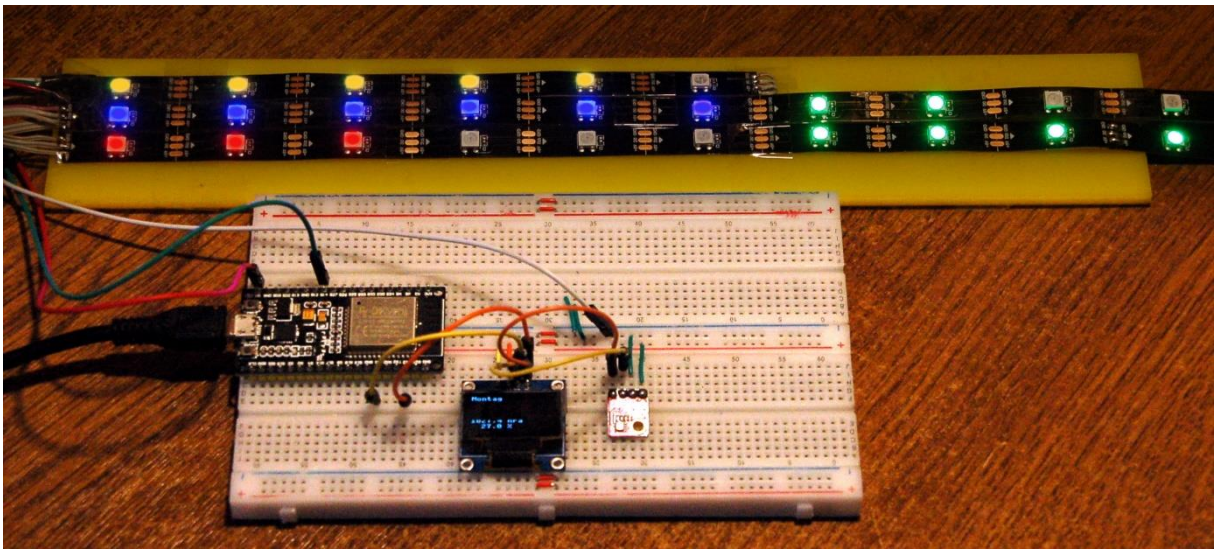

LED-Streifen-Uhr_Ausbaustufe 2

Diesen Beitrag gibt es auch als PDF-Dokument.

Im vorangegangenen Beitrag haben wir die Funktion des NeoPixel-Streifens genauer untersucht und uns Gedanken darüber gemacht, wie damit die Codierung von Uhrzeiten umgesetzt werden könnte. Wir haben uns auch mit den unterschiedlichen Formaten befasst, in denen die Funktionen time.localtime() und RTC.datetime() Datum und Uhrzeit darstellen. Beide Module verwalten die Zeit im Hintergrund in Form eines Zählers, der die Sekunden seit dem Beginn der Epoche erfasst. Dieser Zählerstand wird von den beiden Funktionen in eine lesbare Form gebracht, nur eben in unterschiedlicher Weise.

Wer aber sagt unserem Controller beim Start eigentlich, wieviel Uhr es geschlagen hat? Und wie kann sich ein ESP32 im Lauf der Zeit vergewissern, dass seine Zählung auch der gesetzlichen Zeit entspricht? Das werden wir klären und zwar in der aktuellen Folge aus der Reihe

MicroPython auf dem ESP32 und Raspberry Pi Pico

heute

Die Abakus-Uhr bekommt Unterstützung

In diesem Beitrag wird außerdem noch ein OLED-Display mit 128x64 Pixel und ein BME280 zur Erfassung von Umgebungsdaten wie Temperatur, Luftdruck und relative Luftfeuchtigkeit eingebaut. Dazu verwenden wir folgende Teile.

Hardware

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
1	Oder Raspberry Pi Pico W RP2040 Mikrocontroller-Board
1	WS2812B 30 LEDs/m 30 Pixels/m LED Strip RGB adressierbarer LED Streifen mit 5050 SMD LEDs IP20 Schwarz nicht Wasserdicht
1	Optional für Raspberry Pi Pico W Taste zum Beispiel KY-004 Taster Modul
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
Optional	Logic Analyzer
Optional	Charger Doctor

Für die Ausbaustufe 1 hätte auch ein ESP8266 ausgereicht. Weil sich in dieser neuen Folge ein BME280 dazugesellt, ist der Kleine aber leider zu schwach auf der Speicherbrust, weshalb hier ein ESP32 eingesetzt wird. Wahlweise kann auch ein Raspberry Pi Pico W die Steuerung übernehmen. Der ist, wie der ESP32 ja auch WLAN-fähig, was wir in Stufe 2 für eine Verbindung zu einem NTP-Server im Internet brauchen.

Mit einem Logic Analyzer lassen sich die Signale auf dem NeoPixel-Bus oder dem I2C-Bus sehr schön sichtbar machen. Das ist besonders hilfreich, wenn Probleme auftreten oder wenn man das Verhalten von NeoPixel-Arrays und den Transfer auf dem I2C-Bus studieren möchte.

Der Charger Doctor ermöglicht die Überwachung/Messung von Spannung und Stromstärke am USB-Bus und verrät uns vor allem den Strombedarf der Schaltung.

So sieht der Aufbau von Stufe 2 aus.

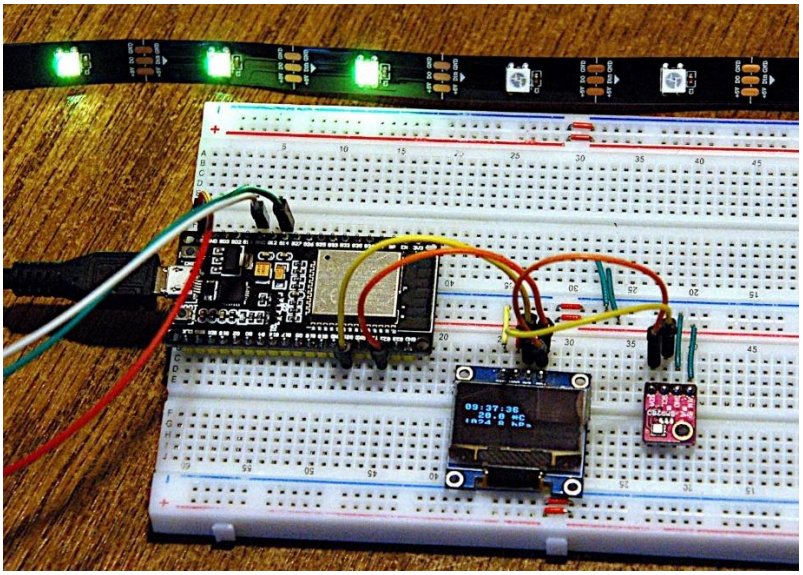

Abbildung 1: LED-Streifen-Uhr mit BME280 und OLED

Und hier sind die Schaltbilder für die beiden Controller.

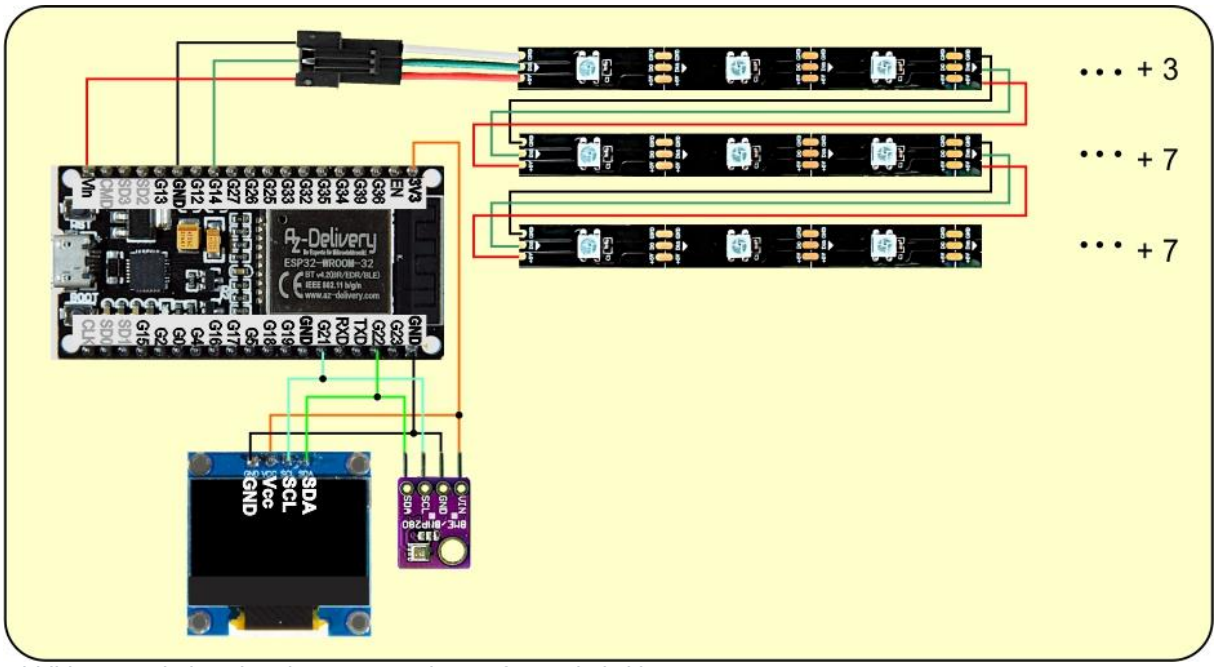

Abbildung 2: Abakusuhr mit ESP32 Ausbaustufe 2 Schaltskizze

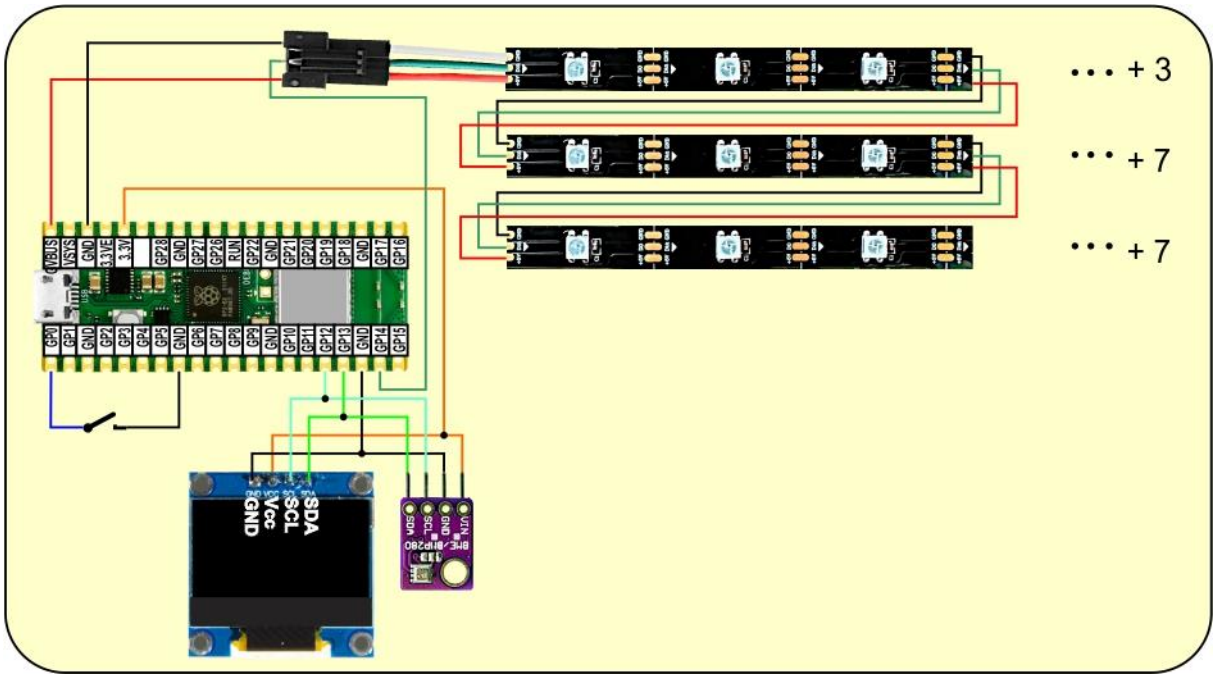

Abbildung 3: Abakusuhr mit Raspberry Py Pico W Ausbaustufe 2 Schaltskizze

Die Steuerleitung für die Neopixels ist in allen Fällen GPIO14, sodass diesbezüglich am Programm, das wir in Kürze besprechen werden, keine Änderungen nötig sind. Berücksichtigen müssen wir allerdings den I2C-Anschluss. Die Leitungen SCL und SDA können beim ESP32 beliebig an GPIOs zugewiesen werden. Beim Raspberry Pi Pico dagegen stehen immer nur bestimmte GPIO-Paare zur Verfügung. Das Problem ist aber eher marginal. Wir schaffen das!

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Zum Darstellen von Bussignalen

SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für den ESP32:

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

Verwendete Firmware für den Raspberry Pi Pico (W):

RPI_PICO_W-20240602-v1.23.0.uf2

Die MicroPython-Programme zum Projekt:

uhr.py Betriebsproramm für die Uhr
uhr2.py Betriebsproramm für die Uhr
ntp_test.py Testverbindung zu einem NTP-Server
rtc_time.py Modul zur Umsetzung der Zeitstrukturen
bme280.py Hardwaretreiber-Modul für den BME280
ssd1306.py Hardwaretreiber-Modul für das OLED 128x64
oled.py API zur bequemen Text-Ansteuerung der Anzeigemoduls

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 25.01.2024) auf den ESP-Chip gebrannt wird. Wie Sie den Raspberry Pi Pico einsatzbereit kriegen, finden Sie hier.

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 **main.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.

Experimente zum Zeit-Dschungel

Wir werden es heute mit drei Systemen zur Zeitverwaltung zu tun bekommen, die lokale Systemzeit (localtime), RTC (Real Time Clock = Echtzeituhr) und NTP (Network Time Protokoll). Die ersten beiden haben wir im letzten Beitrag schon kurz beleuchtet. Dabei haben wir festgestellt, dass die Strukturen, wie Datum und Zeit angezeigt werden, zwar ähnlich aber nicht kongruent sind. Andererseits greifen beide Module **time.localtime**() und **RTC.datetime**() auf dieselbe Quelle der Zeitführung zu. Das ist die Zählung der Sekunden seit Beginn der Epoche oder Ära.

Beim Start des ESP32 am USB-Bus des PCs bekommt der Controller automatisch die Sekundenzahl mitgeteilt und stellt danach seine Uhr. Deshalb zeigt das Programm **uhr.py**, das wir für den LED-Streifen geschrieben haben, auch gleich zu Beginn die korrekte Zeit. Dafür sorgt letztlich unsere Entwicklungsumgebung Thonny. Wenn wir nun die Schaltung nicht mit dem PC verbinden, sondern autonom laufen lassen, was passiert dann?

Probieren wir es aus, Jugend forscht! Damit der Controller autonom starten kann, müssen wir das Programm **uhr.py** in den Editor laden und mit „Speichern unter“ mit dem Namen **main.py** in den Flash des ESP32 hochladen. Ist das erledigt, dann verbinden wir den Aufbau mit einer 5V-Spannungsquelle.

ESP-Pin 5V an Batterie Plus
ESP GND Pin an Batterie Minus

So würden wir das auch machen, wenn wir die Schaltung später alleine betreiben würden. Nur stellen wir jetzt fest, dass die angezeigte Zeit nicht mit der aktuellen Uhrzeit übereinstimmt. Nein, die Zählung beginnt bei 00:00:00! Selbst wenn wir jetzt den ESP32 wieder mit dem PC anstatt mit der Batterie verbinden, beginnt die Zählung erneut bei 00:00:00. Weil das Programm durch die Firmware des ESP32 automatisch gestartet wird, verspürt dieser keine Veranlassung, mit dem PC in einen kuschligen Plausch einzutreten.

Sobald wir das Programm mit der Flash-Taste unterbrechen (diese Möglichkeit haben wir uns ja sicherheitshalber eingebaut ;-)), und erneut das Programm **uhr.py** in Thonny starten, funktioniert auch die Zeitsynchronisation wieder.

Fazit:
Wir müssen dafür sorgen, dass unsere Uhr auch ohne Anschluss an den PC beim Booten synchronisiert wird. Genau das machen wir gleich mit Hilfe eines Zeit-Servers im Internet.

Zuvor aber noch ein paar Fakten zu den drei Zeitsystemen, die über die Darstellung hinausgehen.

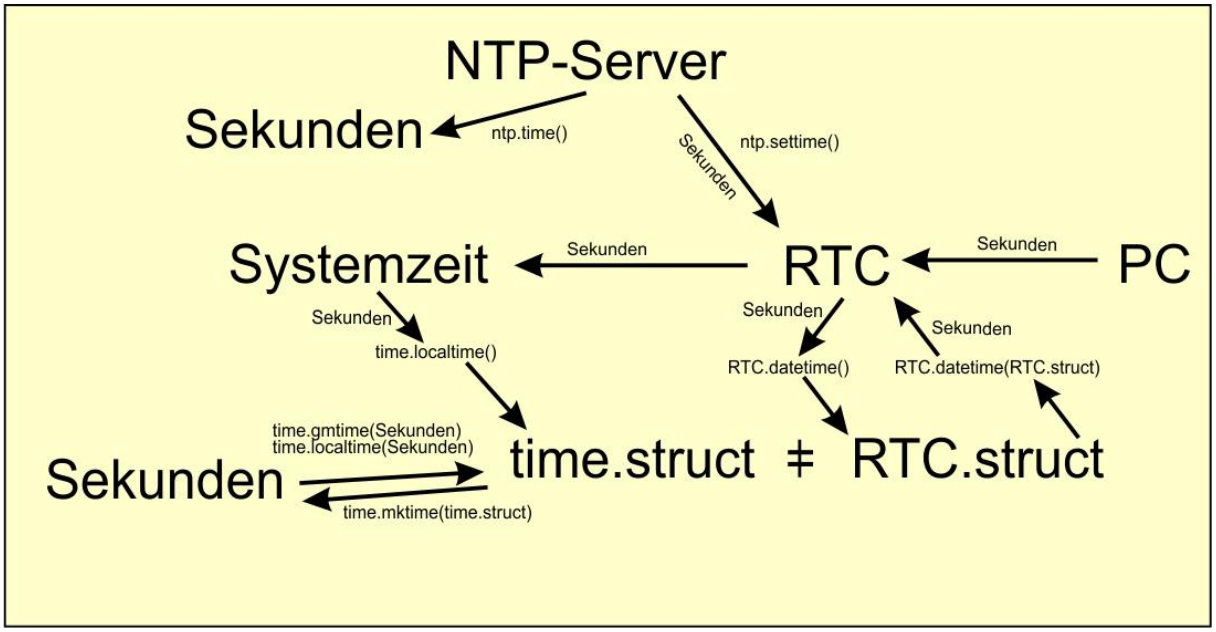

Abbildung 4: Interaktion der Zeitsysteme

Der Zeitmanager auf dem ESP32 ist eindeutig die RTC. Sie verwaltet den Sekundenzähler und beherrscht ebenfalls die Umwandlung von Sekunden in eine RTC-Zeitstruktur und umgekehrt. Nur der Sekundenwert, der in der RTC vorgehalten wird, kann von außen verändert werden, vom PC, von einem MicroPython-Programm und über den Zugriff auf einen NTP-Server. Die RTC ist also lesbar und schreibbar.

Die Systemzeit wird über die RTC synchronisiert und kann nur ausgelesen werden. allerdings kann man der Funktion **time.localtime**() ein Argument übergeben, einen Sekundenwert, der dann in ein time-Struktur-Tupel umgewandelt wird. Das ist praktisch, wenn man Zeitpunkte in der Zukunft oder Vergangenheit berechnen möchte. Dafür können auch die Funktionen **time.mktime**() und **time.gmtime**() herangezogen werden. Erstere erzeugt aus einem time-Tupel einen Sekundenwert, die zweite arbeitet umgekehrt und macht aus einem Sekundenwert ein time-Struktur-Tupel. Wir schauen uns jetzt ein paar Beispiele dazu in REPL an. Wir beginnen mit dem Import der Zutaten.

>>> from time import localtime,gmtime,mktime
>>> from machine import RTC

Dann fragen wir die Systemzeit ab und erhalten ein time-Struktur-Tupel, es hat die folgende Form.
(Y, M, D, h, m, s, dow, doy)

>>> localtime()
(2025, 4, 5, 18, 14, 9, 5, 95)

Daraus machen wir die Sekunden der Epoche

>>> mktime((2025, 4, 5, 18, 14, 9, 5, 95))
797192049

Und aus den Sekunden wird wieder das time-Struktur-Tupel

>>> gmtime(797192049)
(2025, 4, 5, 18, 14, 9, 5, 95)

Ein Tag hat 86400 Sekunden, dann müsste jetzt das Datum von morgen herauskommen.

>>> gmtime(797192049 + 86400)
*(2025, 4, **6**, 18, 14, 9, 6, 96)*

Wetten, dass jetzt der 02.04.2025 rauskommt? **localtime**() ist derselben Meinung.

>>> gmtime(797192049 - 86400*3)
(2025, 4, 2, 18, 14, 9, 2, 92)
>>> localtime(797192049 - 86400*3)
(2025, 4, 2, 18, 14, 9, 2, 92)

Der Beginn der Epoche wird durch die Sekunde 0 angegeben.

>>> localtime(0)
(2000, 1, 1, 0, 0, 0, 5, 1)

Um die Systemzeit zu verändern brauchen wir ein RTC-Objekt.

>>> rtc=RTC()

Das RTC-Tupel hat einen anderen Aufbau.
(Y, M, D, dow, h, m, s, subsec)

>>> rtc.datetime()
(2025, 4, 5, 5, 18, 18, 44, 18560)

Wir setzen jetzt ein anderes Datum und eine andere Uhrzeit. Danach fragen wir sofort die RTC und die Systemzeit ab. Befehle kann man in einer Zeile durch Strichpunkte trennen.

>>> rtc.datetime((2025, 4, 2, 0, 18, 14, 9, 0));rtc.datetime();localtime()
*(2025, 4, 2, **2**, 18, 14, 9, **121**)*
*(2025, 4, 2, 18, 14, 9, **2**, **92**)*

Hier erkennen wir den Unterschied in den Struktur-Tupeln ganz deutlich.

Wie wir noch sehen werden, liefern NTP-Server im Internet keine Struktur-Tupel sondern lediglich Sekundenwerte, die wir mit der Funktion **ntp.time**() abmelken können. Leider sind die Sekundenwerte als UTC-Zeitstempel angegeben. Die UTC (Universal Time Coordinated) entspricht der Zeit am 0. Längengrad (Greenage). Das bedeutet für Deutschland, dass hier eine Stunde zu wenig angegeben wird und daher eine Stunde zur UTC zu addieren ist. Wir sind eben in der Zeitzone +1. Und im Sommer kommt durch die Zeitumstellung am letzten Sonntag im März bis zum letzten Sonntag im Oktober noch eine weitere Stunde dazu. Zwar gibt es im Modul **ntptime** eine Methode, die automatisch unsere RTC synchronisieren kann, aber leider berücksichtigt die Methode **settime**() weder die Zeitzone noch die Sommerzeit. Wir müssen uns also selbst um die Korrektur kümmern, wenn wir die RTC mit einem Zeitserver synchronisieren wollen. Wie das geht, zeigt das Programm **ntp_test.py**. Selbstredend, dass wir für den Zugriff auf einen Zeitserver eine Internetverbindung brauchen. Der Zugriff auf den Server ist eigentlich mit zwei Zeilen erledigt. Aber damit dieser Zugriff möglich wird, dafür brauchen wir erheblich mehr Aufwand.

Wir importieren das Modul **ntptime** unter dem Namen **ntp**. Mit dem Modul **network** bauen wir die Verbindung zum Router auf. Zum Umwandeln der Zeitformate holen wir die bereits bekannten Methoden aus dem Modul **time**. Die Credentials für den Login am Router befinden sich in der Datei **credentials.py**.

```
# credentials.py
#
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySSID = "EMPIRE_OF_ANTS"; myPass = "nightingale"
myIP="10.0.2.182"
myPort=9001
mySocket=(myIP,myPort)
myMask="255.255.255.0"
myGW="10.0.2.200"
myDNS="10.0.2.100"
```

Der Stern holt uns alle Attribute in den globalen Namensraum, das spart uns die Angabe des Prefixes **credentials** bei der Referenzierung. **credentials.py** muss in den Flash des ESP32 hochgeladen werden! Den Rest kennen wir schon aus dem Programm **uhr.py**.

```
# ntp_test.py
import ntptime as ntp
import network
from time import sleep, localtime, gmtime, mktime
from credentials import  *
from timeout import TimeOutMs
from sys import exit
from machine import Pin, RTC
```

Wir sind in Zeitzone 1

```
timeZone=1
```

Das Dictionary **connectStatus** ist eine Übersetzungstabelle für Nummern beim Verbindungsaufbau zum Router in Klartext.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }
```

Funktion **hexMac**() nennt uns die MAC-Adresse des Station-Interfaces. Die brauchen wir, um dem Router unseren Controller bekannt zu machen. In meinem Router habe ich einen Wachhund (MAC-Filtering) engagiert, der beißt jeden in den Hintern, den er nicht kennt und lässt ihn natürlich auch nicht ins System.

```
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
        val="{:02X}".format(byteMac[i])
        macString += val
```

```
        if i <len(byteMac)-1 :           # Trennzeichen
            macString +="-"              # bis auf letztes Byte
    return macString
```

Die Funktion **connect2WLAN**() stellt die Verbindung her und gibt das Interface-Objekt zurück. Es ist wichtig, dass ein existierendes Gateway (Router) und ein Nameserver angegeben werden, denn das **ntptime**-Modul muss den Namen des Zeitservers in eine IP-Adresse umwandeln können und natürlich Pakete (UDP) zum entfernten Server schicken können.

```
def connect2WLAN():
    nic = network.WLAN(network.STA_IF) # WiFi-Objekt erzeugen
    nic.active(True)                   # STA-Objekt nic ein
    sleep(1)
    #
    MAC = nic.config('mac')   # binaere MAC-Adresse abrufen
    myMac=hexMac(MAC)      # in eine Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    #
    # Verbindung mit AP im lokalen Netzwerk aufnehmen,
    # falls noch nicht verbunden
    nic.ifconfig((myIP,myMask,myGW,myDNS))
    if not nic.isconnected():
      # Zum AP im lokalen Netz verbinden und Status anzeigen
      nic.connect(mySSID, myPass)
      # warten bis die Verbindung zum Accesspoint steht
      print("connection status: ", nic.isconnected())
      n=0
      line="..........."
      while (nic.status()!= network.STAT_GOT_IP) and (n < 10):
        n+=1
        print(".",end='')
        sleep(1)
    # Wenn verbunden, zeige Verbindungsstatus und Config-Daten
    nicStatus=nic.status()
    print("\nVerbindungsstatus: ",connectStatus[nicStatus])
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",
          STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    return nic
```

Die Funktion **inTheSummertime**() hat nix mit dem Lied von Mungo Jerry aus den 70-ern zu tun, sondern stellt fest, ob ein Datum Sommerzeitcharakter hat. In diesem Fall wird zur Zeitzone auch noch eine weitere Stunde addiert. Jahr, Monat und Tag werden als einzelne Intergers übergeben.

```
def inTheSummertime(jahr,monat,tag):
    if monat < 3 or monat > 10:
        return timeZone
    if 3 < monat < 10:
        return timeZone + 1
    letzterMaerzSonntag = max([mtag for mtag in range(25,32) \
```

```
        if gmtime(mktime((jahr, 3, mtag, 2, 0, 0, 0, 0,
0)))[6] == 6])
    letzterOktSonntag = max([mtag for mtag in range(25,32) \
        if gmtime(mktime((jahr, 10, mtag, 2, 0, 0, 0, 0,
0)))[6] == 6])
    if monat == 3 and tag >= letzterMaerzSonntag:
        return timeZone + 1
    if monat == 10 and tag < letzterOktSonntag:
        return timeZone + 1
    return timeZone
```

Weil es dafür keine Funktion im MicroPython-Kern gibt, müssen wir die letzten Sonntage im März und Oktober selbst herausfuzeln. Wir setzen dafür eine List-Comprehension ein (ein wenig Python-Magie muss sein!!), die primär eine Liste mit den letzten 7 Monatstagen erstellt. Jeden einzelnen davon verwandeln wir in Sekunden und die wiederum in ein time-Struktur-Tupel. An dessen 6. Position steht der Wochentag und wenn der den Wert 6 (=Sonntag) hat, wird dieser Monatstag in die Liste aufgenommen. Falls es zwei Tage sind, nehmen wir den höheren Wert.

>>> [mtag for mtag in range(25,32)]
[25, 26, 27, 28, 29, 30, 31]
>>> jahr=2025
>>> mtag=25
>>> mktime((jahr, 3, mtag, 2, 0, 0, 0, 0))
796183200
>>> gmtime(mktime((jahr, 3, mtag, 2, 0, 0, 0, 0)))
(2025, 3, 25, 2, 0, 0, 1, 84)
>>> gmtime(mktime((jahr, 3, mtag, 2, 0, 0, 0, 0)))[6]
1
>>> gmtime(mktime((jahr, 3, mtag, 2, 0, 0, 0, 0)))[6] == 6
False

Mit **mtag**=30 erhalten wir **True**. 30 ist der einzige Wert in der Liste und daher auch das Maximum. Von November bis Februar ist Winterzeit, wir geben eine Stunde für den Offset der Zeitzone zurück. Von April bis September ist garantiert Sommerzeit, also geht Zeitzone +1 zurück. Den Rest kriegen wir durch den Vergleich mit dem Datum des letzten Sonntags heraus.

Verbindung zum Router herstellen.

```
nic=connect2WLAN()
```

Taste für den Programmabbruch definieren und ein RTC-Objekt erzeugen. Dann ab in die Hauptschleife. Den Rest erklären die Kommentare.

```
taste=Pin(0,Pin.IN)
r=RTC()

while 1:
    try:
        # Sekundenwert vom Server holen
```

```
        sekunden=ntp.time()
        # Jahr,Monat,Tag extrahieren
        year,month,day=gmtime(sekunden)[0:3]
        # Stundenoffset zur Netzzeit berechnen
        offset=inTheSummertime(year,month,day)
        # ntp.time() liefert Anzahl Sekunden
        # eine Stunde hat 3600 Sekunden
        tag=localtime(sekunden+offset*3600)
        print("NTP-Time       ",tag)
        # Systemzeit-Format durch localtime()
        # year,month,day,hor,minute,second,dow,doy

    except:
        print("Uebertragungsfehler")
    sleep(1)

    if taste() == 0:
        exit()
```

Wenn wir das im Test laufen lassen, dann ergibt sich die folgende Ausgabe in REPL. Die Ausgabe, die aus dem UTC-Timestamp + Lokal-Colorit geformt wurde, ist natürlich im Format der Systemzeit-Struktur,.

```
>>> %Run -c $EDITOR_CONTENT
STATION MAC:      FC-F5-C4-27-09-10

connection status:  False
..
Verbindungsstatus:  STAT_GOT_IP
STA-IP:             10.0.1.182
STA-NETMASK:        255.255.255.0
STA-GATEWAY:        10.0.1.20
NTP-Time       (2025, 4, 5, 18, 57, 59, 5, 95)
NTP-Time       (2025, 4, 5, 18, 58, 0, 5, 95)
NTP-Time       (2025, 4, 5, 18, 58, 1, 5, 95)
```

Der Weg zum fertigen Programm

Was zu tun bleibt, um aus den Bausteinen, die wir bislang entwickelt haben, ein autonom laufendes Programm zu bauen, folgt jetzt. Wir mixen **uhr.py** mit **ntp_test.py** und machen daraus **uhr2.py**. Außerdem spendieren wir noch eine Überwachung der Umgebungsparameter Temperatur, Luftdruck und relativer Luftfeuchte, Taupunkt-Berechnung als Option.

Ich bereite das Listing von uhr2.py schrittweise auf. Arbeitsschritte der Vorlagen habe ich teilweise in Funktionen verpackt. Beim Import wurden alle bisherigen und zusätzlich gebrauchten Module berücksichtigt.

Wichtig:
Damit später alles richtig funktioniert, müssen folgende Dateien in den Flash des ESP32 hochgeladen werden:
credentials.py
timeout.py
oled.py
ssd1306.py
bme280

```
# uhr2.py
import network
from machine import Pin,RTC,Timer,SoftI2C
from neopixel import NeoPixel
from time import sleep_ms, sleep, localtime, gmtime, mktime
from sys import exit
import ntptime as ntp
from credentials import  *
from timeout import TimeOutMs
from oled import OLED
from bme280 import BME280
```

Timer und I2C-Bus müssen an den jeweiligen Controller angepasst werden. Das macht der if-elif-else-Block. Damit wird das Programm für alle aufgeführten Familien lauffähig.

```
if platform == "esp32":
    t0=Timer(0)
    i2c=SoftI2C(scl=Pin(21),sda=Pin(22),freq=100000)
elif platform == "rp2":
    t0=Timer()
    i2c=SoftI2C(scl=Pin(12),sda=Pin(13),freq=100000)
else:
    print("Nicht unterstützter Controller")
    exit()
```

NeoPixel-Ausgang ist GPIO14 für maximal 30 LEDs

```
np=Pin(14, Pin.OUT)
n=NeoPixel(np,30)
```

Das I2C-Bus-Objekt leiten wir an den OLED- und BME280-Konstruktor weiter. Die Umgebungswerte ermitteln wir schon mal vorab.

```
d=OLED(i2c,heightw=64)
bme=BME280(i2c)
temp=bme.calcTemperature()
pres=bme.calcPressureNN(h=465,temp=20)
hum=bme.calcHumidity()
```

RTC-Objekt erzeugen

```
rtc=RTC()
```

Wir lesen schon mal die RTC aus und definieren die Attribute, die für die Verarbeitung wichtig sind, Indizes und Farben.

```
dt=rtc.datetime()
anno=const(0)
mon=const(1)
mday=const(2)
wday=const(3)
hor=const(4)
mnt=const(5)
sec=const(6)
high=(0x40,0,0)
low=(0,0,0x40)
lowH=(0x40,0,0x40)
highH=(0x40,0x40,0)
single=(0,0x40,0)
```

Wir sind in Zeitzone +1, führen jede Stunde das nächste Zeitupdate durch und setzen den Refresh-Zähler auf 0. Für die Ausgabe im OLED erstellen wir eine Liste der Wochentagsnamen.

```
timeZone=1
refreshPoint=3600 # Sekunden zur naechsten Synchronisation
refreshCnt=0
wochentag=[
    "Montag",
    "Dienstag",
    "Mittwoch",
    "Donnerstag",
    "Freitag",
    "Samstag",
    "Sonntag"
    ]
```

Das kennen wir schon.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
```

```
    }
```

Das auch.

```
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
        val="{:02X}".format(byteMac[i])
        macString += val
        if i <len(byteMac)-1 :          # Trennzeichen
            macString +="-"             # bis auf letztes Byte
    return macString
```

Auch das ist bekannt.

```
def connect2WLAN():
    nic = network.WLAN(network.STA_IF) # WiFi-Objekt erzeugen
    nic.active(True)                   # STA-Objekt nic ein
    sleep(1)
    #
    MAC = nic.config('mac')   # binaere MAC-Adresse abrufen
    myMac=hexMac(MAC)      # in eine Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    #
    # Verbindung mit AP im lokalen Netzwerk aufnehmen,
    # falls noch nicht verbunden
    nic.ifconfig((myIP,myMask,myGW,myDNS))
    if not nic.isconnected():
      # Zum AP im lokalen Netz verbinden und Status anzeigen
      nic.connect(mySSID, myPass)
      # warten bis die Verbindung zum Accesspoint steht
      print("connection status: ", nic.isconnected())
      n=0
      line="..........."
      while (nic.status() != network.STAT_GOT_IP) and (n <
10):
        n+=1
        print(".",end='')
        sleep(1)
    # Wenn verbunden, zeige Verbindungsstatus und Config-Daten
    nicStatus=nic.status()
    print("\nVerbindungsstatus: ",connectStatus[nicStatus])
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",
          STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    return nic
```

ISR ist bekannt aus dem ersten Beitrag zum Thema.

```
def tick(t0):
    global ticked
    ticked = True
```

Ebenso.

```
def ledsOff(nbr,immediate=None):
    for i in range(nbr):
        n[i]=(0,0,0)
    if immediate:
        n.write()
```

Immer wieder Sommerzeit.

```
def inTheSummertime(jahr,monat,tag):
    if monat < 3 or monat > 10:
        return timeZone
    if 3 < monat < 10:
        return timeZone + 1
    letzterMaerzSonntag = max([mtag for mtag in range(25,32) \
        if gmtime(mktime((jahr, 3, mtag, 2, 0, 0, 0, 0,
0)))[6] == 6])
    letzterOktSonntag = max([mtag for mtag in range(25,32) \
        if gmtime(mktime((jahr, 10, mtag, 3, 0, 0, 0, 0,
0)))[6] == 6])
    if monat == 3 and tag >= letzterMaerzSonntag:
        return timeZone + 1
    if monat == 10 and tag < letzterOktSonntag:
        return timeZone + 1
    return timeZone
```

Das ist neu, die Synchronisation der RTC mit der Zeit des NTP-Servers. **refreshCnt** wird global erklärt, damit die Wertänderung die Funktion verlassen kann.

Jetzt wird versucht, eine Verbindung zum Zeitserver aufzunehmen, der im NTP-Objekt definiert ist: **pool.ntp.org**. Das bereits bekannte Hin- und Herschaufeln der Zeitdaten führt letztlich dazu, dass die RTC auf dem aktuellen Stand ist. Andernfalls beginnt die Refresh-Periode von vorn und die RTC läuft ohne Update weiter. Den Refreshzähler setzen wir natürlich auf 0 zurück.

```
def synchronize():
    global refreshCnt
    try:
        sekunden=ntp.time()
        year,month,day=gmtime(sekunden)[0:3]
        offset=inTheSummertime(year,month,day)
        # offset=zeitzone+0/1
        sekunden += offset*3600
        # Zeitzone und Sommer/Winterzeit
        Y,M,D,h,m,s,dow,doy=gmtime(sekunden)
        rtc.datetime((Y,M,D,dow,h,m,s,0))
```

```
        print("synchronized: ",rtc.datetime())
    except OSError as e:
        print("Synchron-Fehler",e)
    refreshCnt=0
```

Im OLED-Display lassen wir den Wochentag, das Datum und die Uhrzeit anzeigen. Dazu löschen wir den Bereich von links oben bis zum Ende in Textzeile 2 verdeckt (False). Das heißt, wir schreiben nur in den Puffer im ESP32, ohne den Inhalt sofort zum Display zu schicken. Den Namen des Wochentags holen wir aus der Liste **wochentag**. Als Index dient die Nummer im rtc-Date-Time-Tupel **dt** an Position **wday** (= 3). Für Datum und Uhrzeit verwenden wir eine formatierte Ausgabe. **{:0>2}** bedingt eine Spalte mit zwei Zeichen Breite in der rechtsbündig, gegebenenfalls mit einer führenden 0, die Zahlen ausgegeben werden. In Zeile 2 fehlt das False am Ende. Das bewirkt, dass der Zeichen-Puffer, in den wir bislang geschrieben haben, jetzt zum Display geschickt wird.

```
def showDateTime():
    d.clearFT(0,0,16,2,False)
    d.writeAt(wochentag[dt[wday]],0,0,False)
    d.writeAt("{:0>2}.{:0>2}.{:0>2}".\
              format(dt[mday],dt[mon],dt[anno]),0,1,False)
    d.writeAt("{:0>2}:{:0>2}:{:0>4}".\
              format(dt[hor],dt[mnt],dt[sec]),0,2)
```

Die Funktion **abakus**() sorgt für die korrekte Erleuchtung des LED-Streifens. Wie das funktioniert, haben wir im vorangegangenen Post schon genau beschrieben.

```
def abacus(dt):
    hour=dt[hor] % 12
    sColor = lowH if hour <= 6 else highH
    ledsOff(30)

    if 1 <= hour <= 6:
        for i in range(1,hour+1):
            n[i-1] = sColor
    elif 7 <= hour <= 11:
        for i in range(7,hour+1):
            n[i-7] = sColor


    FifeMin=dt[mnt] // 5
    Minutes=dt[mnt] % 5
    sColor = low if FifeMin <= 6 else high
    if FifeMin <= 6:
        for i in range(FifeMin % 7):
            n[i+6]=sColor
    else:
        for i in range(FifeMin - 6):
            n[i+6]=sColor
    for i in range(1,Minutes+1):
        n[i+5+6]=single
```

```
    FifeSec=dt[sec] // 5
    Seconds=dt[sec] % 5
    sColor = low if FifeSec <= 6 else high
    if FifeSec <= 6:
        for i in range(FifeSec % 7):
            n[i+16]=sColor
    else:
        for i in range(FifeSec - 6):
            n[i+16]=sColor
    for i in range(1,Seconds+1):
        n[i+5+16]=single

    n.write()
    print("RTC-Zeit:",dt[hor],dt[mnt],dt[sec])
    showDateTime()
```

Im Display sind noch drei Zeilen frei und bieten Platz für die Messwerte vom BME280. Wir brauchen die geänderten Werte im globalen Namespace (Namensraum). Was zum Abholen der Werte im Hintergrund alles nötig ist, darum kümmert sich das Modul **bme280**, das natürlich auch in den Flash des ESP32 hochgeladen werden muss.

```
def getEnvironment():
    global temp,pres,hum
    temp=bme.calcTemperature()
    pres=bme.calcPressureNN(h=465,temp=20)
    hum=bme.calcHumidity()
```

Diese Werte geben wir als Fließkommazahlen mit einer Spaltenbreite von 6 Zeichen mit einer Nachkommastelle aus.

```
def showEnvironment():
    d.clearFT(0,3,16,5,False)
    d.writeAt("{:6.1f} *C".format(temp),0,3,False)
    d.writeAt("{:6.1f} hPa".format(pres),0,4,False)
    d.writeAt("{:6.1f} %".format(hum),0,5)
```

Wir verbinden uns mit dem Router, erzeugen ein Tasten-Objekt für den Programmabbruch, löschen alle LEDs, synchronisieren die RTC mit dem Zeitserver auf **pool.ntp.org** und starten den Sekundentimer.

```
nic=connect2WLAN()

taste=Pin(0,Pin.IN, Pin.PULL_UP)
ledsOff(30,True)
synchronize()
ticked=False
t0.init(period=1000,mode=Timer.PERIODIC,callback=tick)
```

Durch die ISR (Interrupt Service Routine) des Sekundentimers wird das Flag **ticked** auf True gesetzt. In der Mainloop wird dadurch die Sequenz zur Zeitanzeige gestartet.

Die Mainloop wurde durch das Auslagern der einzelnen Aufgaben in Funktionen sehr übersichtlich.

Wir stellen das Flag auf False zurück, holen das aktuelle rtc-Tupel nach **dt** und lesen die Environment-Werte vom BME280 ein. Mit **abacus**(dt) wird der LED-Streifen angesteuert.

```
while 1:
    if ticked:
        ticked = False

        dt=rtc.datetime()
        getEnvironment()
        abacus(dt)
```

Den Refresh-Zähler erhöhen wir um 1 und prüfen, ob eine neue Zeitsynchronisation erforderlich ist. Schließlich lassen wir noch die eingelesenen Messwerte anzeigen.

```
        refreshCnt += 1
        if refreshCnt == refreshPoint:
            synchronize()
        showEnvironment()
```

Unabhängig vom Zustand des Sekundentimers prüfen wir auf die Betätigung der Abbruchtaste. Damit können wir das Programm stets zuverlässig und geordnet verlassen, auch wenn Strg+C nicht mehr funktioniert. Vor allem werden dann die LEDs alle ausgeschaltet und der Timer wird deaktiviert. **break** verlässt die while-Schleife und beendet damit das Programm.

```
    if taste() == 0:
        ledsOff(30, True)
        t0.deinit()
        break
```

Ist alles komplett aufgebaut? Sind alle erforderlichen Dateien in den Flash hochgeladen? Die **main.py**, die sich noch im Flash befindet löschen wir (Rechtsklick und Delete).

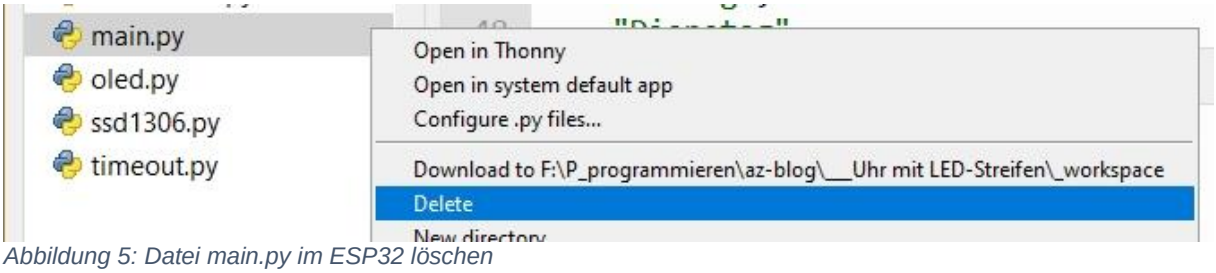

Abbildung 5: Datei main.py im ESP32 löschen

Dann starten wir das Programm uhr2.py erst mal in einem Editorfenster von Thonny. Wenn alles perfekt läuft stoppen wir das Programm mit der Flashtaste vom ESP32.

Damit es autonom läuft, schicken wir es als **main.py** in den Flash-Speicher. Nach einem Reset startet der ESP32 das Programm auch dann, wenn kein PC angeschlossen ist.

Das vorliegende Projekt ist natürlich noch beliebig ausbaufähig. Ich habe eingangs schon angedeutet, dass die Helligkeit der LEDs automatisch durch einen LDR gesteuert werden könnte. Denkbar wäre auch die Übermittlung der Messwerte an einen Server oder an ein Handy. Auch die Aufzeichnung der Werte mit einem Datenlogger via SD-Karten-Modul ist möglich. Beispiele dazu finden Sie in der MicroPython-Blogfolge.

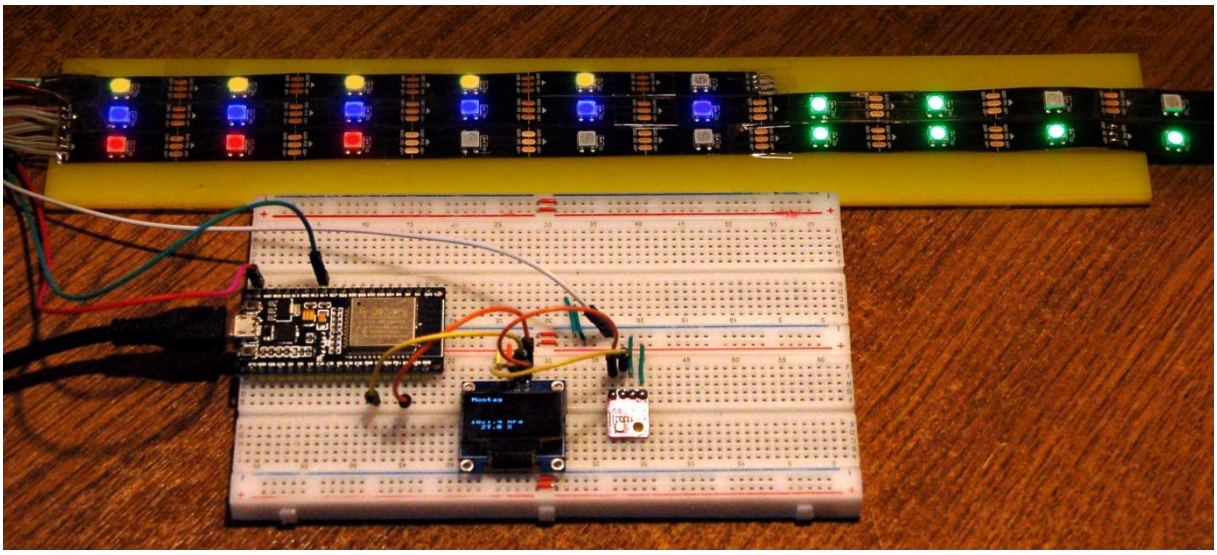

Abbildung 6: Es ist 11 Uhr 32 Minuten und 49 Sekunden