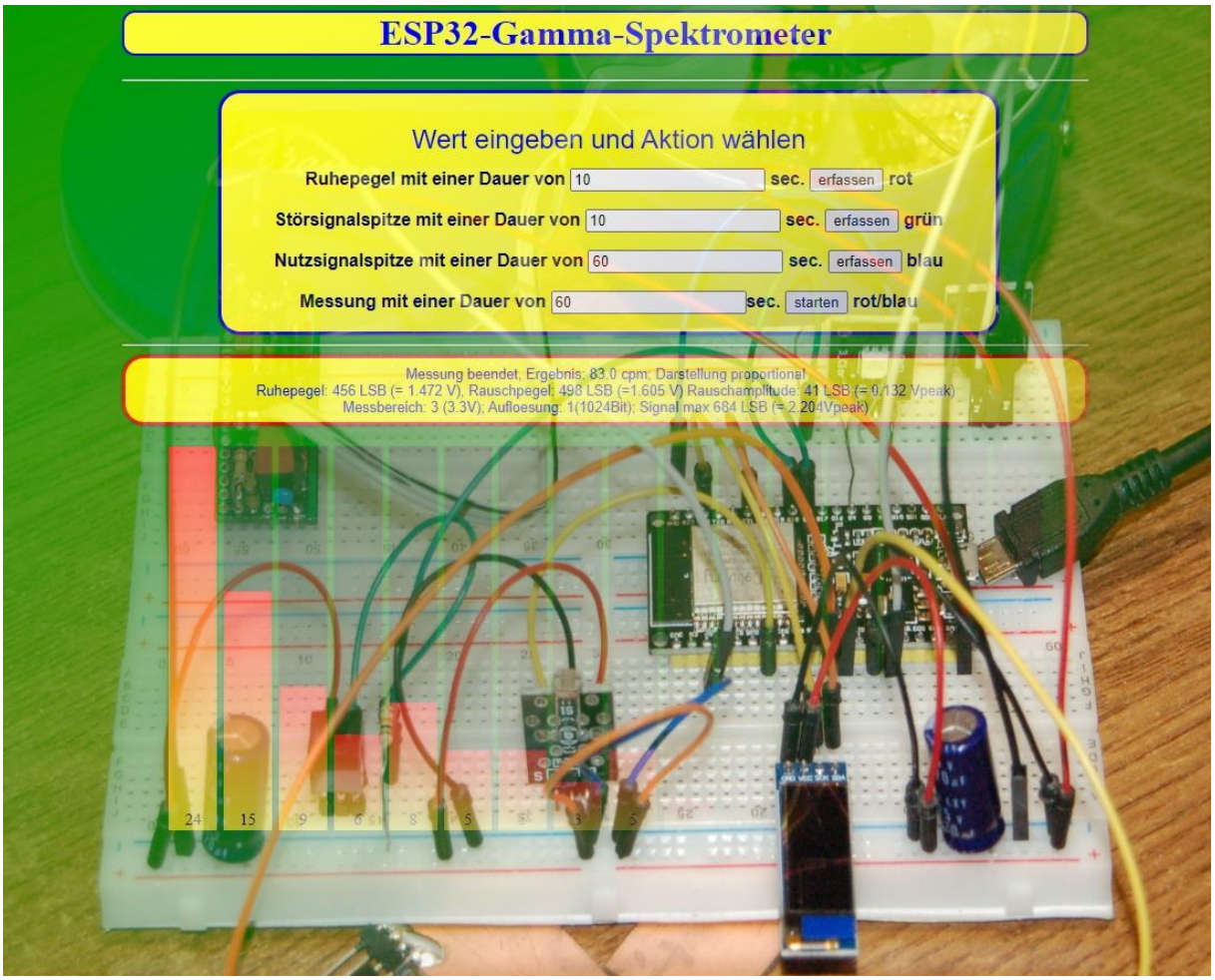


Micropython auf dem ESP32– Teil6

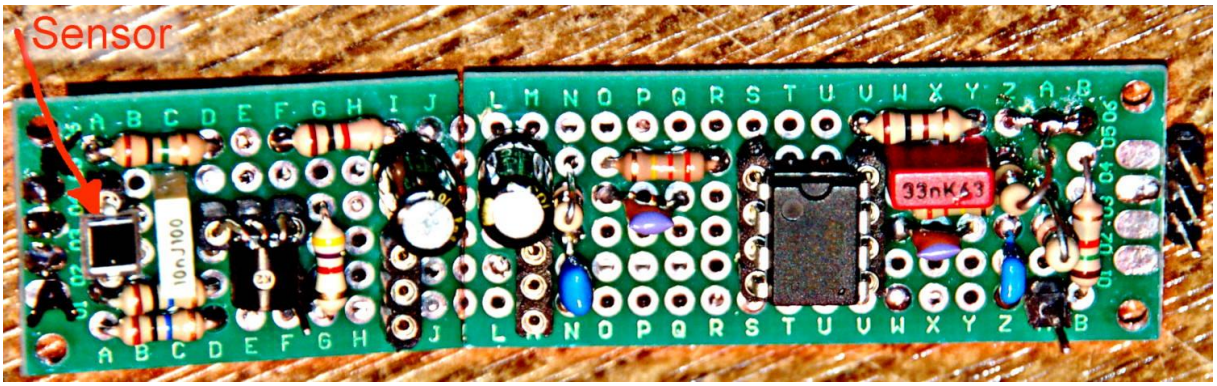
Heute verbinden wir alle Erkenntnisse der vergangenen Folgen dieses Blogs zu einem Gesamtprojekt. Der ESP32 wird gut ausgereizt. Er bedient LEDs, einen aktiven Buzzer, ein OLED-Display, zwei Touchpads und liest Zerfallsdaten von unserem Eigenbau-Strahlungsdetektor ein. Die Einstellung der drei wesentlichsten Parameter für diese Schaltung wird über ein Webinterface passieren, das schließlich auch die Ergebnisse der Messung grafisch darstellt. Gemanagt wird das alles durch den Webserver aus der 2. Folge des Blogs, der für diesen Zweck kannibalisch aufgemufft wird. Wie der ESP32 dann auch noch ohne ein lokales Funknetzwerk auskommt, zeige ich Ihnen am Ende der aktuellen Folge. Das Ganze wird natürlich autonom, auch was den Start angeht.

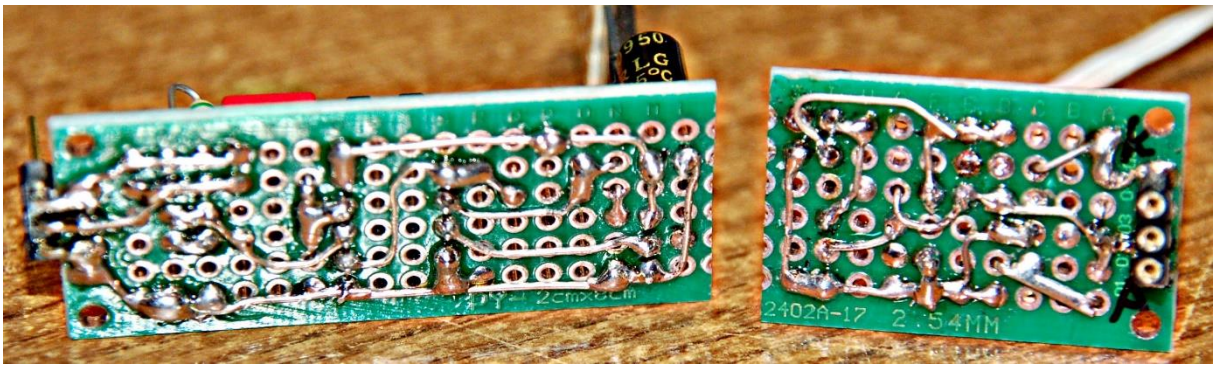
Erstaunlich, was in so einen kleinen Baustein alles reinpasst. Softwaremäßig würde das Programm auch in einem ESP8266 arbeiten, aber bei diesen Boards reicht der RAM-Speicher leider nicht annähernd. Damit ein herzliches Willkommen zum 6. Teil dieser Reihe. Wie bereits beim 5. Teil eingeführt, gibt es auch Teil 6 durch den Übersetzer von Onkel Google als PDF in englischer Sprache. Einen Link dazu finden Sie am Ende des Beitrags.

An Material benötigen Sie die folgenden Teile. Wenn Sie die vorangegangenen Folgen studiert haben, haben Sie wohl bereits das meiste davon.

1	ESP32 NodeMCU Module WLAN WiFi Development Board oder
1	ESP-32 Dev Kit C V4 oder
1	NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F mit CH340
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi oder
1	0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi
1	KY-012 Buzzer Modul aktiv
2	LED (Farbe egal) und
2	Widerstand 330 Ohm für LED oder
1	KY-011 Bi-Color LED Modul 5mm und
2	Widerstand 560 Ohm für LED oder
1	KY-009 RGB LED SMD Modul und
1	Widerstand 330 Ohm für blaue LED
1	Widerstand 2,2k Ohm für rote LED
1	Widerstand 3,9k Ohm für grüne LED
1	KY-004 Taster Modul oder
1	keypad-ttp224-1x4-kapazitiv
2	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
2	Blechstücke ca. 20 x 20 mm (nicht Aluminium!) oder Platinenreste
	einige Steckstifte 0,6x0,6x12mm

Die Beschreibungen für die Schaltungen des Kernstrahlungs-Sensors sind recht umfangreich. Daher verweise ich Sie wegen der benötigten Teile und dem Aufbau auf den Teil 5 des Blogs, wo alles genau beschrieben ist. Fertig bestückt sieht das Teil auf der Lochrasterplatine so aus.





In Folge 5 hatten wir bereits erste Messungen gemacht. Die Programme für die einzelnen Dienste wurden zunächst vom PC aus gestartet. In einer zweiten Stufe lief der ESP32/ESP8266 dann autonom und wurde über Tasten oder Touchpads gesteuert. Die Ergebnisse konnte man am OLED-Display ablesen. Weil die Darstellungsmöglichkeiten damit sehr beschränkt sind, erweitern wir heute das Projekt um einen Webserver.

Alte Liebe

Mit der wiederverwendeten Peripherie kommen auch die entsprechenden Treibermodule wieder zum Einsatz. An einigen Stellen wurde aber noch einmal, teils kräftig, daran herumgefeilt, um die Performance zu verbessern und um ein eigenes Beispiel für die Vererbung von Klassen anbieten zu können. Mit den Erläuterungen dazu fange ich an.

Vererbung von Klassen und Überschreiben

Import

Man kann in einem Programm eine Klasse importieren, um auf deren Namensraum über den Klassennamen oder eine Instanz der Klasse zugreifen zu können. Das kennen wir und haben es auch schon oft benutzt. Was passiert aber, wenn Sie für die Definition einer Klasse D eine Klasse A importieren wollen? Das Modul a.py mit der Klasse A haben wir hier, wir werden es später noch einmal benutzen. Geben Sie den Text ein oder laden Sie die Datei a.py herunter, Sie können sie auch schon mal auf den ESP32/ESP8266 hochladen.

Download: a.py

```
# modul a
modVarA = 50
modConstA = const(80)
#
class A:
    classVarA = 40
    classConstA = const(70)
    Summe=0
    #
    def __init__ (self, ap, bp=classVarA, cp=modVarA):
        self.aI=ap
        self.bI=bp
        self.cI=cp
        produkt = ap*bp
        print("Konstruktor A:", self.aI, self.bI, self.cI, A.classVarA, modVarA)

    def summe(self,x,y,z):
        self.Summe=x+y+z
        return self.Summe

    def summcv(self, x):
        return (A.classVarA + x)

    def produkt(self,x, y=classConstA):
        prod = x * y
        return prod
w=A(5000)
print(w.summcv(60),"\n\n")
```

Ein paar Tests der Klasse A:

>>> from a import A

Konstruktor A: 5000 40 50 40 50
100

Beim manuellen Import meldet sich bereits der Konstruktor, wir erkennen das an seiner Rückmeldung. Das liegt daran, dass in dem Modul a.py neben der Klassendefinition auch noch zwei Zeilen stehen, deren Inhalt sich bereits auf die Klasse bezieht. Es werden zwei Anweisungen ausgeführt. Die Klasseninstanz w wird declariert und das Ergebnis der Methode summcv(60) wird ausgegeben. 60 als Argument, addiert zum Wert der Klassenvariable classVarA von 40, ergibt 100. Das reicht als Test. Laden Sie spätestens jetzt a.py ins Device hoch. Danach erstellen oder laden Sie die Datei d.py und transferieren sie in den WorkSpace.

Die Klasse D baut auf Klasse A auf und importiert sie deshalb.

Download: d.py

```
# modul d importiert a.A
from a import A
modVarD=3.14
class D:
    def __init__(self):
        self.a=A(5)
        print("Konstruktor von D")

    def addStr(self,s1="Hallo",s2="da",s3="draußen."):
        greeting=" ".join([s1,s2,s3])
        return greeting


x=D()
y=x.addStr(s1="Ein",s2="einfacher",s3="Satz.")
print(y,"\n\n")
```

Konstruktor A: **5000** 40 50 40 50
100

Konstruktor A: **5** 40 50 40 50 (Achten Sie auf die **5** im Vergleich zur **5000**)
Konstruktor von D
Ein einfacher Satz.

Das ist die Ausgabe, wenn man d.py im Editor startet. Wir erkennen an den ersten beiden Zeilen, dass beim Import die Datei a.py komplett abgearbeitet wird. Die Attribute und Methoden werden registriert und die zwei Schlusszeilen werden ausgeführt.

Der Konstruktor von A erscheint ein zweites Mal, wenn im Konstruktor von D das Instanzattribut self.a instanziert wird. Der Konstruktor von D meldet sich und schließlich erscheint das Ergebnis der Testzeilen.

Es geht im Terminal händisch weiter.

```
>>> x
<D object at 3ffe5240>
>>> x.a
<A object at 3ffe5950>
>>>
>>> dir(x.a)   (Ausgabe verkürzt)
['classVarA', 'classConstA', 'Summe', 'aI', 'bI', 'cI', 'produkt', 'summe', 'summcv']
```

Also über das Instanzattribut a kann man auch auf dessen Attribute und Methoden zugreifen.

```
>>> x.a.produkt(5,90)
450
```

Nun das Spielchen können Sie fast beliebig weitertreiben, zumindest so lange wie der Speicher reicht. Schieben Sie d.py ins Device und erzeugen Sie e.py im Editorfenster oder laden Sie die Datei herunter. Starten Sie e.py im Editorfnester.

Download: e.py

```
# Modul e importiert D
from d import D

class E:
    classVarE=1000

    def __init__(self):
        self.q=D()
        print(self.q.addStr("Das","ist","erfreulich!"))
        print("Konstruktor von E\n")


    def divide(self,a,b):
        return a/b

s=E()
print(s.divide(8,2))
print(s.q.a)
print(s.q.a.produkt(5,9))
```

```
Konstruktor A: 5000 40 50 40 50
100


Konstruktor A: 5 40 50 40 50
Konstruktor von D
Ein einfacher Satz.


Konstruktor A: 5 40 50 40 50
Konstruktor von D
Das ist erfreulich!
Konstruktor von E
```

```
4.0
<A object at 3ffe6110>
45
```

Solange Sie eine geschlossene Kette von Instanzen der übergeordneten Klasse haben, haben Sie auch Zugriff auf die Methoden der obersten Klasse, in diesem Fall A. Aber leider werden die Aufrufe dafür immer umständlicher und unübersichtlicher. Das wollen wir abstellen, indem wir alles auf eine Eben legen. Das Zauberwort heißt Vererbung. Wie das geht, zeige ich an den Modulen a, b und c in Verbindung mit den entsprechenden Klassen. Danach wenden wir die neuen Erkenntnisse auf das aktuelle Projekt an und bauen die verwendeten Module so um, dass sie zum neuen Wissen passen.

Vererbung

Mit Modul a und Klasse A fangen wir an, das bleibt alles so wie es ist, Sie müssen die Datei also nicht erneut herunterladen – sofern Sie nichts daran verändert haben.

```
>>> from a import A
Konstruktor A: 5000 40 50 40 50
100


>>> x=A(3000)
Konstruktor A: 3000 40 50 40 50
>>> x.summe(3,4,5)
12
>>>
```

Diese Ausgabe können Sie jetzt bereits richtig einordnen. Eine Instanz w wurde mit dem Startwert 5000 für den Parameter ap des Konstruktors erzeugt und danach die Methode w.summvc() mit dem Argument 60 aufgerufen. Eine Instanz x erzeugen und die Methode x.summe() testen. Schön!

Holen wir uns als nächstes das Modul b mit der Klasse B dazu und laden sie ins Device hoch.

Download b.py

```
import a

class B(a.A):
    classVarB="Test"

    def __init__ (self, np, mp=classVarB):
        self.n=np
        self.m=mp
        super().__init__(150,250, cp=350)
        print("Konstruktor von B:",self.n, self.m, B.classVarB, "von A:",
B.classVarA)

    def diff(self, x,y):
        self.differenz=x-y
        return self.differenz

    def summe(self, d,f):
        print("call in B")
        sumB = d+f
        return sumB
```

```
>>> from b import B
Konstruktor A: 5000 40 50 40 50
100


>>> y=B(8000)
Konstruktor A: 150 250 350 40 50
Konstruktor von B: 8000 Test Test von A: 40
>>> y.diff(100,30)
70
```

Der Import von B auf REPL verläuft nach bisherigen Erkenntnissen erwartungsgemäß, die Erzeugung einer Instanz y der Klasse B ebenso. Aber jetzt schauen Sie mal.

```
>>> y.summe(10,20)
call in B
30
```

Hätten Sie jetzt nicht eine Fehlermeldung erwartet, dass ein Positionsparameter beim Aufruf von summe() fehlt? Und wieso 'call in B'? Wir hatten doch die Methode summe() mit drei Parametern in A definiert! Stimmt, aber schauen Sie doch mal die Zeile

class B(a.A):

genau an. Durch das Hinzufügen der Klammer samt Inhalt haben Sie MicroPython angewiesen, alle Informationen von der Klasse A zu übernehmen. Das ist auch geschehen. Klasse B hat von Klasse A alle Objekte geerbt.

```
>>> y.produkt(8,40)
320
```

Aber wir haben in B eine neue Methode summe() mit nur zwei Parametern definiert und die hat diese Definition hat summe() aus A überschrieben. Noch etwas müsste Ihnen jetzt aufgefallen sein, wenn Sie den Aufruf der Methode produkt() aus Klasse A mit dem aus dem letzten Abschnitt vergleichen.

```
>>> x.a.produkt(5,90)
450
```

Um die Methode produkt() jetzt zu erreichen, mussten wir nicht einen Schritt in der Hierarchie zurückgehen, sondern konnten die Methode direkt von der Instanz der Klasse B aus erreichen. Vererbung sorgt also dafür, dass der Namensraum des Erblassers in den des Erben übergeht. Das ist bequem, hat allerdings den Nachteil, dass Objekte aus der Klasse A, also dem Vorgänger, durch gleichnamige Objekte aus der Klasse B, dem Erben, überschrieben werden. Dieser Nachteil kann aber auch zum Vorteil werden, weil durch das Überschreiben von Objekten Dynamik ins System kommt, durch die man Brauchbares übernehmen, aber gleichzeitig auch Altes durch Neues ersetzen kann. Für diesen Zweck bietet Python im allgemeinen und MicroPython im Besonderen Mechanismen an, die diesen Ansatz noch viel mehr unterstützen. Dazu vielleicht an anderer Stelle mehr.

Lassen Sie uns das Ganze noch mit einer Klasse C aus dem Modul c.py deckeln. Dann gehen wir noch auf einen wichtigen Punkt bei der Vererbung ein. Vergessen Sie nicht, auch c.py zum Device zu schicken.

Download c.py

```
import b

class C(b.B):
    classVarC="3.14"

    def __init__ (self, np, mp=classVarC):
        self.v=np
        self.w=mp
        super().__init__(800,mp=300)
        print("Konstruktor von C:",self.v, self.w, C.classVarC, "von A:",
C.classVarA)

    def diff3(self, x,y):
        differenz=3*x-y
        return differenz

    def summe(self, d,f,g):
        print("call in C")
        sumC = d+f+g
        return sumC
```

```
>>> from c import C
Konstruktor A: 5000 40 50 40 50
100
```

```
>>> z=C(77,333)
Konstruktor A: 150 250 350 40 50
Konstruktor von B: 800 300 Test von A: 40
Konstruktor von C: 77 333 3.14 von A: 40
>>> z.diff3(10,20)
10
>>> z.diff(10,20)
-10
>>> z.summe(100,200,300)
in call C
600
>>>
```

Wir sehen, dass sich beim Import von C zunächst der Konstruktor von A meldet. Das liegt immer noch an den beiden letzten Zeilen in a.py. Dann erzeugen wir ein Klasse-C-Objekt. Sofort treten die Konstruktoren der Klassen A, B und C auf. Die Differenz-Methode aus C rechnet richtig, 3 * 10 -20 = 10, aber auch die Methode aus A ist über die Instanz z noch ganz einfach verfügbar. Nur die Summe aus B haben wir erneut mit jener aus C überschrieben.

Aber mal ganz was anderes. Woher haben denn die Konstruktoren von B und A zumindest ihre Positionsparameter? Die Klassen B und A mussten doch auch irgendwie initialisiert werden? Nun, das geschah in den Zeilen

```
    super().__init__(800,mp=300)
```

in C und

```
    super().__init__(150,250, cp=350)
```

in Klasse B. Vergleichen Sie die Parameter mit der Ausgabe der Konstruktormethoden beider Klassen. Mit der Funktion super() wird die jeweils übergeordnete Klasse angesprochen und mittels __init__() initialisiert.

```
>>> dir(z)
['w', 'n', 'diff', 'v', 'classVarC', 'classVarA', 'diff3', 'differenz', 'summe', 'classVarB', 'm',
'classConstA', 'Summe', 'aI', 'bI', 'cI', 'produkt', 'summcv']
```

Mit der dir()-Funktion können Sie sich jetzt davon überzeugen, dass wirklich alles das, was bislang declariert wurde, auch über die Instanz z von C erreichbar ist, über drei Ebenen hinweg und ohne Glimmzüge!

Mit diesem Wissen polieren wir jetzt die Klassen BEEP, TP und OLED auf. TP bekommt das meiste Fett ab, BEEP wird so abgerichtet, dass es zum neuen Gewand von TP, dessen geänderte Version fortan TPX heißt, passt und OLED wird nur ein wenig gestreift. Weil es den wenigsten Aufwand verursacht, beginnen wir mit OLED.

Renovierung guter Bekannter

Die **OLED**-Klasse hat in den folgenden Methoden eine rückwärtskompatible Änderung erfahren. Die schreibenden Methoden des Moduls oled.py, nämlich writeAt(), pillar(), xAxis(), yAxis() und clearFT(), haben in der Parameterliste einen zusätzlichen optionalen Parameter **show** mit dem Defaultwert True erhalten. Ich zeige das stellvertretend am Beispiel der Methode clearFT().

```
  def clearFT(self,x,y,xb=MaxCol,yb=MaxRow, show=True):
    xv = x * 8
    yv = y * 10
    if xb >= self.columns:
      xb = self.columns*8
    else:
      xb = (xb+1) *8
    if yb >= self.rows:
      yb = self.rows*10
    else:
      yb = (yb + 1)*10
    self.display.fill_rect(xv,yv,xb-xv,yb-yv,0)
    if show:
      self.display.show()
```

show = True sorgt dafür, dass wie bisher die Änderung am Inhalt des Framebuffers des Displays am Ende der Methode sofort zum Display gesendet wird. Wird show = False gesetzt, erfolgt nur die Änderung im Framebuffer. Zum Schluss muss dann eine der schreibenden Methoden mit show = True oder ohne Angabe dieses Attributs aufgerufen werden, damit die Änderung auf dem Display erscheint. Weil der Parameter show optional und der letzte in der Reihe ist, kann man ihn auch getrost weglassen. Die Syntax und die Funktion ist dann durch die Vorbelegung mit True genau die gleiche wie bei früheren Versionen. Das ist es, was der Begriff 'rückwärtskompatibel' ausdrückt.

Bei drei Ausgabezeilen ergibt sich daraus eine Geschwindigkeitssteigerung von ca. 225%. Sie können das selbst mit dem Testprogramm newoledtest.py ausprobieren, das auch gleich ein Beispiel für die Anwendung der neuen Syntax darstellt.

Die Klasse BEEP, als Signalklasse wird um vier Methoden für Lichtzeichen oder Tonzeichen erweitert. Die Methoden ledOn() und ledOff() steuern eine RGB-LED und ihrem Namen wird die Methode blink() gerecht. Mit setBuzzPin() kann nachträglich der Buzzerausgang gesetzt oder rückgesetzt werden. Ähnlich arbeitet die Methode setLedPin().

Download: beep.py

```
from machine import Pin, Timer
import os
from time import ticks_ms, sleep_ms

DAUER   =     const(5)

class BEEP:
  dauer   =     DAUER

  # The constructor takes the GPIO numbers of the assigned Pins
  def __init__(self, buzz=None, r=None, g=None, b=None, duration=dauer):
```

```
  self.buzzPin=(Pin(buzz, Pin.OUT) if buzz else None)
  self.tim=Timer(0)
  self.dauer = duration
  self.red   = (Pin(r,Pin.OUT) if r else None)   # LED-Ausgaenge
  self.green = (Pin(g,Pin.OUT) if g else None)
  self.blue  = (Pin(b,Pin.OUT) if b else None)
  if buzz: self.beepOff()
  print("constructor of BEEP")
  if buzz: print("Buzzer at:",buzz,"\nLEDs at: ")
  if r: print("red:{}".format(r))
  if g: print("green:{}".format(g))
  if b: print("blue:{}".format(b))
  print("Dauer={}ms".format(self.dauer))

  # -----------------------------------------------------------------
  # r,g,b is the RGB-Code which makes 7 colors possible
  # 1 means switch the LED on
def ledOn(self,r=1,g=0,b=0):
    if self.red:
        if r:self.red.on()
    if self.green:
        if g:self.green.on()
    if self.blue:
        if b:self.blue.on()
# -----------------------------------------------------------------

# r,g,b is the RGB-Code which makes 7 colors possible
# 1 means switch off the LED
def ledOff(self,r=1,g=1,b=1):
    if self.red:
        if  r:self.red.off()
    if self.green:
        if  g:self.green.off()
    if self.blue:
        if  b:self.blue.off()
# -----------------------------------------------------------------

# lights RGB-LED for dauer microseconds afterwards pauses
# for the same time if pause=None otherwise stays off pause ms
# r,g,b is the RGB-Code which makes 7 colors possible
def blink(self,r,g,b,dauer,pause=None,anzahl=1):
  runden = (anzahl if anzahl>=1 else 1)
  for i in range(runden):
      start = ticks_ms()
      current = start
      end = start+dauer
      self.ledOn(r,b,g)
      while current <= end:
        current=ticks_ms()
      self.ledOff()
      if pause:
          sleep_ms(pause)
      else:
          sleep_ms(dauer)

def beepOff(self):
  self.ledOff()
  if self.buzzPin: self.buzzPin.value(0)
  self.tim.deinit()

def beep(self, pulse=None, r=0, g=0, b=1):
  if pulse == None:
    tick = self.dauer
```

```
    else:
      tick = pulse
    if self.buzzPin: self.buzzPin.value(1)
    self.ledOn(r,g,b)
    self.tim.init(mode=Timer.ONE_SHOT,period=tick,callback=lambda t:
self.beepOff())

  def setDuration(self, duration=dauer):
    self.dauer=duration

  def getDuration(self):
    return self.dauer

  def setBuzzPin(self,buzz):
      if buzz:
          self.buzzPin=Pin(buzz,Pin.OUT)
      else:
          self.buzzPin=None

  def setLedPin(self,color,pin):
      if color in ["r","red","rot","rojo",]:
          self.red   = (Pin(pin,Pin.OUT) if pin else None)
      elif color in ["g","green","gruen","verde"]:
          self.green = (Pin(pin,Pin.OUT) if pin else None)
      elif color in ["b","blue","blau","azul"]:
          self.blue  = (Pin(pin,Pin.OUT) if pin else None)
      else:
          print("No valid color specified")
```

>>>from beep import BEEP
>>>b=BEEP(13,2,b=4,duration=100)

Der Konstruktor nimmt als optionale Argumente die GPIO-Nummern der Ausgänge für den Buzzer, die LEDS rot, grün und blau sowie die Dauer des beep-signals. Alle Platzhalter für IO-Pins sind mit None vorbelegt. Werden diese Argumente beim Aufruf nicht mit (zulässigen) Pinnummern überschrieben, dann behalten diese Argumente den Wert None. Das führt dazu, dass diese Farbe kein Pin belegt und beim Ein- und Ausschalten übergangen wird.

Das obige Beispiel setzt den Buzzer an GPIO13 aktiv, ebenso die rote LED an 2 und die blaue an 4. Für grün wird keine GPIO-Nummer angegeben. Diese Farbe kann später auch nicht angesteuert werden.

Die 3 LEDs der RGB-Einheit, es können auch einzelne HIGH-aktive LED-Typen sein, werden durch **ledOn**() geschaltet, sofern die Ausgänge aktiviert sind. Die Parameter sind optional, was eine Vorbelegung ermöglicht. Diese tritt in Kraft, wenn die Methode ohne Argumente aufgerufen wird. Die Vorgabe ist r=1, g=0, b=0. Das kann aber nach Belieben geändert werden. Der Aufruf

>>> ledOn()

schaltet demnach die am Ausgang für rot liegende LED ein. Der Zustand der blauen LED wird nicht verändert, die grüne LED ist nicht aktiviert und kann daher grundsätzlich nicht angesprochen werden. Beachten Sie bitte, die LEDs müssen unbedingt mit Vorwiderständen größer als 330 Ohm angeschlossen werden, um die

maximale Stromstärke, die die Pins liefern können, nicht zu überschreiten. Die meisten LEDs sind so hell, dass widerstände mit noch viel höheren Werten verwendet werden können. Bei RGB-LEDs ist es ferner sinnvoll, die Widerstandswerte so zu wählen, dass die drei Farben gleich hell erscheinen. Dann klappt das auch mit den Mischfarben

ledOn(1,1,0) – gelb
ledOn(0,1,1) - cyan
ledOn(1,0,1) magenta
ledOn(1,1,1) - weiß.

Die Anweisung

>>> ledOff(r,g,b)

funktioniert ähnlich. Mit einer 1 wird die Farbe gelöscht, mit 0 bleibt der bisherige Status unverändert. ledOff(1,1,1) ist die Defaulteinstellung für das Ausschalten der LEDs, und weil alle Parameter optional sind, macht ledOff() alle LEDs aus.

```
>>> from beep import BEEP
>>> b=BEEP(r=2,g=18,b=4)
constructor of BEEP
red:2
green:18
blue:4
Dauer=5ms
>>> b.ledOn(1,0,1)   ->magenta
>>> b.ledOff(0,0,1)   ->blau aus, rot bleibt an, grün war nicht an, bleibt also aus
```

blink(self,r,g,b,dauer,pause=None,anzahl=1)

Die Positionsparameter **r,g,b** folgen der Beschreibung von ledOn(). Mir **dauer** wird die Leuchtdauer angegeben, die gleich der Pausendauer ist, wenn für **pause** kein Wert übergeben wird. **anzahl** gibt an wie oft eine LED-Einstellung aufleuchten soll.

```
>>> b.blink(1,1,1,200,800,3)
```

Diese Anweisung lässt die LED im Mischlicht weiß dreimal für 200ms aufblinken. Die Periodendauer eines Blinkvorgangs ist 200+800=1000ms = 1s.

Die bereits vorher in BEEP enthaltenen Methoden haben ihre Funktionalität nach außen behalten und wurden nur intern an die neuen Einstellungen für LEDs und Buzzer angepasst. Aus diesem Grund bekam die Methode beep drei optionale Farbparameter dazu. Lässt man die Argumente beim Aufruf weg, blitzt per Default blau auf. Der Buzzer ertönt nur, wenn bei der Instanzierung für buzz die GPIO-Nummer eines Ausgangspins angegeben wurde. Mit einem Aufruf von dieser Art

>>>b.setBuzzPin(13)

kann der Buzzerausgang nachträglich aktiviert werden. Deaktivieren kann man ihn jederzeit durch

>>>b.setBuzzPin(None).

Eine ähnliche Methode existiert auch für das nachträgliche Aktivieren und Deaktivieren der LED-Ausgänge.

setLedPin(color,pin)

Das color-Argument ist ein String in der Form r, red, rot, rojo. Im Argument pin wird die GPIO-Nummer übergeben. Der Wert None deaktiviert den Ausgang

```
>>> from beep import BEEP
>>> c=BEEP()
constructor of BEEP
Dauer=5ms
```

Bisher ist noch kein Ausgang aktiviert.

```
>>> c.ledOn(1,0,0)
```
Keine Reaktion, denn der Rotkanal ist noch nicht aktiviert.

```
>>> c.setLedPin("red",12)
>>> c.ledOn(1,0,0)
```
Die rote LED leuchtet jetzt.

Die **Klasse TP** hat inzwischen mindestens den Status 3.0 erreicht, was Änderungen angeht. Von der einfachen Sammlung von Funktionen im Modul touch.py über die Klasse TP wird jetzt mit der Vererbung der bisher komplexeste Zustand erreicht. Aber der Reihe nach.

Die Module **touch**.py und **touch8266**.py haben zwei neue Methoden dazu bekommen. Die eine erlaubt es, den Grenzwert für das Erkennen von Berührungen einzustellen. Beim ESP32 ist hier eine Plausibilitätsprüfung integriert, sodass nur gültige Werte in die Variable threshold übernommen werden. Außerdem habe ich einen doc-String an den Anfang der Moduldefinition gesetzt, der einen Überblick über die Handhabung der Methoden des Moduls gibt.

Da die Methoden des Moduls touch8266.py rein digital arbeiten, habe ich wegen der Kompatibilität auch in touch.py die Rückgabewerte dahingehend abgeändert. Die getTouch-Methode gibt jetzt, wie die vergleichbare Methode in touch8266.py, True (oder 1) und False (oder 0) zurück und im Fall eines Einlesefehlers den Wert None. Die Methoden waitForTouch und waitForRelease geben als Antwort auf die gewünschte Aktion auch True zurück und bei Timeout None. Beispiele zur Anwendung finden Sie bei der Beschreibung der Funktionen zu wifi_connect2.py und im Anschluss an die Experimente zur Vererbung von Klassen.

Die Methode setThreshold existiert in touch8266.py aus Kompatibilitätsgründen zu touch.py hat aber nur eine Dummyfunktion wie der Wert threshold halt auch.

Die wesentlichste Neuerung kommt zum Schluss, weil die als Voraussetzung für eine weitere Funktionalität eingeführt werden musste. Der Konstruktor wurde amputiert. Seine Aufgabe ist jetzt nur noch, den Grenzwert zu setzen, der als Parameter optional übergeben wird. Natürlich macht er die restlichen Methoden für Instanzen von TP bekannt. Eine dieser Methoden, **initTP(GPIONummer)** ist die neu/alte Erzeugung von Touchpad-Objekten. Der Inhalt dieser Methode ist der Teil, den ich aus dem Konstruktor entfernt habe. Es wird geprüft, ob die übergebene GPIO-Nummer eine gültige Bezeichnung für einen Touchpadanschluss darstellt und die entsprechende Instanz schließlich zurückgegeben. Damit ist es jetzt möglich, innerhalb von TP mehrere Pads zu declarieren.

Download: touch.py

```
from machine import Pin, TouchPad
from time import time
"""
API fuer ESP32
TP([[grenzwert=]integerwert])  Touchpin GPIO, threshold(optional)
touchpad_t: initTP(int:GPIONumber)
bool: getTouch() # touched=True otherwise False, None if error occured
int:  waitForTouch(int: delay) waits delay sec for touch and returns
                               True or None if untouched till delay sec
                               delay = 0 means endless waiting
int:  waitForRelease(int: delay) waits delay sec for release and returns
                               True or None if still touched till delay sec
                               delay = 0 means endless waiting
void: setThreshold(int: grenzwert)    installs the in grenzwert given
                               integer as new threshold for method getTouch()
"""
class TP:
  # touch related values
  # Default-Grenzwert fuer Beruehrungsdetermination
  # ***********************************************
  Grenze = const(150)

  # touch related methods
  # ***********************************************
  def __init__(self, grenzwert=Grenze):
    print("Konstruktor von TP")
    gw=self.setThreshold(grenzwert)

  def initTP(self,pinNbr):
    touchliste = [15,2,0,4,13,12,14,27,33,32]                        #7
    if not pinNbr in touchliste:                                     #8
      print("{} ist keine gueltige GPIO-Nummer fuer Touchpins".format(pinNbr))
#9
      sys.exit()                                                     #10
    return TouchPad(Pin(pinNbr))

  # Liest den Touchwert ein und gibt ihn zur眉ck. Im Fehlerfall wird
  # None zurueckgegeben.
  def getTouch(self,tpin):
    # try to read touch pin
    try:
      tvalue = (tpin.read() < self.threshold)
    except ValueError:
      print("ValueError while reading touch_pin")
```

```
      tvalue = None
    return tvalue

  # delay = 0 wartet ewig und gibt gegf. True zurueck
  # delay <> 0 wartet delay Sekunden, wird bis dann kein Touch bemerkt,
  # wird None zurueckgegeben, sonst True
  def waitForTouch(self, pin, delay):
    start = time()
    end = (start + delay if delay > 0 else start+10)
    current = start
    while current < end:
      val = self.getTouch(pin)
      if (not val is None) and val :
        return val
      current = time()
      if delay==0:
        end=current+10
    return None

  # delay = 0 wartet ewig und gibt gegf. True zurueck
  # delay <> 0 wartet delay Sekunden, wird bis dann kein Release bemerkt,
  # wird None zurueckgegeben, sonst True
  def waitForRelease(self, pin, delay):
    start = time()
    end = (start + delay if delay >0 else start+10)
    current = start
    while current < end:
      val = self.getTouch(pin)
      if (not val is None) and not val:
        return not val
      current = time()
      if delay==0:
        end=current+10
    return None

  def setThreshold(self,grenzwert):
    gw = int(grenzwert)
    gw = (gw if gw >0 and gw <256 else 120)
    print("Als Grenzwert wird {} verwendet.".format(gw))
    self.threshold = gw
    return gw
```

Als Folge dieser Änderung bekamen die Parameterlisten von getTouch(), waitForTouch() und waitForRelease() Zuwachs, Es muss jetzt ein Touchpinobjekt mit übergeben werden, damit die Methode weiß, wen es zu überwachen gilt. TP ist dadurch leider nicht mehr down compatible, hat dafür aber an Vielseitigkeit gewonnen, weil Anwendungsprogramme jetzt dynamisch Touchobjekte erzeugen können. Übrigens der Befehl **del** löscht nicht mehr gebrauchte Objekte und gibt dadurch Speicher frei.

Die Innerei von touch8266.py wurde natürlich auch an die neue Situation angepasst.

Die geänderten Module touch.py und touch8266.py stehen zum Download und zur Untersuchung bereit. Wir werden uns jetzt im Zuge der Vererbung noch etwas damit beschäftigen.

Vererbung in der Anwendung

Problem: Eine Ja/Nein-Abfrage über Touchpads oder Tasten soll implementiert werden. Dazu sind logischerweise zwei Touch- oder Taster-Objekte nötig. Gut, dann erzeugen wir eben die zwei mit touch.py/touch8266.py und verwalten diese im neuen Programm. Das ist möglich, aber was ist, wenn Sie die gleiche Procedur in weiteren Programmen brauchen? Wäre es da nicht praktischer, das Ganze in eine Klasse einzubinden, die zusätzlich auch noch über die bisherigen Eigenschaften von TP verfügt? A, B, C, das klingt nach – ja richtig – Vererbung und das hatten wir erst vor Kurzem.

Wir erzeugen eine neue Klasse TPX in einem neuen Modul touchx.py, in der die Methoden von TP und die neue Methode jaNein() friedlich vereint sind, auf einer gemeinsamen Spielwiese. Und wenn wir schon dabei sind, dann holen wir uns auch noch die Klasse BEEP dazu – und OLED, denn bei Tastenaktionen sind auch Licht- und Tonsignale gut brauchbar und, naja – eine Textanzeige auf dem OLED-Display ist auch nicht verkehrt. "Ganz unabsichtlich" haben wir auch darauf geachtet, dass für touch.py und touch8266.py dieselben Methodennamen und Instanzattribute existieren. Damit besitzen die beiden Klassen dieselbe API. Ich möchte mit den zwei Varianten für den ESP32 und den ESP8266 aufräumen und beide Module im neuen touchx.py zusammenführen. Dazu ist es nötig, den Controller-Typ zu erfahren, das geht mit sys.platform(). Damit beginnen wir den Reigen. Nach diversen Imports erzeugen wir ein Displayobjekt d, importieren je nach Typ touch oder touch8266 und belegen die Vorgaben für die LED-Pins. Danach erzeugen wir das Signalobjekt b. Ein Übersetzer soll bei der Zuordnung der Pinbezeichner des ESP8266 helfen.

Download: touchx.py

```
#import touch
from machine import Pin
from beep import BEEP
from oled import OLED
d=OLED()
from time import time, sleep_ms
import sys
#
device = sys.platform
if device == 'esp32':
  from machine import Touchpad
  import touch
  redLed = 2
  greenLed = 18
  blueLed = 4
elif device == 'esp8266':
  import touch8266 as touch
  redLed = 12
  greenLed = 13
  blueLed = 15
else:
  print("Unbekannter Controller!")
  sys.exit()
b=BEEP(buzz=blueLed,r=redLed,g=greenLed,b=blueLed)

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

```
class TPX(touch.TP):
    SIGNAL=b
    DISPLAY=d
    JA=const(2)
    NEIN=(1)
    BEIDE=(3)
    TIMEOUT=(0)
    ja_nein="JA  <---->  NEIN"
    WhoAmI=device
    if WhoAmI == "esp8266":
        TPJA=16
        TPNEIN=14
    else:
        TPJA=27
        TPNEIN=14

    def __init__(self,tj=TPJA,tn=TPNEIN, sig=SIGNAL, disp=DISPLAY):
        super().__init__()
        self.tpJ=  self.initTP(tj)
        self.tpN=  self.initTP(tn)
        self.b=sig
        print("Construktor of TPX - built: {} {}".format(self.tpJ,self.tpN))

    # Weitere Touch-Objekte können zur Runtime mit obj.initTP(number)
    # erzeugt und in TP- sowie TPX-Methoden verwendet werden
    #-----------------------------------------------------------------

    # method jaNein()
    # Takes touchpad objects in tj and tn and the strings in meldung1
    # and meldung2 and writes them at the first 2 lines of the
    # OLED-Display on the I2C-Bus. Then waits for
    # laufZeit seconds for touch on the Pad objects tj or tn
    # See the above definition. If not noted in the parameter list
    # of the constructor, self.tpJ and self.tpN are used.
    def jaNein(self,tj=None,tn=None,meldung1="",meldung2=ja_nein,laufZeit=5):
        tpj = (tj if tj else self.tpJ)
        tpn = (tn if tn else self.tpN)
        d.clearAll()
        d.writeAt(meldung1,0,0,False)
        d.writeAt(meldung2,0,1,False)
        current = time()
        start = current
        end = (current+laufZeit if laufZeit else current+10)
        antwort=0
        self.b.ledOn(1,1,0)
        d.writeAt("Laufzeit {}s".format(end-current),0,2)
        while current <= end:
            ja=self.getTouch(tpj)
            nein=self.getTouch(tpn)
            if ja:antwort=2
            if nein:antwort+=1
            if antwort:
                b.ledOff()
                d.clearAll()
                return antwort
                break
            current = time()
            end=(end if laufZeit else current + 10)
            sleep_ms(200)
            d.writeAt("Laufzeit {}s".format(end-current),0,2)
        self.b.ledOff()
        return None
```

Die **Definition der Klasse TPX**, die von touch.TP erbt, ist die Stelle, an der sich touch.py und touch8266.py in gewisser Weise vereinen. Dank der Definition

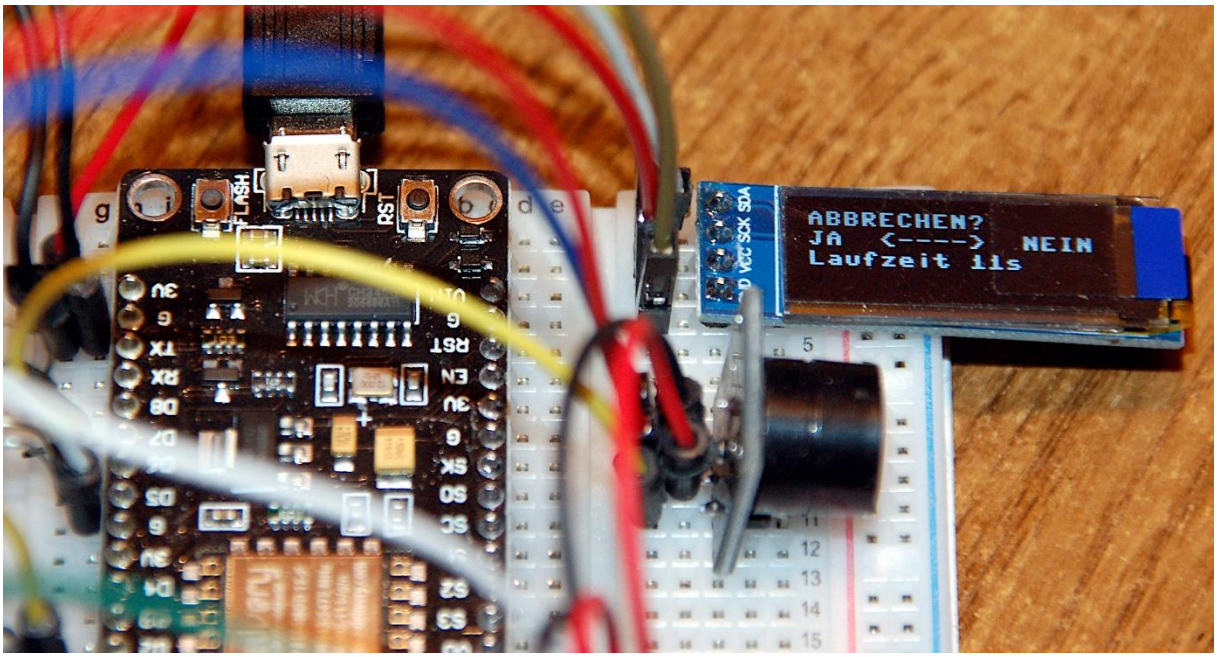
import touch8266 as touch

wird hier je nach Controllertyp immer mit dem richtigen Modul weiter gemacht. touch8266 wird jetzt auch unter dem Namen touch angesprochen. Die Methoden haben die gleichen Namen und die gleiche Parameterliste, es bedarf keiner weiten Unterscheidung, weil auch die Ergebnisse und Rückgaben die gleichen sind. Was innerhalb der Methoden passiert, interessiert ab hier niemand mehr.

Wir legen ein paar Klassenattribute fest und belegen je nach Typ die Eingabe-Pins für ja – nein vor, nur als Option!

Der Konstruktor greift die Optionen als Defaultwerte auf und leitet die Argumente an die Instanzattribute weiter. Bei der Instanzierung können natürlich alle Werte nach eigenem Gutdünken überschreiben werden. Aber denken Sie daran, auch wenn jetzt nur noch die Definition einer einzigen Methode folgt, mit dem Aufruf des Konstruktors haben Sie alle Methoden und Variablen aus der richtigen TP-Klasse zur Verfügung. Ferner stehen unter touchx.b und touchx.d die Signal- und die Displayklasse bereit, die sogar bei der Instanzierung vom aufrufenden Programm hierher weitergegeben werden können. Wir sehen dazu später ein Beispiel im Programm wifi_connect2.py.

Die Methode **jaNein**() nimmt 5 optionale Parameter. **tj** und **tn** sind Pinobjekte des TPX-Objekts oder ohne explizite Angabe die Pins tpJ oder tpN. **meldung1** und **meldung2** nehmen Strings, die in den ersten beiden Zeilen des OLED-Displays dargestellt werden. Nach **laufZeit** Sekunden gibt die Methode None zurück, wenn keine Taste gedrückt wurde. Während die Zeit läuft gibt die RGB-LED ein gelbes Signal und im Display läuft in Zeile 2 ein Countdown.



Wird die Ja-Taste gedrückt – der Pfeil weist darauf hin – gibt die Methode 2 zurück, bei Nein eine 1, beide Tasten/Pads gleichzeitig ergeben 3.
Der Rest besteht aus bekannten Bausteinen und sollte nicht mehr schwer zu entschlüsseln sein.

Netzanmeldung im neuen Gewand

Nach vielen Vorbereitungen nähern wir uns dem Ziel, Stufe 1, die Abteilung Anmeldung am Accesspoint. Zu den Modulen aus Folge 2 kommen drei neue hinzu, oled.py, beep.py und touchx.py. Sie helfen uns, den Controller unabhängig vom PC zu machen.

Download: wifi_connect2.py

```
# **********************Importgeschaeft**************************
import os,sys
from time import time,sleep, sleep_ms, ticks_ms
import network
import ubinascii
from machine import Pin, ADC
from oled import OLED
from beep import BEEP
from touchx import TPX
import esp
esp.osdebug(None)

import gc           # Platz fuer Variablen schaffen
gc.collect()

#********************Variablen deklarieren ************************
ja_nein="JA  <---->  NEIN"
# Die Dictionarystruktur (dict) erlaubt spaeter die Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN"
    }

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

# --------------------------------------------------------------------------
def zeige_ap_liste():
  """
  Scannt die Funkumgebung nach vorhandenen Accesspoints und liefert
  deren Kennung (SSID) sowie die Betriebsdaten zurueck. Nach entsprechender
  Aufbereitung werden die Daten im Terminalfenster ausgegeben.
  """
  # Gib eine Liste der umgebenden APs aus
  liste = nic.scan()
  sleep(1)
  autModus=["open", "WEP", "WPA-PSK", "WPA2-PSK", "WPA/WPA2-PSK"]
```

```
  for AP in liste:
    print("SSID: \t\t",(AP[0]).decode("utf-8"))
    print("MAC: \t\t",ubinascii.hexlify(AP[1],"-").decode("utf-8"))
    print("Kanal: \t\t",AP[2])
    print("Feldstaerke: \t\t",AP[3])
    print("Autentifizierung: \t",autModus[AP[4]])
    print("SSID ist \t\t",end='')
    if AP[5]:
      print("verborgen")
    else:
      print("sichtbar")
    print("")
    sleep(1)
# ------------------------------------------------------------------------
#
d=OLED()                # OLED-Display einrichten
d.clearAll()
d.name="wifi_connect"
#
WhoAmI = sys.platform   # Port ermitteln
if WhoAmI == 'esp32':
  redLed = 2
  greenLed = 18
  blueLed = 4
elif WhoAmI == 'esp8266':
  redLed = 12
  greenLed = 13
  blueLed = 15
else:
  print("Unbekannter Controller!")
  sys.exit()

b=BEEP(blueLed,redLed,greenLed,blueLed) # Buzzer und LEDs anmelden
b.name="wifi_connect"

# Taster/Touchpads aktivieren, Signal-Instanz und Displayobjekt übergeben
t=TPX(sig=b, disp=d)
#
# ************************* Bootsequenz ****************************
# Netzwerk-Interface-Instanz erzeugen und ESP32-Stationmodus aktivieren;
# moeglich sind network.STA_IF und network.AP_IF beide gleichzeitig,
# wie in LUA oder AT-based oder Adruino-IDE ist in MicroPython nicht moeglich
# ----------
# Create network interface instance and activate ESP32 station mode;
# network.STA_IF and network.AP_IF,both at the same time,
# as in LUA or AT-based or Adruino-IDE is not possible in MicroPython

nic = network.WLAN(network.STA_IF)  # Constructor erzeugt WiFi-Objekt nic
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')             # # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)                   # in eine Hexziffernfolge umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben
d.writeAt(myMac,0,0)
#
# Zeige mir verfuegbare APs
# zeige_ap_liste()
#sleep(3) # warten bis gesehen

# Verbindung mit AP im lokalen Netzwerk aufnehmen, falls noch nicht verbunden
# connect to LAN-AP
if not nic.isconnected():
  # Geben Sie hier Ihre eigenen Zugangsdaten an
```

```
  mySid = '<SSIDmeinesAP>; myPass = "PaSsWoRtMeInEsAp"
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySid, myPass)
  # warten bis die Verbindung zum Accesspoint steht
  print("connection status: ", nic.isconnected())
  while nic.status() != network.STAT_GOT_IP:
    #print(".",end='')
    #sleep(1)
    b.blink(1,0,0,500,anzahl=1) # blink red LED while not connected
# Wenn bereits verbunden, zeige Verbindungsstatus und Config-Daten
# print("\nVerbindungsstatus: ",connectStatus[nic.status()])
STAconf = nic.ifconfig()
# print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",STAconf[1],"\nSTA-
GATEWAY:\t",STAconf[2] ,sep='')
#
# Write connection data to OLED-Display
d.writeAt(STAconf[0],0,0)
d.writeAt(STAconf[1],0,1)
d.writeAt(STAconf[2],0,2)
sleep(3)

if t.jaNein(meldung1="ABBRECHEN?",laufZeit=5) != t.JA :
  # tpNein touched between 5 sec or untouched at all start server
  d.clearAll()
  exec(open('server3.py').read(),globals())
else: # falls das Pad an tpJa beruehrt wurde
  print("Die Bootsequenz wurde abgebrochen!")
  d.clearAll()
  d.writeAt("ABGEBROCHEN",0,0)
```

An der Datei hat sich seit der 2. Folge nicht besonders viel geändert. Die Änderungen und Ergänzungen stecken mehrheitlich in den verschiedenen Modulen, die wir inzwischen bearbeitet haben. Wir setzen sie jetzt in wifi_connect2.py ein. An der Verbindungsaufnahme selbst hat sich fast gar nichts geändert. Es gibt nun ein rotes Blinksignal von 1Hz, das angibt, dass noch keine Verbindung besteht. Vorher wurde dieser Zustand durch Punkte im Terminalfenster dargestellt.

Die Funktion zur Anzeige verfügbarer Accesspoints ist noch vorhanden, wird aber nicht benutzt, weil die Liste im Display nicht darstellbar ist. Sie können die Ausgabe zum Debuggen am Terminal ja wieder anzeigen lassen. Entkommentieren Sie dazu einfach die fett formatierten Zeilen.

Dann kommt die Stelle, an der Sie Ihre eigenen Zugangsdaten, Name des Accesspoints (aka SSID) und Ihr Passwort angeben müssen.

Print- und sleep-Befehl in der while-Schleife sind auskommentiert. Sie wurden durch die blink-Anweisung von einer Sekunde Gesamtdauer ersetzt. Beim ESP8266 blinkt die rote LED nur einmal ganz kurz auf, weil die Verbindung zum Accesspoint automatisch hergestellt wird, noch bevor irgendein anderer Befehl ausgeführt wird.

Die folgenden Printbefehle zur Statusmeldung wurden auch auskommentiert, sind aber für ein eventuelles Debugging nicht ganz entfernt worden. Die Statusmeldung erfolgt jetzt über das Display.

Zum Schluss kommt der Auftritt unserer jaNein()-Methode aus dem TPX-Modul. Fünf Sekunden wird auf eine Eingabeaktion gewartet. Mit einer Ja-Antwort bricht das Programm ab, Mit betätigen der nein-Auswahl oder ohne Aktion wird nachfolgend der Serverteil geladen und gestartet.

Der Server wird erwachsen

Die Übersicht

Der Server von Teil 2 hat uns, um niemand zu verschrecken, nur mit Grundfunktionalität beglückt. Das wird jetzt anders. Aus ursprünglich 56 Programmzeilen sind 352 geworden. Das Grundgerüst ist erhalten geblieben, hat aber an drei Stellen Zuwachs bekommen. Von hinten nach vorne: in der Serverschleife selbst wurde die Annahme einer Anfrage und das erste Parsen derselben um eine aufwendigere Darstellung der Webseite erweitert. Ein weiteres Drittel des Umfangs der Serverschleife nimmt die Darstellung des Messergebnisses in Anspruch.

mit ca. 70 Programmzeilen erreicht die Funktion web_page() fast den Umfang der Serverschleife. Sie nimmt den Teil der Browseranfrage, der für die Decodierung des über den Browser erteilten Auftrags wichtig ist. Für uns gültige Anfragen beginnen mit einem GET-Request. Etwas anderes kommt erst einmal gar nicht bis zum Decoder durch.

Weiter aufwärts folgen eine ganze Reihe Strings, welche den Inhalt der zu sendenden Webseite definieren und als nächstes folgen die Angaben des Stylesheets, das die Formatierung der Webseite übernimmt. Ferner steckt darin auch die Balkendefinition für die Ausgabe des Messergebnisses als Säulendiagramm.

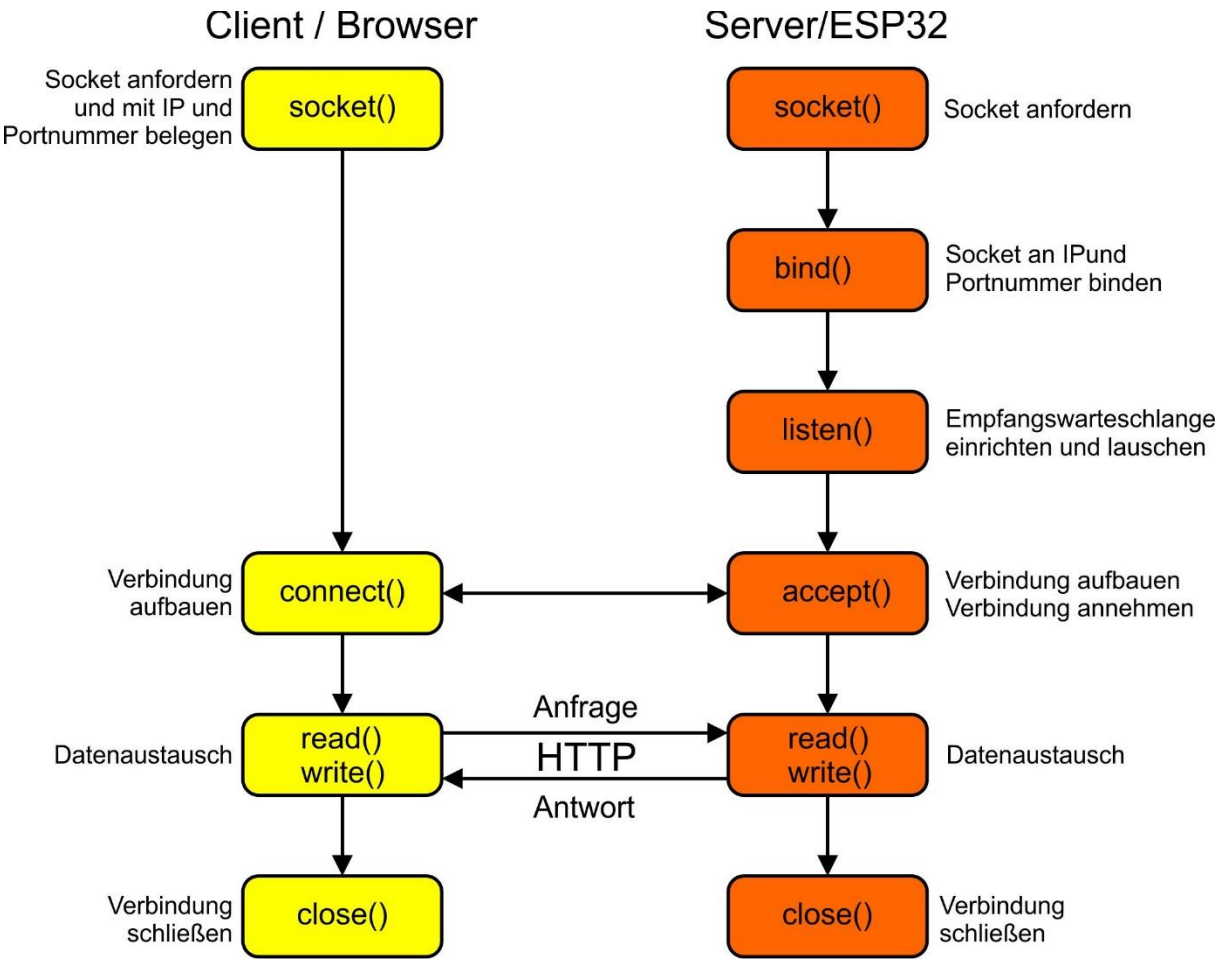
Ganz am Anfang steht, wie immer, eine Reihe von import-Anweisungen. Darunter befindet sich auch die wichtigste für dieses Projekt, die Klasse KS. Die Buchstaben stehen für **K**ern**s**trahlung, und das ist auch die Aufgabe der Methoden dieser Klasse, Kernstrahlung zu messen. Die anderen notwendigen Klassen wie BEEP, TPX, OLED und weitere übernimmt server.py vom aufrufenden wifi_connect2.py. Damit die ESPs einen Autostart hinlegen können, verpacken wir zum Schluss, wenn alles perfekt läuft, den Inhalt von wifi_connect.py in boot.py. Ab dann läuft das System autonom und kann von einem PC, Handy oder Tablet über den Browser ferngesteuert werden.

So arbeitet der Server

Download: server.py

Haben Sie bitte Verständnis dafür, dass ich den Programmtext des server-Files hier nicht als Text wiedergebe. Ich schlage vor, dass Sie sich die Datei herunterladen und wenn möglich, parallel zum Blog in einem eigenen Textfenster vorhalten, zum Beispiel mit Thonny oder einem anderen Editor, der Zeilennummern anzeigen kann.

Die Serverdefinition beginnt in Zeile 271. Der WiFi-Status wird am Display ausgegeben und eine Portnummer für den Anschluss festgelegt. Danach erzeugen wir ein Server-Socket-Objekt, binden die IP, die wir vom DHCP-Server des Accesspoints bekommen haben und die Portnummer an diesen Socket und gehen auf Lauschstation. Wenn wir keine IP-Adresse zugewiesen bekommen haben, könnten wir diese innerhalb der beiden Hochkommata selbst angeben. Im Display erscheint die Adresse, unter der der Server angesprochen werden kann. Mit while 1: beginnt die Serverschleife. Im nachfolgenden Diagramm befinden wir uns in der Box **listen()**.



Eine eintreffende Anfrage veranlasst den Serverprozess die listen-Schleife zu verlassen. Die accept()-Methode des Serversockets instanziert ein Connection-Objekt c und gibt ferner die IP des anfragenden Clients zurück. Der Serverprozess wird beendet und geht wieder auf Horchposten. Der Kommunikationssocket c übernimmt die Abwicklung der Anfrage und stellt schließlich als Antwort die Webseite zusammen. Wir lassen uns die Kontaktdaten am Display ausgeben.

Bevor es speicherintensiv wird, sammeln wir alle Ressourcen, die wir haben. Die mem_info zeigt uns das Ergebnis im Terminal. Wir empfangen den Byte-Stream vom Browser, wandeln ihn in ASCII-Code um und speichern den String in der Variable request. Die eingestreuten print-Anweisungen dienen dem Debugging und können auskommentiert werden, wenn alles läuft. Weil es sich beim Einlesen immerhin um bis zu 1024 Zeichen handeln kann, wende ich hier einen Trick an, der verhindert, dass jedes Mal dem Namen request neuer Speicherplatz zugewiesen werden muss. Am Programmbeginn habe ich gleich nach der ersten Müllsammlung den Speicherplatz für diese Variable fest reserviert (Zeile 43).

```
gc.collect()
request = bytearray(1024)
```

Am Anfang stehen die Chancen, 1024 zusammenhängende Bytes zu bekommen, jedenfalls besser als später mit fragmentiertem Heap. Wichtiger ist aber noch, dass durch dieses Festzementieren des Speichers später stets die eingelesenen Daten an diese Adresse geschrieben werden. Somit muss kein neuer Speicherbereich belegt werden. Das wirkt natürlich auch einer Fragmentierung des Heap entgegen.

Wie sieht eine gültige Anfrage vom Browser aus und wie wird sie geparst?

Getestete Browser sind Opera, Chrome und Edge. Firefox hat die völlig blödsinnige Eigenart, auf https-Verbindungen zu beharren, wodurch keine Verbindung zustande kommt.

Jetzt mal angenommen, wir würden diese Zeile als URL im Browser eingeben:

http://10.0.1.150:9192/?mtime=120&measurement=starten

Dann sehen wir diese Meldung im Serverdisplay:
Got a connection from ('10.0.1.10', 51279)

Und das "bisschen Text" schickt der Browser übers Netz, 556 Zeichen

```
GET /?mtime=120&measurement=starten HTTP/1.1
Host: 10.0.1.150:9192
Connection: keep-alive
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/87.0.4280.141 Safari/537.36
Accept:
text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,image/a
png,*/*;q=0.8,application/signed-exchange;v=b3;q=0.9
Referer: http://10.0.1.150:9192/?mtime=40&measurement=starten
Accept-Encoding: gzip, deflate
Accept-Language: de-DE,de;q=0.9,en-US;q=0.8,en;q=0.7
dnt: 1
```

Hier ist die Antwort unseres Parsers. Diese 30 Zeichen enthalten für uns wichtige Informationen. Den Rest kann man vergessen.

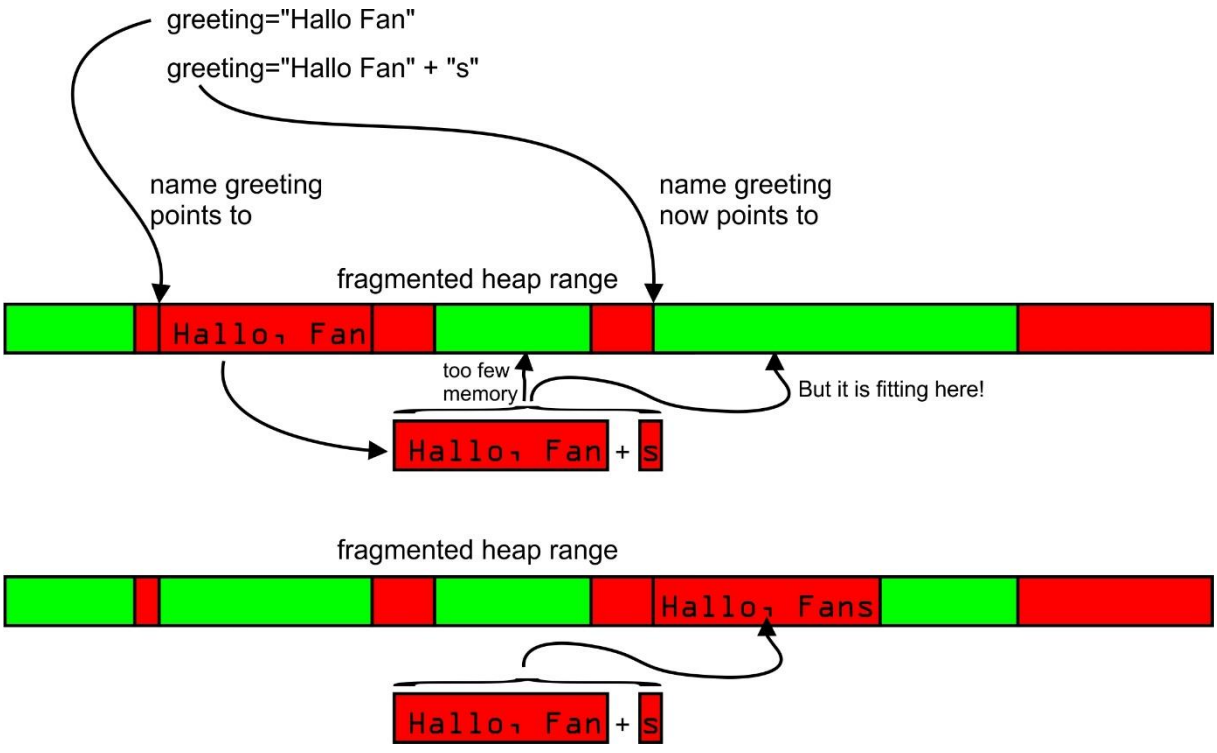
Aktion(30) ==> **?mtime=120&measurement=starten**

Die Anweisungen in den Zeilen 295, 296 und 298 filtern diesen Text heraus.
295: Ist es eine GET-Anfrage und ist das Rootverzeichnis des Servers,"/", angegeben? Merke dir die Position nach dem "/".
296: Suche ab hier nach einem " " (Leerzeichen, aka Blank), merke dir die Position)
298: In die Variable action kommen alle Zeichen nach den "/" bis zum Blank aber nach MicroPython-Manier ohne das Blank, das den Bereich begrenzt.

Der Text in action enthält die Befehle, die wir an den Server schicken und die dieser als nächstes decodieren muss, um dann die entsprechenden Aktionen einzuleiten. Bevor wir uns aber darum kümmern, sehen wir uns zuerst noch an, was mit den Ergebnissen der Aktionen passiert.

Wenn die Funktion web_page True zurückgibt, war das Decoding erfolgreich, und es liegt eine Rückmeldung vor, falls eine Messung durchgeführt wurde, gilt es auch noch, das Ergebnis darzustellen.

Die zu übertragende Webseite kann man sich als einen String von ca. 4000 Zeichen vorstellen. Als statische Seite könnte man diesen String am Stück über das Netzwerk senden. Strings sind in Python immutable, also unveränderbar. Wird ein String zum Beispiel zur Laufzeit um ein Zeichen verlängert, dann wird nicht etwa einfach das Zeichen im Speicher in die nächste Speicherstelle nach dem String geschrieben, sondern es wird ein komplett neuer String an einer anderen Speicherposition angelegt. Bei umfangreichen Zeichenketten geht das schnell an die Grenzen des RAM-Speichers, zumindest fördert es die Fragmentierung.



Deshalb habe ich den Webseitentext in kleinere Teile mit konstantem Inhalt gehäckselt. Getrennt habe ich überall dort, wo dynamisch Texte ausgetauscht werden müssen oder wo Zahlenwerte einzufügen sind. Weil auf diese Weise keine Stringoperationen durchgeführt werden müssen, hält sich auch der Speicherbedarf in Grenzen.

Was für den Text der Webseite gilt, gilt auch für die Erzeugung des Balkendiagramms aus der Liste **spektrum** der Messwerte. Die konstanten Teile werden durch ein Stylesheet definiert. Dazwischen werden im if-Konstrukt ab Zeile 317 dann die Zahlenwerte eingestreut.

Woher weiß der ESP32 aber, welche Aktion er ausführen muss? Wir sagen es ihm in der Funktion web_page() ab Zeile 197.

Die Funktion muss die globalen Variablen aenderung und ergebnis mit Werten versehen, also müssen wir sie als global kennzeichnen. (199,200)

Leere Anfragen und solche mit favicon.ico als Inhalt werden nicht weiter geparst. Von dem, was übrig bleibt gehen nur Anfragen durch, die eines der Schlüsselwörter enthalten. Woher diese Begriffe kommen, erkläre ich später. Der Parameter act kann verschiedene Anfragen enthalten. Eine Form haben wir vor kurzem gesehen.

?mtime=120&measurement=starten

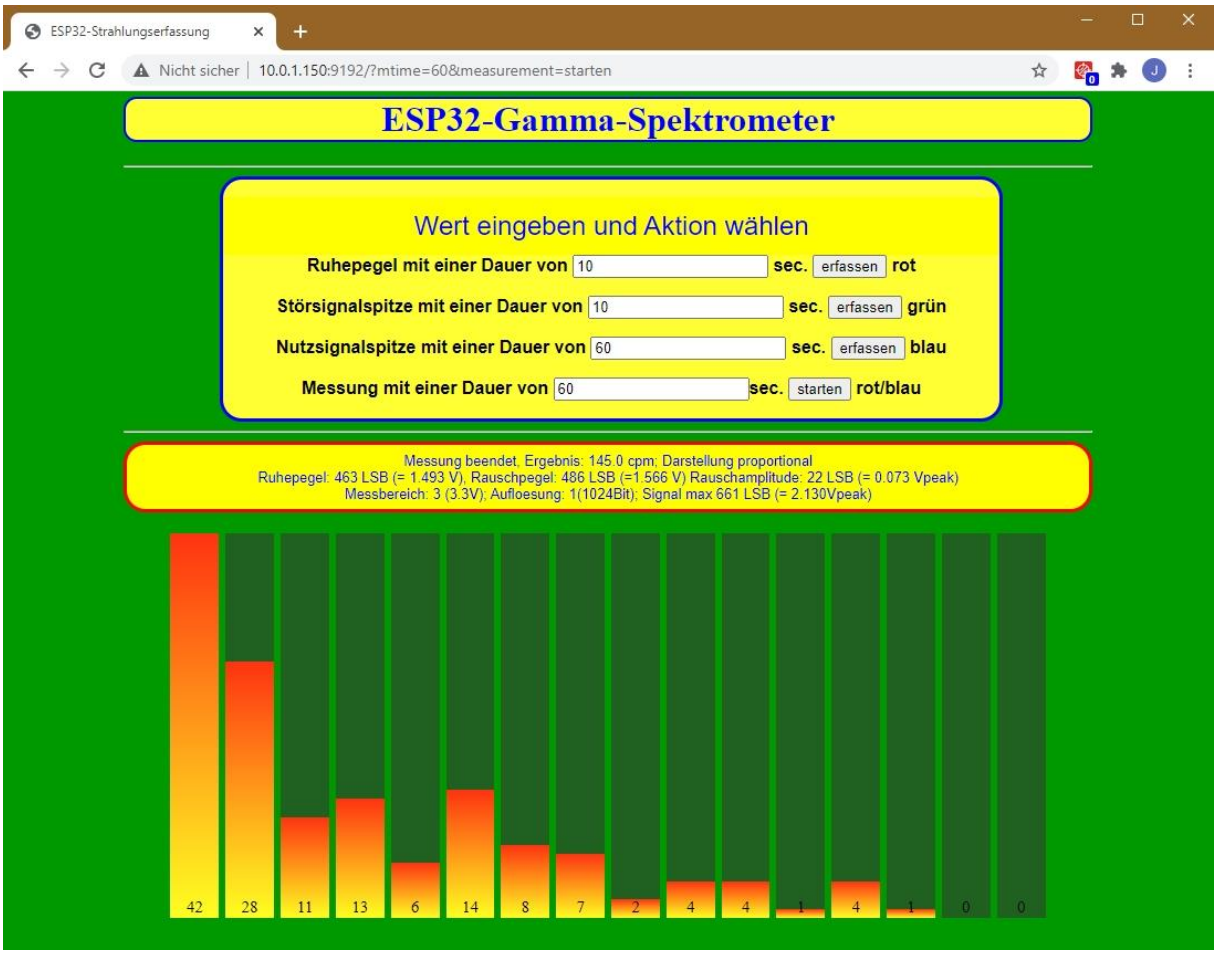
Es gibt aber auch noch eine kurze Form, etwa so.

?start

Bei der ersten Form interessieren wir uns für den Teil bis zum "&". Bei der zweiten Form gibt es kein "&", deshalb müssen diese Fälle anders geparst werden. In jedem Fall muss das führende "?" weg. Das macht Zeile 206. Wenn dann bei der Suche nach einem "&" -1 herauskommt (nicht gefunden), dann reicht es, sich das Wort nach dem Fragezeichen zu merken. Andernfalls teilen wir den Inhalt von act bis zum "&" exklusive in Name und Wert.

Der Rest ist einfach und wiederholt sich für die verschiedenen Befehle, die in entsprechende Aktionen umgesetzt werden, welche die Methoden der Klasse KS auszuführen haben. Die Namen dieser Methoden kennen wir bereits von der Folge 5, wo dieser Art Messungen bereits durchgeführt wurden. Jetzt können wir vom Browser aus die Messdauer verändern und die einzelnen Aktionen per Mausklick starten.

Nachdem die Aktion abgeschlossen ist, geht in **aenderung** ein String zurück an die Serverschleife und falls eine komplette Messung durchgeführt wurde, wird **ergebnis** auf 1 gesetzt. Die Serverschleife weiß jetzt, dass das Resultat der Messung als Balkendiagramm dargestellt werden muss. Die Abbildung zeigt diesen Status.



Bleiben noch zwei Fragen:

1. Wie ist die Webseite codiert?
2. Wer macht die Messarbeit?

Das Webinterface und CSS

Den grundlegenden Aufbau einer HTML-Seite habe ich bereits in der zweiten Folge des Blogs beschrieben. Wir erweitern das jetzt durch den Einsatz von Forms und CSS. Eine Form ist ein HTML-Element,das durch Textfelder, Kontrollkästchen, Optionbuttons und Aktionsschaltflächen, um nur einige zu nennen, eine Webseite interaktiv macht. CSS steht für "Cascading Style Sheets", eine Technik, die eine komfortable Formatierung einer Webseite erlaubt.

Das Webinterface wird aus vier Formtags zusammengebaut, das sind die vier Zeilen mit den Eingabefeldern im oberen Rahmen. Dazu kommt das rot eingerahmte Meldungsfenster. Das Säulendiagramm darunter wird nur nach einer Messung angezeigt.

Ich habe den Baustein für eine Form aus dem Programmkontext herausgeschnitten und den MicroPython-Code entfernt, um daran die Wirkungsweise im HTML-Dokument zu erklären. Es handelt sich um die Zeilen 140 bis 150.

```
<form method="get" action="http://10.0.1.150:9192/">
  <div align="center" bgcolor="#009900"><b><font face="Arial, Helvetica, sans-
serif">Ruhepegel mit einer Dauer von
  <input type="text" name="groundtime" value=
HIER STEHT DER ZAHLENWERT FÜR DIE VARIABLE GROUNDTIIME
> sec.
    <input type="submit" name="getgroundlevel" value="erfassen"> rot
    </font></b></div>
</form>
```

Der HTML-Code definiert ein Formkonstrukt mit einer Texteingabezeile und einem Submit-Button. Der Inhalt dieser Felder wird beim Klicken auf den Submit-Button unter Angabe von GET als Anfragemethode an die Adresse http://10.0.1.150:9192 gesendet. Der Browser bastelt daraus die Anfrage, deren Syntax wir schon kennen.

http://10.0.1.150:9192/?groundtime=10&getgroundlevel=erfassen

Wie das am ESP32 ankommt, wissen wir ja auch bereits. Wie kann der Browser aber einen einfachen Text grafisch ansprechend aufbereiten? Selbst wenn man jeden Absatz mit entsprechenden Formatierungstags versehen würde, käme nicht das heraus, was Sie in der Darstellung oben sehen. Das Stichwort dazu lautet Cascading Stylesheets oder kurz CSS. Mit diesem Hilfsmittel kann man zwei Dinge ganz einfach erledigen. Man kann zum einen sehr effektiv den Text einer ganzen Site einheitlich und übersichtlich formatieren und, falls erforderlich zentral von einer Stelle aus, dokumentübergreifend ändern. CSS lassen sich nämlich in eigenen Dateien mit der Endung .css ablegen und machen so Eingriffe in einzelne HTML-Dokumente überflüssig. So gesehen sind CSS für Webseiten das, was Klassen in MicroPython für Programmdateien darstellen. Darüber hinaus bieten CSS zum Beispiel auch die Möglichkeit, grafische Elemente in eine Webseite zu integrieren, wie ich es hier mit den Rahmen und dem Säulendiagramm getan habe.

Die Definition eines CSS befindet sich der Sektion zwischen <head> und </head> und wird in die Tags <style> und </style> eingeschlossen. In diesem Rahmen finden Sie in den Zeilen 54 bis 124 die Klassendefinitionen für die Textformate und die Progress-Bars, die ich als Säulen in meinem Diagramm missbrauche. Neben den Klassendefinitionen, die mit einem "." eingeleitet werden, arbeiten zwei der Definitionen, body und h1, elementbezogen, das heißt, sie sind auf HTML-Tags ausgerichtet und immer dann aktiv, wenn das entsprechende Tag im HTML-Text auftaucht wie in Zeile 135 die h1-Überschrift.

Die CSS-Klassen werden in den öffnenden Tags für Abschnitte wie und <p> mit dem Schlüsselwort "class" aktiviert. Beispiele zur Textformatierung sehen Sie in den Zeilen 137 und 138. Den Einsatz für die Formatierung der Säulen für das Diagramm finden Sie in den Zeilen 126, 128 und 130. Um ein Gefühl für die Wirkung der diversen Vorgaben zu bekommen schlage ich vor, einzelne Zuweisungen zu verändern und die Auswirkungen zu studieren. Das lohnt besonders bei den Progress-Bars, bei den Textformaten sind die Namen und Zuweisungen offensichtlicher. Während die Feldnamen ähnlich wie die reservierten Wörter in MicroPython festgelegt sind, können die Klassennamen selbst gewählt werden.

Die Erzeugung und Formatierung der Progressbars zeige ich an einem kompakten Beispiel. Anschließend teilen wir den Beispieltext so auf, dass dynamisch Werte eingefügt werden können. Das Tag <style> leitet die Definition eines Stylesheets ein, durch </style > wird die Definition abgeschlossen. Wir gehen vom Groben zum Feinen. Der erste Absatz definiert die Hintergrundfarbe eines Balkens.

Im zweiten Absatz legen wir fest, dass es ein vertikaler Balken werden soll. Breite und Höhe in Prozent der Arbeitsfläche werden festgelegt. Diese wird durch das <body>-Tag bereits auf 80% der Fensterbreite reduziert.

Im dritten Absatz weisen wir dem Fortschrittsbalken die gesamte Breite des Hintergrunds zu und lassen mit absolute die Säulen von unten nach oben wachsen.

Im vierten Absatz ergänzen wir die Klasse progress-bar mit der Füllung, die einen Farbverlauf von gelb von unten (0%) bis rot (oben=100%) bekommen soll.

Die Klasse tagging beschreibt schließlich, wie die Beschriftung der Balken aussehen soll.

```
  <style>

    .progress { background-color: #206020; }

    .progress.vertical {
      position: relative;
      width: 5%;
      height: 50%;
      display: inline-block;
      margin: 1px;
    }

    .progress.vertical > .progress-bar {
      width: 100% !important;
      position: absolute;
        bottom: 0;
```

```
    }

    .progress-bar { background: linear-gradient(to top, #ffff22 0%, #ff3311
100%); }

    .tagging {
      position: absolute;
      font-size: 15;
      top: 93%;
      left: 50%;
      transform: translate(-50%, -50%);
      z-index: 5;
    }

    body {
      padding: 5px;
      margin: auto;
      width: 80%;
      text-align: center;
        background-color: #009900
    }
</style>
…
…
…
<body>
…
<div class="progress vertical">
    <p class="tagging">36</p>
    <div role="progressbar" style="height:36%;" class="progress-bar">
    </div>
</div>
```

Im <body>-Bereich erzeugt die mittlere der letzten 5 Zeilen den Balken Die Formatierung erfolgt durch die Angabe der Style-Klassen. Der String für diesen div-Bereich muss dort aufgetrennt werden, wo im Ernstfall dynamisch Zahlenwerte eingefügt werden müssen. Das ist hier die 36 und 36%. Genau diese Aufteilung geschieht in den Zeilen 126,128 uns 130.

Die Klasse KS enthält die Methoden und Attribute aus der 5. Blogfolge, die zur Durchführung von Messungen mit unserem Kernstrahlungssensor nötig sind. Sie sind dort ausführlich beschrieben. Damit dabei Licht- und Tonsignale abgegeben werden können, nimmt der KS-Konstruktor ein BEEP-Objekt und eine OLED-Instanz sowie optional die Nummer des Analogeingangs. Der Defaultwert ist GPIO34. Das Programm server.py übernimmt von wifi_connect2.py das BEEP-Objekt b und das Display-Objekt d. Wir geben beide einfach als Parameter an die KS-Instanz k weiter.

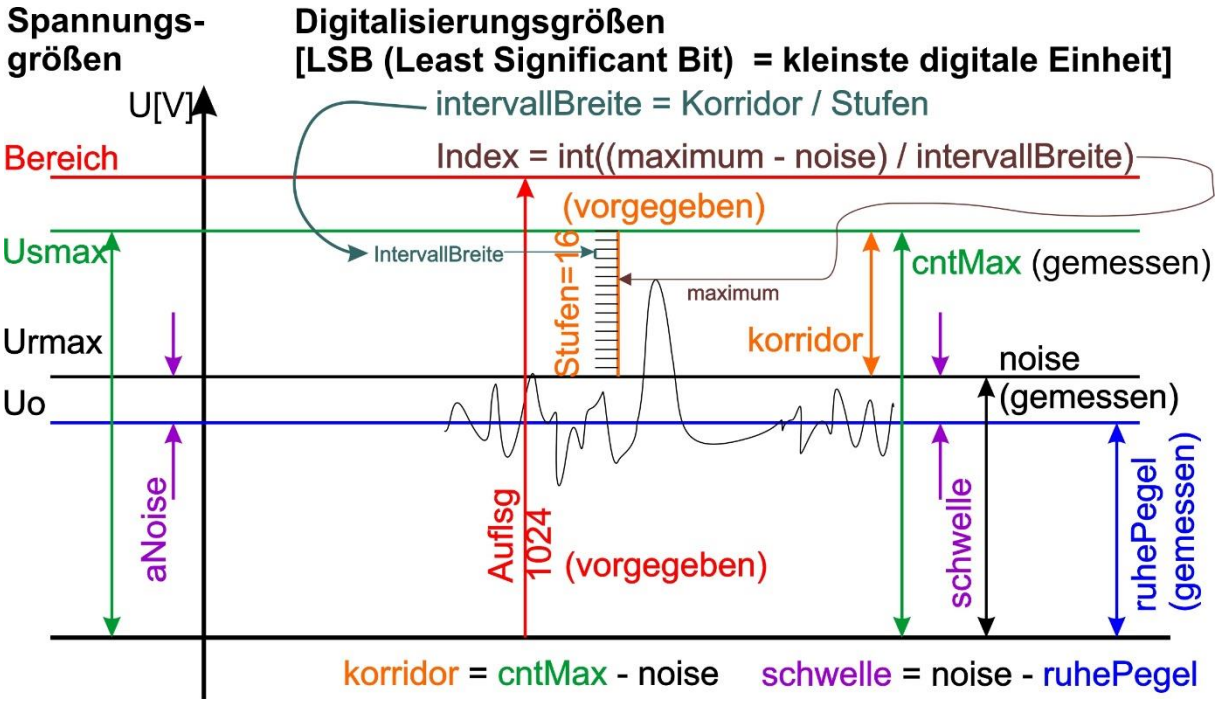
Neben den essentiellen Programmteilen und ein paar einfachen Helferlein benutzen wir das erste Mal das Dateisystem des ESP32 als Heimat für eine Datei **ks.ini**, in welche der Controller nach der Änderungsmessung einer der drei veränderbaren Messvorgaben ruhePegel, noisePegel und cntMax deren Werte und die damit unmittelbar zusammenhängenden Größen schreibt (**writeConfig**(), ks.py, Zeile 201). Beim Programmstart wird diese Datei eingelesen (**readConfig**(), ks.py Zeile 237). Somit stehen die Werte, ohne neu gemessen zu werden, sofort zur Verfügung. Nur wenn andere Umstände eintreten, welche diese Größen mutmaßlich verändern

könnten oder unsinnige Ergebnisse einer Zerfallsmessung auftreten, dann sollten die Basismessungen wiederholt werden.

Das Programmschnippsel zeigt die Methode writeConfig(). Das "wt" in der open-Anweisung bedeutet: schreibe Textdaten. Ist der Vorgang beendet muss die Datei geschlossen werden.

```
def writeConfig(self):
  d=open("ks.ini","wt")
  d.write(str(self.auflsg)+"\n")   #int
  d.write(str(self.bereich)+"\n")   #int
  d.write(str(self.ruhePegel)+"\n")   #float
  d.write(str(self.noise)+"\n")   # int
  d.write(str(self.schwelle)+"\n")   #float
  d.write(str(self.korridor)+"\n")   # int
  d.write(str(self.cntMax)+"\n")   # int
  d.close()
```

Die Zuordnung der darin aufgeführten sieben Instanzvariablen bis auf auflsg und bereich finden Sie in der folgenden Abbildung.



Beim Einlesen werden Leseprobleme durch try – except abgefangen. Wenn es hier Probleme gibt, empfiehlt sich der Anschluss des Controllers an ein Terminal, um die Fehlermeldung zu sehen. Hat das Einlesen geklappt, muss die Datei geschlossen werden. Danach werden die ADC-Attribute auf Bereichswert und Auflösung gesetzt und die Intervallbreite wird berechnet. **showConfig**() gibt die Werte am Terminal aus.

Nachdem eine Messreihe über den Browser gestartet wurde, wartet der Browser auf die Bekanntgabe des Ergebnisses vom Modul KS. In der Liste **spektrum** stehen jetzt die Zählwerte von Zerfallsereignissen der jeweiligen Energiestufe. Diese Liste nutzt das Serverscript, um daraus die Balkenhöhe des Diagramms zu setzen, so wie ich das oben schon beschrieben habe.

Zum Testlauf der Anwendung starten Sie jetzt wifi_connect2.py aus dem Editorfenster. Nach dem Verbindungsaufbau zum Accesspoint im WLAN werden die Verbindungsdaten im Display für 3 Sekunden angezeigt. Danach leuchtet die RGB-LED für 5 Sekunden in gelb, im Display läuft ein Countdown und während dieser Zeit haben Sie die Möglichkeit, den Programmlauf mit der Ja-Taste abzubrechen. Geschieht das nicht, startet spätestens nach 5 Sekunden der Server und meldet seine Empfangsbereitschaft am Display. Sie können jetzt von einem beliebigen Endgerät aus via Browser das Gammameter mit der URL http://10.0.1.150:9192/?start starten und dann durch die Formularfelder dessen Funktionen steuern.

Wenn dieser Test mit allen Teilen der Steuerung ohne Probleme durchgelaufen ist, bleibt noch die Organisation des Autostarts. Im Rootverzeichnis des ESP32 gibt es eine Datei boot.py. Kopieren Sie diese Datei in den WorkSpace Ihrer IDE (µPyCraft oder Thonny oder…). Benennen Sie diese Datei via Windows Explorer in boot.org um und schieben Sie die umbenannte Datei über die IDE wieder zurück ins Device. Dieser Schritt ist Ihre "Reiserücktrittskostenversicherung" für den Fall, dass nach dem nächsten Schritt via IDE keine Verbindung zum ESP32 mehr möglich ist. Das ist selten, kann aber eintreten. Versuchen Sie nicht, boot.org in der IDE zu öffnen, das funktioniert weder vom WorkSpace aus noch vom Device. Im Fall eines GAU besteht so die Möglichkeit, über Putty wieder ins System zu kommen, die Datei in boot.py umzutaufen und so einen "offenen" Start in der IDE zu ermöglichen.

Für einen Autostart benennen Sie jetzt wifi_connect2.py im WorkSpace über den Explorer in boot.py um. Schieben Sie boot.py aufs Device. Die Datei server.py sollte sich ja bereits dort befinden und so müsste nach dem nächsten Neustart des ESP32 nach der Netzwerkanmeldung der Server durchstarten. Ab jetzt läuft der Server autonom und kann über das Webinterface gesteuert werden. Für diese Lösung ist aber immer noch ein lokales Netzwerk mit Funkrouter nötig. Dass es auch anders geht, zeigt das nächste Kapitel.

Der ESP32 als Accesspoint

Wenn Sie nun alle Teile des Projektaufbaus schön in ein Gehäuse verpackt haben und in der Pampa unterwegs sind, um zum Beispiel Messungen an Mineralien zu machen, dann haben Sie vermutlich selten ein WLAN zur Hand, über das Sie Kontakt mit Ihrem ESP32 aufnehmen können. Und in der Tat, es geht auch ohne Router und PC. Wenn Sie Ihren ESP32 selbst Accesspoint spielen lassen, können Sie sich vom Handy aus damit verbinden und über den Browser wie gehabt das Gammameter steuern.

Die bestehenden Dateien wifi_connect2.py und server.py habe ich dazu einfach auf accesspoint2.py und server2.py kopiert und beide an einigen Stellen leicht verändert.

Für die Bereitstellung einer Accesspoint-Funktionalität durch den ESP32 wurde im letzten Fünftel von wifi_connect2.py ein Teil des Programmtextes gelöscht oder durch folgende Zeilen ersetzt.

```
# ************************* Bootsequenz ******************************
nic = network.WLAN(network.AP_IF)  # Constructoraufruf erzeugt WiFi-Objekt nic
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')             # # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)                   # in eine Hexziffernfolge umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben
d.writeAt(myMac,0,0)
#
ssid = 'gammameter'; passwd = "uranium238"
nic.ifconfig(("10.0.2.100","255.255.255.0","10.0.2.100","10.0.2.100"))
print(nic.ifconfig())
print("Authentication mode:",nic.config("authmode"))
nic.config(essid=ssid, password=passwd)
while not nic.active():
  # blink red LED while not activated
  b.blink(1,0,0,200,300,anzahl=1)
  pass

print("Server gammameter ist empfangsbereit")
# Write connection data to OLED-Display
d.clearAll()
d.writeAt("10.0.2.100/24",0,0,False)
d.writeAt("Ready to serve",0,2,False)
d.writeAt("Port: 80",0,1)
sleep(3)

if t.jaNein(meldung1="ABBRECHEN?",laufZeit=5) != t.JA :
  # tpNein touched between 5 sec or untouched at all start server
  d.clearAll()
  exec(open('server2.py').read(),globals())
else: # falls das Pad an tpJa beruehrt wurde
  print("Die Bootsequenz wurde abgebrochen!")
  d.clearAll()
  d.writeAt("ABGEBROCHEN",0,0)
```

Für die Funktion als Accesspoint sind die fett-formatierten Stellen wichtig, der Rest wurde entweder nicht verändert oder ist nur Makulatur.

Die Notierung eines Passworts ist eigentlich überflüssig, da mein ESP32 von vornherein nur den Authentifizierungsmodus 0 = Open anbietet und sich auch nicht überreden lässt, einen anderen Modus zu akzepieren.

Die Datei server.py hat an mehr Stellen Änderungen abgekriegt. Das hat vor allem damit zu tun, dass ich den Accesspoint mit der IP 10.0.2.100 in ein anderes Teilnetz ausquartiert habe, 10.0.2.0/24. Die 24 stellt eine Kurzform der Netzwerkmaske 255.255.255.0 dar. Da die Anweisung ifconfig als Parameter ein 4-Tupel erfordert, habe ich an 3. und 4. Stelle für die Gatewayadresse und den DNS-Server auch die Adresse des Accesspoints angegeben. Obwohl beides nicht genutzt wird, muss eine gültige IP angegeben werden, 0.0.0.0 wird nicht akzeptiert. Weil eine andere Serverdatei nachzuladen ist, musste auch der Name in der exec-Anweisung in accesspoint2.py angepasst werden.

Im Serverscript server2.py wurden in den bodyX-Strings schon mal die IP-Adressen für die action-Feld geändert, von 10.0.1.150:9192 auf 10.0.2.100:80. An der Funktion web_page() musste nichts geändert werden, erst wieder die Variable portNum gleich

im Anschluss von 9192 aus 80. Die Ausgaben auf das OLED-Display wurden umorganisiert. Innerhalb der Serverschleife blieb wieder alles beim Alten.

Im Testlauf wird accesspoint2.py im Editorfenster gestartet, das danach server2.py nachlädt, wenn am Checkpoint nicht abgebrochen wird.

Im Terminal sollte sich dann eine Meldung der folgenden Form zeigen.

STATION MAC: ac-67-b2-2a-7b-41

('10.0.2.100', '255.255.255.0', '10.0.2.100', '10.0.2.100')
Authentication mode: 0
Server gammameter ist empfangsbereit
Aufloesung: 1024 Bit; Bereich: 3.3 V
Konfiguration wurde eingelesen
Ruhe: 453.7291
Noise max: 480
Schwelle: 26.27087

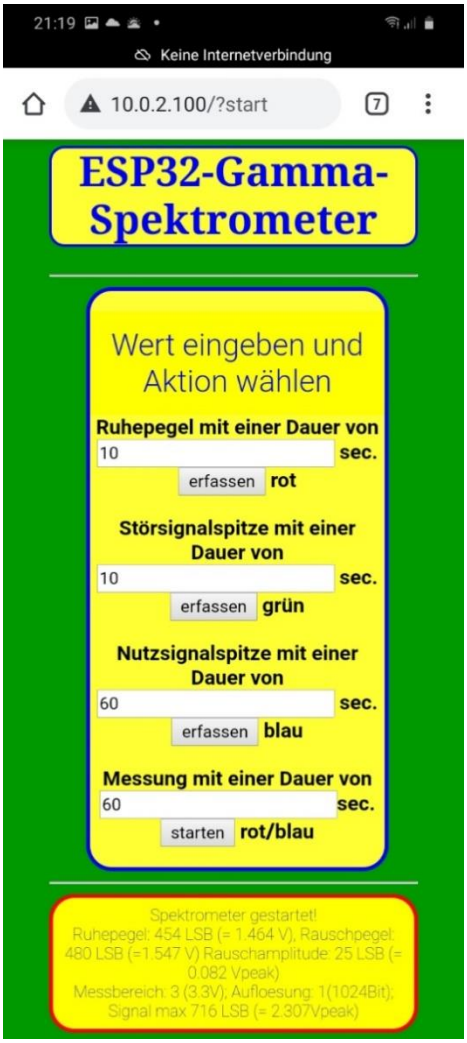
cntMax: 710
Korridor: 230
Intervallbreite: 42.73307

Empfange Anfragen auf 10.0.2.100:80

Zum Testlauf der Anwendung auf dem Accesspoint starten Sie jetzt accesspoint2.py aus dem Editorfenster. Nach dem internen Aufbau der Funktion als Accesspoint ohne WLAN werden die Kontaktdaten im Display für 3 Sekunden angezeigt. Danach leuchtet die RGB-LED für 5 Sekunden in gelb, im Display läuft ein Countdown und während dieser Zeit haben Sie die Möglichkeit, den Programmlauf mit der Ja-Taste abzubrechen. Geschieht das nicht, startet spätestens nach 5 Sekunden der Server und meldet seine Empfangsbereitschaft am Display. Sie können jetzt von einem beliebigen Endgerät aus via Browser das Gammameter mit der URL http://10.0.2.100/?start starten und dann durch die Formularbuttons dessen Funktionen steuern.

Gehen Sie zu diesem Zweck auf Ihrem Handy auf Einstellungen – Verbindungen – WLAN. Lassen sie das Gerät nach neuen Accesspoints suchen oder geben Sie die SSID gammameter direkt ein (je nach Möglichkeit). Verbinden Sie dann ihr Handy mit dem Accesspoint des ESP32. Sie werden über diese Verbindung keinen Internetzugriff haben, können sich aber mit dem ESP32 unterhalten.

Danach starten Sie mit der URL http://10.0.2.100/?start das Gammameter und über die Formbuttons die jeweilige Aktion. Die Zeitwerte können einfach übernommen oder neu eingegeben werden.



Das System läuft jetzt als Insellösung, ohne Zugriff auf ein lokales Netzwerk aber noch nicht autonom. Ohne den ersten Zugriff über die genannte URL besteht kein Zugriff auf die Messeinrichtung, weil das Interface nicht angezeigt wird. Unser Server frisst jeden Zugriff, der nicht den Parameterdaten in der URL-Zeile entspricht. Durch entsprechende Handshake-Methoden ließe sich auch ohne Authentifizierung über den Accesspoint eine zusätzliche Absicherung der Steuerung erreichen.

Wenn dieser Test mit allen Teilen der Steuerung ohne Probleme durchgelaufen ist, bleibt auch hier noch die Organisation des Autostarts. Im Rootverzeichnis des ESP32 gibt es eine Datei boot.py. Kopieren Sie diese Datei in den WorkSpace Ihrer IDE (µPyCraft oder Thonny oder…). Benennen Sie diese Datei via Windows Explorer in boot.org um und schieben Sie die umbenannte Datei über die IDE wieder zurück ins Device. Dieser Schritt ist Ihre "Reiserücktrittskostenversicherung" für den Fall, dass nach dem nächsten Schritt via IDE keine Verbindung zum ESP32 mehr möglich ist. Das ist selten, kann aber eintreten. Versuchen Sie nicht, boot.org in der IDE zu öffnen, das funktioniert nicht. Im Fall eines GAU besteht so aber die Möglichkeit, über Putty wieder ins System zu kommen, die Datei boot.py zu löschen und boot.org in boot.py umzutaufen und so einen "offenen" Start in der IDE zu ermöglichen.

Für einen Autostart benennen Sie jetzt accesspoint2.py im WorkSpace in boot.py um. Die Datei server2,py sollte sich bereits auf dem Device befinden und so müsste nach dem nächsten Neustart des ESP32 nach dem Aufbau des boardeigenen Accesspoints der Server durchstarten. Ab jetzt läuft der Server autonom und kann nach der Anmeldung des Smartphones am Accesspoint des ESP32 über das Webinterface gesteuert werden, wie es oben beschrieben ist..

Leider stellt der ESP8266 für dieses Projekt viel zu wenig RAM-Speicher zur Verfügung, ich habe das eingangs ja schon bedauert. Beim Zuladen des Serverprogramms stellt MicroPython fest, dass knapp 10000 Byte RAM fehlen. Es lohnt sich also doch die Anschaffung des größeren Bruders oder Sie geben sich mit der Magerlösung von Blogfolge 5 zufrieden.

Threading auf dem ESP32 funktioniert unter MicroPython grundsätzlich im Beta-Ansatz. **Leider** aber nicht im Zusammenhang mit WiFi oder Sockets. daher sollte

man als Messzeit nicht mehr angeben, als der Webbrowser geneigt ist, auf eine Antwort vom Server zu warten. Das ist von System zu System unterschiedlich, probieren Sie es aus. Mit 60 Sekunden (ich habe auch schon 300 Sekunden mit Erfolg getestet) ist man auf der sicheren Seite. Im Moment arbeite ich noch an einer Android-App, welche die Steuerung und Anzeige ohne Browser besser erledigt. Lassen wir uns überraschen!

Wrapping up

Mit dem, was Sie in den sechs Folgen lernen konnten, sind sie jedenfalls gut gerüstet, um eigene Projekte anzugehen. Sie können die Auswahl aus unterschiedlicher Programmiersoftware nutzen, kennen die Basistypen von Variablen, haben einen Einblick in serielle Datenstrukturen wie Listen und können GPIO-Pins bedienen oder analoge Signale einlesen. Mit Modulen und Klassen können Sie Programmteile gliedern und durch Importieren oder gar Vererben in anderen Projekten gewinnbringend einsetzen. Sie wissen, wie Sie Funktionen erstellen müssen, welche Parameter platzgebunden oder optional sind, was lokale und globale Variablen sind, und wie man damit umgeht. Außerdem können Sie Daten in Files auf dem ESP32/ESP8266 sichern und wieder einlesen. Gewiss ist diese Auflistung nicht vollständig, aber sie zeigt, was alles in MicroPython steckt, wenn man damit umgehen kann. Eines kann ich Ihnen versprechen, es warten noch viel, viel mehr interessante Features dieser Programmiersprache darauf, von Ihnen entdeckt zu werden..

Sicher gibt es in der Zukunft weitere Beiträge, die tiefer in die interessante Materie von MicroPython eindringen.

Dann bis zum nächsten Mal!

Links zu den früheren Teilen:
Teil1 im Web und als PDF deutsch
Teil2 im Web und als PDF deutsch
Teil3 im Web und als PDF deutsch
Teil4 im Web und als PDF deutsch
Teil5 im Web und als PDF deutsch als PDF englisch
Teil6 im Web und als PDF deutsch als PDF englisch