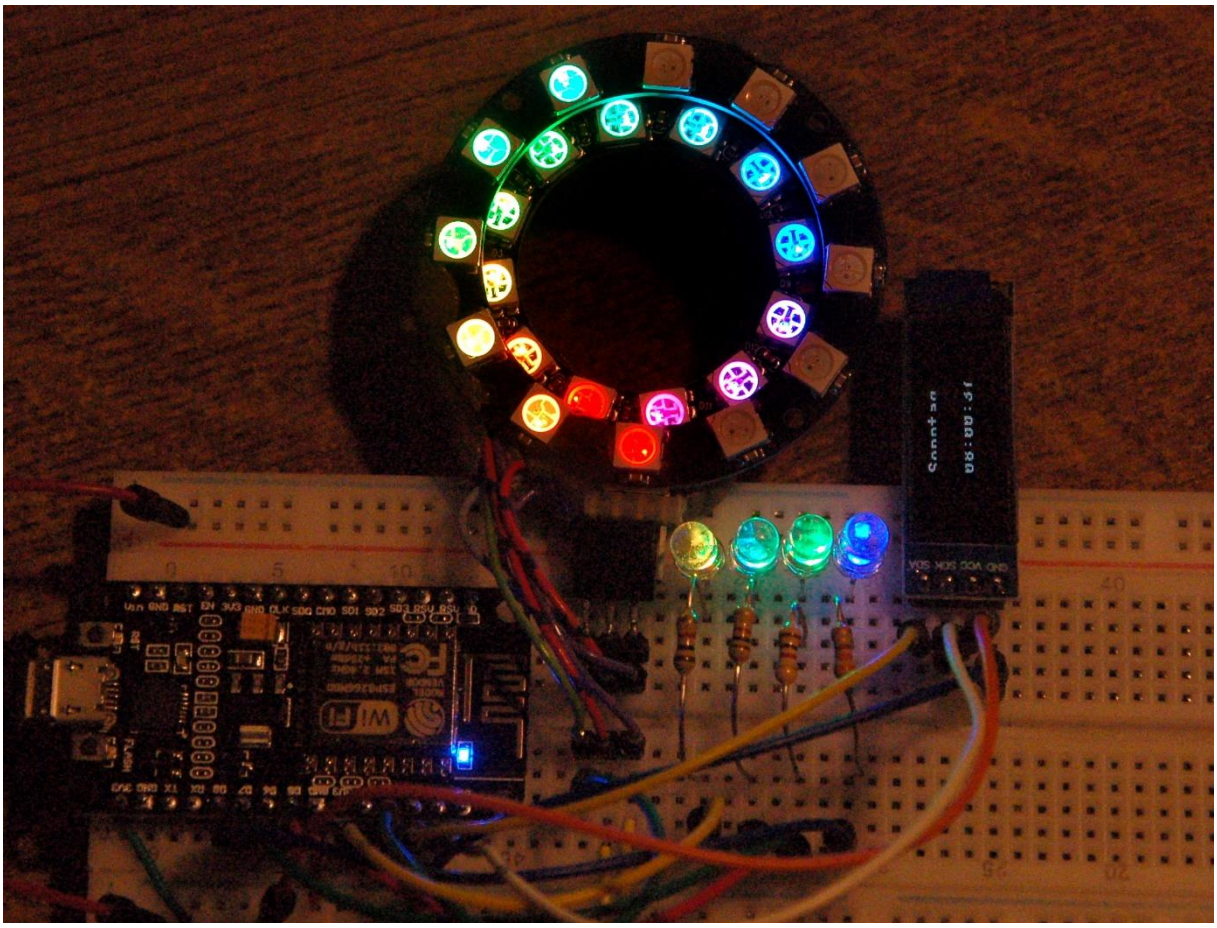

Advents-Kranz-Kalender

Diese Anleitung gibt es auch als PDF-Dokument.

Apfent, Apfent, der Bärwurz brennt.
Erst trinks oan, dann zwao, drei, vier,
dann hautsde mit dein Hirn an d' Tür.

In diesem Vierzeiler aus dem Buch "Weihnachtsgeschichten von Toni Lauerer – Apfent" brennt zwar auch etwas, aber was bei uns brennen soll, ist kein Bärwurz und auch nicht nur die vier Kerzen am Adventskranz. Es sollen schon ein paar Lichter mehr sein, wie wäre es mit 28 Stück? Tatsächlich beginnen im August bereits die ersten Discounter, die Regale mit Lebkuchen und Spekulatius zu füllen, damit wir nicht 28 Wochen auf Weihnachten warten müssen, aber wie verhält es sich jetzt mit unseren 28 Lichtern? Das erzähle ich Ihnen gern in einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Der Advents-Kranz-Kalender mit dem ESP8266

Wenn Sie den Prospekt vom Discounter vor zwei, drei Wochen durchgeblättert haben, werden Sie sicher über diverse Adventskalender mit dubiosen Inhalten wie

Spielsachen, Bier, Wein, Schnaps und noch ganz anderen delikaten Sachen gestolpert sein, je nach Anbieter. Von dieser Art ist unser Kalender nicht. Der leuchtet nur, das aber kräftig. Dafür sorgen zwei Neopixel-Ringe, ein kleiner (37mm Ø) und ein großer (50mm Ø). Hintereinander gelegt bilden sie den Kranz. Natürlich brauchen wir auch Kerzen, die durch Blink-LEDs ersetzt werden. Das dient der Sicherheit und schützt vor Brandgefahr durch unbeaufsichtigtes Abbrennen von Wachskerzen. Außerdem kann man das Ding auch ohne Probleme an die Wand hängen. Probieren Sie das mal mit einem handelsüblichen Adventskranz. Damit jederzeit Datum und Uhrzeit ersichtlich sind, habe ich dem Adventskranzkalender auch noch ein kleines OLED-Display spendiert, das hinter dem inneren kleinen Ring platziert wird. Zusammen mit dem ESP8266, der das Kommando übernimmt, haben wir die Hardware auch schon beieinander.

Hardware

1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Neopixel-Ring 50mm
1	Neopixel-Ring 37mm
4	Widerstand 1 kΩ
1	LED Leuchtdioden Sortiment Kit, 350 Stück, 3mm & 5mm, 5 Farben - 1x Set
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
Evtl.	Basisbrett 17cm x 17cm

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber zum OLED-Display
oled.py API für das OLED-Display

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 05.02.2022) 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

Wie eine Übertragung auf dem I2C-Bus abläuft und wie die Signalfolge ausschaut, das können Sie in meinem Beitrag mammutmatrix_2_ger.pdf nachlesen. Ich verwende dort ein interessantes kleines Tool, mit dem Sie die I2C-Bus-Signale auf Ihren PC holen und analysieren können.

Jetzt geht's rund mit dem Adventskranz

Ja, die Teile sind halt mal nicht eckig, auch wenn ein Kalender dahintersteckt. Sicher ist Ihnen aufgefallen, dass die beiden Ringe miteinander 24 LEDs auf den Tisch bringen. Was liegt näher, als denen je einen Tag vom 1.12. bis zum 24.12. zuzuordnen und zu gegebener Zeit dann die "Kerzen" dazuzuschalten. Hier ist der übersichtliche Schaltplan.

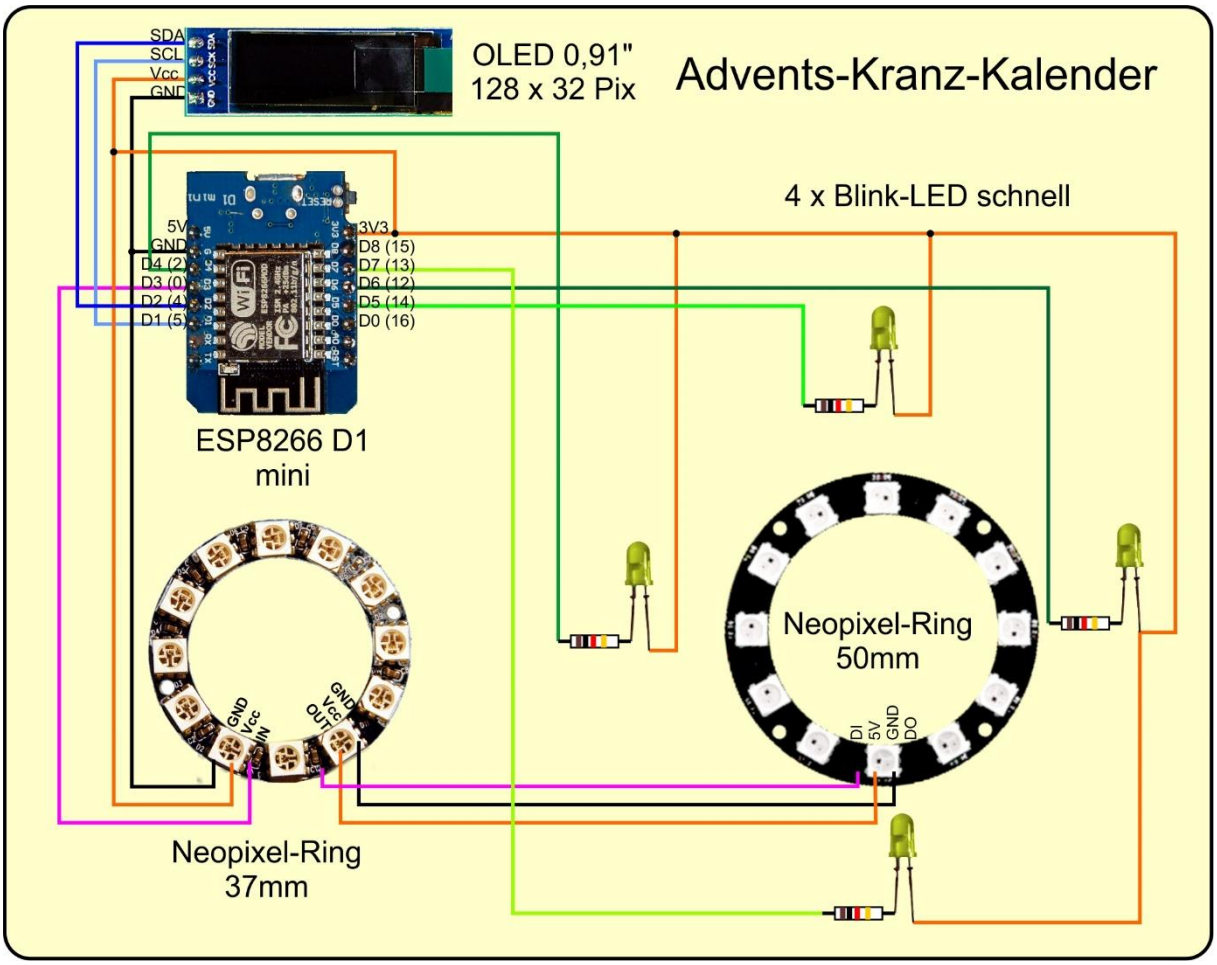

Abbildung 1: Advents-Kranz-Kalender - Schaltung

Für die Neopixelringe, sie sind kaskadiert, brauche ich nur einen GPIO-Pin, das ist D3 auf dem ESP8266-Board. MicroPython-technisch steckt der Pin GPIO0 dahinter.

Die vier Blink-LEDs aus dem Set werden von den Anschlüssen GPIO2 (D4), GPIO14 (D5), GPIO12(D6) und GPIO13 (D7) bedient.

Bleibt noch das OLED-Display, welches über den I2C-Bus angesteuert wird. Die Busleitungen sind SCL=GPIO5 (D1) und SDA=GPIO4 (D2).

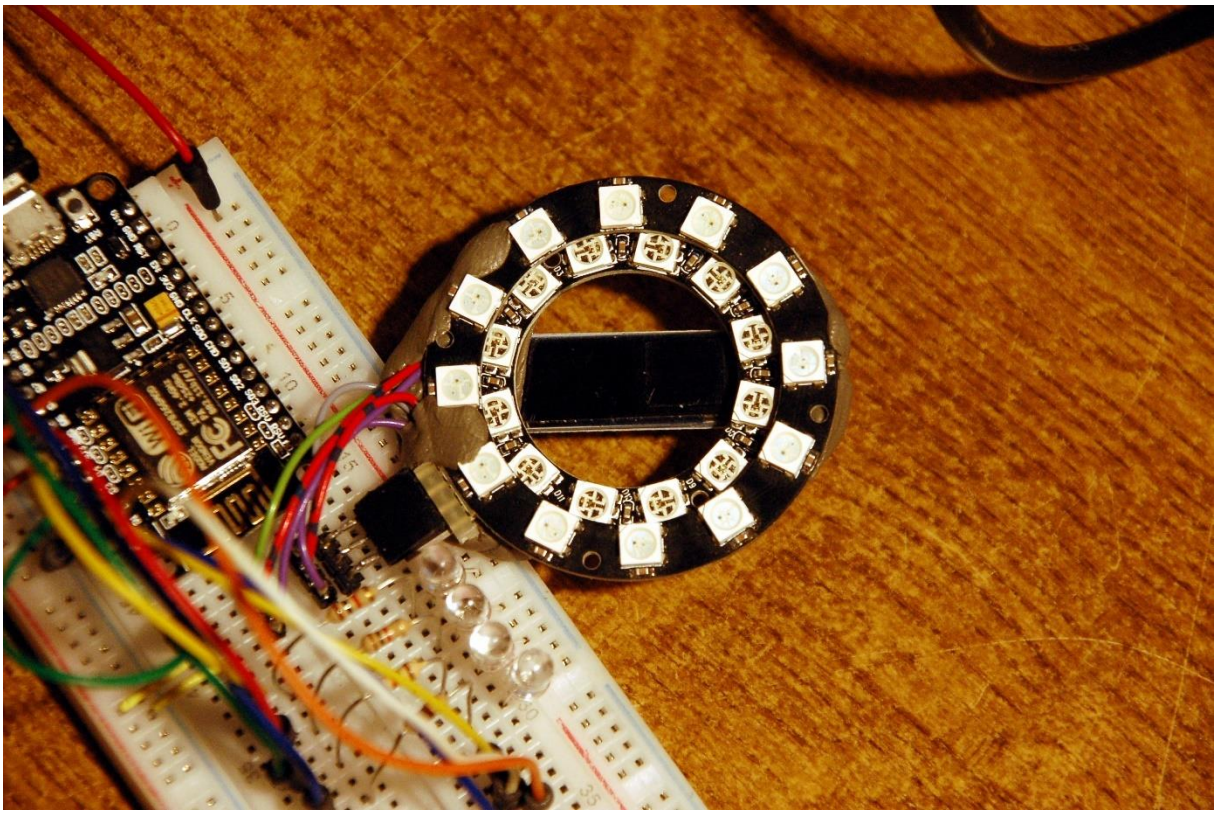

Abbildung 2: Advents-Kranz-Kalender - Testschaltung

Abbildung 2 zeigt die Anordnung der beiden Ringe und dahinter das Display. Die vier einzelnen LEDs werden als "Kerzen" verwendet und im finalen Aufbau um die Ringe herum verteilt, wie beim echten Adventskranz.

Eine Liste der Zuordnungen zwischen den Arduino-Bezeichnungen der IO-Pins und der nativen von MicroPython zeigt die folgende Tabelle aus dem Listing.

```
# Pintranslator fuer ESP8266-Boards
# ARDUINO-Pins D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                SC SD
```

Das war's dann auch schon mit der Hardware. Kommen wir zum Programm. Das wartet mit ein paar netten Besonderheiten und Details auf, die nicht zuletzt Eigenheiten des ESP8266 sind.

Was wäre ein Adventskranz ohne Programm?

Oh nein, ich will den Gärtnereien nicht absprechen, sie hätten keinen gestalterischen oder gar künstlerischen Hintergedanken bei der Zusammenstellung von Adventskränzen. Dasselbe gilt natürlich auch für Objekte, die im trauten Heim sorgsam hergerichtet wurden. Oft steckt ein Motto, ein Programm dahinter.

Auch unsere Hardware kommt ohne Programm, allerdings anderer Art, nämlich eine Steuerung durch ein MicroPython-Programm, nicht aus.

Der Betrieb bietet dreierlei Modi – Test-Modus – synchronisiert durch einen NTP-Server – RTC-basiert im Offline-Betrieb. Das alles geht mit einem Programm. Zugegeben, das ist etwas umfangreicher geworden, als ursprünglich geplant. Aber – die Auswahl des Betriebsmodus erfolgt lediglich über eine Variable (Testmodus) oder automatisch darüber, ob ein WLAN erreichbar ist, oder eben nicht.

Im Testmodus wird die Variable **debug**, zu der Sie später Genaueres erfahren, auf **True** gesetzt. Ist **debug = False**, dann versucht das Programm eine Verbindung zu einem, von Ihnen angegebenen Accesspoint aufzubauen. Gelingt das nicht, dann greift es auf die Angabe einer Startbedingung in der Variablen **rtcTag** zurück, welches die Tageswerte in einem 8-Tuple mit er folgenden Bedeutung enthält.

(Jahr, Monat, Tag, Wochentag, Stunde, Minute, Sekunde, Millisekunden)

Tuples sind Zusammenstellungen von verschiedenen Daten zu einem Verbund. Notiert werden sie in runden Klammern. Ein Tuple ist in MicroPython ein immutabler Datentyp. Das bedeutet, dass Sie die Felder einer solchen Datenstruktur nicht nachträglich verändern können. Änderungen sind aber möglich, bevor das Programm startet und natürlich, wenn Sie aus einem Tuple zur **Runtime** (= während des Programmlaufs) ein neues erstellen.

Leider gibt es, bedingt durch die Schnittstellen zwischen den drei Datenformaten der Datumsverwaltung bei den verwendeten Modulen Probleme, die ich aber durch den Einsatz einer Funktion auf eine gemeinsame Schiene herunterbrechen konnte. Mein Standard sieht daher wie folgt aus. Ich habe mich hier an das Format von **localtime**() gehalten.

(Jahr, Monat, Tag, Stunde, Minute, Sekunde, Wochentag, Tag im Jahr)

Eigenheiten des ESP8266

Im Kernel des ESP8266 ist es verankert, dass der Controller, falls er bereits einmal eine Verbindung mit einem Accesspoint aufgebaut hat, beim Neustart versucht, diese Verbindung erneut herzustellen. Meistens ist das nervig, weil man darauf keinen Einfluss nehmen kann. Erstens verzögert es den Start, wenn der Accesspoint nicht (mehr) zur Verfügung steht. Zweitens versucht der ESP8266 eine Verbindung aufzubauen, in der selbst einen Accesspoint spielt. Das kann zu nervenden ständigen Neustarts führen.

Der erste Schritt dieses Verhalten abzustellen, ist, dass man nach dem erneuten Flashen der Firmware an der Kommandozeile, im Terminalfenster von Thonny zum Beispiel, folgenden Befehl eingibt:

>>> import webrepl_setup

Danach erschein die Zeile:

> d fuer disable

Geben Sie hier **d** ein und rebooten Sie den ESP8266 danach mit der RST-Taste. Damit versucht der ESP8266 wenigstens nicht mehr webREPL, die Funkkommandozeile, zu starten.

Den zweiten Schritt der Problemlösung erläutere ich bei der Programmbesprechung.

Nicht alle GPIOs sind durch den Programmierer voll nutzbar. Das betrifft vor allem die Pins GPIO16 (D0) und GPIO15 (D8), die im Programm deswegen auch nicht verwendet werden.

Module

Zum Sprachumfang von MicroPython gehören eine Vielzahl von Modulen. Das sind Bibliotheken, die Spezialaufgaben erfüllen, wie die Ein- Ausgabe von Daten via GPIO-Pins, Timer, Bus-Leitungen wie I2C oder RS232, analoges Input usw. Weitere Hardware wird ebenfalls über Module bedient, die aber extra, bei Programmbeginn, als externe Dateien auf den ESP8266 hochgeladen werden müssen. In unserem Fall sind das die Dateien **oled.py** und **ssd1306.py**. Kopieren Sie diese Dateien nach dem Herunterladen in Ihr Arbeitsverzeichnis (_workspace) im Projektverzeichnis, das Sie an einem beliebigen Platz auf Ihrer Festplatte anlegen. In Thonny navigieren Sie zu Ihrem Arbeitsverzeichnis und rufen mit einem Rechtsklick auf die hochzuladende Datei das Kontextmenü auf. Dann wählen aus dem Kontextmenü den Punkt **Upload to */***.

Interne sowie externe Module werden beim Programmstart importiert und damit dem MicroPython-Interpreter zur Kenntnis gebracht.

```
from machine import Pin, SoftI2C, RTC
from time import sleep, time, localtime, ticks_ms, mktime
import ntptime
from neopixel import NeoPixel
from oled import OLED
import network, socket
from sys import exit
```

Ein Import wie durch

```
import network, socket
```

bindet alle Zeilen des Moduls mit dem Prefix **network** oder **socket** in den Namensraum des Programms ein. Dagegen importiert

```
from time import sleep, time, localtime, ticks_ms, mktime
```

nur die aufgeführten Methoden, die aber dann ohne das Prefix **time** zu verwenden sind.

Durch Angabe des Namens einer Klasse, wird nur der Inhalt dieser Klasse importiert, aber nicht das, was sich eventuell außerhalb davon in dem Modul an Definitionen von Objekten befindet.

```
from oled import OLED
```

Einzelheiten, auch dazu, erfahren Sie beim Durcharbeiten der MicroPython-Reihe.

Wie arbeitet das Programm

Nach dem Import der Hilfsgüter definiere ich, ob ich einen Testlauf durchführen möchte, oder ob es bereits Ernst ist. Die boolsche Variable **debug** erledigt das. Mit **debug = False** erkläre ich den Ernstfall. Mit **debug = True** fahre ich den Testmodus hoch. Dann geht es an die Einrichtung der nötigen Objekte.

```
debug=False

# ************** Objekte declarieren ******************
#
neo=0 # D3
neoPin=Pin(neo,Pin.OUT)
neoCnt=24
np = NeoPixel(neoPin, neoCnt) #

i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
d=OLED(i2c,heightw=32) # 128x32-Pixel-Display
d.clearAll()
d.writeAt("WILLKOMMEN ZUM",0,0)
d.writeAt("ADVENTS-KRANZ-",0,1)
d.writeAt("KALENDER",3,2)
sleep(3)
```

Das Objekt **np** wird über den Pin GPIO0 die 24 Neopixel-LEDs steuern. Klar, dass GPIO0 als Ausgang fungieren muss. Die Funktionsweise von Neopixel-LEDs habe ich in Bandit – Spiele mit dem ESP32 in MicroPython beschrieben.

Dann erzeuge ich eine I2C-Bus-Instanz und reiche diese an das Display-Objekt **d** weiter, das mit 32 Pixeln Anzeigenhöhe arbeitet. Die 128 Pixel Breite sind als Delfaultwert im Molul **OLED** festgelegt. Sie können sich das Modul in Thonny durch Doppelklick auf den Dateinamen in den Editor laden, um das Innenleben des Moduls zu studieren.

Ich lösche die Anzeige komplett und gebe eine Willkommensbotschaft aus. Danach schnarcht das Programm ganze 3 Sekunden.

Es folgt die Definition der "Kerzen". Damit sie auch im Verbund ansprechbar sind, fülle ich mit den Objekten eine Liste. Jetzt kann ich zum Beispiel alle "Kerzen" mittels for-Schleife löschen oder anmachen.

```
k1 = Pin(2,  Pin.OUT, value = 1) # D4
k2 = Pin(14, Pin.OUT, value = 1) # D5
k3 = Pin(12, Pin.OUT, value = 1) # D6
k4 = Pin(13, Pin.OUT, value = 1) # D7
kerze=[k1,k2,k3,k4] # Kerzenliste
```

Der Container **sonntage** nimmt die Daten der vier Adventsonntage auf und mit **timeZone** lege ich die Zeitzone fest, mit der ich die UTC (Koordinierte Weltzeit) eines NTP-Servers in die lokale Zeit (CET Central European Time) umrechne. Sonntage ist eine sogenannte Liste. Dieser sequenzielle Datentyp ist mutable. Das heißt, dass die Felder auch während des Programmlaufs veränderlich sind. Angesprochen werden die Felder durch ihre Platznummer, den sogenannten Index, der in eckigen Klammern an den Listennamen angehängt wird. Kerze[2] spricht also k3 an, denn die Indexzählung beginnt bei 0.

```
sonntage=[0,0,0,0]# Liste der Advents-Sonntage
timeZone=+1 # Zeitzone Berlin
```

Wenn kein Zeitserver erreichbar ist, weil der WLAN-Zugang fehlt, verwende ich das im ESP8266 eingebaute RTC-Modul (Real Time Clock). Deren Ganggenauigkeit lässt zwar zu wünschen übrig, aber bei unserer Zeitauflösung in Tagen ist das allemal OK. Während es der Zugriff auf einen Zeitserver erlaubt, unsere Schaltung zu einem beliebigen Zeitpunkt im Jahr einzuschalten, sie synchronisiert sich selbst, muss der Aufbau mit RTC-Zugriff zum richtigen Zeitpunkt gestartet werden. Gleiches gilt für den Testbetrieb. Die beiden 8-Tuples **tag** und **rtcTag** definieren den jeweiligen Startzeitpunkt. Die Felder sind wie folgt angeordnet.

tag:
(Jahr, Monat, Tag, Stunde, Minute, Sekunde, Wochentag, Tag im Jahr)

rtcTag:
(Jahr, Monat, Tag, Wochentag, Stunde, Minute, Sekunde, Millisekunden)

```
rtc=RTC()
tag=(2022,12,17,8,0,0,6,0)  # debug-Tagesstempel Start
rtcTag=(2022,11,27,6,8,0,0,0) # RTC-Tagesstempel Start
weekday=[
    "Montag",
    "Dienstag",
    "Mittwoch",
    "Donnerstag",
    "Freitag",
    "Samstag",
    "Sonntag"
    ]
syncTime=60000 # ms
refreshTime =20000 # ms
```

Damit die Wochentage im Klartext ausgegeben werden können, sind deren Namen in der Liste **weekday** zusammengefasst der Index läuft von 0 für Montag bis 6 für Sonntag, entsprechend dem Wert in den Tuples. Weekday[4] liefert also Freitag.

syncTime ist die Zeitspanne, nach der eine Synchronisation mit einem Timeserver per NTP durchgeführt wird, während **refreshTime** das Intervall für eine erneutes Aufbauen der Beleuchtung der Ringe und der "Kerzen"-LEDs festlegt.

Die Helligkeit der Neopixels steuere ich über die Variable **faktor**, die Farben legen die Tuples in der Liste **color** fest. Sie sehen, dass Listen nicht nur einfache Datentypen enthalten müssen. Jedes Farb-Tuple ist durch seinen Index ansprechbar.

```
faktor=0.3 # Helligkeits-Faktor
color=[(160,0,0), # red
       (120,40,0),
       (80,80,0), # yellow
       (40,120,0),
       (0,160,0), # green
       (0,120,40),
       (0,80,80), # cyan
       (0,40,120),
       (0,0,160), # blue
       (40,0,120),
       (80,0,80), # magenta
       (120,0,40),
      ]
```

Für den Zugriff auf einen NTP-Server brauchen wir das WLAN. Die Zugriffsparameter für die SSID und das Passwort richten sich nach den Vorgaben Ihres WLAN-Routers. Tragen Sie hier also unbedingt ihre eigenen Credentials ein. Die Portnummer ist fast frei wählbar und darf zwischen 1024 und 65535 liegen.

```
# **************WLAN-Zugriff definieren******************
#
mySSID="Here goes your SSID"
myPass="Here goes your password"
myPort=9009
```

Das Netzwerk-Interface (NIC) des ESP8266 gibt beim Verbindungsaufbau verschiedene Statusmeldungen zurück. Das Dictionary (kurz Dict) **connectStatus** übersetzt die Zahlencodes 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",
    }
```

Wie die Werte in einem Tuple sind die Schlüssel in einem Dict immutable, sie können nicht nachträglich geändert werden. Während Schlüssel zusätzlich einzigartig sein müssen, dürfen die Werte auch mehrfach vorkommen. Dicts werden durch geschweifte Klammern eingefasst. Die Schlüssel-Wert-Paare werden durch einen Doppelpunkt getrennt. Zwischen den Paaren steht ein Komma.

Ein Großteil der Programmarbeit wird durch Funktionen erledigt. Die Funktion **hexMac**() übersetzt ein, vom WLAN-Modul geliefertes Bytes-Objekt in eine menschenlesbare Zeichenkette. Dieser String besteht aus den üblichen Hexadezimalziffern 0-9 und A-F, durch welche die MAC-Adresse des Station-Interfaces des ESP8266 festgelegt ist.

Dieses Six-Pack muss dem WLAN-Router bekanntgemacht werden, damit er dem ESP8266 Zugang gewährt. Dazu ist die Adresse in die Liste der zugelassenen Geräte aufzunehmen. Die finden Sie in der Regel im Pflegemenü Ihres Routers unter dem Punkt WLAN – Sicherheit. Für das genaue Vorgehen ziehen Sie bitte das Handbuch Ihres Geräts zu Rate. Es ist übrigens aus Sicherheitserwägungen heraus keine gute Idee, allen sich anmeldenden Geräten, generell den Zugang zu erlauben, indem man die MAC-Filterung ausschaltet. Der nächste Hacker freut sich tierisch, wenn Sie ihm alle Türen offen halten.

Alle weiteren Funktionen beschäftigen sich mit der Beleuchtungssteuerung. So steuert **setPixel**() die LED mit der Nummer **num** mit dem Farbenmuster in **r, g** und **b** an, wobei der Faktor **f** die Helligkeit beeinflusst. Die finalen werte dürfen aber 255 nicht übersteigen.

```
def setPixel(num,r,g,b,f):
    r=int(r*f)
    g=int(g*f)
    b=int(b*f)
    np[num]=(r,g,b)
    np.write()
```

Die Funktionen **ringTest**() und **candleTest**() lassen eine Überprüfung des Gesundheitszustands der LEDs im Ring und der "Kerzen" zu. In beiden Funktionen werden alle Objekte der Gruppe mit einer for-Schleife durchlaufen. **pause** definiert die Verzögerung zwischen den LEDs in den Ringen. In MicroPython wird die Obergrenze eines Bereichs stets ausgeschlossen. Der Laufindex der for-Schleife nimmt daher Werte von 0 bis 23 an. Die Schleife wird somit 24 mal durchlaufen.

```
def ringTest(pause,faktor):
    for i in range(24):
        setPixel(i,160,160,160,faktor)
        sleep(pause)
    sleep(2)
    clearCalender()

def candleTest():
    for i in range(4):
        kerze[i].value(0)
        sleep(0.5)
    sleep(2)
    clearCandles()
```

Anders als Ostern und Pfingsten ist Weihnachten nicht durch einen Wochentag, sondern durch ein Monatsdatum festgelegt. Daher verschiebt sich von Jahr zu Jahr die kalendarische Position der Adventsonntage. Die Aufgabe der Funktion **adventSonntage**() ist es, die Tagesdaten aus dem Wochentag von Heilig Abend zu ermitteln.

```
def adventSonntage(jahr):
    global sonntage
    z=mktime((jahr,12,24,0,0,0,0,0))
    dateTime=localtime(z)
    wt=dateTime[6] # Wochentag des 24.12.
    if wt==6:
        sonntage=[3,10,17,24]
    else:
        s4=24-(wt+1)
        s1=s4-21 if wt <= 1 else (s4-21)+30
        sonntage=[s1,s4-14,s4-7,s4]
    # print (sonntage)
```

global sonntage
sorgt dafür, dass Änderungen an der Liste innerhalb der Funktion außerhalb verfügbar werden. Eine lokale Variable wie **z**, die innerhalb einer Funktion deklariert wird, ist außerhalb der Funktion nicht referenzierbar, nicht vorhanden. Sobald die Funktion verlassen wird kennt niemand mehr dieses **z**. Sehr wohl kann man innerhalb einer Funktion auf Werte von außerhalb jederzeit lesend zugreifen.

z=mktime((jahr,12,24,0,0,0,0,0))
Die Funktion **mktime**() ist eine Methode aus der Klasse **time**. Sie erzeugt aus dem 8-Tuple eines bestimmten Zeitpunkts einen Timestamp in Sekunden seit dem 01.01.2000, 00:00 Uhr. Mit dem Argument **jahr** ist damit der aktuelle Heilig Abend

dieses Jahres festgelegt. Aus dem Timestamp kann ich jetzt umgekehrt leicht den Wochentag des 24.12. ermitteln. Das macht

```
dateTime=localtime(z)
wt=dateTime[6]
```

Wenn **wt** den Wert **6** hat, ist die Suppe schon gegessen, dann ist der 24.12.ein Sonntag und der Advent findet sicher innerhalb des Monats Dezember statt. Das werden wir in 2023 haben.

```
if wt==6:
    sonntage=[3,10,17,24]
```

Andernfalls kann der erste Adventsonntag aber auch bereits im November liegen, wie das heuer (2022) der Fall ist. Wir müssen also herausfinden, welches Tagesdatum der letzte Sonntag vor Heilig Abend hat. Wir subtrahieren die um 1 erhöhte Nummer des Wochentags von 24. Heuer ist der 24.12. ein Samstag mit der Nummer 5. 24 – 6 = 18, der letzte Sonntag vor Weihnachten ist also der 18.12.

```
else:
    s4=24-(wt+1)
    s1=s4-21 if wt <= 1 else (s4-21)+30
    sonntage=[s1,s4-14,s4-7,s4]
```

Von da gehen wir 21 Tage = 3 Adventswochen zurück. Ist das Ergebnis positiv, dann ist der erste Advent sofort dingfest gemacht. Kommt dabei aber eine negative Zahl heraus, dann befinden wir uns im November. Wenn jetzt 30 addiert wird, haben wir den ersten Advent erwischt, es ist der 27.11.

Ahh, Moment mal, -21 + 30 = 9 und 18 + 9 = 27, dann könnte man ja gleich, s4 + 9 schreiben, statt (s4 – 21) +30. Ja das könnte man schon, arithmetisch stimmt das, aber das Vorgehen ist dann nicht so ohne Weiteres nachvollziehbar. Woher kommt die 9? Letztlich hat das etwas mit Zahlentheorie zu tun und zwar mit der Modulo-Rechnung. Würde ich nämlich eine analoge Überlegung vom Januar zurück in den Dezember durchführen, dann müsste ich mit dem Modul 31 statt mit dem Modul 30 rechnen, und dann stimmt die 9 eben nicht mehr.

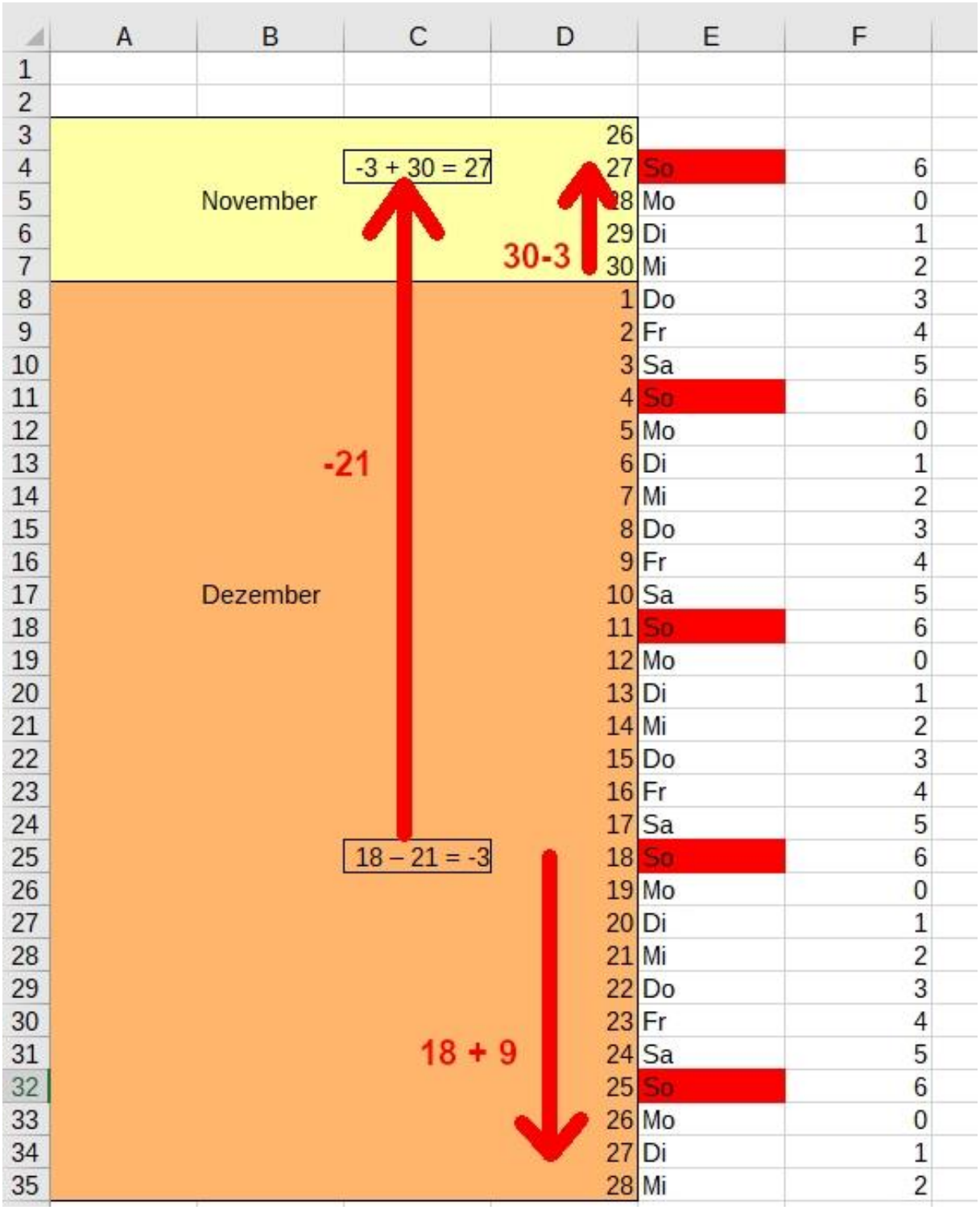

Abbildung 3: Adventsonntage

18 + 9 ist rein logisch der 27. Dezember, aber nicht der 27. November.

Jetzt geht es um das Anzünden der Kerzen. Der Funktion **lightCandles**() übergebe ich ein Standard-Datums-Tuple. Falls sie keines bekommt, dann holt sich die Funktion selbst eines von der Systemzeit. Hier muss ich die Zeitzone berücksichtigen. Mit der Jahreszahl im Feld **dateTime**[0] bestimme ich die Tagesdaten der Adventsonntage des aktuellen Jahres.

```
def lightCandles(dt=None):
    if dt is not None:
        dateTime=dt
    else:
        dateTime=localtime(time()+timeZone*3600)
    adventSonntage(dateTime[0])
    if (sonntage[0] <= dateTime[2] and dateTime[1] == 11) or\
       (sonntage[0] >=27 and dateTime[2]<=24 \
        and dateTime[1]==12):
        kerze[0].value(0)
        print(0,sonntage[0], dateTime[2])
    for i in range(1,4):
        if (sonntage[i] <= dateTime[2]) and dateTime[1]==12:
            print(i,sonntage[i], dateTime[2])
            kerze[i].value(0)
            kerze[0].value(0)
```

Liegt das aktuelle Tagesdatum auf oder nach dem des ersten Adventsonntags und befinden wir uns im November oder liegt der erste Advent im November und das aktuelle Tagesdatum liegt zwischen dem 1.12 und 24.12. inclusive, dann muss die erste Kerze brennen. Die Anweisung

kerze[0].value(0)

legt die Kathode der Kerze k1 auf GND-Potenzial, die LED geht an und beginnt zu flackern, wie das Kerzen eben machen.

Wenn das Tagesdatum auf oder nach dem zweiten, dritten oder vierten Adventsonntag liegt, muss auch die jeweilige Kerze angezündet werden, wenn es so weit ist. Sicherheitshalber machen wir auch k1 noch an. Die Liste **kerze** und die **for**-Schleife ersparen uns, dass wir eine ähnliche Sequenz dreimal codieren müssen.

Ähnlich wie die Testroutinen arbeiten die Funktionen, die die Ringe und "Kerzen" zum Erlöschen bringen.

```
def clearCandles():
    for i in range(4):
        kerze[i].value(1)

def clearCalender():
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
```

Bevor die Ringe ausgemacht werden, müssen sie erst einmal an sein. Das macht die Funktion **setCalender**(). Auch ihr wird in der Regel ein dateTime-Tuple übergeben.

```
def setCalender(dt=None):
    if dt is not None:
        dateTime=dt
    else:
        dateTime=localtime(time()+timeZone*3600)
    dom=dateTime[2]
    month=dateTime[1]
    if month==12 and dom <= 24:
        for tag in range(dom):
            r,g,b=color[tag % 12]
            setPixel(tag,r,g,b,faktor)
            sleep(0.5)
```

Das Monatsdatum wird extrahiert, und wenn wir uns im Dezember befinden, werden alle LEDs von der ersten bis zum aktuellen Tagesdatum illuminiert. Die for-Schleife erledigt das mit Hilfe der Funktion **setPixel**(). Hier wird auch der Helligkeitsfaktor, den wir eingangs definiert haben berücksichtigt. Damit wir die Farben einzeln genießen können, gibt's zwischendurch eine kleine lila Pause von 0,5 Sekunden. Das ganze Spektakel kann also bis zu 12 Sekunden dauern. Variieren Sie diesen Wert ganz nach Belieben, nur achten Sie darauf, dass die Gesamtdauer das Refresh-Intervall nicht übersteigt.

Damit für jeden Betriebsmodus der Abruf des Zeit-Tuples in korrekter Weise erfolgt, habe ich die Funktion **getDayTime**() eingeführt. Sie erkennt an den Statuswerten, was zu tun ist.

```
def getDayTime():
    if debug:
        return tag
        #print("debugging")
    if nicStatus != 4:
        return localtime(time()+timeZone*3600)
        #print("Local time")
    yr,mon,day,dow,hor,minute,sec,ms=rtc.datetime()
    #print("RTC-Time")
    return(yr,mon,day,hor,minute,sec,dow)
```

Wurde **debug** auf **True** gesetzt, dann sorgen die entsprechenden Sequenzen in der Hauptschleife selbst für die Zeitführung in **tag**.

Ist der **nicStatus** ungleich 4, dann besteht sehr wahrscheinlich eine WLAN-Verbindung, und die Systemzeit wird über NTP synchronisiert. Wir beziehen die Zeitzone mit ein und geben das standardisierte 8-Tuple zurück.

Ist weder das Eine noch das Andere der Fall, dann basiert die Zeitführung auf der RTC. Die Feldreihenfolge in deren Zeitstempel muss aber in das Standardformat umgebaut werden.

Der Goldflitter auf dem Adventskranz ist die Funktion **TimeOut**(), klein aber fein und voller Raffinesse. Die Funktion gibt keinen Wert zurück, sondern die Funktion **compare**(). Genau genommen wird nicht die Funktion, sondern eine Referenz darauf zurückgegeben. Ich habe weiter oben schon geschrieben, dass Objekte, die innerhalb einer Funktion definiert werden, außerhalb nicht sichtbar sind, sie sind lokal. Wird die Funktion verlassen, sterben alle lokal definierten Objekte. Das wird bei **TimeOut**() dadurch umgangen, dass die innerhalb definierte Funktion **compare**() auf den Parameter **t** und die außerhalb von **compare**() definierte, zu **TimeOut**() lokale Variable **start** zugreift. Aus der Funktion **TimeOut**() wird damit so eine sogenannte Closure.

Durch diesen Klimmzug ist es mir möglich, beliebig viele, einfach zu verwaltende Softwaretimer in meine Programme einzubauen. Jeder Timer arbeitet unabhängig von den anderen im Hintergrund, blockiert also den Programmablauf in keiner Weise. Erst beim Aufruf der Referenz auf die zurückgegebene Funktion erwacht diese aus ihrem Dornröschenschlaf und liefert die Information, ob der Timer abgelaufen ist oder nicht.

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

Der nächste Schritt ist die Einrichtung eines WLAN-Zugangs. Die Meldung im Display setzt uns darüber in Kenntnis.

```
d.clearAll()
d.writeAt("CONNECTING TO",1,0)
d.writeAt(mySSID,4,1)
sleep(3)
```

Zuerst drücke ich das AP-Interface bewusst aus dem Spiel, weil das im Zusammenhang mit dem ESP8266 gerne zu Irritationen führt. Das ist der zweite Schritt nach dem Abschalten von webREPL, das ich eingangs schon angesprochen habe.

```
nic = network.WLAN(network.AP_IF)  # AP-Interface-Objekt
nic.active(False)                  # sicher ausschalten
```

Dann schalte ich das Station-Interface (STA) ein und aktiviere es. Die Methode **config**() mit dem Argument **'mac'** (als String!) aufgerufen, liefert mir die MAC-Adresse, die ich mir von **hexMac**() in Klartext übersetzen lasse.

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

Ein ESP8266 hat zu diesem Zeitpunkt bereits automatisch eine WLAN-Verbindung aufgebaut, wenn er zuvor schon einmal mit diesem Accesspoint verbunden war. Falls nicht, dann versucht die folgende Sequenz eine Verbindung herzustellen. Dazu werden die Credentials **mySSID** und **myPass** benötigt.

```
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='')
    d.writeAt(line[0:n],0,2)
    sleep(1)
```

Das Spiel wird nun zwischen Accesspoint und ESP8266 ausgekartet, was ein paar Sekunden dauern kann. Solange dem ESP8266 vom DHCP des Routers noch keine IP zugeteilt wurde, zeigen Punkte im Terminal und im Display den Fortschritt der Verhandlungen an.

Dauert das Ganze länger als 10 Sekunden, dann kann der ESP8266 den Router wohl nicht erreichen oder bekommt von diesem keine Zutrittsgenehmigung. Haben Sie die MAC-Adresse beim Router eingetragen und die Credentials anfangs fehlerfrei eingesetzt?

Wir holen und merken uns den Status und geben den Zustand im Terminal und auf dem Display bekannt.

```
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='')

d.clearAll()
if nicStatus == 5: # got IP
    d.writeAt(STAconf[0],0,0)
    d.writeAt(STAconf[1],0,1)
    d.writeAt(STAconf[2],0,2)
else:
    d.writeAt("STATION CONNECT",0,0)
    d.writeAt("FAILED",4,1)
    d.writeAt("USING RTC",2,2)
sleep(3)
```

Der Timer für die Synchronisation von Datum und Uhrzeit wird gestellt, ebenso der für das Erneuern der Anzeige der Ringe und "Kerzen". Dann machen wir beide LED-Gruppen aus.

```
syncIt=TimeOut(syncTime) # Uhr-Synchronisierung
renew =TimeOut(refreshTime)    # Anzeige erneuern
clearCandles()
clearCalender()
```

Ich versuche, einen NTP-Server zu erreichen. Gelingt das, dann wird die Systemzeit damit synchronisiert.

```
try:
    ntptime.settime()
    print("Synchonized",localtime(time()+timeZone*3600))
```

Im anderen Fall stelle ich die Zeit der RTC auf den eingangs festgelegten Wert in **rtcTag**. Das Tuple sollte natürlich zutreffend den aktuellen Tag und die aktuelle Uhrzeit enthalten.

```
except:
    rtc.datetime(rtcTag)
    print("RTC-Time
set",rtc.datetime())#localtime(time()+timeZone*3600))
```

Nun hole ich mit **getDayTime**() die Zeit-Daten im Standard-Format ab und initialisiere damit Kalender und "Kerzen". Danach, Anzeige löschen.

```
dayTime=getDayTime()
setCalender(dayTime)
lightCandles(dayTime)
d.clearAll()
```

Wir betreten die Hauptschleife.

```
while 1:
    dayTime=getDayTime()
    if renew():
        clearCandles()
        clearCalender()
        setCalender(dayTime)
        lightCandles(dayTime)
        if debug:
            jahr=tag[0]
            day=tag[2]+1
            day= 1 if day == 31 else day
            sec=tag[5]+1
            hor=tag[3]
            mn =tag[4]
            month= 12 if 1 <= day <= 25 else 11
            dow=tag[6]
            tag=(2022,month,day,hor,mn,sec,dow,0)
            dayTime=tag
            print(tag)
        renew=TimeOut(refreshTime)
```

Erneut besorgen wir uns ein standardisiertes Zeit-Tuple.

Ist der Refresh-Timer abgelaufen? Über den Bezeichner **renew** rufe ich eigentlich die Funktion **compare**() auf, die **True** zurückgibt, wenn die in **t** an **TimeOut()** übergebene Zeit in Millisekunden überschritten wurde.

Dann müssen die "Kerzen" und die Ringe gelöscht und mit dem neuen Timestamp gesetzt werden.

Im Debug-Modus wird jetzt im Timestamp in **tag** das Tagesdatum erhöht. Wir berücksichtigen dabei einen eventuellen Monatswechsel. Abschließend stellen wir den Timer neu ein.

Auch der Synchronisationstimer wird abgefragt. Wenn der gemerkte **nicStatus** gleich **5** ist, besteht eine WLAN-Verbindung und der Systemtimer kann mit dem NTP-Server abgeglichen werden. Wenn's keine Verbindung gibt, gibt's auch nix zu tun. Synchronisationstimer neu stellen, und gut is.

Im Debugmodus muss jetzt der Sekundeneintrag in **tag** upgedatet werden.

```
    if debug:
        jahr=tag[0]
        mon=tag[1]
        day=tag[2]
        sec=tag[5]+1
        if sec == 60:
            sec = 0
            mn=tag[4]+1
        else:
            mn=tag[4]
        hor=tag[3]
        dow=tag[6]
        tag=(2022,mon,day,hor,mn,sec,dow,0)
        dayTime=tag
        print(tag)
```

Das Programm kann beendet werden, wenn der 25.12. erreicht ist. Na dann: Frohe Weihnachten, Veselé Vánoce, Merry Christmas, Feliz Navidad, Boldog Karácsonyt, Joyeux noël!

```
    if dayTime[2]==25 and dayTime[1] == 12:
        print("Programm beendet")
        d.clearAll()
        d.writeAt("FROHE",5,0)
        d.writeAt("WEIHNACHTEN",2,1)
        sleep(5)
        exit()
```

Bleibt noch, den aktuellen Wochentag, samt Datum und Uhrzeit am Display auszugeben. Dazu verwende ich Formatierungsstrings, die Tag, Monat und die Uhrzeit zweistellig, gegebenenfalls mit führender 0 ausgeben. Das Jahr bleibt vierstellig. Eine Sekunde Pause, dann auf zum nächsten Schleifendurchlauf.

```
    d.clearAll(False)
    d.writeAt(weekday[dayTime[6]],4,0, False)
d.writeAt("{:02}.{:02}.{:04}".format(dayTime[2],dayTime[1],\
                                dayTime[0]),3,1,False)
d.writeAt("{:02}:{:02}:{:02}".format(dayTime[3],dayTime[4],\
                                dayTime[5]),4,2, True)
sleep(1)
```

Funktionsprüfung

Natürlich können Sie nicht bis zum 25.12 warten, um herauszufinden, ob Schaltung und Programm funktionieren. Deswegen habe ich den Debug-Modus eingebaut. Sie können ein beliebiges Datum um die Adventszeit herum im Zeit-Tuple **tag** codieren und **debug** auf **True** setzen. Wenn dann das Programm startet, wird mit jedem neuen Refresh-Intervall das Tagesdatum hochgezählt, und Sie können überprüfen, ob die LEDs zur richtigen Zeit angehen.

Normalbetrieb

Für den Normalbetrieb ist die Synchronisation mit einem Zeitserver via WLAN die aller einfachste Variante. Sie können den Aufbau jederzeit ein- und ausschalten. Die Verbindung mit dem Zeitserver stellt innerhalb kurzer Zeit die tagesaktuelle Beleuchtung ein, immer am Puls der Weltzeit.

Steht kein WLAN zur Verfügung, dann gehen Sie ähnlich vor bei der Funktionsprüfung. Nur geben Sie einen ganzen Zeitstempel mit Jahr, Monat, Tag, Stunde, Minute, Sekunde, Wochentag und einer finalen 0 im Tuple **rtcTag** ein. Starten Sie dann umgehend das Programm. Mit einigen Sekunden Abweichung wird daraufhin die Anzeige im Display reagieren. Die Gangungenauigkeit der Real Time Clock kann natürlich zu weiteren Abweichungen von der Normalzeit führen. Mir liegen dazu im Moment noch keine weiteren Erkenntnisse vor. Ich habe auch den Einsatz einer externen RTC erwogen, aber in der Kürze der zur Verfügung stehenden Zeit bin ich auch da noch zu keinem Ergebnis gekommen. Nur eines steht fest, dass ein ESP8266 in Verbindung mit dem DS1302-BOB (Break Out Board) für diese Schaltung überfordert ist, was digitale Ein- und Ausgänge angeht. Alternativ käme höchstens ein DS3231 mit I2C-Interface in Frage. Der war aber auf die Schnelle nicht zur Hand.

Nun, vielleicht gibt es in nächster Zukunft einen Beitrag zum Thema RTC. Die Geschichte muss sich ja nicht unbedingt um den Advent drehen.

Übrigens, die Geschichte von Toni Lauerer habe ich auch im Internet gefunden. Wenn Sie dann Ihren Advents-Kranz-Kalender fertig gebaut, programmiert, festlich mit Tannenreisern geschmückt und angeworfen haben, stehen Ihnen genug ruhige Stunden bis Weihnachten zur Verfügung. Ich empfehle Ihnen, sich dann diese Geschichte reinzuziehen. Glauben Sie mir, es lohnt sich! Und was Ihnen mit dem Advents–Kranz–Kalender auf keinen Fall passieren kann:

Wenn die fünfte Kerze brennt, dann haben Sie Weihnachten verpennt!

Einen schönen Advent und viel Freude bei der Umsetzung des Projekts.