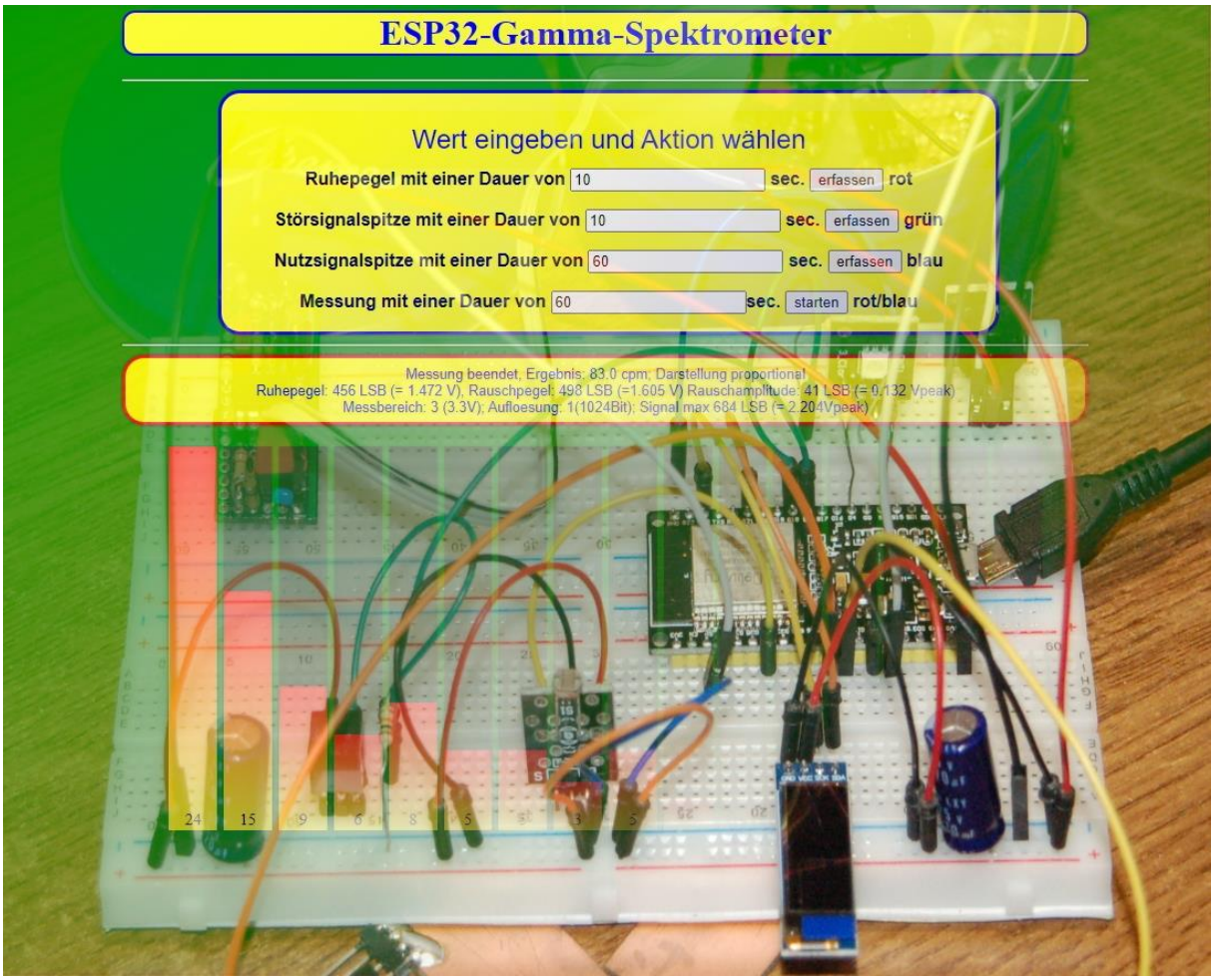


Micropython on the ESP32/ESP8266 - part6

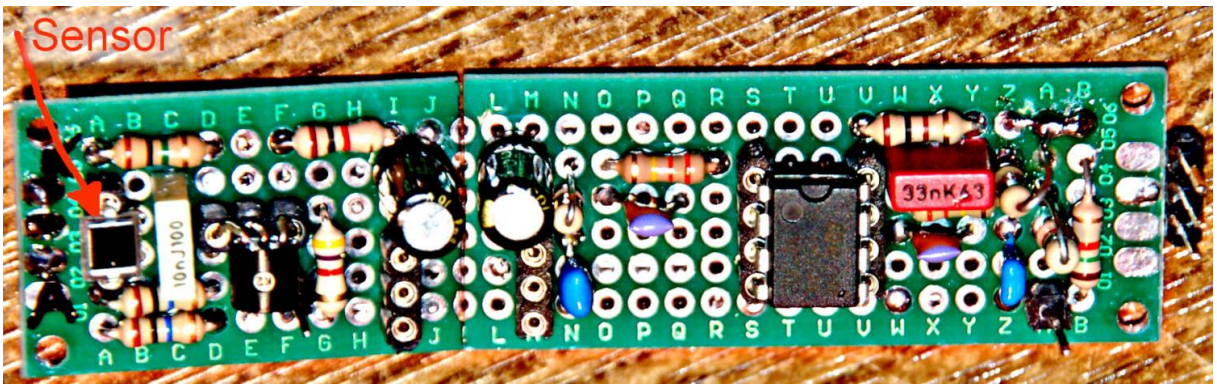
Today we combine all the findings from the past episodes of this blog into one overall project. The ESP32 is well utilized. It operates LEDs, an active buzzer, an OLED display, two touchpads and reads in decay data from our self-made radiation detector. The three most important parameters for this circuit are set via a web interface, which ultimately also graphically displays the results of the measurement. All of this is managed by the web server from the second episode of the blog, which is cannibalized for this purpose. At the end of the current episode, I'll show you how the ESP32 can manage without a local wireless network. The whole thing is of course autonomous, including when it comes to the start.

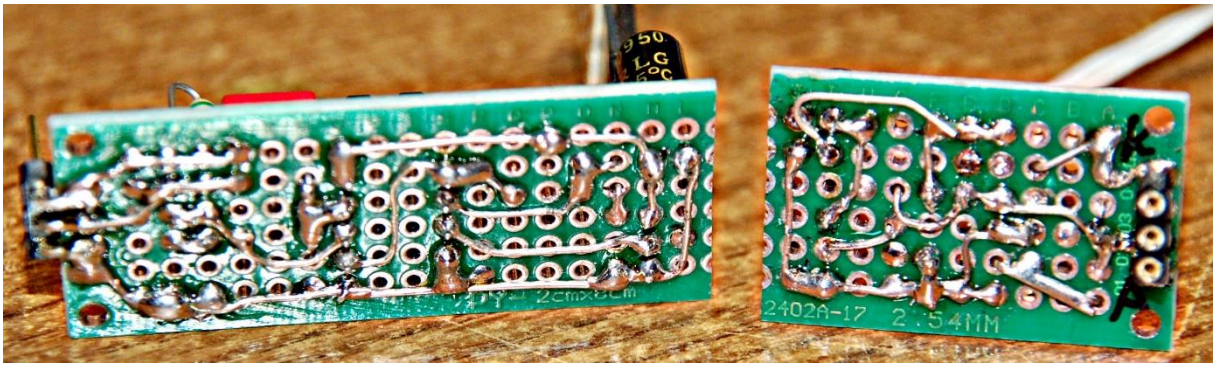
It's amazing what fits into such a small building block. In terms of software, the program would also work in an ESP8266, but unfortunately the RAM memory on these boards is not nearly enough. So a warm welcome to the 6th part of this series. As already introduced in Part 5, Part 6 is also available as a PDF in English through the translator from Onkel Google. You can find a link to this at the end of the article.

You will need the following parts in terms of material. If you've studied the previous episodes, you probably already have most of them.

1 ESP32 NodeMCU Module WLAN WiFi Development Board or
1 ESP-32 Dev Kit C V4 or
1 NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F with CH340
1 0.91 inch OLED I2C display 128 x 32 pixels for Arduino and Raspberry Pi or
1 0.96 inch OLED I2C display 128 x 64 pixels for Arduino and Raspberry Pi
1 KY-012 buzzer module active
2 LED (color does not matter) and
2 resistor 330 ohm for LED or
1 KY-011 Bi-Color LED module 5mm and
2 resistor 560 ohm for LED or
1 KY-009 RGB LED SMD module and
1 resistor 330 Ohm for blue LED
1 resistor 2.2k Ohm for red LED
1 resistor 3.9k Ohm for green LED
1 KY-004 button module or
1 keypad-ttp224-1x4-capacitive
2 mini breadboard 400 pin with 4 power rails
1 jumper wire cable 3 x 40 PCS. 20 cm each M2M / F2M / F2F
2 pieces of sheet metal approx. 20 x 20 mm (not aluminum!) Or remnants of circuit boards
some pins 0.6x0.6x12mm

The descriptions for the circuits of the nuclear radiation sensor are quite extensive. Therefore I refer you to part 5 of the blog, where everything is described in detail, because of the required parts and the structure. When fully assembled, the part looks like this on the breadboard.





In episode 5 we had already made the first measurements. The programs for the individual services were initially started from the PC. In a second stage, the ESP32 / ESP8266 then ran autonomously and was controlled via buttons or touchpads. The results could be read on the OLED display. Because the display options are very limited, we are adding a web server to the project today.

Old love

With the reused peripherals, the corresponding driver modules are also used again. In some places, however, it was tweaked again, sometimes vigorously, in order to improve the performance and to be able to offer our own example for the inheritance of classes. I will start with the explanations.

Class inheritance and overwriting

Import

A class can be imported into a program in order to be able to access its namespace via the class name or an instance of the class. We know that and have used it many times. But what happens if you want to import a class A to define a class D? We have the a.py module with class A here, we will use it again later. Enter the text or download the a.py file, you already can upload it to the ESP32 / ESP8266.

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")
```

A couple of Class A tests:

>>> from an import A

Constructor A: 5000 40 50 40 50
100

During the manual import, the constructor already reports, we recognize this from its reply. This is because the a.py module contains two lines in addition to the class definition, the content of which already relates to the class. Two statements are executed. The class instance w is declared and the result of the method summcv (60) is output. 60 as an argument, added to the value of the class variable classVarA of 40, results in 100. That is sufficient as a test. Upload a.py to the device now at the latest. Then create or load the d.py file and transfer it to the WorkSpace.

Class D builds on class A and therefore imports it.

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.

This is the output when you start d.py in the editor. We can see from the first two lines that the a.py file is completely processed during import. The attributes and methods are registered and the last two lines are executed.

A's constructor appears a second time when the instance attribute self.a is instantiated in D's constructor. D's constructor answers and finally the result of the test lines appears.

We continue manually in the terminal.

```
>>> 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']
```

You can also access its attributes and methods via the instance attribute a.

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

Now you can play the game almost as you like, at least as long as the memory lasts.
Slide d.py into the device and create e.py in the editor window or download the file.
Start e.py in the editor window.

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
```

As long as you have a closed chain of instances of the higher-level class, you also have access to the methods of the top-level class, in this case A. But unfortunately the calls for them are becoming more and more cumbersome and confusing. We want to turn that off by putting everything on one level. The magic word is inheritance. I will show you how to do this using modules a, b and c in conjunction with the corresponding classes. Then we apply the new knowledge to the current project and modify the modules used so that they fit the new knowledge.

Vererbung

We start with module a and class A, everything stays as it is, so you don't have to download the file again - unless you have changed anything.

```
>>> 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
>>>
```

You can now classify this issue correctly. An instance w was created with the start value 5000 for the parameter ap of the constructor and then the method w.summvc () was called with the argument 60. Create an instance x and test the x.sum () method. Nice!

Next, let's get module b with class B and upload it to the device.

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
```

According to previous knowledge, the import from B to REPL proceeds as expected, as does the creation of an instance y of class B. But now take a look.

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

Wouldn't you have expected an error message saying that a positional parameter was missing when calling sum ()? And why 'call in B'? We had defined the sum () method with three parameters in A! That's right, but look at the line

```
class B (a.A):
```

exactly on. By adding the bracket and its contents, you have instructed MicroPython to take all information from class A. That happened too. Class B inherited all objects from class A.

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

But we have defined a new method sum () in B with only two parameters and it has overwritten this definition sum () from A. You should have noticed something else if you compare the call of the product () method from class A with the one from the previous section.

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

In order to reach the method product () now, we did not have to go back one step in the hierarchy, but could reach the method directly from the instance of class B. Inheritance ensures that the testator's namespace is transferred to that of the heir. This is convenient, but has the disadvantage that objects from class A, i.e. the predecessor, are overwritten by objects of the same name from class B, the inheritor. However, this disadvantage can also become an advantage, because overwriting objects creates dynamics in the system, through which one can take over useful things, but at the same time also replace old with new. For this purpose, Python in general and MicroPython in particular offer mechanisms that support this approach even more. Maybe more on that elsewhere.

Let's cover the whole thing with a class C from the c.py module. Then we go into an important point about inheritance. Don't forget to send c.py to the device as well.

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
>>>
```

We can see that when importing C, the constructor of A first responds. This is still due to the last two lines in a.py. Then we create a class C object. The class A, B, and C constructors appear immediately. The difference method from C calculates correctly, 3 * 10 -20 = 10, but the method from A is still easily available via the instance z. We only overwritten the sum from B with that from C.

But something completely different. Where did the constructors of B and A get their positional parameters from? Didn't classes B and A have to be initialized somehow? Well, that happened in the lines

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

in C und

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

in class B. Compare the parameters with the output of the constructor methods of both classes. The super () function is used to address the superordinate class and initialize it with __init __ ().

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

With the dir () function, you can now convince yourself that everything that has been declared so far can also be reached via the instance z of C, across three levels and without glowing trains!

With this knowledge we are now polishing up the BEEP, TP and OLED classes. TP gets most of the fat, BEEP is dressed in such a way that it fits the new look of TP, the modified version of which is now called TPX, and OLED is only slightly striped. Because it causes the least amount of effort, let's start with OLED.

Renovation of good friends

The OLED class has seen a backward-compatible change in the following methods. The writing methods of the oled.py module, namely writeAt (), pillar (), xAxis (), yAxis () and clearFT (), have an additional optional parameter show with the default value True in the parameter list. I will show this using the example of the clearFT () method.

```
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 ensures that, as before, the change to the content of the display's frame buffer is immediately sent to the display at the end of the method. If show = False is set, only the change is made in the frame buffer. Finally, one of the writing methods must be called with show = True or without specifying this attribute, so that the change appears on the display. Since the show parameter is optional and the last in the series, it can safely be omitted. The syntax and function is then exactly the same as in earlier versions due to the pre-setting with True. That's what the term 'backward compatible' means.

With three output lines, this results in a speed increase of approx. 225%. You can try this out yourself with the test program newoledtest.py, which is also an example for the application of the new syntax.

The BEEP class, as a signal class, is expanded by four methods for light signals or tone signals. The ledOn () and ledOff () methods control an RGB LED and the blink () method does justice to its name. With setBuzzPin () the buzzer output can be set or reset later. The setLedPin () method works in a similar way.

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)

The constructor takes the GPIO numbers of the outputs for the buzzer, the red, green and blue LEDs and the duration of the beep signal as optional arguments. All

placeholders for IO pins are pre-assigned with None. If these arguments are not overwritten with (permitted) pin numbers when they are called, these arguments retain the value None. This means that this color does not occupy a pin and is ignored when switching on and off.

The above example activates the buzzer on GPIO13, as well as the red LED on 2 and the blue one on 4. No GPIO number is given for green. This color cannot be activated later either.

The 3 LEDs of the RGB unit, which can also be individual HIGH-active LED types, are switched by ledOn () if the outputs are activated. The parameters are optional, which enables them to be pre-assigned. This takes effect when the method is called with no arguments. The default is r = 1, g = 0, b = 0. But that can be changed at will. The call

```
>>> ledOn ()
```

therefore switches on the red LED at the output. The status of the blue LED is not changed, the green LED is not activated and therefore cannot be addressed. Please note that the LEDs must be connected with series resistors greater than 330 ohms in order not to exceed the maximum current that the pins can deliver. Most LEDs are so bright that resistors with even higher values can be used. In the case of RGB LEDs, it is also useful to choose the resistance values so that the three colors appear equally bright. Then it also works with the mixed colors

ledOn (1,1,0) - yellow
ledOn (0,1,1) - cyan
ledOn (1,0,1) magenta
ledOn (1,1,1) - white.

The instruction

```
>>> ledOff (r, g, b)
```

works similarly. With a 1 the color is deleted, with 0 the previous status remains unchanged. ledOff (1,1,1) is the default setting for switching off the LEDs, and because all parameters are optional, ledOff () turns off all LEDs.

```
>>> 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)

The position parameters r, g, b follow the description of ledOn (). The duration of the lighting is given to me, which is the same as the duration of the pause if no value is transferred for pause. number indicates how often an LED setting should light up.

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

This instruction causes the LED in the mixed light to flash white three times for 200ms. The period of a flashing process is 200 + 800 = 1000ms = 1s.

The methods previously contained in BEEP have retained their external functionality and have only been internally adapted to the new settings for LEDs and buzzer. For this reason, the beep method was given three optional color parameters. If you omit the arguments when calling, flashes blue by default. The buzzer only sounds if the GPIO number of an output pin was specified during the instantiation for buzz. With a call of this kind

>>> b.setBuzzPin (13)

the buzzer output can be activated later. You can deactivate it at any time

>>> b.setBuzzPin (None).

A similar method also exists for the subsequent activation and deactivation of the LED outputs.

setLedPin (color, pin)

The color argument is a string of the form r, red, rot, rojo. The GPIO number is passed in the pin argument. The value None deactivates the output

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

No output has yet been activated.

>>> c.ledOn (1,0,0)
No reaction, because the red channel is not yet activated.

>>> c.setLedPin ("red", 12)
>>> c.ledOn (1,0,0)
The red LED is now on.

Class TP has now at least achieved status 3.0 in terms of changes. From the simple collection of functions in the touch.py module to the TP class, the most complex state to date is now achieved with inheritance. But one after anonther.

The modules touch.py and touch8266.py have got two new methods. One allows the limit value for the detection of touches to be set. A plausibility check is integrated into the ESP32 so that only valid values are transferred to the threshold variable. I have also put a doc string at the beginning of the module definition, which gives an overview of the handling of the module's methods.

Since the methods of the touch8266.py module work purely digitally, I have also changed the return values in touch.py to ensure compatibility. Like the comparable method in touch8266.py, the getTouch method now returns True (or 1) and False (or 0) and, in the event of a read error, the value None. The methods waitForTouch and waitForRelease also return True as a response to the desired action and, in the event of a timeout, None. Examples of application can be found in the description of the functions for wifi_connect2.py and after the experiments on class inheritance.

The method setThreshold exists in touch8266.py for reasons of compatibility with touch.py but only has a dummy function like the value threshold halt.

The most important innovation comes at the end because it had to be introduced as a prerequisite for further functionality. The constructor was amputated. His only task now is to set the limit value, which is optionally passed as a parameter. Of course, he makes the remaining methods known for instances of TP. One of these methods, initTP (GPIONummer), is the new / old generation of touchpad objects. The content of this method is the part that I removed from the constructor. It is checked whether the transferred GPIO number represents a valid designation for a touchpad connection and the corresponding instance is finally returned. It is now possible to declare multiple pads within TP.

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
```

As a result of this change, the parameter lists of getTouch (), waitForTouch () and waitForRelease () have grown. A touchpin object must now be transferred so that the method knows who to monitor. Unfortunately, TP is no longer down compatible, but has gained in versatility because application programs can now dynamically create touch objects. Incidentally, the del command deletes objects that are no longer used, thereby freeing up memory.

The innards of touch8266.py have of course been adapted to the new situation.

The modified modules touch.py and touch8266.py are available for download and examination. We will now deal with this a little more in the course of inheritance.

Inheritance at work

Problem: A yes / no query via touchpads or keys is to be implemented. Logically, two touch or button objects are required for this. Well, let's create the two with touch.py/touch8266.py and manage them in the new program. It is possible, but what if you need the same procedure in other programs? Wouldn't it be more practical to include the whole thing in a class that also has the previous properties of TP? A, B, C, that sounds like - yes, right - inheritance and we only had that recently.

We create a new class TPX in a new module touchx.py, in which the methods of TP and the new method yesNo () are peacefully united on a common playground. And while we're at it, we'll also get the BEEP class - and OLED, because light and sound signals can also be used well for button actions and, well - a text display on the OLED display is also not wrong. "Quite unintentionally" we made sure that the same method names and instance attributes exist for touch.py and touch8266.py. This means that the two classes have the same API. I would like to clean up the two variants for the ESP32 and the ESP8266 and merge both modules in the new touchx.py. To do this, it is necessary to find out the controller type, this can be done with sys.platform (). So we start the dance. After various imports, we create a display object d, import depending on the type touch or touch8266 and fill the specifications for the LED pins. Then we create the signal object b. A translator should help with the assignment of the pin names of the ESP8266.

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
```

The definition of the class TPX, which inherits from touch.TP, is where touch.py and touch8266.py in some way unite. Thanks to the definition:

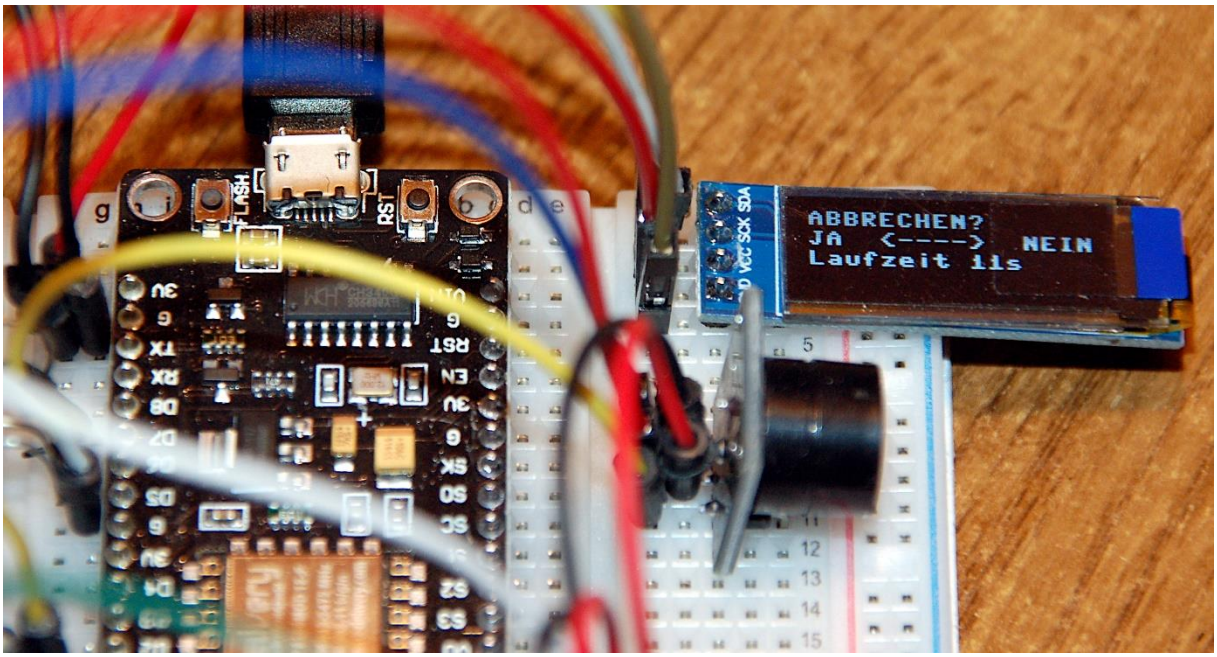
import touch8266 as touch

is always continued with the correct module depending on the type of controller. touch8266 is now also referred to as touch. The methods have the same names and the same parameter list; there is no need to make a wide distinction because the results and returns are the same. From here on, nobody is interested in what happens within the methods.

We define a few class attributes and, depending on the type, assign the input pins for yes - no, only as an option!

The constructor takes the options as default values and forwards the arguments to the instance attributes. When instantiating, all values can of course be overwritten as you see fit. But remember, even if you only have to define a single method, by calling the constructor you have all the methods and variables from the correct TP class available. In addition, the signal and display classes are available under touchx.b and touchx.d, which can even be passed on from the calling program during instantiation. We will see an example of this later in the wifi_connect2.py program.

The yesNo () method takes 5 optional parameters. tj and tn are pin objects of the TPX object or, without explicit specification, the pins tpJ or tpN. message1 and message2 take strings that are shown in the first two lines of the OLED display. After running time seconds, the method returns None if no key has been pressed. While the time is running, the RGB-LED gives a yellow signal and a countdown runs in line 2 in the display.



If the Yes button is pressed - the arrow indicates this - method 2 returns, if No a 1, both buttons / pads at the same time result in 3.

The rest is made up of familiar building blocks and shouldn't be difficult to decipher.

Network registration in a new guise

After much preparation, we are approaching the goal, level 1, the department registration at the access point. Three new modules have been added to the modules from episode 2, oled.py, beep.py and touchx.py. They help us to make the controller independent of the PC.

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)
```

Not much has changed in the file since the second episode. Most of the changes and additions are in the various modules that we have now worked on. We'll put it in wifi_connect2.py now. Almost nothing has changed about the establishment of the

connection itself. There is now a red flashing signal of 1Hz, which indicates that there is no connection yet. This state was previously represented by dots in the terminal window.

The function for displaying available access points is still available, but is not used because the list cannot be shown in the display. You can display the debugging output on the terminal again. To do this, simply uncomment the lines in bold.

Then comes the point where you have to enter your own access data, name of the access point (aka SSID) and your password.

The print and sleep commands in the while loop are commented out. They have been replaced by the blink instruction with a total duration of one second. On the ESP8266, the red LED flashes only once very briefly because the connection to the access point is established automatically before any other command is executed.

The following print commands for the status message have also been commented out, but have not been completely removed for possible debugging. The status message is now shown on the display.

Finally, our yesNo () method from the TPX module appears. An input action is waited for five seconds. With a yes-answer the program aborts, with pressing the no-selection or without action the server part is subsequently loaded and started.

The server is growing up

The overview

In order not to scare anyone, the server of part 2 only provided us with basic functionality. It will be different now. The original 56 program lines have become 352. The basic structure has been preserved, but has been expanded in three places. From back to front: in the server loop itself, the acceptance of a request and the first parsing of the same have been expanded to include a more complex display of the website. Another third of the circumference of the server loop is used to display the measurement result.

With around 70 program lines, the web_page () function almost reaches the size of the server loop. It takes that part of the browser request that is important for decoding the order placed via the browser. Inquiries that are valid for us begin with a GET request. Something else doesn't get through to the decoder at first.

Further up there are a number of strings that define the content of the website to be sent and then the information from the stylesheet that takes over the formatting of the website follows. It also contains the bar definition for the output of the measurement result as a bar chart.

At the very beginning, as always, there is a series of import statements. Among them is the most important one for this project, the KS class. The letters stand for nuclear

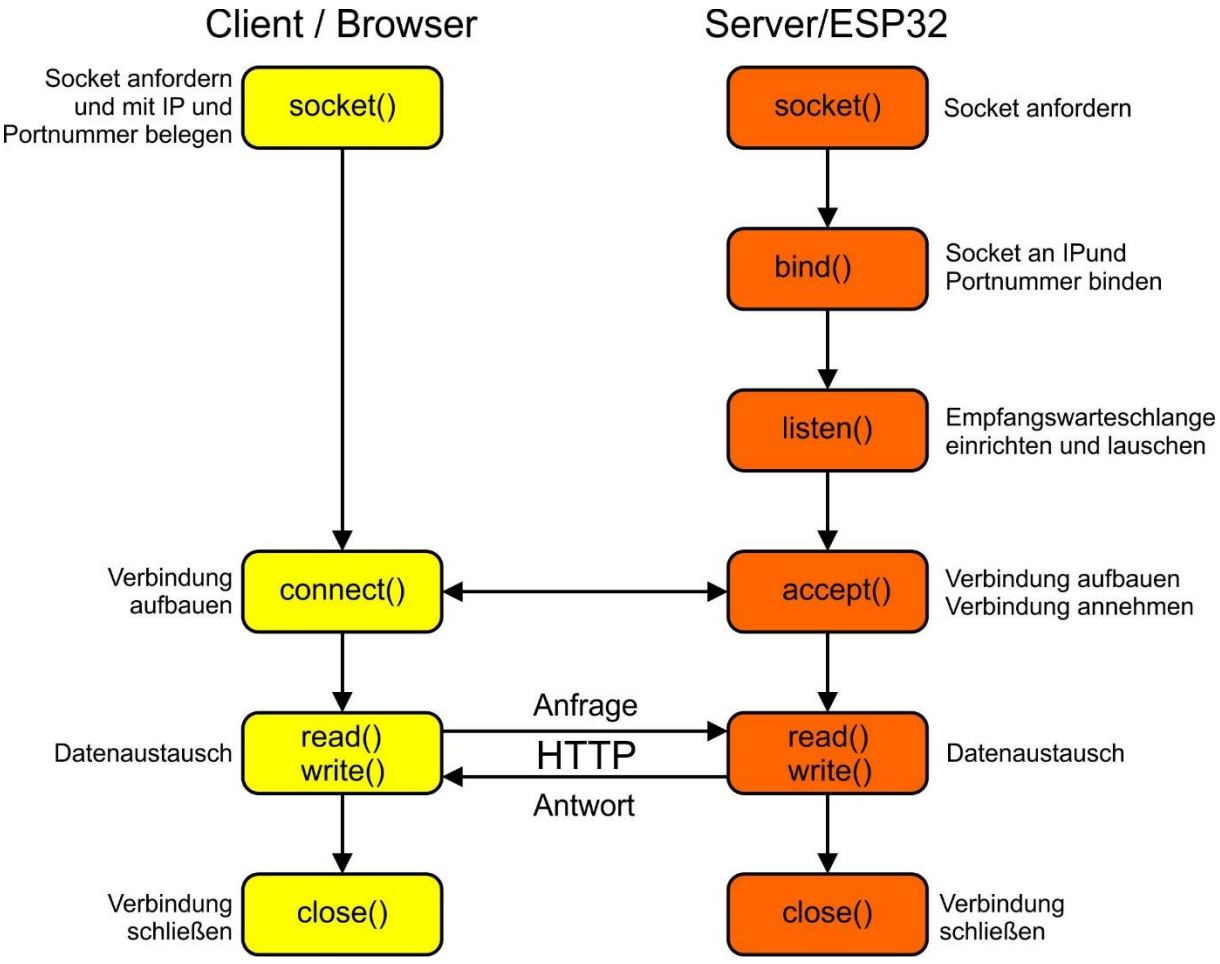
radiation, and that is also the task of the methods of this class to measure nuclear radiation. The other necessary classes such as BEEP, TPX, OLED and others are taken over by server.py from the calling wifi_connect2.py. Finally, if everything goes perfectly, we pack the contents of wifi_connect.py in boot.py so that the ESPs can do an autostart. From then on, the system runs autonomously and can be remotely controlled from a PC, mobile phone or tablet via the browser.

Whow the server works

Download: server.py

Please understand that I am not reproducing the program text of the server file as text here. I suggest that you download the file and, if possible, hold it in a separate text window parallel to the blog, for example with Thonny or another editor that can display line numbers.

The server definition begins in line 271. The WiFi status is shown on the display and a port number is specified for the connection. Then we create a server socket object, bind the IP that we received from the DHCP server of the access point and the port number to this socket and go to the eavesdropping station. If we have not been assigned an IP address, we could enter it ourselves within the two quotation marks. The address under which the server can be addressed appears in the display. The server loop begins with while 1: In the diagram below we are in the list box ().



An incoming request causes the server process to leave the listen loop. The accept () method of the server socket instantiates a connection object c and also returns the IP of the requesting client. The server process is terminated and is back to listening. The communication socket c takes over the processing of the request and finally compiles the website as a response. We have the contact details shown on the display.

Before it gets memory-intensive, let's collect all the resources we have. The mem_info shows us the result in the terminal. We receive the byte stream from the browser, convert it into ASCII code and save the string in the request variable. The interspersed print statements are used for debugging and can be commented out if everything is running. Because the reading can be up to 1024 characters, I use a trick here that prevents the name request from having to be allocated new storage space every time.
At the beginning of the program, right after the first garbage collection, I reserved the memory space for this variable (line 43).

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

At the beginning there is a chance of getting 1024 contiguous bytes, at least better than later with a fragmented heap. But it is even more important that this cementing of the memory means that the read-in data is always later written to this address. This means that no new memory area has to be allocated. Of course, this also counteracts fragmentation of the heap.

What does a valid request from the browser look like and how is it parsed?

Tested browsers are Opera, Chrome and Edge. Firefox has the completely stupid quirk of insisting on https connections, which means that no connection can be established.

Now suppose we would enter this line as a URL in the browser:

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

Then we see this message in the server display:
Got a connection from ('10 .0.1.10 ', 51279)

And the "little bit of text" the browser sends over the Internet, 556 characters

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/apng,*/*;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

Here is our parser's answer. These 30 characters contain important information for us, the rest can be forgotten.

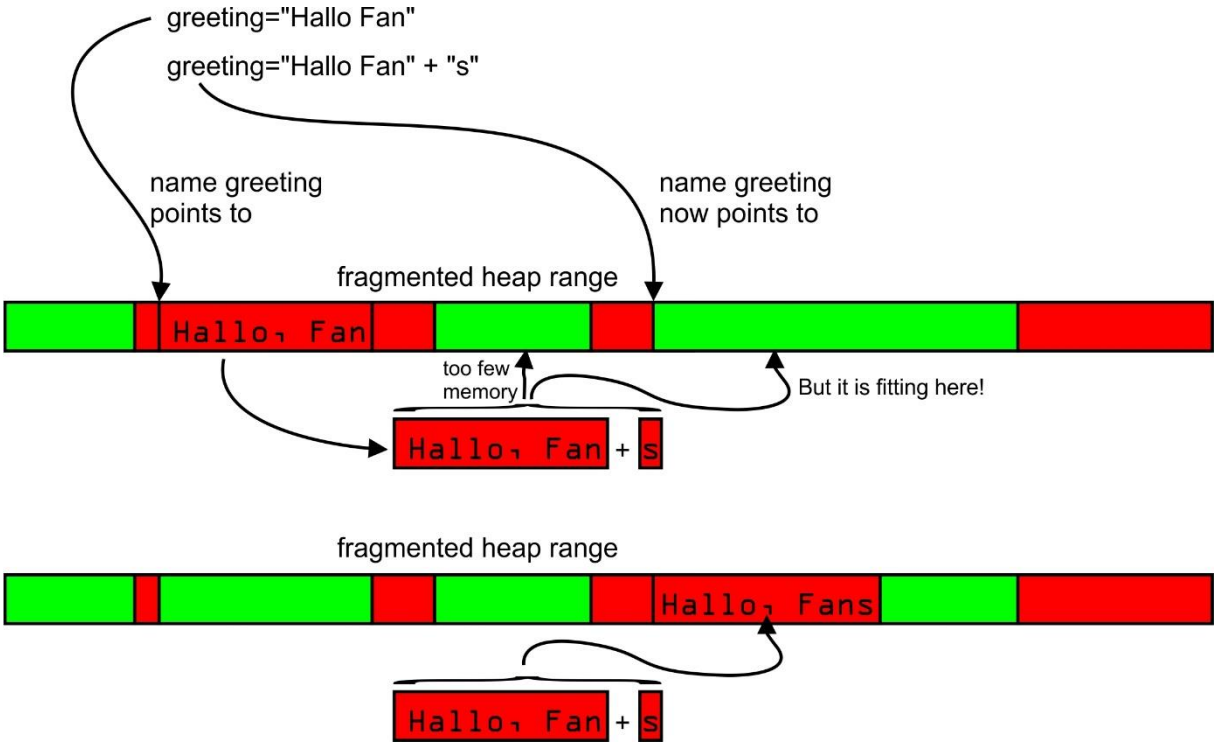
Aktion(30) ==> **?mtime=120&measurement=starten**

The instructions on lines 295, 296 and 298 filter out this text.
295: Is it a GET request and is the root directory of the server, "/", specified? Remember the position after the "/".
296: Search from here for a "" (space, aka blank), remember the position)
298: All characters after the "/" up to the blank come in the variable **action**, but in MicroPython fashion without the blank that limits the area.

The text in action contains the commands that we send to the server and that it must decode next in order to then initiate the appropriate actions. But before we deal with it, let's first see what happens to the results of the actions.

If the function web_page returns True, the decoding was successful and there is a response; if a measurement has been carried out, the result must also be displayed.

The website to be transmitted can be imagined as a string of approx. 4000 characters. As a static page, this string could be sent over the network in one piece. Strings are immutable in Python, i.e. they cannot be changed. For example, if a string is extended by one character at runtime, then the character in the memory is not simply written to the next memory location after the string, but a completely new string is created in a different memory position. With large strings of characters this quickly reaches the limits of the RAM memory, at least it promotes fragmentation.



So I chopped up the website text into smaller pieces with constant content. I have separated them wherever texts have to be exchanged dynamically or where numerical values have to be inserted. Because no string operations have to be performed in this way, the memory requirement is also limited.

What applies to the text of the website also applies to the generation of the bar chart from the list spectrum of measured values. The constant parts are defined by a stylesheet. In between, the numerical values are then interspersed in the if construct from line 317.

But how does the ESP32 know what action to take? We tell him in the function web_page () from line 197.

The function has to value the global variables change and result, so we have to mark them as global. (199,200)

Empty requests and those with favicon.ico as content are no longer parsed. Of what is left, only queries that contain one of the key words go through. I will explain later where these terms come from. The act parameter can contain various requests. We saw one manifestation recently.

?mtime=120&measurement=starten

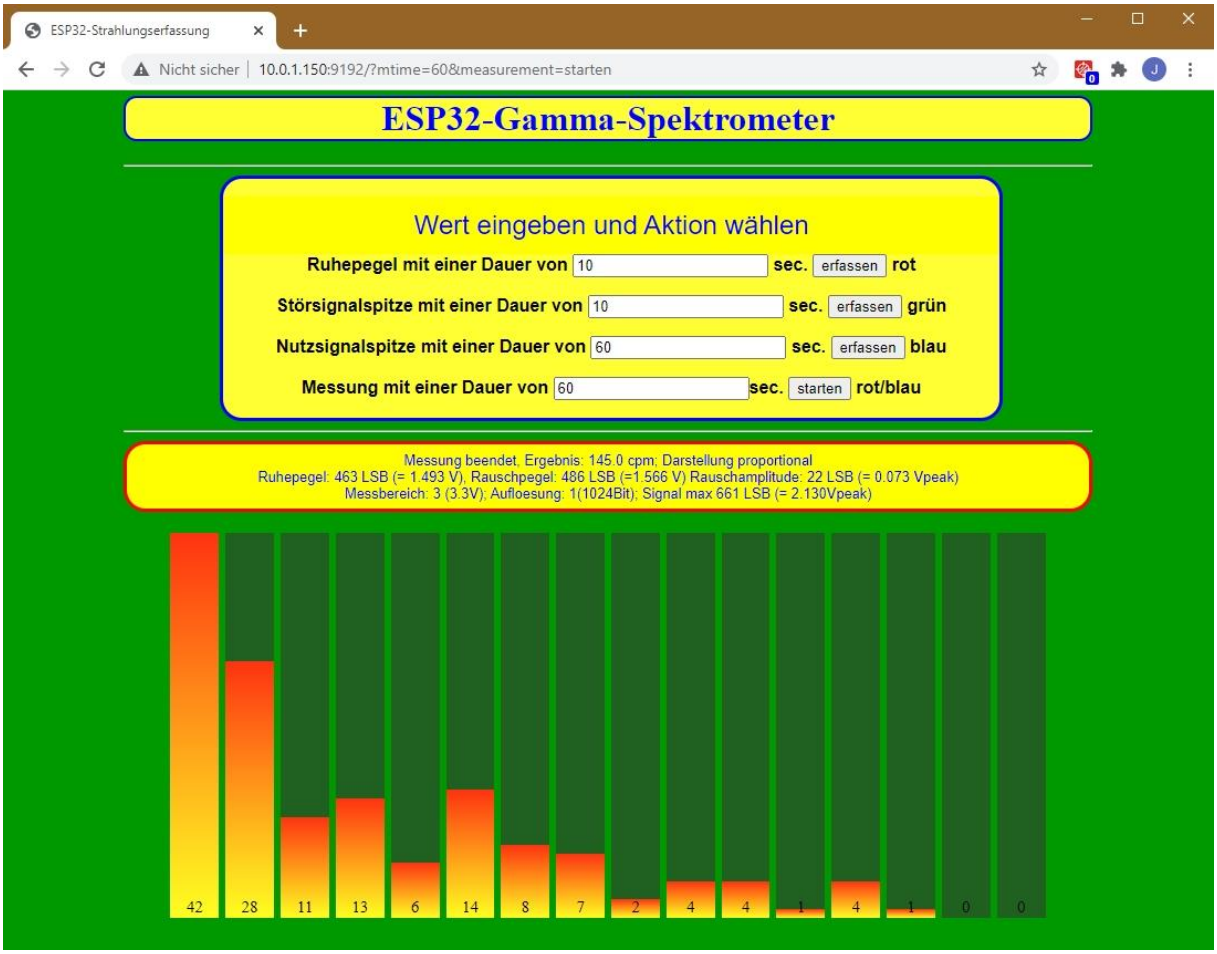
But there is also a short form, like this.

?start

In the first form, we are interested in the part up to the "&". There is no "&" in the second form, so these cases have to be parsed differently. In each case the leading "?" path. That does line 206. If the search for a "&" results in -1 (not found), then it is sufficient to remember the word after the question mark. Otherwise we divide the content from act up to the "&" into name and value.

The rest is simple and is repeated for the various commands that are converted into corresponding actions that the methods of the KS class have to execute. We already know the names of these methods from episode 5, where this type of measurement has already been carried out. Now we can change the measurement duration from the browser and start the individual actions with a click of the mouse.

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.
After the action has been completed, a string is returned in **aenderung** to the server loop and if a complete measurement has been carried out, the variable **ergebnis** is set to 1. The server loop now knows that the result of the measurement must be displayed as a bar chart. The illustration shows this status.



Two questions remain:

1. Whow ist he webside coded?
2. Who does the work at all?

The webinterface und CSS

I already described the basic structure of an HTML page in the second part of the blog. We are now expanding that using forms and CSS. A shape is an HTML element that uses text fields, check boxes, option buttons and action buttons, to name just a few, to make a web page interactive. CSS stands for "Cascading Style Sheets", a technique that allows a website to be conveniently formatted.

The web interface is assembled from four form tags, these are the four lines with the input fields in the upper frame. There is also the message window framed in red. The bar chart below is only displayed after a measurement.

I cut the block for a form out of the program context and removed the MicroPython code to explain how it works in the HTML document. These are lines 140 to 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>
```

The HTML code defines a form construct with a text input line and a submit button. The content of these fields is sent to the address http://10.0.1.150:9192 when you click the Submit button, specifying GET as the request method. The browser uses this to create the request, the syntax of which we already know.

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

We already know how this is received by the ESP32. But how can the browser prepare a simple text in a graphically appealing way? Even if each paragraph were to be provided with the appropriate formatting tags, what you see in the illustration above would not come out. The keyword here is Cascading Style Sheets or CSS for short. There are two things that you can do very easily with this tool. On the one hand, you can very effectively format the text of an entire site in a uniform and clear manner and, if necessary, change it centrally from one place, across all documents. CSS can be stored in its own files with the extension .css, thus making interventions in individual HTML documents superfluous. Seen in this way, CSS is for websites what classes in MicroPython are for program files. In addition, CSS also offers the option of integrating graphic elements into a website, as I did here with the frame and the bar chart.

The definition of a CSS is in the section between <head> and </head> and is enclosed in the tags <style> and </style>. In this frame, in lines 54 to 124, you will find the class definitions for the text formats and the progress bars, which I misuse as columns in my diagram. In addition to the class definitions that start with a "." are introduced, two of the definitions, body and h1, work element-related, that is, they are aligned with HTML tags and are always active when the corresponding tag appears in the HTML text, such as the h1 heading in line 135.

The CSS classes are activated in the opening tags for sections like and <p> with the keyword "class". You can see examples of text formatting in lines 137 and 138. You can find the use of formatting the columns for the diagram in lines 126, 128 and 130. To get a feeling for the effect of the various specifications, I suggest individual assignments to change and study the effects. This is particularly worthwhile with the progress bars, with the text formats the names and assignments are more obvious. While the field names are set similarly to the reserved words in MicroPython, the class names can be chosen by yourself.

I show the creation and formatting of the progress bars with a compact example. Then we split up the sample text so that values can be inserted dynamically. The <style> tag introduces the definition of a style sheet, </ style> completes the definition. We go from coarse to fine. The first paragraph defines the background color of a bar.

In the second paragraph we specify that it should be a vertical bar. The width and height as a percentage of the working area are specified. This is already reduced to 80% of the window width by the <body> tag.

In the third paragraph we assign the entire width of the background to the progress bar and let the columns grow from bottom to top with absolute.

In the fourth paragraph we add the progress-bar class with the filling, which should have a color gradient from yellow from below (0%) to red (above = 100%).

Finally, the tagging class describes how the bars should be labeled.

```
  <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>
```

In the <body> area, the middle of the last 5 lines creates the bar . The formatting is done by specifying the style classes. The string for this div area has to be separated where numerical values have to be inserted dynamically in an emergency. That's the 36 and 36% here. This is exactly what happens in lines 126, 128 and 130.

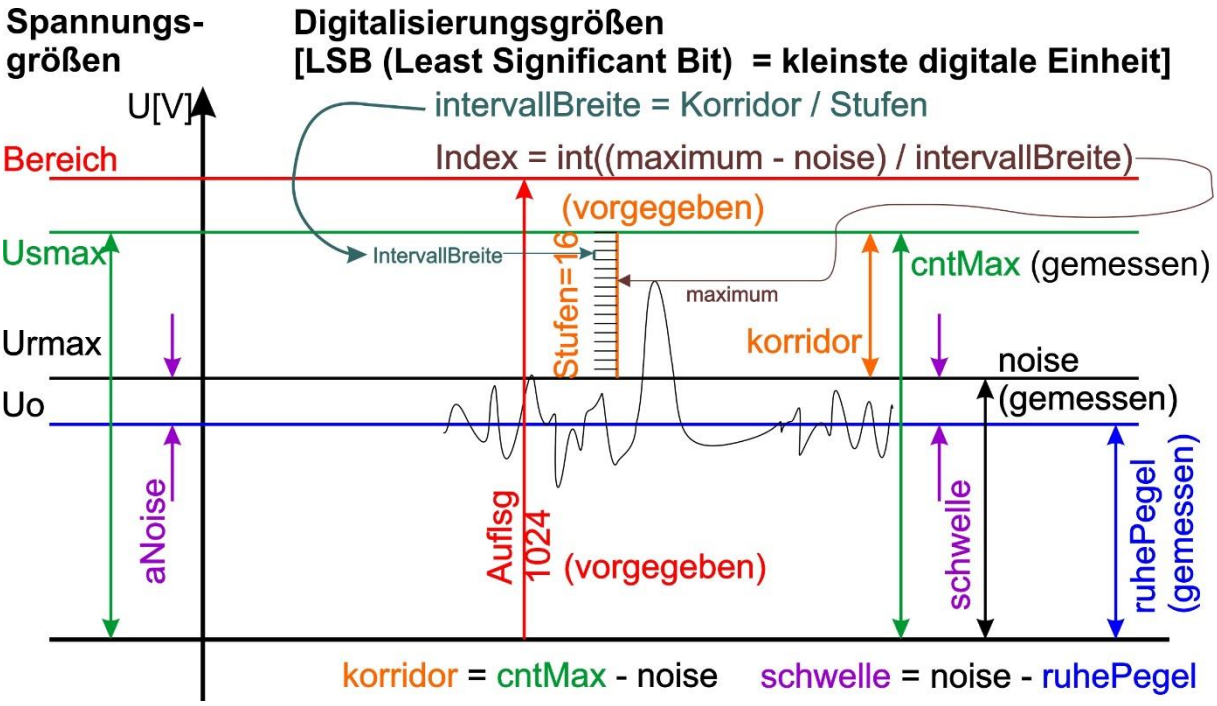
The KS class contains the methods and attributes from the 5th blog series that are necessary to carry out measurements with our nuclear radiation sensor. They are described in detail there. So that light and sound signals can be emitted, the KS constructor takes a BEEP object and an OLED instance and optionally the number of the analog input. The default value is GPIO34. The server.py program takes over the BEEP object b and the display object d from wifi_connect2.py. We simply pass both on as parameters to the KS instance k.

In addition to the essential program parts and a few simple helpers, we are using the file system of the ESP32 as home for a ks.ini file for the first time, in which the controller, after measuring one of the three changeable measurement specifications, restPegel, noisePegel and cntMax, their values and thus directly writes related sizes (writeConfig (), ks.py, line 201). This file is read in when the program starts (readConfig (), ks.py line 237). This means that the values are immediately available without having to be measured again. The basic measurements should only be repeated if other circumstances arise which could presumably change these quantities or if nonsensical results of a decay measurement occur.

The program snippet shows the method writeConfig (). The "wt" in the open statement means: write text data. When the process is finished, the file must be closed.

```
  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()
```

The assignment of the seven instance variables listed in it except for **auflsg** and **bereich** can be found in the following figure.



When reading in, reading problems are caught by try - except. If there are problems here, we recommend connecting the controller to a terminal in order to see the error message. If the import was successful, the file must be closed. Then the ADC attributes are set to range value and resolution and the interval width is calculated. showConfig () outputs the values on the terminal.

After a series of measurements has been started via the browser, the browser waits for the result of the KS module to be announced. The spectrum list now contains the count values of decay events for the respective energy level. The server script uses this list to set the bar height of the diagram, as I have already described above.

To test the application, start wifi_connect2.py from the editor window. After the connection to the access point in the WLAN has been established, the connection data is shown on the display for 3 seconds. Then the RGB-LED lights up yellow for 5 seconds, a countdown is running in the display and during this time you have the option of canceling the program run with the Yes button. If this does not happen, the server starts after 5 seconds at the latest and reports that it is ready to receive on the display. You can now start the gammameter with the URL http://10.0.1.150:9192/?start from any device via browser and then control its functions through the form fields.

If this test has passed with all parts of the control system without any problems, the organization of the autostart remains. In the root directory of the ESP32 there is a file boot.py. Copy this file into the WorkSpace of your IDE (µPyCraft or Thonny or ...). Rename this file via Windows Explorer to boot.org and push the renamed file back into the device via the IDE. This step is your "travel cancellation insurance" in the event that after the next step no connection to the ESP32 is possible via IDE. This is rare, but it can happen. Do not try to open boot.org in the IDE, it will not work from the WorkSpace or from the device. In the event of a disaster, it is possible to get back

into the system via Putty, rename the file as boot.py and thus enable an "open" start in the IDE.

For an autostart, rename wifi_connect2.py in the WorkSpace via the Explorer to boot.py. Slide boot.py onto the device. The file server.py should already be located there and so the server should restart after the next restart of the ESP32 after logging on to the network. From now on the server runs autonomously and can be controlled via the web interface. A local network with wireless router is still required for this solution. The next chapter shows that there is another way.

The ESP32 as accesspoint

If you have now nicely packed all the parts of the project setup in a housing and are out and about in the pampas, for example to take measurements on minerals, then you probably rarely have a WLAN at hand through which you can contact your ESP32. And in fact, you can do it without a router and PC. If you let your ESP32 play access point yourself, you can connect to it from your mobile phone and control the gammameter via the browser as usual.

I simply copied the existing files wifi_connect2.py and server.py to accesspoint2.py and server2.py and changed both slightly in some places.

For the provision of an access point functionality by the ESP32, in the last fifth of wifi_connect2.py part of the program text was deleted or replaced by the following lines.

```
# ************************* 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)
```

The bold characters are important for the function as an access point, the rest has either not been changed or is just waste.

There is actually no need to write down a password, as my ESP32 only offers the authentication mode 0 = Open from the outset and you cannot be persuaded to accept another mode.

The server.py file has received changes in more places. This has mainly to do with the fact that I moved the access point with the IP 10.0.2.100 to another subnet, 10.0.2.0/24. The 24 is a short form of the network mask 255.255.255.0. Since the ifconfig instruction requires a 4-tuple as a parameter, I have also specified the address of the access point in the 3rd and 4th positions for the gateway address and the DNS server. Although neither is used, a valid IP must be specified, 0.0.0.0 is not accepted. Because another server file has to be reloaded, the name in the exec statement in accesspoint2.py also had to be adapted.

In the server2.py server script, the IP addresses for the action field in the bodyX strings have already been changed, from 10.0.1.150:9192 to 10.0.2.100:80. Nothing had to be changed in the web_page () function, only the variable portNum immediately after 9192 from 80. The outputs on the OLED display were reorganized. Everything stayed the same within the server loop.

In the test run, accesspoint2.py is started in the editor window, which then reloads server2.py if it is not canceled at the checkpoint.

A message in the following form should then appear in the terminal.

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

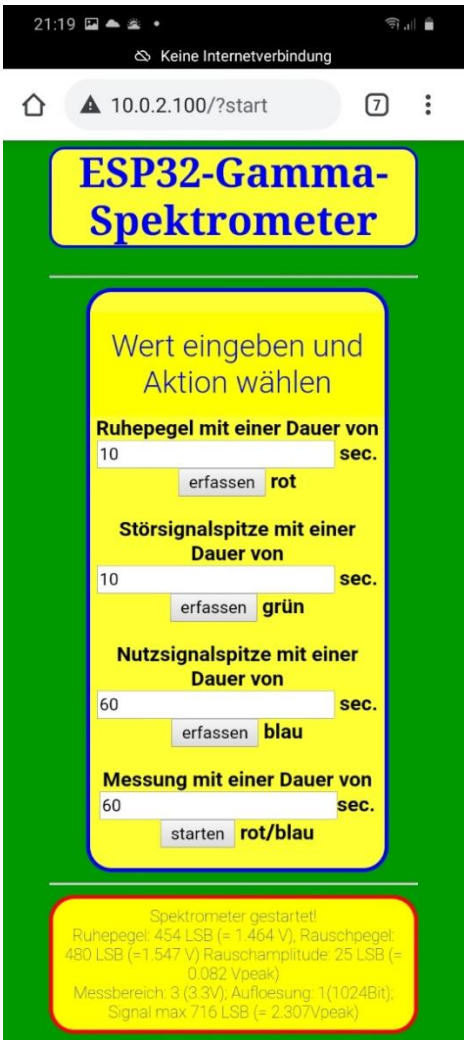
cntMax: 710
Korridor: 230
Intervallbreite: 42.73307

Empfange Anfragen auf 10.0.2.100:80

To test the application on the access point, start accesspoint2.py from the editor window. After the internal structure of the function as an access point without WLAN, the contact details are shown on the display for 3 seconds. Then the RGB-LED lights up yellow for 5 seconds, a countdown runs in the display and during this time you have the option of canceling the program run with the Yes button. If this does not happen, the server starts after 5 seconds at the latest and reports that it is ready to receive on the display. You can now start the gammameter with the URL http://10.0.2.100/?start from any end device via browser and then control its functions with the form buttons.

To do this, go to Settings - Connections - WiFi on your mobile phone. Let the device search for new access points or enter the SSID gammameter directly (depending on the possibility). Then connect your mobile phone to the access point of the ESP32. You will not have internet access via this connection, but you can talk to the ESP32.

Then start the gammameter with the URL http://10.0.2.100/?start and the respective action via the form buttons. The time values can simply be accepted or re-entered.



The system is now running as an isolated solution, without access to a local network but not yet autonomous. Without the first access via the URL mentioned, there is no access to the measuring device because the interface is not displayed. Our server eats every access that does not correspond to the parameter data in the URL line. Using appropriate handshake methods, additional protection of the control can be achieved even without authentication via the access point.

If this test has been carried out with all parts of the control without problems, the organization of the autostart remains here as well. In the root directory of the ESP32 there is a file boot.py. Copy this file into the WorkSpace of your IDE (µPyCraft or Thonny or ...). Rename this file via Windows Explorer to boot.org and push the renamed file back into the device via the IDE. This step is your "travel cancellation insurance" in the event that after the next step no connection to the ESP32 is possible via IDE. This is rare, but it can happen. Do not try to open boot.org in the IDE, it will not work. In the event of a GAU, however, there is the possibility of getting back into the system via Putty, deleting the boot.py file and renaming boot.org to boot.py, thus enabling an "open" start in the IDE

For an autostart rename accesspoint2.py in the WorkSpace to boot.py. The file server2, py should already be on the device and so the server should restart after the next restart of the ESP32 after the on-board access point has been set up. From now on, the server runs autonomously and can be controlled via the web interface after the smartphone has registered at the access point of the ESP32, as described above.

Unfortunately, the ESP8266 has far too little RAM memory available for this project, I regretted that at the beginning. When loading the server program, MicroPython finds that almost 10000 bytes of RAM are missing. So it is worth buying the bigger brother or you will be satisfied with the lean solution from blog episode 5.

Threading on the ESP32 basically works in the beta approach under MicroPython. Unfortunately not in connection with WiFi or sockets. therefore you should not specify more as the measurement time than the web browser is inclined to wait for a response from the server. It differs from system to system, give it a try. With 60 seconds (I have already tested 300 seconds with success) you are on the safe side. At the moment I'm still working on an Android app that does better control and display without a browser. Let us surprise!

Wrapping up

With what you learned in the six episodes, you are well equipped to tackle your own projects. You can use the selection from different programming software, know the basic types of variables, have an insight into serial data structures such as lists and can operate GPIO pins or read in analog signals. With modules and classes you can structure program parts and use them profitably in other projects by importing or even inheriting them. You know how to create functions, which parameters are space-bound or optional, what local and global variables are, and how to use them. You can also save data in files on the ESP32 / ESP8266 and read them in again. This list is certainly not exhaustive, but it shows what MicroPython can do if you can handle it. I can promise you one thing, there are still many, many more interesting features of this programming language waiting to be discovered by you.

Certainly there will be more articles in the future that delve deeper into the interesting subject matter of MicroPython.

See you next time!

Links to the earlier parts:

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