Air Quality Monitoring System

Objective: Air Quality Monitoring System is a real-time tracking system which analyzes the air quality using IoT devices and visualizes the percentage of air pollutants present in the atmosphere. By using this system, society can check the air quality of the city where they are living.

IoT Technology and Sensor: The IoT technology which we decided to use is NodeMCU(WiFi-ESP32) and the sensor we are going to use is BME680 which is a gas sensor which detects temperature, humidity, pressure and air quality. The data from the gas sensor will be displayed to the client as real-time data through esp-32.

Project Overview: Start by clearly defining the project's scope and objectives. What do you want to achieve with your air quality monitoring system? This might include monitoring pollutants like particulate matter (PM2.5, PM10), carbon dioxide (CO2), volatile organic compounds (VOCs), temperature, and humidity.

Hardware Selection: Choose the appropriate hardware for your project. ESP32 is a popular IoT microcontroller, and BM3680 seems to be a sensor module. Ensure that these components are compatible and can measure the required parameters. For instance, the BM3680 could contain sensors for air quality, and the ESP32 would serve as the controller.

Sensor Calibration: Calibrate the sensors to ensure accurate readings. This step is essential to correct any sensor drift or inaccuracies.

Wiring and Assembling: Connect the BM3680 sensor module to the ESP32. Pay close attention to the datasheets and pin configurations to ensure proper wiring. You may need additional components like resistors and capacitors.

Programming the ESP32: Write a Python script for the ESP32 using the MicroPython or CircuitPython environment. This script should configure the sensor, read data from it, and transmit the data to a central server or cloud platform. Make sure the script runs at regular intervals to continuously monitor air quality.

Connectivity: Implement connectivity options for the ESP32 to transmit data. You can use Wi-Fi, cellular, or other communication methods based on your project requirements. MQTT or HTTP can be used for data transfer.

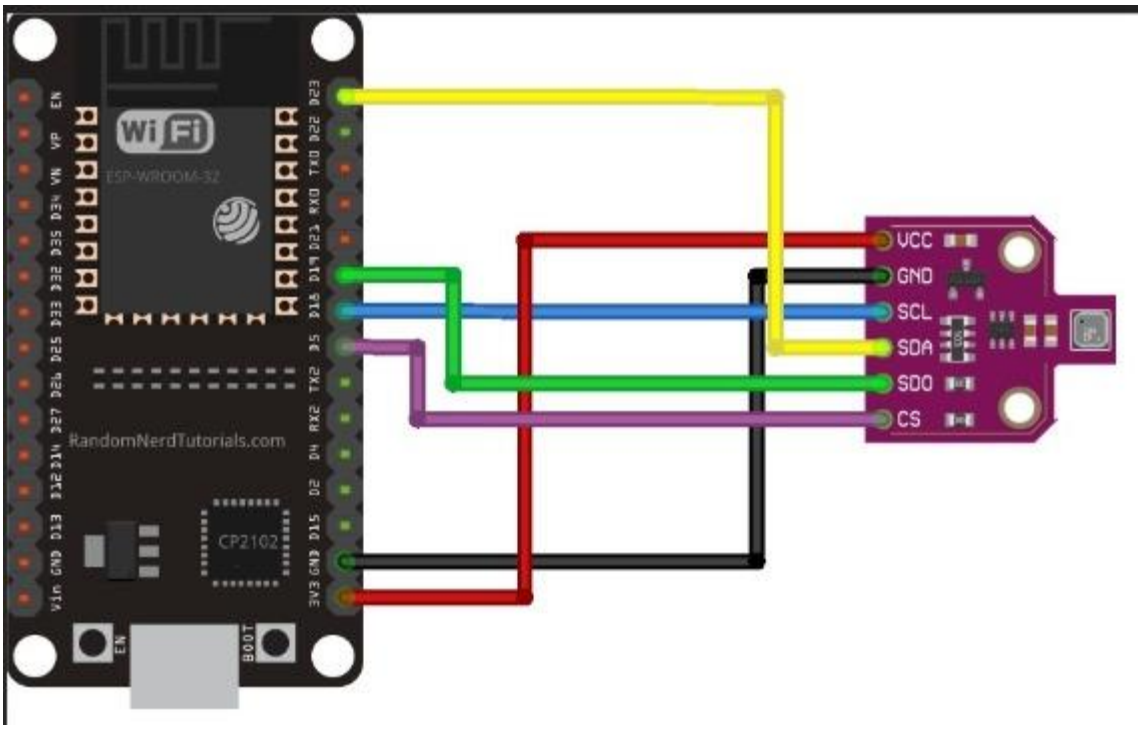
Data Storage and Visualization: Set up a data storage solution to collect and store the air quality data. You can use a database system or cloud services like AWS, Google Cloud, or Azure. Create a web interface or use a dashboard to visualize the data in real-time.

Alerting and Notifications: Implement an alerting system to notify users or administrators when air quality reaches predefined thresholds. This could be done through email, SMS, or push notifications.

Power Management: Optimize power consumption to prolong the battery life of the IoT devices. Use sleep modes when the device is not actively monitoring air quality.

Deployment: Deploy the air quality monitoring system in the intended locations, ensuring that the devices are securely mounted and protected from environmental factors.

Data Analysis and Reporting: Over time, analyze the collected data to identify trends and patterns in air quality.



Program:

```
import time
import math
from micropython import const
from ubinascii import hexlify as hex
try:
  import struct
except ImportError:
  import ustruct as struct
_BME680_CHIPID = const(0x61)
_BME680_REG_CHIPID = const(0xD0)
```

```
_BME680_BME680_COEFF_ADDR1 = const(0x89)
_BME680_BME680_COEFF_ADDR2 = const(0xE1)
_BME680_BME680_RES_HEAT_0 = const(0x5A)
_BME680_BME680_GAS_WAIT_0 = const(0x64)
_BME680_REG_SOFTRESET = const(0xE0)
_BME680_REG_CTRL_GAS = const(0x71)
_BME680_REG_CTRL_HUM = const(0x72)
_BME280_REG_STATUS = const(0xF3)
_BME680_REG_CTRL_MEAS = const(0x74)
_BME680_REG_CONFIG = const(0x75)
_BME680_REG_PAGE_SELECT = const(0x73)
_BME680_REG_MEAS_STATUS = const(0x1D)
_BME680_REG_PDATA = const(0x1F)
_BME680_REG_TDATA = const(0x22)
_BME680_REG_HDATA = const(0x25)
_BME680_SAMPLERATES = (0, 1, 2, 4, 8, 16)
_BME680_FILTERSIZES = (0, 1, 3, 7, 15, 31, 63, 127)
_BME680_RUNGAS = const(0x10)
_LOOKUP_TABLE_1 = (2147483647.0, 2147483647.0, 2147483647.0,
2147483647.0, 2147483647.0,
  2126008810.0, 2147483647.0, 2130303777.0, 2147483647.0,
2147483647.0,
```

```
  2143188679.0, 2136746228.0, 2147483647.0, 2126008810.0,
2147483647.0,
  2147483647.0)
_LOOKUP_TABLE_2 = (4096000000.0, 2048000000.0, 1024000000.0,
512000000.0, 255744255.0, 127110228.0,
  64000000.0, 32258064.0, 16016016.0, 8000000.0, 4000000.0,
2000000.0, 1000000.0,
  500000.0, 250000.0, 125000.0)
def _read24(arr):
  ret = 0.0
  for b in arr:
    ret *= 256.0
    ret += float(b & 0xFF)
  return ret
class Adafruit_BME680:
  def __init__(self, *, refresh_rate=10):
    self._write(_BME680_REG_SOFTRESET, [0xB6])
    time.sleep(0.005)
    chip_id = self._read_byte(_BME680_REG_CHIPID)
    if chip_id != _BME680_CHIPID:
      raise RuntimeError('Failed 0x%x' % chip_id)
    self._read_calibration()
    self._write(_BME680_BME680_RES_HEAT_0, [0x73])
```

```
    self._write(_BME680_BME680_GAS_WAIT_0, [0x65])
    self.sea_level_pressure = 1013.25
    self._pressure_oversample = 0b011
    self._temp_oversample = 0b100
    self._humidity_oversample = 0b010
    self._filter = 0b010
    self._adc_pres = None
    self._adc_temp = None
    self._adc_hum = None
    self._adc_gas = None
    self._gas_range = None
    self._t_fine = None
    self._last_reading = 0
    self._min_refresh_time = 1000 / refresh_rate
  @property
  def pressure_oversample(self):
    return _BME680_SAMPLERATES[self._pressure_oversample]
  @pressure_oversample.setter
  def pressure_oversample(self, sample_rate):
    if sample_rate in _BME680_SAMPLERATES:
      self._pressure_oversample =
_BME680_SAMPLERATES.index(sample_rate)
```

```
    else:
      raise RuntimeError("Invalid")
  @property
  def humidity_oversample(self):
    return _BME680_SAMPLERATES[self._humidity_oversample]
  @humidity_oversample.setter
  def humidity_oversample(self, sample_rate):
    if sample_rate in _BME680_SAMPLERATES:
      self._humidity_oversample =
_BME680_SAMPLERATES.index(sample_rate)
    else:
      raise RuntimeError("Invalid")
  @property
  def temperature_oversample(self):
      return _BME680_SAMPLERATES[self._temp_oversample]
  @temperature_oversample.setter
  def temperature_oversample(self, sample_rate):
    if sample_rate in _BME680_SAMPLERATES:
      self._temp_oversample =
_BME680_SAMPLERATES.index(sample_rate)
    else:
      raise RuntimeError("Invalid")
```

```
@property
def filter_size(self):
  return _BME680_FILTERSIZES[self._filter]
@filter_size.setter
def filter_size(self, size):
  if size in _BME680_FILTERSIZES:
    self._filter = _BME680_FILTERSIZES[size]
  else:
    raise RuntimeError("Invalid")
@property
def temperature(self):
  self._perform_reading()
  calc_temp = (((self._t_fine * 5) + 128) / 256)
  return calc_temp / 100
@property
def pressure(self):
  self._perform_reading()
  var1 = (self._t_fine / 2) - 64000
  var2 = ((var1 / 4) * (var1 / 4)) / 2048
  var2 = (var2 * self._pressure_calibration[5]) / 4
  var2 = var2 + (var1 * self._pressure_calibration[4] * 2)
  var2 = (var2 / 4) + (self._pressure_calibration[3] * 65536)
```

```
    var1 = (((((var1 / 4) * (var1 / 4)) / 8192) *
      (self._pressure_calibration[2] * 32) / 8) +
      ((self._pressure_calibration[1] * var1) / 2))
    var1 = var1 / 262144
    var1 = ((32768 + var1) * self._pressure_calibration[0]) / 32768
    calc_pres = 1048576 - self._adc_pres
    calc_pres = (calc_pres - (var2 / 4096)) * 3125
    calc_pres = (calc_pres / var1) * 2
    var1 = (self._pressure_calibration[8] * (((calc_pres / 8) * (calc_pres /
8)) / 8192)) / 4096
    var2 = ((calc_pres / 4) * self._pressure_calibration[7]) / 8192
    var3 = (((calc_pres / 256) ** 3) * self._pressure_calibration[9]) /
131072
    calc_pres += ((var1 + var2 + var3 + (self._pressure_calibration[6] *
128)) / 16)
    return calc_pres/100
  @property
  def humidity(self):
    self._perform_reading()
    temp_scaled = ((self._t_fine * 5) + 128) / 256
    var1 = ((self._adc_hum - (self._humidity_calibration[0] * 16)) -
      ((temp_scaled * self._humidity_calibration[2]) / 200))
    var2 = (self._humidity_calibration[1] *
```

```
     (((temp_scaled * self._humidity_calibration[3]) / 100) +
      (((temp_scaled * ((temp_scaled * self._humidity_calibration[4]) / 100)) /
        64) / 100) + 16384)) / 1024
    var3 = var1 * var2
    var4 = self._humidity_calibration[5] * 128
    var4 = (var4 + ((temp_scaled * self._humidity_calibration[6]) / 100)) / 16
    var5 = ((var3 / 16384) * (var3 / 16384)) / 1024
    var6 = (var4 * var5) / 2
    calc_hum = (((var3 + var6) / 1024) * 1000) / 4096
    calc_hum /= 1000
    if calc_hum > 100:
      calc_hum = 100
    if calc_hum < 0:
      calc_hum = 0
    return calc_hum
  @property
  def altitude(self):
    pressure = self.pressure
    return 44330 * (1.0 - math.pow(pressure / self.sea_level_pressure, 0.1903))
  @property
```

```
  def gas(self):
    self._perform_reading()
    var1 = ((1340 + (5 * self._sw_err)) *
(_LOOKUP_TABLE_1[self._gas_range])) / 65536
    var2 = ((self._adc_gas * 32768) - 16777216) + var1
    var3 = (_LOOKUP_TABLE_2[self._gas_range] * var1) / 512
    calc_gas_res = (var3 + (var2 / 2)) / var2
    return int(calc_gas_res)
  def _perform_reading(self):
    if (time.ticks_diff(self._last_reading, time.ticks_ms()) *
time.ticks_diff(0, 1)
        < self._min_refresh_time):
      return
    self._write(_BME680_REG_CONFIG, [self._filter << 2])
    self._write(_BME680_REG_CTRL_MEAS,
     [(self._temp_oversample << 5)|(self._pressure_oversample << 2)])
    self._write(_BME680_REG_CTRL_HUM, [self._humidity_oversample])
    self._write(_BME680_REG_CTRL_GAS, [_BME680_RUNGAS])
    ctrl = self._read_byte(_BME680_REG_CTRL_MEAS)
    ctrl = (ctrl & 0xFC) | 0x01
    self._write(_BME680_REG_CTRL_MEAS, [ctrl])
    new_data = False
```

```
    while not new_data:
      data = self._read(_BME680_REG_MEAS_STATUS, 15)
      new_data = data[0] & 0x80 != 0
      time.sleep(0.005)
    self._last_reading = time.ticks_ms()
    self._adc_pres = _read24(data[2:5]) / 16
    self._adc_temp = _read24(data[5:8]) / 16
    self._adc_hum = struct.unpack('>H', bytes(data[8:10]))[0]
    self._adc_gas = int(struct.unpack('>H', bytes(data[13:15]))[0] / 64)
    self._gas_range = data[14] & 0x0F
    var1 = (self._adc_temp / 8) - (self._temp_calibration[0] * 2)
    var2 = (var1 * self._temp_calibration[1]) / 2048
    var3 = ((var1 / 2) * (var1 / 2)) / 4096
    var3 = (var3 * self._temp_calibration[2] * 16) / 16384
    self._t_fine = int(var2 + var3)
  def _read_calibration(self):
    coeff = self._read(_BME680_BME680_COEFF_ADDR1, 25)
    coeff += self._read(_BME680_BME680_COEFF_ADDR2, 16)
    coeff = list(struct.unpack('<hbBHhbBhhbbHhhBBBHbbbBbHhbb',
bytes(coeff[1:39])))
    coeff = [float(i) for i in coeff]
    self._temp_calibration = [coeff[x] for x in [23, 0, 1]]
```

```
    self._pressure_calibration = [coeff[x] for x in [3, 4, 5, 7, 8, 10, 9, 12,
13, 14]]
    self._humidity_calibration = [coeff[x] for x in [17, 16, 18, 19, 20, 21,
22]]
    self._gas_calibration = [coeff[x] for x in [25, 24, 26]]
    self._humidity_calibration[1] *= 16
    self._humidity_calibration[1] += self._humidity_calibration[0] % 16
    self._humidity_calibration[0] /= 16
    self._heat_range = (self._read_byte(0x02) & 0x30) / 16
    self._heat_val = self._read_byte(0x00)
    self._sw_err = (self._read_byte(0x04) & 0xF0) / 16
  def _read_byte(self, register):
    return self._read(register, 1)[0]
  def _read(self, register, length):
    raise NotImplementedError()
  def _write(self, register, values):
    raise NotImplementedError()
class BME680_I2C(Adafruit_BME680):
  def __init__(self, i2c, address=0x77, debug=False, *, refresh_rate=10):
    self._i2c = i2c
    self._address = address
    self._debug = debug
```

```
    super().__init__(refresh_rate=refresh_rate)
  def _read(self, register, length):
    result = bytearray(length)
    self._i2c.readfrom_mem_into(self._address, register & 0xff, result)
    if self._debug:
      print("\t${:x} read ".format(register), " ".join(["{:02x}".format(i) for i
in result]))
    return result
  def _write(self, register, values):
    if self._debug:
      print("\t${:x} write".format(register), " ".join(["{:02x}".format(i) for i
in values]))
    for value in values:
      self._i2c.writeto_mem(self._address, register, bytearray([value &
0xFF]))
      register += 1


try:
  import usocket as socket
except:
  import socket


from time import sleep
```

```
from machine import Pin, I2C
import network

import esp
esp.osdebug(None)

import gc
gc.collect()

from bme680 import *

# ESP32 - Pin assignment
i2c = I2C(scl=Pin(22), sda=Pin(21))
# ESP8266 - Pin assignment
#i2c = I2C(scl=Pin(5), sda=Pin(4))

ssid = 'PCET'
password = 'pcet@1973'

station = network.WLAN(network.STA_IF)
```

```
station.active(True)
station.connect(ssid, password)

while station.isconnected() == False:
  pass

print('Connection successful')
print(station.ifconfig())
def web_page():
  bme = BME680_I2C(i2c=i2c)
  html = """<html><head><title>ESP with BME680</title>
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <link rel="icon" href="data:,"><style>body { text-align: center; font-family: "Trebuchet MS", Arial;}
  table { border-collapse: collapse; margin-left:auto; margin-right:auto; }
  th { padding: 12px; background-color: #0043af; color: white; }
  tr { border: 1px solid #ddd; padding: 12px; }
  tr:hover { background-color: #bcbcbc; }
  td { border: none; padding: 12px; }
  .sensor { color:white; font-weight: bold; background-color: #bcbcbc; padding: 1px;
  </style></head><body><h1>ESP with BME680</h1>
```

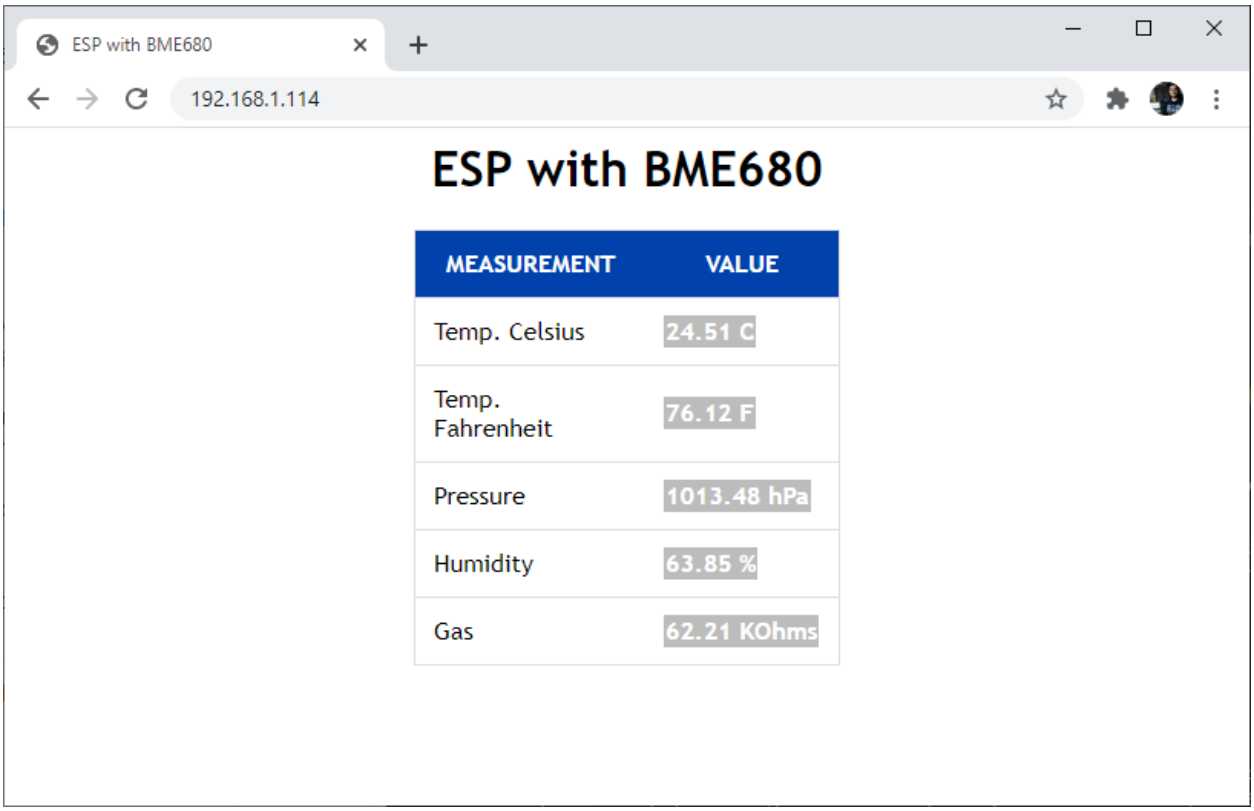
```
  <table><tr><th>MEASUREMENT</th><th>VALUE</th></tr>
  <tr><td>Temp. Celsius</td><td><span class="sensor">""" +
str(round(bme.temperature, 2)) + """ C</span></td></tr>
  <tr><td>Temp. Fahrenheit</td><td><span class="sensor">""" +
str(round((bme.temperature) * (9/5) + 32, 2))  + """ F</span></td></tr>
  <tr><td>Pressure</td><td><span class="sensor">""" +
str(round(bme.pressure, 2)) + """ hPa</span></td></tr>
  <tr><td>Humidity</td><td><span class="sensor">""" +
str(round(bme.humidity, 2)) + """ %</span></td></tr>
  <tr><td>Gas</td><td><span class="sensor">""" +
str(round(bme.gas/1000, 2)) + """
KOhms</span></td></tr></body></html>"""
  return html

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind(('', 80))
s.listen(5)

while True:
  try:
    if gc.mem_free() < 102000:
      gc.collect()
    conn, addr = s.accept()
    conn.settimeout(3.0)
```

```
  print('Got a connection from %s' % str(addr))
  request = conn.recv(1024)
  conn.settimeout(None)
  request = str(request)
  print('Content = %s' % request)
  response = web_page()
  conn.send('HTTP/1.1 200 OK\n')
  conn.send('Content-Type: text/html\n')
  conn.send('Connection: close\n\n')
  conn.sendall(response)
  conn.close()
except OSError as e:
  conn.close()
  print('Connection closed')
```

Output:


ESP with BME680
192.168.1.114
ESP with BME680
MEASUREMENT
VALUE
Temp. Celsius
24.51 C
Temp. Fahrenheit
76.12 F
Pressure
1013.48 hPa
Humidity
63.85 %
Gas
62.21 KOhms