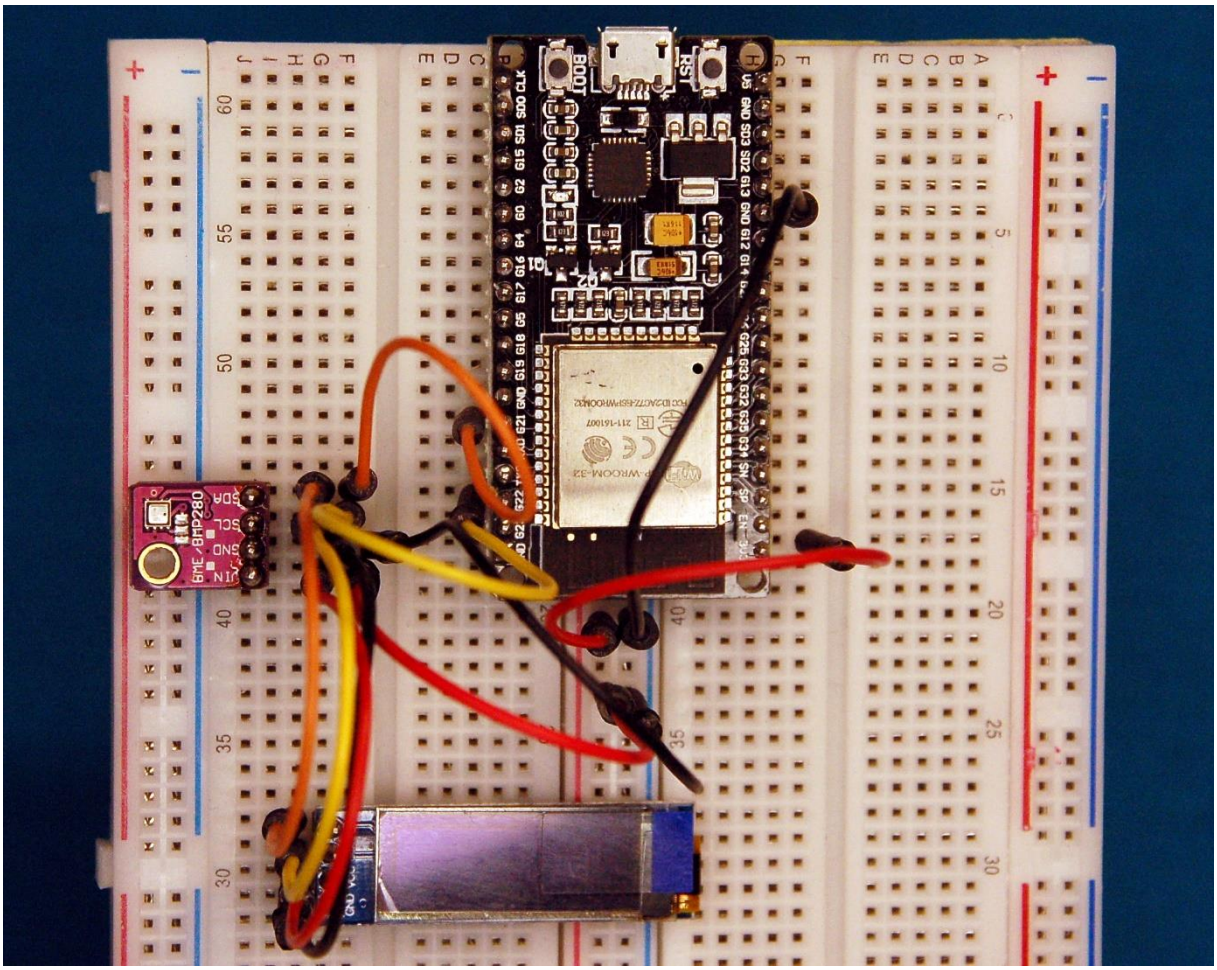

e-Mails vom ESP32

Diesen Beitrag gibt es auch als PDF-Dokument.

Mit der Fähigkeit e-Mails versenden zu können, haben wir aus dem ESP32 einen Packesel gemacht, der weltweit Post zustellen kann. In diesem Beitrag werden wir für die entsprechende Payload sorgen. Dazu schauen wir uns einen BME280 näher an, um dann ein Programm zu entwickeln, das uns die Daten vom Sensor via e-Mail zustellt. Willkommen bei einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Teil 2 - e-Mail-Nachrichten von ESP32 und BME280

Dass der ESP8266 für diesen Job nicht zu gebrauchen ist, liegt an dem knapp bemessenen Speicher. Unter MicroPython kann nur 1MB angesprochen werden und davon belegt bereits der Kernel einen Großteil. So kommt es, dass schon beim Importieren des Moduls BME280 ein Speicherproblem gemeldet wird. Das bedeutet, dass für diesen Beitrag ein ESP32 zwingend erforderlich ist.

Hardware

Um den Zustand der Schaltung jederzeit auch direkt vor Ort einsehen zu können, habe ich dem ESP32 ein kleines Display spendiert, das über den I2C-Bus angesteuert wird. Es ist sogar grafikfähig und könnte daher auch zeitliche Änderungen des Messsignals als Kurve darstellen. Über die Flash-Taste ist ein geordneter Abbruch des Programms möglich. Das ist nützlich, falls zum Beispiel Aktoren sicher ausgeschaltet werden müssen oder ein Abbruch über Strg + C erfolglos ist.

Als Messanwendung habe ich mich für einen Klimamonitor mit dem BME280 entschieden. Der Bosch-Sensor kann Luftdruck, relative Luftfeuchte und Temperatur erfassen. Mit diesen Daten werden wir den Luftdruck auf Meeresspiegelhöhe (NHN Normalhöhennull) und den Taupunkt berechnen.

1	ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	GY-BME280 Barometrischer Sensor für Temperatur, Luftfeuchtigkeit und Luftdruck
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard
optional	Logic Analyzer

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
bme280.py API für den Bosch-Sensor
bme280-test.py Demo- und Testprogramm für den BME280
umail.py Micro-Mail-Modul
e-mail.py Demoprogramm für den e-Mailversand

bme280-monitor.py Demo-Messprogramm

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Signale auf dem I2C-Bus

Immer wenn es Probleme bei der Datenübertragung gibt, setze ich gerne das DSO (Digitales Speicher Oszilloskop) ein, oder ein um Welten billigeres, kleines Tool, einen Logic-Analyzer (LA) mit 8 Kanälen. Das Ding wird an den USB-Bus angeschlossen und zeigt mittels einer kostenlosen Software, was auf den Busleitungen los ist. Dort, wo es nicht auf die Form von Impulsen ankommt, sondern lediglich auf deren zeitliche Abfolge ist ein LA Gold wert. Und, während das DSO nur Momentaufnahmen des Kurvenverlaufs liefert, kann man mit dem LA über längere Zeit abtasten und sich dann in die interessanten Stellen hineinzoomen. Eine Beschreibung zu dem Gerät finden Sie übrigens in dem Blogpost "Logic Analyzer - Teil 1: I2C-Signale sichtbar machen" von Bernd Albrecht. Dort ist auch beschrieben, wie man den I2C-Bus abtastet.

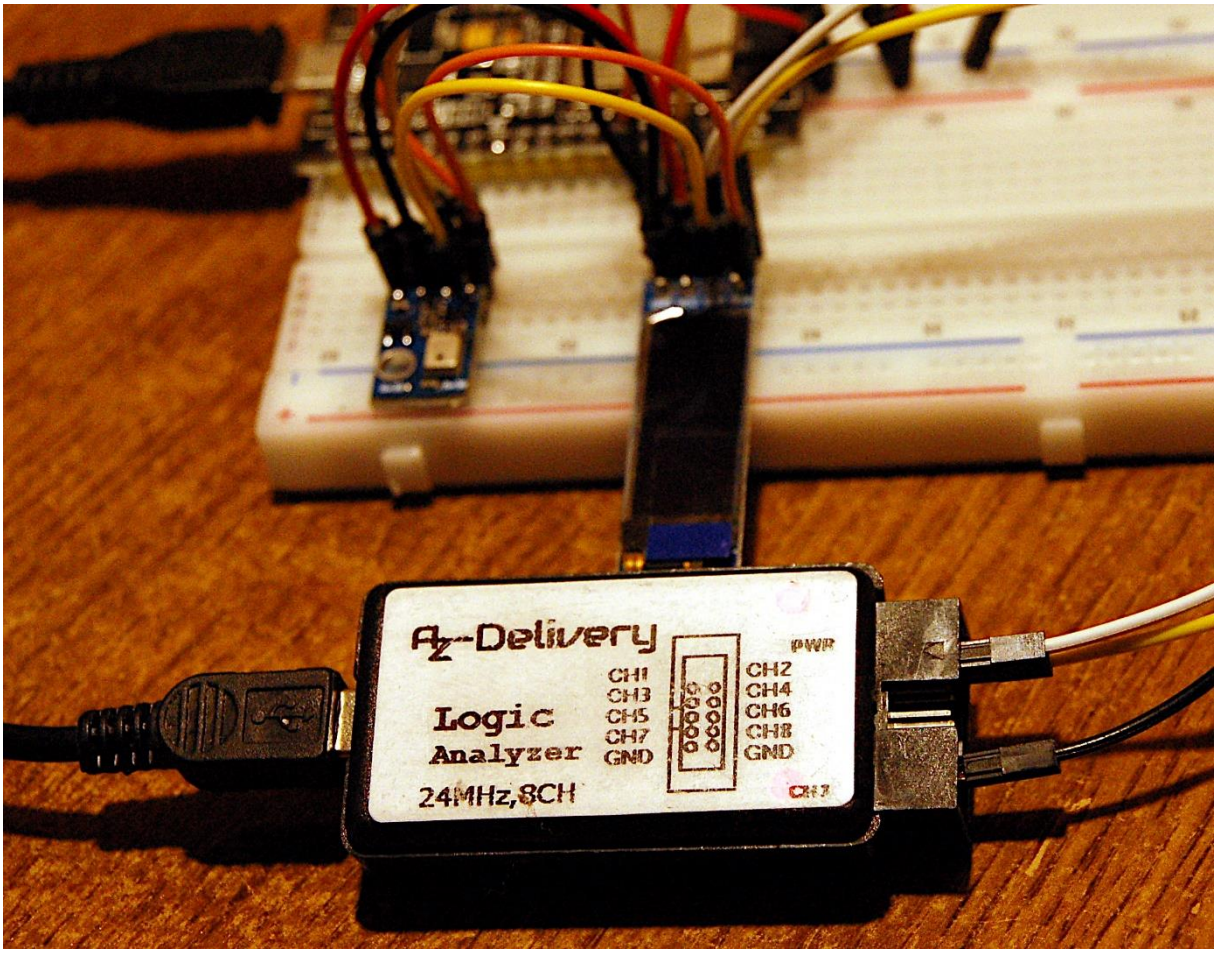

Abbildung 1: Logic Analyzer am I2C-Bus

Die Schaltung

Die drei Teile für die Schaltung sind schnell zusammengesteckt.

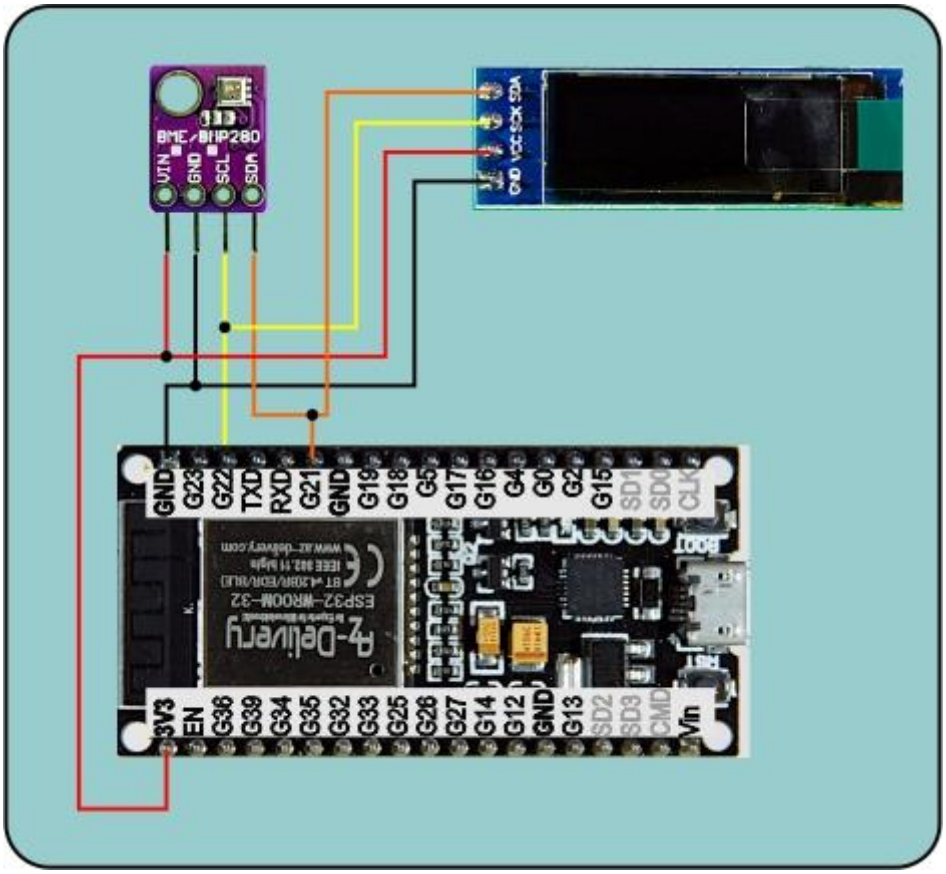

Abbildung 1: e-Mail - Schaltung

Auf dem BME280-Board befinden sich neben dem Sensor selbst noch ein Spannungswandler und ein Pegelwandler für die SCL und SDA-Leitung. Somit müssen keine externen Pullup-Widerstände angebracht werden, denn die sind Teil des Pegelwandlers. Die Spannungsversorgung erfolgt über die USB-Buchse aus dem PC oder durch ein Steckernetzteil.

Der BME280

Während der BMP280 nur Temperatur und Luftdruck erfassen kann, kann sein großer Bruder auch die relative Luftfeuchte messen. Die Nummern und Bedeutung der Register des BMP280 sind beim BME280 gleich. Bei letzterem kommt eben nur der Bereich Feuchte dazu. Das ist praktisch, denn dadurch lässt sich das Modul BME280 auch für den BMP280 verwenden.

Beide Sensoren können über den I2C- oder SPI-Bus als Slave angesteuert werden. Allerdings ist beim vorliegenden Modul der I2C-Bus fest eingestellt, was mich nicht stört, weil auch das Display denselben Bus benutzt. Das I2C-Bus-Objekt wird daher auch im Hauptprogramm erzeugt und an die Konstruktoren **OLED**() und **BME280**() übergeben.

Daten können zum BME280 im Single-Byte-Mode oder im Multi-Byte-Mode, dann als Adress-Wert-Paare, gesendet werden. Beim Auslesen der Messwerte ist es praktisch, dass nur die Adresse des ersten Registers einer ganzen Folge angegeben werden muss und der BME280 die Adresse für weitere Lesezugriffe selbst erhöht (Autoinkrement). Im MicroPython-Modul **bme280**.py gibt es für den Datentransfer entsprechend zwei Methoden.

```
    def writeByteToReg(self,reg,data):
        buf=bytearray(2)
        buf[0]=reg
        buf[1]=data & 0xFF
        self.i2c.writeto(self.hwadr,buf)

    def readBytesFromReg(self,reg,num):
        buf=bytearray(num)
        buf[0]=reg
        self.i2c.writeto(self.hwadr,buf[:1])
        self.i2c.readfrom_into(self.hwadr,buf)
        return buf
```

Beim Lesen gebe ich also nur das Startregister an und die Anzahl zu lesender Bytes. Damit wird ein Bytearray der gewünschten Länge erzeugt. Zum Senden der Adresse verwende ich nur das erste Element, in welches ich die Adresse schreibe. Dann werden so viele Bytes abgeholt wie in das Array passen.

Das Datenblatt des BME280 gibt Aufschluss über die Registerlandschaft. In drei Registern werden die Konfigurationsdaten gehalten. Die Routinen zum Schreiben und Auslesen der Register greifen auf die Konfigurationsattribute des BME280-Objekts und auf die Routinen für den I2C-Transfer zurück.

Ein Statusregister informiert über den Systemzustand des Sensors. Nach jeder Messung wird der Satz an Rohdaten in Schattenregister geschrieben. Das Bit **status.measuring** ist 0, wenn die Daten zum Auslesen bereitstehen.

Im Read-Only-Register **id** = 0xD0 steht ein Indentifikationsbyte, das den Typ des Sensors verrät. Die Routine **readIDReg**() liefert die Klartextbezeichnung zurück.

0x55: "BMP180",
0x58: "BMP280",
0x60: "BME280"

Werksseitig ist jeder Sensor mit einem Satz an Kalibrierdaten versehen worden. Damit der BME280 korrekte Werte liefert, müssen diese aus den Registern 0xE1 bis 0xF0 ausgelesen werden. Das macht die Methode **getCalibrationData**() automatisch beim Instanziieren eines BME280-Objekts. Zusammen mit den Rohdaten für Temperatur, relative Feuchte und Luftdruck, die mit der Methode **readDataRaw**() ausgelesen werden, berechnet man dann nach den Formeln im Datenblatt die Endwerte. Das erledigen die Methoden **calcTemperature**(), **calcPressureH**() und **calcHumidity**(). Die Methode **calcPressureNN**() rechnet auf den Luftdruck auf Meeresniveau zurück. **calcDewPoint**() berechnet den Taupunkt, das ist die

Temperatur ab der, der in der Luft befindliche unsichtbare Wasserdampf beginnt zu kondensieren und Nebeltröpfchen zu bilden.

Die Genauigkeit der Messungen lässt sich durch die Oversampling-Werte über die Control-Register in jeweils 5 Stufen einstellen, x1, x2, x4, x8 und x16. Dazu muss zuerst die Konfiguration gesetzt werden, danach werden die Attribute in die Control-Register geschrieben. Das Oversampling der Feuchtemessung besitzt ein eigenes Register.

>>> b.setControl(OST=5)
>>> b.writeContrlReg()
>>> b.setControlH(OSH=3)
>>> b.writeControlHReg()

Die Startwerte, die der Konstruktor einstellt, sind über die Methoden **showControl**() und **showConfig**() abrufbar.

>>> b.showConfig()
Standby= 125 Filter= 16
(2, 16)
>>> b.showControl()
OST= 1 OSP= 3 Mode= 3 OSH= 2
(1, 3, 3, 2)

Dann wird es Zeit für ein erstes Testprogramm.

```
# bme280-test.py
from machine import Pin, SoftI2C
from bme280 import BME280
from time import sleep
import os,sys

i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=100000)

try:
    b=BME280(i2c)
except OSError as e:
    print("Kein BME280 ansprechbar",e)

# b.getCalibrationData()
# b.printCalibrationData()
print("")

b.readDataRaw()
sleep(0.5)
print(b.calcTemperature(),"°C")
print(b.calcPressureH(),"hPa @ Standort")
print(b.calcPressureNN(450),"hPa @ Sea level")
print(b.calcHumidity(),"%RH")
print(b.calcDewPoint(),"°C")
```

Weil bei einem ESP8266 gleich beim Start dieses Magerprogramms eine Fehlermeldung aufploppt, muss die I2C-Schnittstelle nicht durch das Programm ermittelt werden, sondern ist direkt auf einen ESP32 eingestellt. Der ESP8266 ist speichermäßig einfach zu schmalbrüstig.

>>> %Run -c $EDITOR_CONTENT
Traceback (most recent call last):
* File "<stdin>", line 3, in <module>*
MemoryError: memory allocation failed, allocating 360 bytes

Während der Entwicklung ist es nützlich, die Calibrierdaten aus dem NVM (Non Volatile Memory = nicht flüchtiger Speicher) des BME280 zu kennen, um die berechneten Endwerte mit Hilfe der im Datenblatt angegebenen Formeln händisch zu überprüfen. **b.getCalibrationData**() und **b.printCalibrationData**() rufen die Werte ab und zeigen sie an.

Ich hatte oben schon erwähnt, dass die Messwerte in einen Schattenspeicher übertragen werden, sobald die Messung beendet ist. Um nach dem Start zuverlässige Werte zu erhalten, wird die erste Gruppe eingelesen und verworfen. Die nachfolgenden **calcXY**()-Aufrufe fordern dann jedes Mal einen neuen Satz an Rohdaten an. Damit ist sichergestellt, dass für die Druck- und Feuchteberechnung auch der korrekte Temperaturwert zur Verfügung steht. Ein Programmlauf liefert nun eine Ausgabe im Terminal, die ähnlich wie die folgende aussehen sollte.

17.78 °C
983.2547 hPa @ Standort
1037.413 hPa @ Sea level
42.30078 %RH
4.789577 °C

Es ist nicht schwierig, diese Daten per e-Mail zu versenden, wir müssen nur das Programm **e-mail.py** aus der letzten Folge mit **bme280-test.py** kombinieren. Das Endprodukt habe ich **bme280-monitor.py** genannt. Schauen wir uns an, wie es arbeitet.

```
import umail
import network
import sys
from time import sleep, ticks_ms
from machine import SoftI2C,Pin
from bme280 import BME280
from oled import OLED
```

Die beiden projektspezifischen Importe sind **umail** und **bme280**. Die entsprechenden Dateien müssen zum ESP32 hochgeladen werden, weil sie nicht Teil des MicroPython-Kernels sind, wie die anderen Beigaben.

Für den Zugriff auf das WLAN müssen Sie ihre eigenen Credentials eintragen.

```
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySSID = 'EMPIRE_OF_ANTS'
myPass = 'nightingale'
```

Außerdem benötigen Sie Daten für den G-Mailzugang inklusive App-Passwort. Wie Sie zu beiden kommen, ist in der vorangehenden Folge erklärt. Vergessen Sie auch nicht bei **recipient_email** Ihre Mailadresse einzutragen.

```
# e-Mail-Daten
sender_email = 'ernohub@gmail.com'
sender_name = 'ESP32' #sender name
sender_app_password = 'xxxxxxxxxxxxxxx'
recipient_email ='meine@mail.org'
```

Für das OLED-Display und den BME280 brauchen wir den I2C-Bus. Beide Bausteine unterstützen Geschwindigkeiten bis 400000kHz. Den ersten Datensatz vom BME280 verwerfen wir, das heißt wir tun nix damit.

```
i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=400000)

d=OLED(i2c,heightw=32)
d.writeAt("BME280-MAILER",0,0)

bme=BME280(i2c)
bme.readDataRaw()
```

Die Flash-Taste ist die Taste für den geordneten Ausstieg aus der Hauptschleife.

```
taste=Pin(0,Pin.IN,Pin.PULL_UP)

intervall=3600 # Sendeintervall in Sekunden
```

Im Zwei-Stundenabstand wird nach einer Datenerfassung eine Mail versandt. Den Zeitraum können Sie natürlich Ihren Bedürfnissen entsprechend anpassen.

```
temp,location,seaLevel,relHum,taupunkt=0,0,0,0,0
```

Die Variablen für die Messwerte initialisieren wir mit 0. Die Methode dafür ist etwas ungewöhnlich. Wieso geht das? Wir nutzen dabei die Methode des Packens und Entpackens von Tupeln. Der MicroPython-Interpreter macht aus den fünf Nullen und dem Zuweisungsoperator "=" erst einmal ein Tupel. Diesen Vorgang nennt man Packen.

>>> x=0,0,0,0,0
>>> x
(0, 0, 0, 0, 0)

Der Inhalt des Tupels wird aber sofort wieder auf die fünf Variablen verteilt, also entpackt.

>>> a,b,c,d,e=(0, 0, 0, 0, 0)
>>> a; b; c; d; e
0
0
0
0
0

>>> a,b,c,d,e = 0,0,0,0,0

Packen und entpacken geschieht also in einer Zeile.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN"
    }
```

Das Dictionary **connectStatus** hilft bei der Übersetzung der Status-Codes von der WLAN-Schnittstelle in Klartext.

Das Ergebnis der Abfrage der MAC-Adresse durch die Funktion **nic.config('mac')** ist als bytes-Objekt recht kryptisch.

>>> nic = network.WLAN(network.STA_IF)
>>> nic.active(True)
>>> nic.config('mac')
b'\xf0\x08\xd1\xd2\x1e\x94'

Die Funktion **hexMac**() macht daraus Klartext, den Sie im Router eintragen müssen, damit der MAC-Filter dem ESP32 Einlass gewährt.

>>> hexMac(b'\xf0\x08\xd1\xd2\x1e\x94')
'f0-8-d1-d2-1e-94'

```
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode und
  bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]  # String ab 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen
      macString +="-"                 # bis auf letztes Byte
  return macString
```

Die Closure **TimeOut**() erzeugt beim Aufruf einen Software-Timer. Die zurückgegebene Funktion **compare**() weisen wir später dem Bezeichner **senden** zu. Dieser Name ist ein Alias für die Funktion **compare**(). Rufen wir **senden**() auf, dann erhalten wir als Ergebnis **True** oder **False**. Das sagt uns, ob der Timer schon abgelaufen ist oder nicht.

```
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_diff(ticks_ms(),start)) >= t
    return compare
```

Die Funktion **getWerte**() liest die Werte vom BME280 ein und baut daraus Strings, die an die Variablen **temp**, **location**, **sealevel**, **relHum** und **taupunkt** übergeben werden. Weil das aus einer Funktion heraus erfolgt, müssen die Variablen als **global** deklariert sein. Ohne das Schlüsselwort global wären diese Variablen für die Funktion lokal, die zugewiesenen Inhalte könnten außerhalb der Funktion nicht abgerufen werden.

Durch die Formatangabe {:0.2f} werden die Fließkommawerte, die von den calcXY()-Funktionen zurückkommen, auf zwei Stellen nach dem Komma ausgegeben. Diese Art der Formatierung ist die einfachste Möglichkeit, Strings und numerische Werte zu mischen.

```
def getWerte():
    global temp,location,seaLevel,relHum,taupunkt
    bme.readDataRaw()
    sleep(0.5)
    bme.readDataRaw()
    temp="{:0.2f} *C".format(bme.calcTemperature())
    location="{:0.2f} hPa @ Standort".\
             format(bme.calcPressureH())
    seaLevel="{:0.2f} hPa @ Sea level".\
             format(bme.calcPressureNN(450))
    relHum="{:0.2f} %RH".format(bme.calcHumidity())
    taupunkt="{:0.2f} *C".format(bme.calcDewPoint())
```

Es folgt die Verbindungsaufnahme zum WLAN-Router. Damit das Accesspoint-Interface des ESP32 uns nicht in die Suppe spukt, wird es deaktiviert. Das ist vor allem beim ESP8266 wichtig, schadet aber auch beim ESP32 nicht.

```
# ************** Zum Router verbinden ******************
#
nic=network.WLAN(network.AP_IF)
nic.active(False)

nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt
nic.active(True)                    # nic einschalten
MAC = nic.config('mac')   # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)         # in Hexziffernfolge umwandeln
print("STATION MAC: \t"+myMac+"\n") # ausgeben
```

Dann erzeugen wir ein Station-Interface-Objekt und aktivieren es. Die ausgelesene MAC-Adresse übergeben wir zum Übersetzen an **hexMac**().

Nach einer kurzen Verschnaufpause bauen wir die Verbindung auf. SSID und Passwort werden übergeben und der Status wird abgefragt. Solange wie der ESP32 noch keine IP-Adresse vom DHCP-Server erhalten hat, werden im Display und im Terminalbereich von Thonny im Sekundenraster Punkte ausgegeben. Im Display geschieht das durch Slicing des Strings **points**. Um im Terminal auf einer Zeile zu bleiben, teilen wir der print-Anweisung mit, dass das Zeilenende-Zeichen "\n" durch nichts ' ' ersetzt werden soll.

Dann fragen wir den Status erneut ab und lassen uns die Verbindungsdaten anzeigen.

```
print("\nStatus: ",connectStatus[nic.status()])
d.clearAll()
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
      STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
print()
d.writeAt(STAconf[0],0,0)
d.writeAt(STAconf[1],0,1)
d.writeAt(STAconf[2],0,2)
sleep(3)
```

Vor dem Eintritt in die Hauptschleife, stellen wir den Timer auf die Zeit in **intervall**. Weil der Timer in Millisekunden tickt, wird der Wert mit 1000 multipliziert. Damit die erste Mail sofort verschickt wird, setzen wir **jetzt** auf **True**. Diese Variable hat aber noch eine zweite Bedeutung. Tritt nämlich ein Ereignis ein, das eine sofortige Versendung der Mail erforderlich macht, dann kann der auslösende Vorgang **jetzt** auf **True** setzen und so den Mailversand auch außerhalb der festen Zeitspanne veranlassen.

```
senden=TimeOut(intervall*1000)
jetzt=True
kontrolle=TimeOut(1800000)
oldPres= oldPres=float(seaLevel.split(" ")[0])
```

Dann holen wir die aktuellen Werte. Wir merken uns schon mal den Luftdruck in **oldPres**. Und stellen einen weiteren Timer auf eine halbe Stunde. In der Hauptschleife prüfen wir, ob der Timer **kontrolle**() schon abgelaufen ist. In diesem Fall merken wir uns den aktuellen Luftdruckwert und stellen den Timer neu.

```
while 1:
    if kontrolle():
        oldPres=float(seaLevel.split(" ")[0])
        kontrolle=TimeOut(1800000)
```

Die neuen Werte werden eingelesen und die Änderung des Luftdrucks überprüft. Fällt der Luftdruck in der Kontrollzeitspanne um mehr als zwei hPa, dann könnte ein Gewitter im Anzug sein. und eine Wetterwarnung wird vorbereitet, der Betreff wird geändert und **jetzt** wird **True**. Für den Vergleich der Zahlenwerte müssen diese aus dem String extrahiert werden. Dazu lasse ich den String an den Leerstellen " " aufteilen. Aus der erhaltenen Liste nehme ich das erste Element und wandle es in eine Fließkommazahl um.

>>> seaLevel
'1037.62 hPa @ Sea level'
>>> seaLevel.split(" ")
['1037.62', 'hPa', '@', 'Sea', 'level']
>>> seaLevel.split(" ")[0]
'1037.62'
>>> float(seaLevel.split(" ")[0])
1037.62

```
    getWerte()
    if oldPres - float(seaLevel.split(" ")[0]) > 2:
        email_subject ='Unwetter im Anzug'
        jetzt=True

    d.clearAll(False)
    d.writeAt(temp,0,0,False)
    d.writeAt(seaLevel,0,1,False)
    d.writeAt(relHum,0,2)
```

Die Ausgabe der Werte im Terminal und im Display ist nicht spektakulär. **False** in den **writeAt**-Befehlen zum Display verhindert das flackern der Anzeige. Es bewirkt, dass die Änderungen zuerst nur im Hintergrund im Puffer erfolgen. Erst mit dem letzten **writeAt**-Befehl wird der Pufferinhalt zum OLED-Display gesendet.

Eine e-Mail wird versandt, falls der Intervall-Timer abgelaufen ist, oder wenn **jetzt** den Wert **True** hat.

```
    if senden() or jetzt:
```

```
        # ************  Eine Mail versenden  ***************
        #
        print(temp)
        print(location)
        print(seaLevel)
        print(relHum)
        smtp = umail.SMTP('smtp.gmail.com', 465,
                          ssl=True, debug=True)
        smtp.login(sender_email, sender_app_password)
        smtp.to(recipient_email)
        smtp.write("From:" + sender_name + "<"+ \
                   sender_email+">\n")
        smtp.write("Subject:" + email_subject + "\n")
        smtp.write("Klimawerte vom ESP32\n")
        smtp.write(temp+"\n")
        smtp.write(location+"\n")
        smtp.write(seaLevel+"\n")
        smtp.write(relHum+"\n")
        smtp.write(taupunkt+"\n")
        smtp.send()
        smtp.quit()
        if email_subject == 'Unwetter im Anzug'
            senden=TimeOut(intervall*1000)
        jetzt=False
        email_subject ='Wettermeldung'
```

Wir bauen eine Verbindung zum Provider auf und senden Username und App-Passwort, danach die Empfängeradresse, den Absender und den Betreff. Nach der Übertragung der Messdaten veranlassen wir die Versendung an den Empfänger, also uns selbst und beenden die Verbindung.

Es folgen Aufräumarbeiten. Der Intervall-Timer wird neu gestellt, wenn er abgelaufen war, und der Alarmgeber wird rückgesetzt sowie der Betreff auf Normalbetrieb gebracht.

Die Abfrage des Zustands der Flash-Taste schließt das Programm ab.

```
    if taste.value() == 0:
        sys.exit()
```

Die Versendung einer e-Mail "on request", hier die Gewitterwarnung, kann natürlich auch durch verschiedene andere Ereignisse ausgelöst werden. Alles, was sich durch einen Sensor erfassen lässt, kann einen Mailversand triggern. Die Hauptschleife läuft im Dauerlauf und kann jederzeit weitere Sensoren abfragen. Personen, die sich in einem Raum bewegen, der Wasserstand im Regenfass, Sturm, Licht an oder aus sind nur ein paar Möglichkeiten. Lassen Sie Ihrer Phantasie freien Lauf. Das komplette Programm liegt natürlich zum Download bereit.

Viel Vergnügen beim Basteln und Programmieren!