ESP32 Digital Function Generator with WEB Functionality

By,

Roy H Guerra Jr.

Notes:
1) Follow intructions on OLED display to set up WiFi, and Display WEB Page Function Generator Parameters
2) Keep amplitude at 100% or the frequency will be attenuated starting at 10KHz to 1 MHz
3) The AD9833 Function Generator has a 0.328 volt offset and the square wave is more than double that of the sine and traingle waveforms (this is the chip). To make the levels consistent and avoid rolloff due to the digital POT capacitance, use an Analog OP-AMP Signal Conditioner. This will also increase the output current.

AD9833 Generator Connections:

VCC -> 3.3V

GND -> GND

DATA(MOSI)-> GPIO14

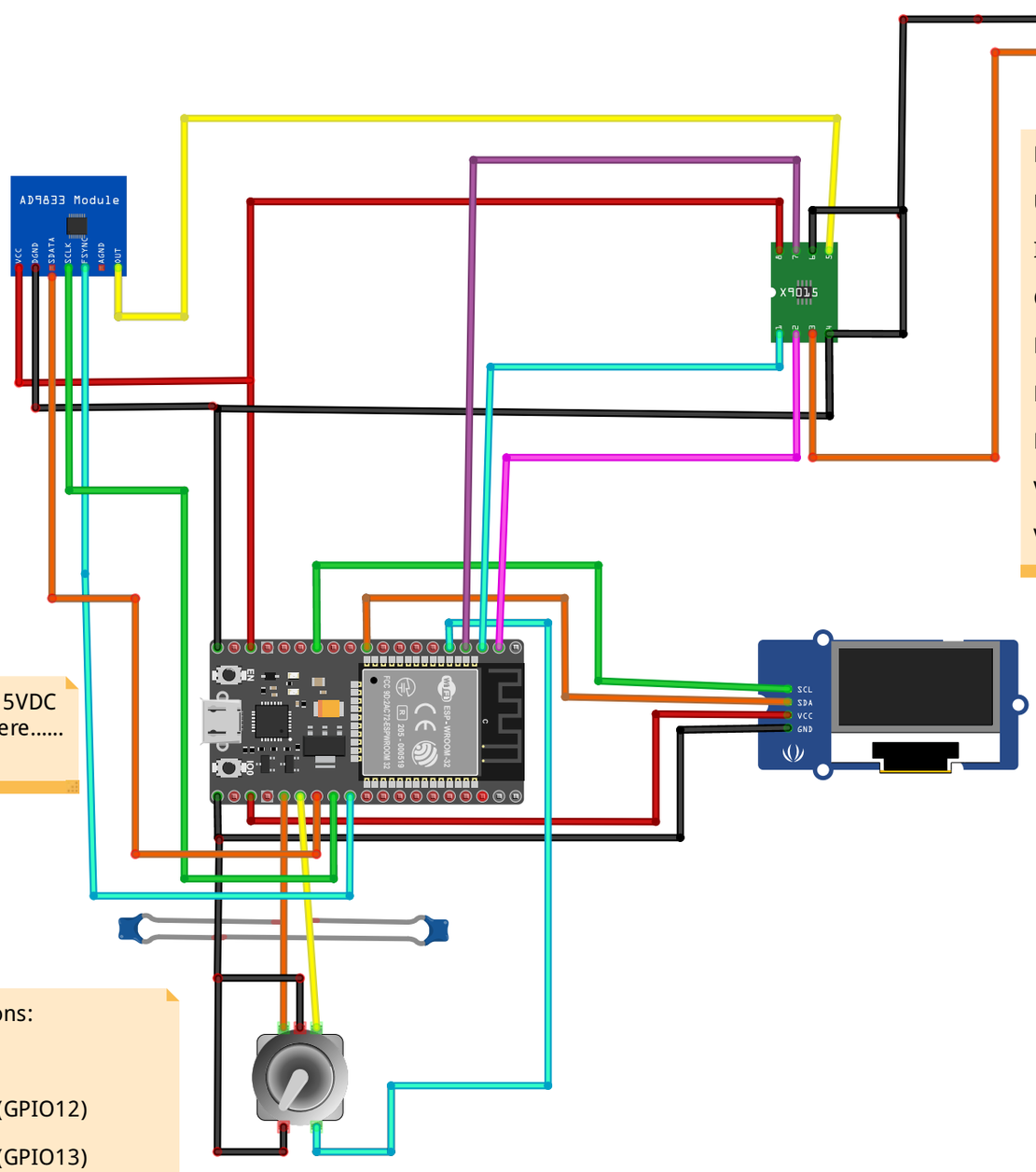
DATA(MISO)-> GPIO18 (dummy pin to make software SPI work, do not use this pin for other I/O)

CLK(MSCK)-> GPIO27

FSYNC -> GPIO26

Ref -> Master OSC Check (25 MHz)

Out -> Frequency Output to (Pin_Wiper_H of X9C104P-pin 3)



Digital POT Connections:

UD = ESP32(GPIO21) --> X9C104P pin 2

INC = ESP32(GPIO22) --> X9C104P pin 1

CS = ESP32(GPIO19) --> X9C104P pin 7

Pin_Wiper --> X9C104P pin 5 -->(Generator Output)

Pin_Wiper_L --> X9C104P pin 6 -->(GND)

Pin_Wiper_H --> X9C104P pin 3 --> (AD9833 Out)

Vss(Gnd) --> X9C104P pin 4

Vcc(3.3v) --> X9C104P pin 8

Add a micro-USB 5VDC Power Adapter Here.....

Note-could use a Heltec ESP32 WiFi Module with a built in (128X64) OLED, or use a separate (128X64) OLED with IIC interface. Wiring is:
- GPIO13 to orange wire
For an External OLED Module:
- Vcc = 3.3 VDC
- GND = GND
- SCL = GPIO15
- SDA = GPIO4

Encoder Connections:

1) Pin "C" = GND

2) Pin "A" = ESP32(GPIO12)

3) Pin "B" = ESP32(GPIO13)

4) Pin_3(Switch1)= ESP32(GPIO23)

5) Pin 5(Switch2) = GND

```
"""
Pin Mapping for X9C104P (100K Digital Potentiometer)
-----------------------------------------------------
UD = ESP32(GPIO21) --> X9C104P pin 2
INC = ESP32(GPIO22) --> X9C104P pin 1
CS = ESP32(GPIO19) --> X9C104P pin 7
Pin_Wiper --> X9C104P pin 5 -->(Generator Output)
Pin_Wiper_L --> X9C104P pin 6 -->(GND)
Pin_Wiper_H --> X9C104P pin 3 --> (AD9833 Out)
Vss(Gnd)  --> X9C104P pin 4
Vcc(3.3v) --> X9C104P pin 8

Note(s) -
A)Power Rating is 10mW for 100K Pot
B) Place a 10uF capacitor between Pin_Wiper_H and AD9833 (Out) observing DC offset

Pin Mapping for Encoder (24 pulses / Rev); 2 phase quadrature:
-------------------------------------------------------------
1) Pin "C" = GND
2) Pin "A" = ESP32(GPIO12)
3) Pin "B" = ESP32(GPIO13)
4) Pin_3(Switch1)= ESP32(GPIO23)
5) Pin 5(Switch2) = GND

Pin Mapping for AD9833:
-----------------------
VCC -> 3.3V
GND -> GND
DATA(MOSI)-> GPIO14
DATA(MISO)-> GPIO18 (dummy pin to make software SPI work, do not use this pin for other I/O)
CLK(MSCK)-> GPIO27
FSYNC -> GPIO26
Ref -> Master OSC Check (25 MHz)
Out -> Frequency Output to (Pin_Wiper_H of X9C104P-pin 3)

Note(s) - Add the following support libraries:
A) SSD1306 (OLED library)
B) encoderLib (Encoder Library)
C) My wifimanager

Displays parameters on a simple web page using the "socket method" as follows:
Server Socket Methods

1   s.bind() #This method binds address (hostname, port number pair) to socket.
2   s.listen() # This method sets up and start TCP listener.
3   s.accept() # This passively accept TCP client connection, waiting until connection arrives (blocking).

Client Socket Methods

1   s.connect() # This method actively initiates TCP server connection.



"""

# Import Modules:
# ---------------
from time import sleep, sleep_ms, sleep_us
import time, socket, select, network, machine
import wifimanager # Start the Wifi Manager Program (this is a separate file)
from network import WLAN
import machine
from machine import Pin, I2C, SPI
from ssd1306 import SSD1306_I2C
import encoderLib
```

```
66
67  # ESP32 GPIO Pins:
68  # ----------------
69  INC_pin = 22
70  CS_pin = 19
71  UD_pin = 21
72  ENC_SW_PIN = 23
73  WIFI_LED_PIN = 25
74  OLED_RST_PIN = 16
75  FSYNC_PIN = 26
76
77  # Hardare ESP32 Setup:
78  # --------------------
79  led1 = Pin(WIFI_LED_PIN, Pin.OUT, value=0) # Sets up WiFi LED to Low
80  OLED_RST = Pin(OLED_RST_PIN, Pin.OUT, value=1) # Sets up OLED RST to high
81  i2c = I2C(sda = Pin(4), scl = Pin(15)) # Set Up OLED IIC Pins
82  display = SSD1306_I2C(128, 64, i2c) # IIC parameters for OLED Display
83  UD = Pin(UD_pin, Pin.OUT, value=0) # Sets UD to low
84  INC = Pin(INC_pin, Pin.OUT, value=0) # Sets INC to low
85  CS = Pin(CS_pin, Pin.OUT, value=0) # Sets CS to low
86  FSYNC = Pin(FSYNC_PIN, Pin.OUT, value=0) # Sets FSYNC to low
87  ENC_SWITCH = (Pin(ENC_SW_PIN, Pin.IN, Pin.PULL_UP))# Encoder Knob Switch with pull-up
    resistor enabled
88
89  # Global Variables & Constants:
90  # -----------------------------
91  position = 0 # Amplitude
92  freq = 10 # Frequency (start at 10)
93  mult = 0 # Frequency multiplier (start at X1)
94  mult_index = 1 #Frequency multiplier Index
95  wave = 0 # Initial encoder waveform
96  waveform = 0x2000 # Initial encoder waveform (sine)
97  wave_display = "Sine Wave" # OLED Display Text
98  last = 0 # Encoder POT last position
99  last1 = 0 # Encoder Frequency last position
100 last2 = 0 # Frequency Multiplier last position
101 last3 = 0 # Waveform last position
102 value = 0 # Encoder POT initial Value
103 count = 5 # Menu Count Variable to Start at Main Menu
104 ClockFreq = 25000000 # Clock Frequency
105 phase = 0 # Phase Shift
106 selection = 0 # Menu Selection Variable
107 label = "" # String Null Variable
108 menu_rst = 0 # Variable to reset "count"
109 hold = 0 # Menu lock
110 z = 0 # Timeout Variable
111 #new_Hz = " " # Calculated KHz/MHz
112
113 # Establish Internet Connection:
114 # ------------------------------
115 wlan = wifimanager.get_connection()
116 if wlan is None:
117     print("Could Not Initialize Network Connection.")
118     L1 = "Could Not"
119     L2 = "Initialize"
120     L3 = "Network"
121     L4 = "Connection"
122     L5 = "Please Turn Off"
123     L6 = "And Try Again"
124     delayx = 3
125     Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
126     #while True:  # Program will not go further and stat_mode1() will not operate
        unless line is commented out.
127         #pass
128
129 # Shut off Access Point
130 # ---------------------
```

```
ap_if = network.WLAN(network.AP_IF)
sleep(5) # 5 second time delay
ap_if.active(False) # Turn off AP Mode

# Initiate Software SPI Bus:
# --------------------------
# The -1 is software SPI, 0 or 1 is the hardware SPI port, miso pin is placeholder to make library work (miso is not utilized).
spi = SPI(-1, baudrate=9600, polarity=1, phase=0, firstbit=SPI.MSB, sck=Pin(27), mosi=Pin(14), miso=Pin(18))

# Initiate the Encoder library with pin CLK on 12 and pin DT on 13:
# -----------------------------------------------------------------
e = encoderLib.encoder(12, 13)

# Generic Function to Display 6 lines on OLED
# -------------------------------------------
def Display_Stat1(line1,line2,line3,line4,line5,line6,delaytime):
    display.fill(0) # Clear Previous Text
    display.text(line1,0,0,1) #Line 1
    display.text(line2,0,10,1) # Line 2
    display.text(line3,0,20,1) # Line 3
    display.text(line4,0,30,1) #Line 4
    display.text(line5,0,40,1) # Line 5
    display.text(line6,0,50,1) # Line 6
    display.show() # Display New Text
    sleep(delaytime) # Delay secs

# Function Check to see if Station mode wifi is active
# ----------------------------------------------------
def stat_mode1():
    sta1 = WLAN(network.STA_IF)
    if not sta1.isconnected():
        led1.value(0) # Turn off LED
        L1 = "Press Reset"
        L2 = "To Re-Start"
        L3 = "WiFi Connection"
        L4 = "or"
        L5 = "Check Router"
        L6 = "Control Settings"
        delayx = 2
        Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto function
    else: # connected to station wifi
        led1.value(1) # Turn on LED

def do_something_else():
    #Script to be executed besides of the blocking socket
    global freq, mult_index, position, wave_display
    stat_mode1() # Goto Function
    print("Selection is Main Menu") # Used for debug
    print() # Space
    if (freq == 100) and (mult_index == 10000):
        label = "MHz"
        new_Hz = "{0:.2f}".format(freq * mult_index / 1000000) # Convert to a unit and a string
    else:
        label = "KHz"
        new_Hz = "{0:.2f}".format(freq * mult_index / 1000) # Convert to a unit and a string
    # Main Selection Screen Page Stuff
    L1 = " Function Gen."
    L2 = "----------------"
    L3 = "Freq = " + str(round(freq * mult_index)) + "Hz"
    L4 = label + " = " + new_Hz
    L5 = "Amplitude = " + str(position) + "%"
    L6 = wave_display
    delayx = 0.1
```

```
194         Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
195         print("New_Hz = " + new_Hz)
196 
197     def Client_handler(client_obj):
198         #Execute this Script when there's a socket connection
199         global freq, mult_index, position, wave_display
200 
201         html = """<!DOCTYPE html>
202         <html>
203         <head>
204         <meta charset= "utf-8">
205         <!-- The below code refreshes page every 5 seconds -->
206         <meta http-equiv="refresh" content="5" >
207         <title>ESP32 WiFi Function Generator</title>
208         </head>
209         <body bgcolor="#00FFFF"><center><h2>A Simple Webserver for Displaying ESP32
            Function Generator Parameters</h2></center>
210         <center><h3>(Open a Browser, and type the "ESP32 IP Address:80 + Enter" to bring up
            Web Page)</h3></center>
211         <br><br><br><br>
212         <form>
213         <center><h3>Waveform = "%(Waveform)s"</h3></center>
214         <br><br>
215         <center><h3>Frequency = "%(New_Frequency)s"</h3></center>
216         <br><br>
217         <center><h3>Amplitude = "%(Amplitude1)s"</h3></center>
218         </form>
219         </body>
220         </html>
221         """ %dict(Waveform = wave_display, New_Frequency = freq*mult_index, Amplitude1 =
            position)
222 
223         # Send the custom HTML Script
224         client_obj.send(html)
225 
226         # Close the connection with the client
227         client_obj.close()
228 
229     # reserve a port on your computer in our
230     # Next bind to the port
231     # we have not typed any ip in the ip field
232     # instead we have inputted an empty string
233     # this makes the server listen to requests
234     # coming from other computers on the network
235 
236     server_port = 80
237     incoming_addr = ''
238     address = (incoming_addr, server_port)
239     server_socket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
240     print("Socket successfully created")
241     server_socket.bind(address)
242     print("socket binded to %s" %(server_port))
243     # put the socket into listening mode
244     server_socket.listen(5)
245     print("Server socket is listening")
246 
247     # Store Wiper Setting in Memory:
248     # ------------------------------
249     def store():
250         INC.value(1) # Set to High
251         sleep_us(100) # 100uS delay
252         CS.value(0) # Set chip select
253         sleep_ms(50) # 50 mS delay
254         CS.value(1) # Set chip select
255         sleep_ms(50) # 50mS delay
256 
257     # Move POT UP:
```

```
# ------------
def POT_UP(pos_num):
    global position, last
    if pos_num > last: # Check to see if encoder is turned right
        CS.value(0) # Set chip select
        sleep_us(100) # 100uS delay
        UD.value(1) # Set up/down value to up
        INC.value(1) # High to Low transition steps to move wiper
        sleep_us(100) # 100uS delay
        INC.value(0)
        sleep_us(100) # 100uS delay
        position += 1 # Move POT up one step at a time
        if position >= 100: # Force to upper position if it goes past
            position = 100
        store() # Store wiper position
        return position # Return wiper position for display

# Move POT DOWN:
# --------------
def POT_DOWN(neg_num):
    global position, last
    if neg_num < last: # Check to see if encoder is turned left
        CS.value(0) # Set chip select
        sleep_us(100) # 100uS delay
        UD.value(0) # Set up/down value to down
        INC.value(1) # High to Low transition steps to move wiper
        sleep_us(100) # 100uS delay
        INC.value(0)
        sleep_us(100) #100uS delay
        position -= 1 # Move POT down one step at a time
        if position <= 0: # Force to zero position if it goes past
            position = 0
        store() # Store wiper position
        return position # Return wiper position for display

# Move Frequency UP:
# ------------------
def FR_UP(pos_freq):
    global freq, last1
    if pos_freq > last1: # Check to see if encoder is turned right
        freq += 1 # Move frequency up one step at a time
        if freq >= 100: # Force to upper position if it goes past
            freq = 100
        return freq # Return the selected frequency for display

# Move Frequency DOWN:
# --------------------
def FR_DOWN(neg_freq):
    global freq, last1
    if neg_freq < last1: # Check to see if encoder is turned left
        freq -= 1 # Move frequency down one step at a time
        if freq <= 0: # Force to first position if it goes past
            freq = 1
        return freq # Return the selected frequency for display

# Move Waveform UP:
# -----------------
def WAVEFRM_UP(pos_wv):
    global wave, last3
    if pos_wv > last3: # Check to see if encoder is turned left
        wave += 1 # Move wave down one step at a time
        if wave >= 2: # Force to upper position if it goes past
            wave = 2
        return wave # Return wiper position

# Move Waveform DOWN:
# -------------------
```

```
def WAVEFRM_DOWN(neg_wv):
    global wave, last3
    if neg_wv < last3: # Check to see if encoder is turned left
        wave -= 1 # Move wave down one step at a time
        if wave <= 0: # Force to zero position if it goes past
            wave = 0
        return wave # Return wiper position

# Move Multiplier UP:
# -------------------
def MULT_UP(pos_mult):
    global mult, last2, mult_index
    if pos_mult > last2: # Check to see if encoder is turned right
        mult += 1 # Move frequency multiplier up one step at a time
        if mult >= 4: # Force to upper position if it goes past
            mult = 4
        return mult # Return the selected mult

# Move Multiplier DOWN:
# ---------------------
def MULT_DOWN(neg_mult):
    global mult, last2, mult_index
    if neg_mult < last2: # Check to see if encoder is turned left
        mult -= 1 # Move frequency multiplier down one step at a time
        if mult <= 0: # Force to zero position if it goes past
            mult = 0
        return mult # Return the selected mult

# Reset POT to Zero Position:
# ---------------------------
def Reset_Pot():
    global position
    sleep_ms(50) # Stabilization Time
    for i in range(100, -1, -1): # Count down by one and reset low
        CS.value(0) # Set chip select
        sleep_us(100) # 100uS delay
        UD.value(0) # Set up/down value to down
        INC.value(1) # High to Low transition steps to move wiper
        sleep_us(100) # 100uS delay
        INC.value(0)
        sleep_us(100) # 100uS delay
        position -= 1 # Move position down
        if position <= 0: # Force to zero position if it goes past
            position = 0
    store() # Store wiper position
    return position  # Return wiper position for display

# Main Menu Selection:
# --------------------
def menu(rst = 0):
    global count, hold
    if rst == 1: # Reset count variable
        count = 0
    if ENC_SWITCH.value() == 1: # Encoder button is UP
        hold = 1 # State to unlock entry
    if (ENC_SWITCH.value() == 0) and (hold == 1): # See if Encoder Switch was depressed
        sleep_ms(200) # 200mS delay for de-bounce
        count += 1 # Increase count by one
        if count > 7: # menu choices (0-7)
            count = 0 # Reset Menu selection
        hold = 0  # Lock Entry
    print("Count is :" + str(count))
    return count # Return Menu selection

# Waveform Generator Function:
# ----------------------------
def generator(Frequency = 10, multiplier = 1, Waveform = 0x2000, Phase = 0):
```

```
392         global ClockFreq
393 
394         # Calculate Frequency "Word" to send:
395         # ----------------------------------
396         if Waveform == 0x2020:
397             word = hex(round((2*Frequency*multiplier*2**28)/ClockFreq)) # Doble frequency
                for square wave only
398         else:
399             word = hex(round((Frequency*multiplier*2**28)/ClockFreq))
400 
401         # Split Frequency "Word" into separate "bytes" (14 bits each):
402         # ------------------------------------------------------------
403         MSB = (int(word) & 0xFFFC000)>>14
404         LSB = int(word) & 0x3FFF
405 
406         # Set control bits DB15 to 0 and DB14 to 1; for frequency register 0:
407         # -------------------------------------------------------------------
408         MSB |= 0x4000 # Logic "Or" with MSB above and 4000 hex
409         LSB |= 0x4000 # Logic "Or" with LSB above and 4000 hex
410 
411         # Set the Control Register and Send:
412         # ----------------------------------
413         Send(0x2100)
414 
415         # Set the frequency and Send:
416         # ---------------------------
417         Send(LSB) #Lower 14 bits (Goto Send Function)
418         Send(MSB) #Upper 14 bits (Goto Send Function)
419 
420         # Set Phase Register and Send(if Using):
421         # --------------------------------------
422         #Send(Phase |= 0xC000) # Set the phase shift
423 
424         # Set the Waveform, Send and Exit:
425         # --------------------------------
426         Send(Waveform) # square: 0x2020, sin: 0x2000, triangle: 0x2002) # Goto Send
            Function
427 
428     # Function to send the data:
429     # --------------------------
430     def Send(data):
431         high = data >>8 # Get the High Byte
432         low = data & 0xFF # Get the Low Byte
433         FSYNC.value(0) # Make Sync pin Low
434         spi.write(bytearray([high])) # Write the High Byte Array
435         spi.write(bytearray([low])) # Write the Low Byte Array
436         FSYNC.value(1) # Make Sync pin High
437 
438     # Display Start Message(s)
439     # ------------------------
440     L1 = "Digital Funct."
441     L2 = "Generator"
442     L3 = "Created By"
443     L4 = "Roy Guerra Jr."
444     L5 = "Starting....."
445     L6 = "Please Wait"
446     delayx = 3
447     Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
448 
449     # Next Message
450     # ------------
451     L1 = "Frequency"
452     L2 = "Range Is:"
453     L3 = ""
454     L4 = "10Hz - 1MHz"
455     L5 = ""
456     L6 = ""
```

```
457   delayx = 3
458   Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
459
460   # Next Message
461   # ------------
462   L1 = "Push Encoder"
463   L2 = "Knob To:"
464   L3 = ""
465   L4 = "Change The"
466   L5 = "Menu"
467   L6 = "Parameter(s)"
468   delayx = 3
469   Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
470
471   # Next Message
472   # ------------
473   L1 = "Turn Encoder"
474   L2 = "Left or Right:"
475   L3 = ""
476   L4 = "To Change The"
477   L5 = "Value"
478   L6 = "Parameter(s)"
479   delayx = 3
480   Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
481
482   # Next Message
483   # ------------
484   L1 = "Server Started"
485   L2 = "Open Web Browser"
486   L3 = "  Type in: "
487   L4 = "IP Address:80"
488   L5 = "Only When Using"
489   L6 = "Main Menu"
490   delayx = 5
491   Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
492
493   # Main Program:
494   # -------------
495   Reset_Pot() # Goto Reset POT Function
496   generator() # Start generator with initial values in Function declared above
497   while True: # Infinite Loop
498       try:
499           selection = menu() # Goto Menu Select Function, and use initial parameters
500           print("Selection is : " + str(selection))
501           z = 0 # Reset timeout variable
502           while (selection >= 0) and (selection < 5):
503               menu_rst = 0 # Menu Reset Variable
504               selection = menu(menu_rst) # Goto Menu Select Function
505               sleep_ms(1) # 1mS delay for timeout loop
506               z +=1 # Increment counter
507               print("Z = " + str(z)) # Used for Debug
508               print() # Space
509               if z >= 1600: # Approx. 60 second timeout
510                   z = 0 # Reset timeout variable
511                   print("Selection Timed Out")  # Used for Debug
512                   print() # Space
513                   # Main Selection Screen Page Stuff
514                   L1 = "Menu"
515                   L2 = "Timeout"
516                   L3 = ""
517                   L4 = "Going To:"
518                   L5 = "Main Menu"
519                   L6 = ""
520                   delayx = 2
521                   Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
522                   if (mult_index == 10000) and (freq >= 100): # Force to upper frequency
                      position if Multiplier is 10000
```

```
523                     freq = 100
524                 elif (mult_index == 1) and (freq <= 10): # Force to lower frequency
                    position of 10 for given multiplier
525                     freq = 10
526                 elif (mult_index == 10) and (freq <= 1): # Force to lower frequency
                    position of 1 for given multiplier
527                     freq = 1
528                 generator(freq, mult_index, waveform, phase) # Goto Function
529                 count = 5 # Set up for Main Menu after Break Statement
530                 break # Exit this inner loop to main menu
531             if selection == 0: # Waveform Stuff
532                 wv = e.getValue()  # Get rotary encoder value
533                 if wv != last3:  # See if there is a new value
534                     z = 0 # Reset timeout variable
535                     print("Encoder Value = " + str(wv))  # Used for Debug
536                     print() # Space
537                     print ("Waveform = " + str(wave)) # Used for Debug
538                     print() # Space
539                     WAVEFRM_UP(wv) # Goto multiplier Up Function
540                     WAVEFRM_DOWN(wv) # Goto multiplier DOWN Function
541                     last3 = wv # Store new value
542                 # square: 0x2020, sin: 0x2000, triangle: 0x2002
543                 if wave == 0:
544                     wave_display = "Sine Wave"
545                     waveform = 0x2000
546                 elif wave == 1:
547                     wave_display = "Square Wave"
548                     waveform = 0x2020
549                 elif wave == 2:
550                     wave_display = "Triangle Wave"
551                     waveform = 0x2002
552                 else:
553                     print("wave_display = Error")
554                 # Display Routine for Waveform
555                 L1 = "Function Gen."
556                 L2 = "Waveform Type"
557                 L3 = "Is:"
558                 L4 =  wave_display
559                 L5 = "Choices are sine"
560                 L6 = "square, triangle"
561                 delayx = 0.01
562                 Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
563                 print("Selection is Waveform")  # Used for Debug
564                 print() # Space
565             elif selection == 1: # Frequency Stuff
566                 fr = e.getValue()  # Get rotary encoder value
567                 if fr != last1:  # See if there is a new value
568                     z = 0 # Reset timeout variable
569                     print("Encoder Value = " + str(fr))  # Used for Debug
570                     print() # Space
571                     print ("Frequency = " + str(freq)) # Used for Debug
572                     print() # Space
573                     FR_UP(fr) # Goto frequency Up Function
574                     FR_DOWN(fr) # Goto frequency DOWN Function
575                     last1 = fr # Store new value
576                 # Display Routine for Frequency
577                 L1 = "Function Gen."
578                 L2 = "Frequency in Hz"
579                 L3 = ""
580                 L4 = "Is: " + str(freq)
581                 L5 = ""
582                 L6 = "Range is 0-100"
583                 delayx = 0.01
584                 Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
585                 print("Selection is Frequency") # Used for Debug
586                 print() # Space
587             elif selection == 2:  # X1, X10, X100, X1000, X10000
```

```
                mu = e.getValue()  # Get rotary encoder value
                if mu != last2:  # See if there is a new value
                    z = 0 # Reset timeout variable
                    print("Encoder Value = " + str(mu))  # Used for Debug
                    print() # Space
                    print ("Mult. Index = " + str(mult)) # Used for Debug
                    print() # Space
                    MULT_UP(mu) # Goto multiplier Up Function
                    MULT_DOWN(mu) # Goto multiplier DOWN Function
                    last2 = mu # Store new value
                if mult == 0 :
                    mult_index = 1 # X1 Scale
                elif mult == 1 :
                    mult_index = 10 # X10 Scale
                elif mult == 2 :
                    mult_index = 100 # X100 Scale
                elif mult == 3 :
                    mult_index = 1000 # X1000 Scale
                elif mult == 4 :
                    mult_index = 10000 # X10000 Scale
                # Display Routine for Scale Multiplier
                L1 = "Function Gen."
                L2 = "Multiplier Index"
                L3 = "Is:"
                L4 = str(mult_index)
                L5 = ""
                L6 = ""
                delayx = 0.01
                Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
                print("Selection is Multiplier") # Used for Debug
                print() # Space
            elif selection == 3: # Digital POT
                value = e.getValue()  # Get rotary encoder value
                if value != last:  # See if there is a new value
                    z = 0 # Reset timeout variable
                    print("Encoder Value = " + str(value))  # Used for Debug
                    print() # Space
                    print ("POT Position = " + str(position)) # Used for Debug
                    print() # Space
                    POT_UP(value) # Goto Pot Up Function
                    POT_DOWN(value) # Goto POT DOWN Function
                    last = value # Store new value
                # Display Routine for POT position
                L1 = "Function Gen."
                L2 = "Signal Amplitude"
                L3 = ""
                L4 = "Is: " + str(position) + "%"
                L5 = ""
                L6 = "Range = 0-100%"
                delayx = 0.01
                Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
                print("Selection is Amplitude")  # Used for Debug
                print() # Space
            elif selection == 4:
                print("Selection is Menu Exit") # Used for debug
                print() # Space
                if (mult_index == 10000) and (freq >= 100): # Force to upper frequency
                position if Multiplier is 10000
                    freq = 100
                elif (mult_index == 1) and (freq <= 10): # Force to lower frequency
                position of 10 for given multiplier
                    freq = 10
                elif (mult_index == 10) and (freq <= 1): # Force to lower frequency
                position of 1 for given multiplier
                    freq = 1
                generator(freq, mult_index, waveform, phase) # Goto Function
                count = 5 # Set up for Main Menu after Break Statement
```

```
652                 break # Exit While Loop
653         if selection == 5: # Main Menu
654             r, w, err = select.select((server_socket,), (), (), 1)
655             if r:
656                 for readable in r:
657                     client, client_address = server_socket.accept()
658                     print("Got a connection from ", client_address)
659                     try:
660                         Client_handler(client) # Goto Function
661                     except Exception as e:
662                         print("A New Program Exception is: ")
663                         print(e)
664                         pass
665             do_something_else() # Goto Function
666         if selection >= 6: # Reset menu to beginning
667             print("Selection is Reset") # Used for debug
668             print() # Space
669             menu_rst = 1 # Menu Reset Variable
670             menu(menu_rst) # Goto Menu Function
671 
672     except Exception as e:
673                         print("A New Program Exception is: ")
674                         print(e)
675                         # Main Selection Screen Page Stuff
676                         L1 = "Bad Hardware"
677                         L2 = ""
678                         L3 = "and / or a"
679                         L4 = ""
680                         L5 = "Program Issue"
681                         L6 = ""
682                         delayx = 2
683                         Display_Stat1(L1,L2,L3,L4,L5,L6,delayx) # Goto Display
                            function
684                         #pass
685 
686 
```

```
"""
Wifi Manager Routine
====================
"""

# Import Modules
# --------------
import network
import socket
import ure
import time
import os
import machine
from network import WLAN
from machine import Pin, I2C
from time import sleep
from ssd1306 import SSD1306_I2C

# Constants
# ---------
ap_ssid = "ESP32 Gen. WifiManager"
ap_password = "password"
ap_authmode = 3  # WPA2
WIFI_LED_PIN = 25 # ESP32 on board LED
OLED_RST_PIN = 16 # ESP32 OLED display Reset pin

# Initial Setup
# -------------
led = Pin(WIFI_LED_PIN, Pin.OUT, value=0) # Sets up LED and starts
OLED_RST = Pin(OLED_RST_PIN, Pin.OUT, value=1) # Sets up OLED RST to high
i2c = I2C(sda = Pin(4), scl = Pin(15)) # ESP32 OLED I2C Pins
display = SSD1306_I2C(128, 64, i2c) # ESP32 I2C display resolution

# Network Data File
# -----------------
NETWORK_PROFILES = 'wifi.dat'

# Access Point and Staion Network Class Objects
# ---------------------------------------------
ap = WLAN(network.AP_IF)
sta = WLAN(network.STA_IF)

# Generic Function to Display 6 lines on OLED
# -------------------------------------------
def Display_Stat(line1,line2,line3,line4,line5,line6,delaytime):
    display.fill(0) # Clear Previous Text
    display.text(line1,0,0,1) #Line 1
    display.text(line2,0,10,1) # Line 2
    display.text(line3,0,20,1) # Line 3
    display.text(line4,0,30,1) #Line 4
    display.text(line5,0,40,1) # Line 5
    display.text(line6,0,50,1) # Line 6
    display.show() # Display New Text
    sleep(delaytime) # Delay secs

# Function to Reboot the ESP32
# ----------------------------
def reboot():
    print("ESP32 is Rebooting....")
    L1 = "SSID and Pass"
    L2 = "Is Saved in a"
    L3 = ""
    L4 = "Configuration"
    L5 = "File"
    L6 = ""
    delayx = 3
    Display_Stat(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
```

```
68      machine.reset() # Hardware reset command
69
70  server_socket = None
71
72  # Function to return a working STA instance or "None" on the Main.py Program
73  # ----------------------------------------------------------------------------
74  def get_connection():
75      L1 = "Hello"
76      L2 = "Starting Wi-Fi"
77      L3 = "Let's Go..."
78      L4 = "Attemping to"
79      L5 = "Connect....."
80      L6 = "Please Wait"
81      delayx = 3
82      Display_Stat(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
83      if sta.isconnected(): # First check if there already is any connection:
84          led.value(1) # Turn on LED
85          return sta
86      connected = False
87      try: # ESP32 WiFi takes time, wait a bit and try once more
88          time.sleep(5) # 5 second delay
89          if sta.isconnected():
90              led.value(1) # Turn on LED
91              return sta
92          # Read known network profiles from file
93          profiles = read_profiles()
94          # Search WiFis networks in range
95          sta.active(True)
96          networks = sta.scan()
97          AUTHMODE = {0: "open", 1: "WEP", 2: "WPA-PSK", 3: "WPA2-PSK", 4: "WPA/WPA2-PSK"}
98          for ssid, bssid, channel, rssi, authmode, hidden in sorted(networks, key=lambda
            x: x[3], reverse=True):
99              ssid = ssid.decode('utf-8')
100             encrypted = authmode > 0
101             print("ssid: %s chan: %d rssi: %d authmode: %s" % (ssid, channel, rssi,
                AUTHMODE.get(authmode, '?')))
102             if encrypted:
103                 if ssid in profiles:
104                     password = profiles[ssid]
105                     connected = do_connect(ssid, password)
106                 else:
107                     print("skipping unknown encrypted network")
108             else:  # open
109                 connected = do_connect(ssid, None) # Unencrypted network
110             if connected:
111                 break
112     except OSError as e:
113         print("exception", str(e))
114
115     # start web server for connection manager:
116     if not connected:
117         connected = start()
118     return sta if connected else None
119
120 # Function to read Network SSID Profiles
121 # --------------------------------------
122 def read_profiles():
123     with open(NETWORK_PROFILES) as f:
124         lines = f.readlines()
125     profiles = {}
126     for line in lines:
127         ssid, password = line.strip("\n").split(";")
128         profiles[ssid] = password
129     return profiles
130
131 # Function to Write Network SSID Profiles
132 # ---------------------------------------
```

```
133 def write_profiles(profiles):
134     lines = []
135     for ssid, password in profiles.items():
136         lines.append("%s;%s\n" % (ssid, password))
137     with open(NETWORK_PROFILES, "w") as f:
138         f.write(''.join(lines))
139 
140 # Function to connect WiFi
141 # ------------------------
142 def do_connect(ssid, password):
143     sta.active(True)
144     if sta.isconnected():
145         return None
146     print('Trying to connect to %s...' % ssid)
147     sta.connect(ssid, password)
148     attempt = 0
149     for retry in range(11):
150         connected = sta.isconnected()
151         if connected:
152             break
153         L1 = "Trying to"
154         L2 = "Connect....."
155         L3 = ""
156         L4 = "Attempt: " + str(attempt)
157         L5 = ""
158         L6 = ""
159         delayx = 1
160         Display_Stat(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
161         attempt += 1
162         print("Attempts = " + str(attempt))
163     if connected:
164         print('\nConnected. Network config: ', sta.ifconfig())
165         led.value(1) # Turn on LED
166         #ap.active(False) # Turn off AP Mode
167         L1 = "WiFi Connected!"
168         L2 = "IP Address Is:"
169         L3 = str(sta.ifconfig())
170         L4 = ""
171         L5 = "Returning to"
172         L6 = "Main Program"
173         delayx = 3
174         Display_Stat(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
175     else:
176         print('\nFailed. Not Connected to: ' + ssid)
177         led.value(0) # Turn off LED
178         L1 = "Wi-Fi"
179         L2 = "Connection"
180         L3 = "Failed"
181         L4 = ""
182         L5 = ""
183         L6 = ""
184         delayx = 3
185         Display_Stat(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
186     return connected
187 
188 # Function to Send Web Page Total Response
189 # ----------------------------------------
190 def send_response(client, payload, status_code=200):
191     client.sendall("HTTP/1.0 {} OK\r\n".format(status_code))
192     client.sendall("Content-Type: text/html\r\n")
193     client.sendall("Content-Length: {}\r\n".format(len(payload)))
194     client.sendall("\r\n")
195     if len(payload) > 0:
196         client.sendall(payload)
197 
198 # Function to Display HTML Web Browser
199 # ------------------------------------
```

```
def handle_root(client):
    sta.active(True)
    ssids = sorted(ssid.decode('utf-8') for ssid, *_ in sta.scan())
    response = []
    response.append("""\
        <html>
          <body bgcolor="#00FFFF">
            <h1 style="color: #5e9ca0; text-align: center;">
                <span style="color: #ff0000;">
                    ESP32 WiFi Setup
                </span>
            </h1>
            <form action="configure" method="post">
                <table style="margin-left: auto; margin-right: auto;">
                    <tbody>
    """)
    for ssid in ssids:
        response.append("""\
                        <tr>
                            <td colspan="2">
                                <input type="radio" name="ssid" value="{0}" />{0}
                            </td>
                        </tr>
        """.format(ssid))

    response.append("""\
                        <tr>
                            <td>Password:</td>
                            <td><input name="password" type="password" /></td>
                        </tr>
                    </tbody>
                </table>
                <p style="text-align: center;">
                    <input type="submit" value="Submit" />
                </p>
            </form>
            <p> </p>
            <hr />
            <h5>
                <span style="color: #ff0000;">
                    Your ssid and password information will be saved into the
                    "%(filename)s" file in your ESP module for future usage.
                    Be careful about security!
                </span>
            </h5>
            <hr />
          </body>
        </html>
    """ % dict(filename=NETWORK_PROFILES))
    send_response(client, "\n".join(response))

# Function to Get SSID & Password
# -------------------------------
def handle_configure(client, request):
    match = ure.search("ssid=([^&]*)&password=(.*)", request)
    if match is None:
        send_response(client, "Parameters not found", status_code=400)
        return False
    try:
        ssid = match.group(1).decode("utf-8").replace("%3F", "?").replace("%21", "!")
        password = match.group(2).decode("utf-8").replace("%3F", "?").replace("%21", "!")
    except:
        ssid = match.group(1).replace("%3F", "?").replace("%21", "!")
        password = match.group(2).replace("%3F", "?").replace("%21", "!")
    if len(ssid) == 0:
        send_response(client, "SSID must be provided", status_code=400)
        return False
```

```
267         if do_connect(ssid, password):
268             response = """\
269                 <html>
270                     <center>
271                         <br><br>
272                         <h1 style="color: #5e9ca0; text-align: center;">
273                             <span style="color: #ff0000;">
274                                 ESP32 successfully connected to WiFi network %(ssid)s.
275                             </span>
276                         </h1>
277                         <br><br>
278                     </center>
279                 </html>
280             """ % dict(ssid=ssid)
281             send_response(client, response)
282             try:
283                 profiles = read_profiles()
284             except OSError:
285                 profiles = {}
286             profiles[ssid] = password
287             write_profiles(profiles)
288             return True
289         else:
290             response = """\
291                 <html>
292                     <center>
293                         <h1 style="color: #5e9ca0; text-align: center;">
294                             <span style="color: #ff0000;">
295                                 ESP32 could not connect to WiFi network %(ssid)s.
296                             </span>
297                         </h1>
298                         <br><br>
299                         <form>
300                             <input type="button" value="Go back!"
                                onclick="history.back()"></input>
301                         </form>
302                     </center>
303                 </html>
304             """ % dict(ssid=ssid)
305             send_response(client, response)
306             return False
307 
308     # Path Error Function
309     # -------------------
310     def handle_not_found(client, url):
311         send_response(client, "Path not found: {}".format(url), status_code=404)
312 
313     # Web Socket Close Function
314     # -------------------------
315     def stop():
316         global server_socket
317         if server_socket:
318             server_socket.close()
319             server_socket = None
320 
321     # Web Server Start Function
322     # -------------------------
323     def start(port=80):
324         global server_socket
325         addr = socket.getaddrinfo('0.0.0.0', port)[0][-1]
326         stop()
327         sta.active(True)
328         ap.active(True)
329         ap.config(essid=ap_ssid, password=ap_password, authmode=ap_authmode)
330         server_socket = socket.socket()
331         server_socket.bind(addr)
332         server_socket.listen(1)
```

```
333         print('Connect to WiFi ssid ' + ap_ssid + ', default password: ' + ap_password)
334         print('and access the ESP via your favorite web browser at 192.168.4.1.')
335         print('Listening on:', addr)
336         L1 = "Server Started"
337         L2 = "Connect to AP"
338         L3 = "Type 'password'"
339         L4 = "Open Web Browser"
340         L5 = "Type in Address"
341         L6 = "192.168.4.1"
342         delayx = 3
343         Display_Stat(L1,L2,L3,L4,L5,L6,delayx) # Goto Display function
344         while True:
345             if sta.isconnected():
346                 return True
347             client, addr = server_socket.accept()
348             print('client connected from', addr)
349             try:
350                 client.settimeout(5.0)
351                 request = b""
352                 try:
353                     while "\r\n\r\n" not in request:
354                         request += client.recv(512)
355                 except OSError:
356                     pass
357                 print("Request is: {}".format(request))
358                 if "HTTP" not in request:
359                     # skip invalid requests
360                     continue
361                 # version 1.9 compatibility
362                 try:
363                     url = ure.search("(?:GET|POST) /(.*?)(?:\\?.*?)? HTTP",
                        request).group(1).decode("utf-8").rstrip("/")
364                 except:
365                     url = ure.search("(?:GET|POST) /(.*?)(?:\\?.*?)? HTTP",
                        request).group(1).rstrip("/")
366                 print("URL is {}".format(url))
367                 if url == "":
368                     handle_root(client)
369                 elif url == "configure":
370                     handle_configure(client, request)
371                 else:
372                     handle_not_found(client, url)
373             finally:
374                 client.close()
375                 #reboot() # Goto Reboot function
376
```

```
from machine import Pin
from machine import Timer

class encoder:
    # Init variables
    encoder_clk_prev = False
    i = 0

    def __init__(self, clk_pin, dt_pin):
        # Configure the rotary encoder pins and interupt
        self.clk = Pin(clk_pin, Pin.IN, Pin.PULL_UP)
        self.dt = Pin(dt_pin, Pin.IN, Pin.PULL_UP)

        tim = Timer(-1)
        tim.init(  # Timer to run self.update every 5ms
            period=5,
            mode=Timer.PERIODIC,
            callback=self.update
        )

    def getValue(self):
        return(self.i)  # Return rotary encoder value

    # Non blocking delay of 5ms
    def update(self, p):
        # Read the rotary encoder pins
        self.encoder_clk = self.clk.value()
        self.encoder_dt = self.dt.value()

        # If rotary encoder rotated
        if not self.encoder_clk and self.encoder_clk_prev:
            # Get direction of rotation
            if self.encoder_dt:
                self.i += 1
            else:
                self.i -= 1

        self.encoder_clk_prev = self.encoder_clk


```

```
# MicroPython SSD1306 OLED driver, I2C and SPI interfaces

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()

    def poweroff(self):
        self.write_cmd(SET_DISP | 0x00)
```

```
68
69      def contrast(self, contrast):
70          self.write_cmd(SET_CONTRAST)
71          self.write_cmd(contrast)
72
73      def invert(self, invert):
74          self.write_cmd(SET_NORM_INV | (invert & 1))
75
76      def show(self):
77          x0 = 0
78          x1 = self.width - 1
79          if self.width == 64:
80              # displays with width of 64 pixels are shifted by 32
81              x0 += 32
82              x1 += 32
83          self.write_cmd(SET_COL_ADDR)
84          self.write_cmd(x0)
85          self.write_cmd(x1)
86          self.write_cmd(SET_PAGE_ADDR)
87          self.write_cmd(0)
88          self.write_cmd(self.pages - 1)
89          self.write_framebuf()
90
91      def fill(self, col):
92          self.framebuf.fill(col)
93
94      def pixel(self, x, y, col):
95          self.framebuf.pixel(x, y, col)
96
97      def scroll(self, dx, dy):
98          self.framebuf.scroll(dx, dy)
99
100     def text(self, string, x, y, col=1):
101         self.framebuf.text(string, x, y, col)
102
103
104 class SSD1306_I2C(SSD1306):
105     def __init__(self, width, height, i2c, addr=0x3c, external_vcc=False):
106         self.i2c = i2c
107         self.addr = addr
108         self.temp = bytearray(2)
109         # Add an extra byte to the data buffer to hold an I2C data/command byte
110         # to use hardware-compatible I2C transactions.  A memoryview of the
111         # buffer is used to mask this byte from the framebuffer operations
112         # (without a major memory hit as memoryview doesn't copy to a separate
113         # buffer).
114         self.buffer = bytearray(((height // 8) * width) + 1)
115         self.buffer[0] = 0x40  # Set first byte of data buffer to Co=0, D/C=1
116         self.framebuf = framebuf.FrameBuffer1(memoryview(self.buffer)[1:], width, height)
117         super().__init__(width, height, external_vcc)
118
119     def write_cmd(self, cmd):
120         self.temp[0] = 0x80 # Co=1, D/C#=0
121         self.temp[1] = cmd
122         self.i2c.writeto(self.addr, self.temp)
123
124     def write_framebuf(self):
125         # Blast out the frame buffer using a single I2C transaction to support
126         # hardware I2C interfaces.
127         self.i2c.writeto(self.addr, self.buffer)
128
129     def poweron(self):
130         pass
131
132
133 class SSD1306_SPI(SSD1306):
134     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()
```