x5F, 0x44, 0x44, # + or -

0x40, 0x51, 0x4A, 0x44, 0x40, # >=

0x40, 0x44, 0x4A, 0x51, 0x40, # <=

0x00, 0x00, 0xFF, 0x01, 0x03, # top of integral

0xE0, 0x80, 0xFF, 0x00, 0x00, # bottom of integral

0x08, 0x08, 0x6B, 0x6B, 0x08,

0x36, 0x12, 0x36, 0x24, 0x36, # approximately

0x06, 0x0F, 0x09, 0x0F, 0x06, # Degree

0x00, 0x00, 0x18, 0x18, 0x00,

0x00, 0x00, 0x10, 0x10, 0x00,

0x30, 0x40, 0xFF, 0x01, 0x01, # sq root

0x00, 0x1F, 0x01, 0x01, 0x1E, # n superscript

0x00, 0x19, 0x1D, 0x17, 0x12, # squared (^2)

0x00, 0x3C, 0x3C, 0x3C, 0x3C,

0x00, 0x00, 0x00, 0x00, 0x00 # 255 also a <space>

])

def character(asc,xt,yt,sz,c): # Single character sz is size:1 or 2

code = asc \* 5 # 5 bytes per character

for ii in range(5):

line = FONT[code + ii]

for yy in range(8):

if (line >> yy) & 0x1:

oled.pixel(ii\*sz+xt,yy\*sz+yt,c)

if sz > 1:

oled.pixel(ii\*sz+xt+1,yy\*sz+yt,c)

oled.pixel(ii\*sz+xt,yy\*sz+yt+1,c)

oled.pixel(ii\*sz+xt+1,yy\*sz+yt+1,c)

if sz == 3:

oled.pixel(ii\*sz+xt, yy\*sz+yt+2,c)

oled.pixel(ii\*sz+xt+1,yy\*sz+yt+2,c)

oled.pixel(x-i,y-a,c)

# Pointer controlled by Potentiometer

r = 25 # Length of pointer

xn = xc+1 # Dummy values for end of old pointer

yn = yc+1 # Used in first loop pass

circle(xc,yc,27,0)

running = True

while running: # Loop start

p = get\_ADC(100) # Get pot reading in range 0 - 100

oled.line(xc,yc,xn+xc,yc+yn,0) # Remove old pointer

theta = p \* 2.7 - 45 # Angle of pointer. Zero is left horizontal

# Calculate coordinates of end of pointer

theta\_rad = math.radians(theta)

theta\_rad = math.radians(theta)

yn = -int(r \* math.sin(theta\_rad))

xn = -int(r \* math.cos(theta\_rad))

# Draw new pointer

oled.line(xc,yc,xn+xc,yc+yn,1) # Draw pointer

circle(xc,yc,3,1) # Centre of pointer - blob

block(70,15,60,16,0) # Rub out previous percentage

st = " " + str(p) +" %" # Right align text

prnt\_st(st[-5:],70,15,2,1)

oled.show() # Update screen

#if button.value() == 1: # If button is pressed STOP looping

if button.value() == 0: # If button is pressed STOP looping

running = False

time.sleep(0.1) # short delay

# Tidy up

oled.fill(0) Files: **1. SSD1306+Instr+Dial3+TEXT.py**

oled.show() **2. ssd1306.py-🡪on the device ESP32**

cntr\_st("BYE",18,3,1)

oled.show()

time.sleep(3)

oled.fill(0)

oled.show()

oled.pixel(x+i,y-a,c)

oled.pixel(x-i,y+a,c)

oled.pixel(x+i,y+a,c)

# === MAIN ===

# Flash screen

oled.fill(0)

oled.show()

width = 128

cntr\_st("SSD1306",2,2,1)

cntr\_st("Dial",20,2,1)

cntr\_st("Gauge",38,2,1)

cntr\_st("Tony Goodhew",55,1,1)

oled.show()

time.sleep(5)

oled.fill(0)

oled.show()

r = 31 # Radius for dial

xc = 32 # Centre of dial

yc = 31

ring(xc,yc,r,1) # Draw edge of dial

# Draw the scale ticks - lines from centre

for p in range(0,101,10):

theta = p \* 2.7 - 45 # Angle of pointer. Zero is left horizontal

theta\_rad = math.radians(theta) # Angle in radians

yn = -int(r \* math.sin(theta\_rad)) # Calculate outer tick coordinates

xn = -int(r \* math.cos(theta\_rad))

oled.line(xc,yc,xc+xn,yc+yn,1) # Radial line from centre to ring

# Clear centre & bottom of dial

circle(xc,yc,27,0) # Leave the ticks on the ring

block(xc-31,yc+22,64,15,0) # Remove bottom of ring

oled.text("Tony",82,45,1) # Write Title

oled.text("Goodhew",72,55,1) # Write Title

oled.show() # Display the dial

# Move the pointer slowly from 0 to 100



r = 25 # Length of pointer

xn = xc+1 # Dummy values for end of old pointer

yn = yc+1 # Use in first loop pass

for p in range(0,101,1): # Move pointer in single steps (0 -> 100)

oled.line(xc,yc,xn+xc,yc+yn,0) # Remove old pointer

theta = p \* 2.7 - 45 # Angle of pointer. Zero is left horizontal

# Calculate coordinates of end of pointer

theta\_rad = math.radians(theta)

theta\_rad = math.radians(theta)

yn = -int(r \* math.sin(theta\_rad))

xn = -int(r \* math.cos(theta\_rad))

# Draw new pointer

oled.line(xc,yc,xn+xc,yc+yn,1) # Draw pointer - radial line

circle(xc,yc,3,1) # Centre of pointer - blob

block(70,15,60,16,0) # Rub out previous percentage

st = " " + str(p) +" %" # Right align text

prnt\_st(st[-5:],70,15,2,1)

oled.show() # Update screen

import machine

**File Name: 1. RTC\_LeftCorner\_Clock\_OLED.py**

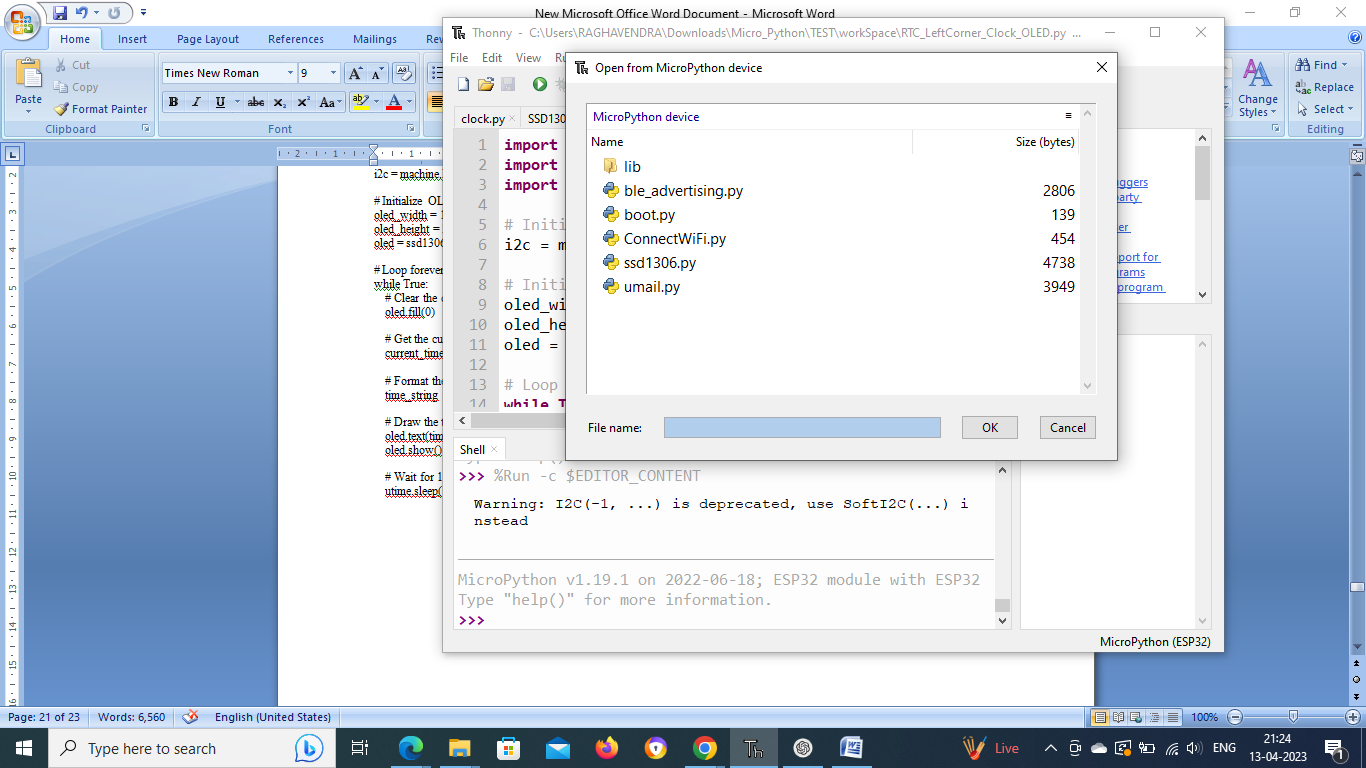
**2. ssd1306.py (on device:ESP32) 4738Bytes**

import ssd1306

import utime

# Initialize I2C bus

i2c = machine.I2C(scl=machine.Pin(22), sda=machine.Pin(21))



# Initialize OLED display

oled\_width = 128

oled\_height = 32 #when 64 small letters

oled = ssd1306.SSD1306\_I2C(oled\_width, oled\_height, i2c)

# Loop forever

while True:

# Clear the display

oled.fill(0)

# Get the current time

current\_time = utime.localtime()

# Format the time as a string

time\_string = "{:02d}:{:02d}:{:02d}".format(current\_time[3], current\_time[4], current\_time[5])

# Draw the time on the display

oled.text(time\_string, 0, 0)

oled.show()

# Wait for 1 second

utime.sleep(1)

**File Name: MAC\_TEST.py**

**#https://stackoverflow.com/questions/71902740/how-to-retrieve-and-format-wifi-mac-address-in-micropython-on-esp32**

from machine import Pin, I2C

import ssd1306

import network

import ubinascii

i2c = I2C(-1, Pin(22), Pin(21)) # Create I2C object

oled\_width = 128

oled\_height = 64

oled = ssd1306.SSD1306\_I2C(oled\_width, oled\_height, i2c)

ssid = "KGM"

password = "nalini\_raghavendra\_195863"

station = network.WLAN(network.STA\_IF)

station.active(True)

station.connect(ssid, password)

while not station.isconnected():

pass

print("Connection successful")

mac = station.config('mac')

mac = (ubinascii.hexlify(mac).decode())

oled.fill(0)

oled.text("MAC Address:", 0, 0)

oled.text(mac, 0, 16)

oled.show()

import machine

import ssd1306

**File Name: 1. Dial\_Clock\_OLED\_1.py**

**2. ssd1306.py (on device:ESP32) 4738 Bytes**

import time

import math

# initialize OLED display

i2c = machine.I2C(scl=machine.Pin(22), sda=machine.Pin(21))

oled = ssd1306.SSD1306\_I2C(128, 64, i2c)

# set up clock hand lengths

hour\_hand\_len = 20

minute\_hand\_len = 30

second\_hand\_len = 25

while True:

# clear display

oled.fill(0)

# get current time

now = time.localtime()

# calculate hand angles

hour\_angle = (now[3] % 12) \* 30 + now[4] / 2 + now[5] / 120

minute\_angle = now[4] \* 6 + now[5] / 10

second\_angle = now[5] \* 6

# calculate hand endpoints

hour\_x = int(math.cos(math.radians(hour\_angle - 90)) \* hour\_hand\_len + 64)

hour\_y = int(math.sin(math.radians(hour\_angle - 90)) \* hour\_hand\_len + 32)

minute\_x = int(math.cos(math.radians(minute\_angle - 90)) \* minute\_hand\_len + 64)

minute\_y = int(math.sin(math.radians(minute\_angle - 90)) \* minute\_hand\_len + 32)

second\_x = int(math.cos(math.radians(second\_angle - 90)) \* second\_hand\_len + 64)

second\_y = int(math.sin(math.radians(second\_angle - 90)) \* second\_hand\_len + 32)

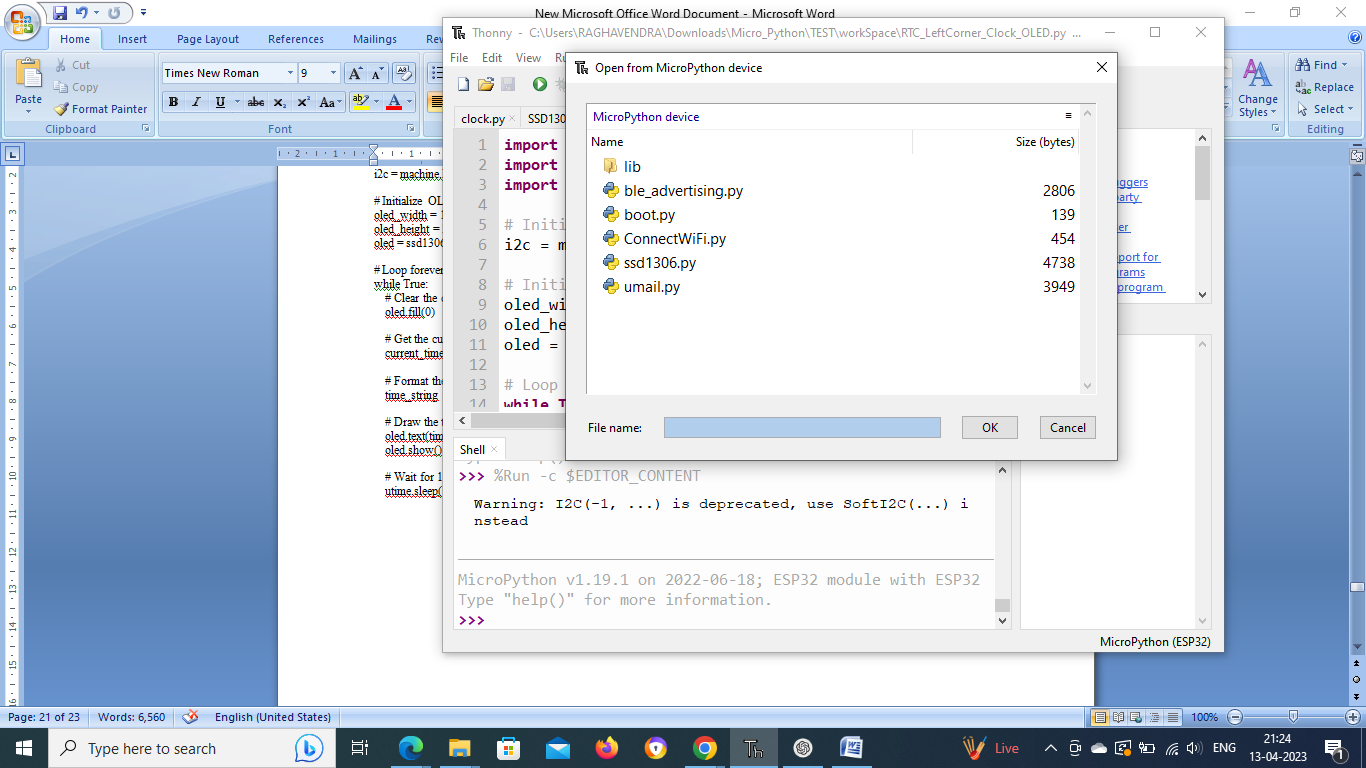
# draw clock hands

oled.line(64, 32, hour\_x, hour\_y, 1)

oled.line(64, 32, minute\_x, minute\_y, 1)

oled.line(64, 32, second\_x, second\_y, 1)

def ring(x,y,r,c): # Edge of circle



oled.pixel(x-r,y,c)

oled.pixel(x+r,y,c)

oled.pixel(x,y-r,c)

oled.pixel(x,y+r,c)

for i in range(1,r):

a = int(math.sqrt(r\*r-i\*i)) # Pythagoras!

oled.pixel(x-a,y-i,c)

oled.pixel(x+a,y-i,c)

oled.pixel(x-a,y+i,c)

oled.pixel(x+a,y+i,c)

oled.pixel(x-i,y-a,c)

oled.pixel(x+i,y-a,c)

oled.pixel(x-i,y+a,c)

oled.pixel(x+i,y+a,c)

# update display

oled.show()

# wait for one second

time.sleep(1)

r= 32

xc= 64

yc= 32

# draw clock face

ring(xc, yc, r, 1)

# Import necessary libraries

**ESP32\_Params.py**

**File: ssd1306.py 4738Bytes is on the Device ESP32**

from machine import I2C, Pin

import ssd1306

import network

import time

# Initialize I2C interface and OLED display

# i2c = I2C(0, Pin(22), Pin(21))

i2c =I2C(0,sda=Pin(21),scl=Pin(22),freq=100000)

oled = ssd1306.SSD1306\_I2C(128, 32, i2c) # Larger size

# oled = ssd1306.SSD1306\_I2C(128, 64, i2c) # Smaller size

# Connect to Wi-Fi network

wifi = network.WLAN(network.STA\_IF)

wifi.active(True)

wifi.connect("KGM", "nalini\_raghavendra\_195863")

while not wifi.isconnected():

pass

print("Connection successful")

# Display ESP32 parameters on OLED display

while True:

oled.fill(0)

# oled.text("IP Address:", 0, 0)

oled.text(wifi.ifconfig()[0], 0, 0)

oled.text("Free Memory:", 0, 10)

oled.text(str(gc.mem\_free()), 0, 20)

oled.show()

time.sleep(1)

-------------------------------------------------------------------------------------------------------------------------------------------------------

from machine import Pin, I2C

**ESP32\_Params\_1.py**

**File: ssd1306.py 4738Bytes is on the Device ESP32**

import ssd1306

import network

import time

import ubinascii #**https://stackoverflow.com/questions/71902740/how-to-retrieve-and-format-wifi-mac-address-in-micropython-on-esp32**

i2c = I2C(-1, Pin(22), Pin(21)) # Create I2C object

oled\_width = 128

oled\_height = 64

oled = ssd1306.SSD1306\_I2C(oled\_width, oled\_height, i2c)

ssid = "KGM"

password = "nalini\_raghavendra\_195863"

station = network.WLAN(network.STA\_IF)

station.active(True)

station.connect(ssid, password)

while not station.isconnected():

pass

print("Connection successful")

oled.text(str(gc.mem\_free()), 0, 50)

oled.show()

time.sleep(1)

oled.fill(0)

oled.show()

time.sleep(1)

while True:

oled.fill(0)

mac = station.config('mac')

mac = (ubinascii.hexlify(mac).decode())

oled.text(mac, 0, 0)

oled.text(str(station.ifconfig()[0]), 0, 10)

oled.text(str(station.ifconfig()[1]), 0, 20)

oled.text(str(station.ifconfig()[2]),0,30)

oled.text("Free Memory:", 0, 40)

from machine import Pin, I2C

import time

**File Name: MAC\_TEST.py**

**File: ssd1306.py** 4732Bytes **is on the Device ESP32**

import ssd1306

import network

import ubinascii

import numpy as np

import machine

from sklearn.linear\_model import LogisticRegression

# Initialize microphone module

mic = machine.ADC(0)

# Function to extract audio features

def get\_features():

samples = []

for i in range(100):

sample = mic.read()

samples.append(sample)

samples = np.array(samples)

features = [np.mean(samples), np.std(samples)]

return features

# Create dataset

dataset = []

for i in range(50):

dataset.append((get\_features(), 0))

for i in range(50):

dataset.append((get\_features(), 1))

# Train machine learning model

X = [data[0] for data in dataset]

y = [data[1] for data in dataset]

clf = LogisticRegression(random\_state=0).fit(X, y)

# Function to classify audio samples in real-time

def classify():

features = get\_features()

prediction = clf.predict([features])[0]

return prediction

# Main loop

while True:

label = classify()

if label == 0:

print("Sound A detected")

else:

print("Sound B detected")

import binascii

import machine

i2c = I2C(-1, Pin(22), Pin(21)) # Create I2C object

oled\_width = 128

oled\_height = 64

oled = ssd1306.SSD1306\_I2C(oled\_width, oled\_height, i2c)

ssid = "KGM"

password = "nalini\_raghavendra\_195863"

station = network.WLAN(network.STA\_IF)

station.active(True)

station.connect(ssid, password)

while not station.isconnected():

pass

print("Connection successful")

while True:

mac = station.config('mac')

mac = (ubinascii.hexlify(mac).decode())

oled.fill(0)

oled.text("MAC Address:", 0, 0)

oled.text(mac, 0, 16)

oled.show()

time.sleep(1)

oled.fill(0)

chip\_id = machine.unique\_id() **# Chip ID is its MAC Address**

hex\_id = binascii.hexlify(chip\_id).decode()

oled.text("ESP32 ChipID:", 0, 20)

oled.text(hex\_id, 0, 34)

oled.show()

time.sleep(1)

Step 1: Install Required Libraries To begin, you'll need to install the TensorFlow Lite Micro library for Micro Python on your ESP32. You can do this by following the instructions on the TensorFlow Lite Micro GitHub page.

Step 2: Collect Data Next, you'll need to collect data that you'll use to train your machine learning model. For this example, let's say you want to create a simple binary classification model that distinguishes between two types of sounds (sound A and sound B).

To collect data, you'll need to record audio samples of sound A and sound B. You can use a microphone module connected to your ESP32 to record these samples.

Step 3: Train Your Model Now that you have your data, you can train your machine learning model. For this example, let's use a simple logistic regression model.

To train your model, you'll need to extract features from your audio samples. For this example, let's use the mean and standard deviation of the audio signal as features. You can use the following code to extract these features:

The above is answer for the query to Chat GPT**: Simple Machine Learning Example on ESP32 using Micro Python**