

## MicroPython with ESP32/ESP8266 – Part 5

---

Today the secret of what the project is about after all the preparations will be revealed. We build a circuit board to measure nuclear radiation and then work on the topic `esptool.py`. Discussing the homework brings a lot of additional know-how. So welcome to the fifth part of the series.

Over 100 years ago, Marie Skłodowska-Curie published her doctoral thesis entitled "RECHERCHES SUR LES SUBSTANCES RADIOACTIVES" "Investigations into radioactive substances". Since 1903 the methods of measuring nuclear radiation have changed and improved enormously. The measuring device designed by Messrs. Geiger and Müller works with a special tube and with voltages between 300V and 500V. The circuit I am describing today is content with 5V.

Of the three types of radiation, alpha, beta, and gamma radiation, which come from the interior of the atomic nucleus, our structure can detect the latter type quite well, because it is an electromagnetic wave, similar to visible light. So what could be more obvious than using a photodiode as a sensor. Gamma radiation can penetrate the plastic layer of our sensor like light, which the other two types cannot. Alpha radiation consists of fast helium nuclei and can be stopped with a sheet of paper. The fast electrons of the beta radiation get stuck in 1mm aluminum or plastic.

Our sensor is a PIN diode. The 500 to 1000 nm thick, weakly negatively doped intrinsic region is strongly positive on one side and strongly negatively doped on the other. Light can enter through the thin p-doped side (approx. 35 nm), which can release charge carriers in the broad intrinsic range, which are accelerated in the applied electrical field, which leads to a current pulse. In contrast to normal PN diodes, the broad intrinsic range means that there is a higher probability that charge separation will be caused by light quanta. In our case, this task falls to the gamma quanta of the nuclear radiation.

So that the much stronger surrounding daylight does not come into play and all weak gamma signals are superimposed, the measurement must take place in the dark. That is one reason why the sensor part of the circuit has to be put into a well-closable tin can. On the other hand, the can simply serves as an electrical shield so that the "normal" electrosmog around us cannot disturb the sensitive circuitry. That's why it has to be a tin can, plastic shields light but does not shield electrical fields. More on that later.

We build the sensor for today's experiments from individual parts on a breadboard. The following material is needed for this.

1	LM 358 DIP Operationsverstärker, 2-fach, DIP-8
1	36pol. Stiftleiste, gerade, RM 2,54
1	Buchsenleiste, 20-polig, einreihig, RM 2,54, gerade
1	BF 256B N-Kanal J-FET, 30V, 13mA, 350mW, TO-92
1	BPW 34 Silizium-PIN-Fotodiode, 50µA / 430...1100nm
1	100 Ohm Widerstand, Metallschicht
1	1,00K Widerstand, Metallschicht
1	4,7K Widerstand, Kohleschicht
2	10,0K Widerstand, Metallschicht
1	15,0K Widerstand, Metallschicht
1	33k Widerstand, Metallschicht
2	330K Widerstand, Metallschicht
1	1,00M Widerstand, Metallschicht
2	10M Widerstand, Kohleschicht
3	100N Vielschicht-Keramikkondensator
2	KERKO 47P Keramik-Kondensator 47P
1	33N Vielschicht-Keramikkondensator
2	Elko, radial, 100 µF, 25 V
1	<a href="#">PCB Board Set Lochrasterplatte</a>
1	Blechdose mit Blechdeckel ca. 9cm Ø und 4cm hoch, lichtdicht
1	Schraube M3 mit Beilage und Mutter
	etwas Schaltdraht

- 1 LM 358 DIP operational amplifier, 2-way, DIP-8
- 1 36pol. Pin header, straight, pitch 2.54
- 1 socket strip, 20-pin, single row, RM 2.54, straight
- 1 BF 256B N-channel J-FET, 30V, 13mA, 350mW, TO-92
- 1 BPW 34 silicon PIN photodiode, 50µA / 430 ... 1100nm
- 1 100 ohm resistor, metal layer

- 1 1.00K resistor, metal layer
- 1 4.7K resistor, coal layer
- 2 10.0K resistor, metal layer
- 1 15.0K resistor, metal layer
- 1 33k resistor, metal layer
- 2 330K resistor, metal layer
- 1 1.00M resistor, metal layer
- 2 10M resistor, carbon layer
- 3 100N multilayer ceramic capacitor
- 2 KERKO 47P ceramic capacitor 47P
- 1 33N multilayer ceramic capacitor
- 2 electrolytic capacitors, radial, 100 µF, 25 V
- 1 PCB board set breadboard
- 1 tin can with sheet metal lid approx. 9cm Ø and 4cm high, light-tight
- 1 M3 screw with washer and nut some hookup wire

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

1	<a href="#">ESP32 NodeMCU Module WLAN WiFi Development Board</a> oder
1	<a href="#">ESP-32 Dev Kit C V4</a> or
1	<a href="#">NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F mit CH340</a>
1	<a href="#">0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi</a> oder
1	<a href="#">0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi</a>
1	<a href="#">KY-012 Buzzer Modul aktiv</a>
2	LED and
2	Resistor 330 ohm for LED or
1	<a href="#">KY-011 Bi-Color LED Modul 5mm</a> und
2	Widerstand 560 Ohm für LED or
1	<a href="#">KY-009 RGB LED SMD Modul</a> and
1	Resistor 330 Ohm for blue LED
1	Resistor 2,2 kOhm for red LED
1	Resistor 3,3 kOhm for green LED
1	<a href="#">KY-004 Taster Modul</a> or
1	<a href="#">keypad-tp224-14-kapazitiv</a>
2	<a href="#">Mini Breadboard 400 Pin mit 4 Stromschienen</a>
1	<a href="#">Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F</a>
2	Sheet metal pieces approx. 20 x 20 mm (not aluminum!) Or remnants of circuit boards
	some plug pins 0,6x0,6x12mm

**The following tool is helpful.**

- Soldering iron or station, fine tip,
- tin solder small side cutter
- small screwdriver
- 3mm drill (manual operation or minimot)
- Hand saw with metal blade
- possibly a "third hand" to hold the board while soldering
- possibly digital voltmeter
- small flat file or a piece of sandpaper grit 120-150



## Structure of the sensor and amplifier board

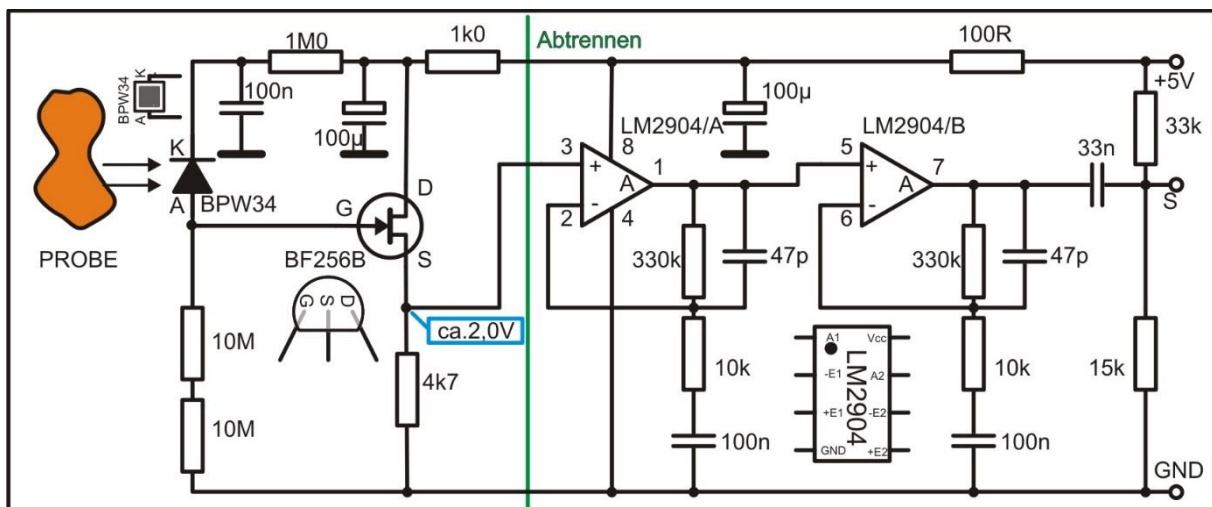
For my construction I used a breadboard from this set PCB Board Set breadboard. If you don't want to go into series production, a breadboard structure is sufficient.

The columns and lines of the circuit board are provided with letters and numbers so that the position of the component connections can be easily understood and checked. I wanted to know and actually managed to build both circuit parts on one of the narrow strips of the set. If you're not that sure about soldering, use 2 strips, maybe even the wider ones. Of course, the original drawing is then unfortunately no longer correct.



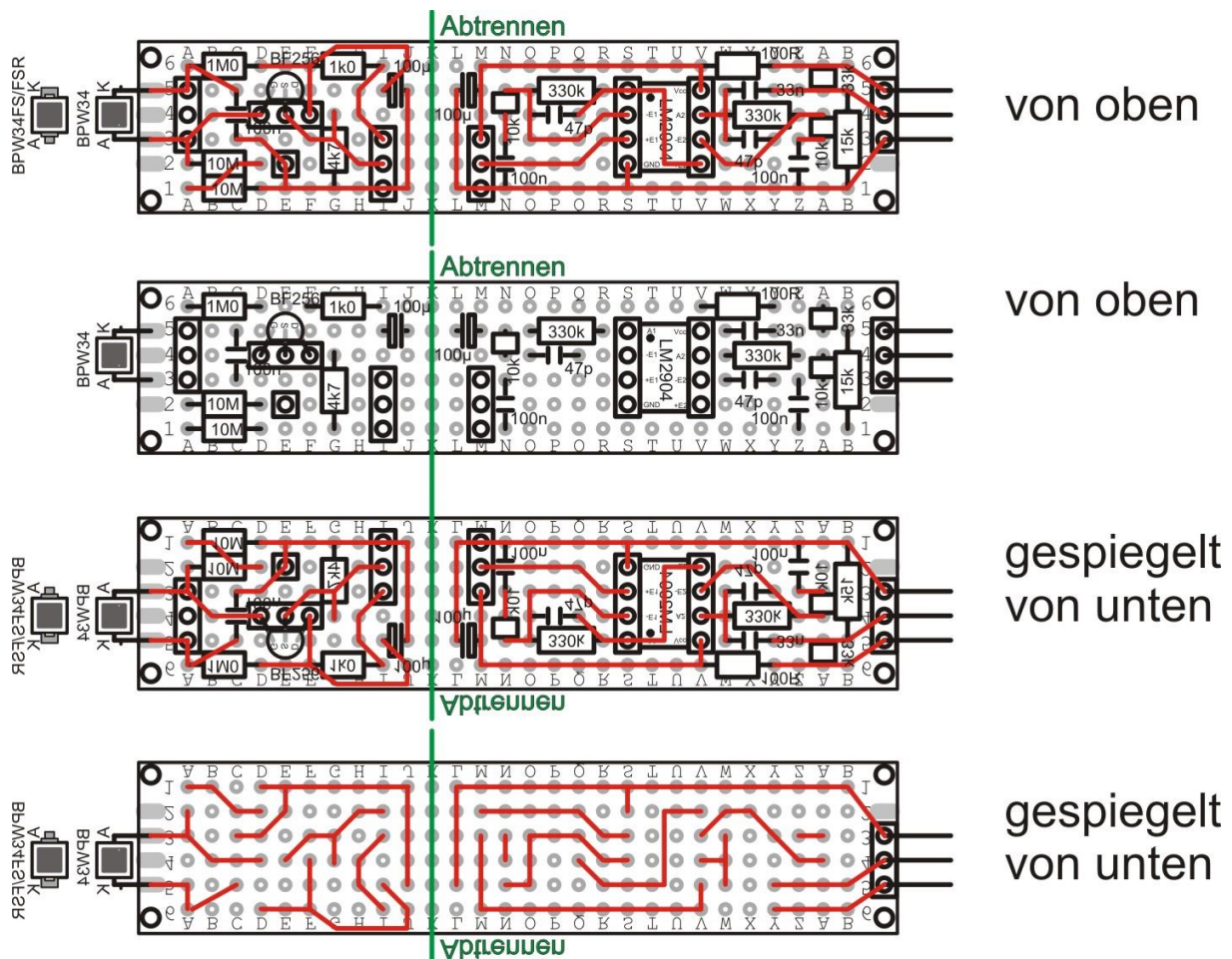
The illustration shows the top of the board. The letters then run in alphabetical order from left to right, the numbers from bottom to top. The back is labeled in the same way so that, for example, hole A01 is the same when viewed from both sides.

There are different approaches on the Internet, all of which ultimately have the same goal of reducing the short and weak voltage signals of the PIN diode to a duration and magnitude that can be evaluated by the microcontroller. This can be done purely digitally, in that the captured decay product is translated into a digital pulse of a defined length. I found the basis for the circuit used here in Elektor years ago. I added a high-pass filter to the output of the amplifier, which suppresses low-frequency noise up to 300Hz and significantly improves the useful signal. The 33kΩ resistor is used to increase the level so that no negative voltages can occur at the ESP32 / ESP8266 analog input. Here is the schematic of the circuit.



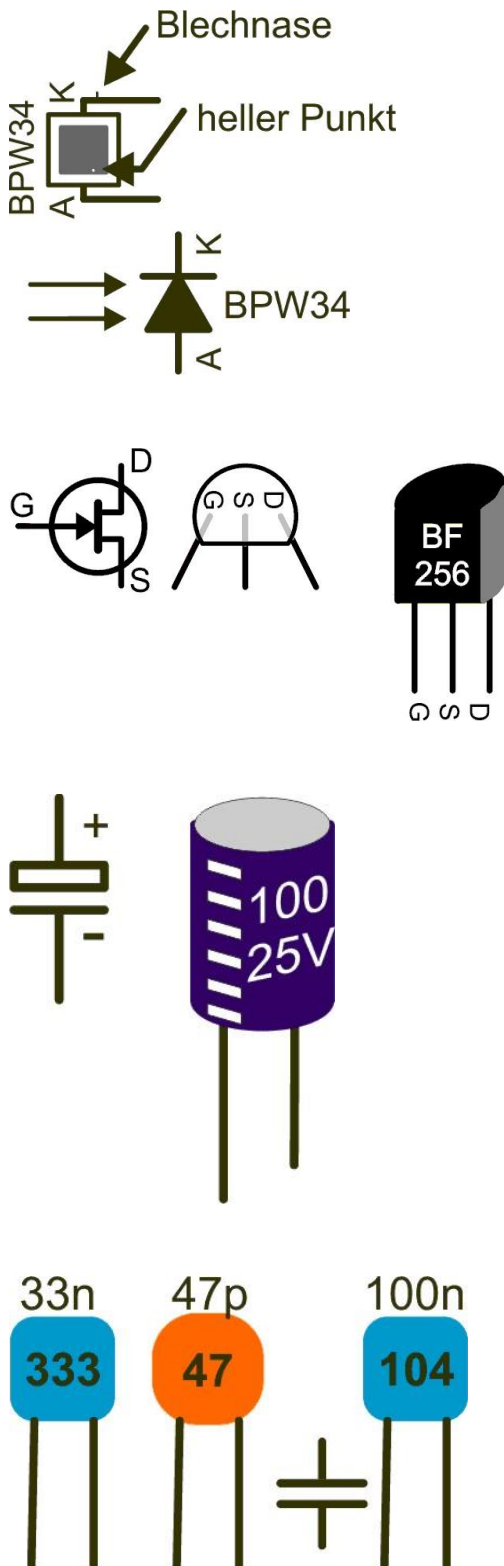
The implementation of the circuit on the board is shown in the next figure. In the top picture you look from above at the components and circuit board. The conductor tracks lie as if the circuit board were transparent. In the second picture, same direction of view, the conductor tracks have been left out so that the labeling of the components can be read better.

The equipment is always from above. Check carefully the position and alignment of the components. Photodiode BPW34, transistor BF256 and the electrolytic capacitors must be inserted the right way round according to their polarity. If you start from the left, you can also follow your progress on the schematic. The coordinates with letters and numbers will help you.

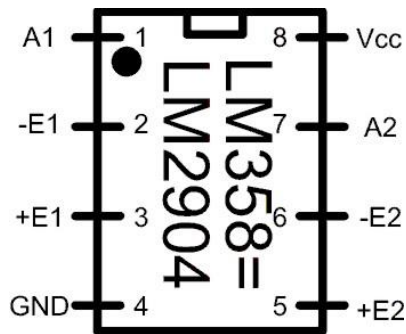
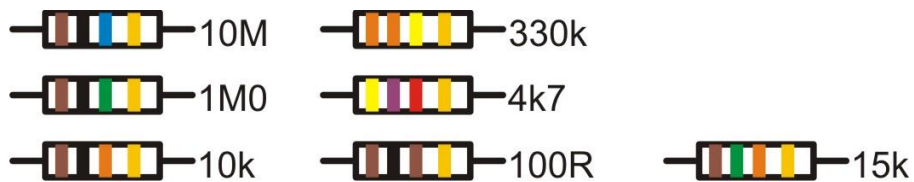


For the photodiode, the transistor and the IC LM2904 = LM358, parts of the socket strip are used as a socket, which makes it easier to replace them. Parts of the pin header are used for the plug connections from board to board and to the ESP32. The wiring to the amplifier itself can then be done with jumper cables.

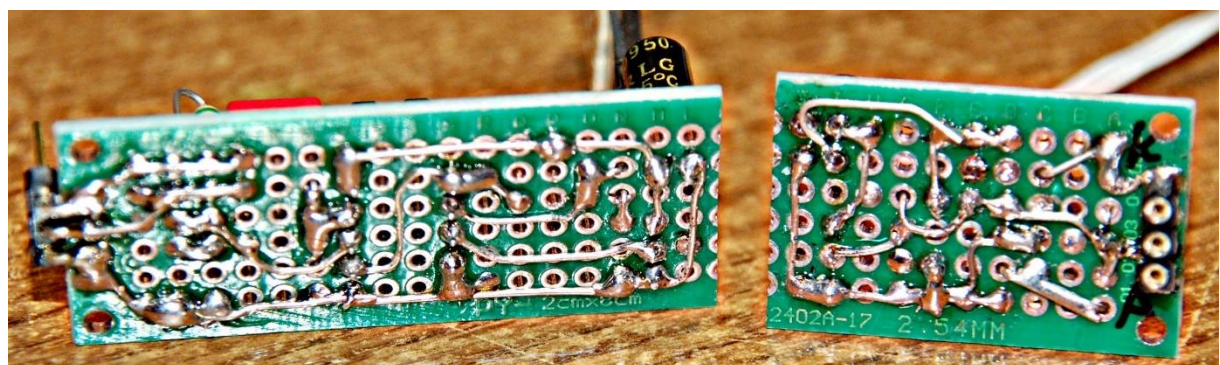
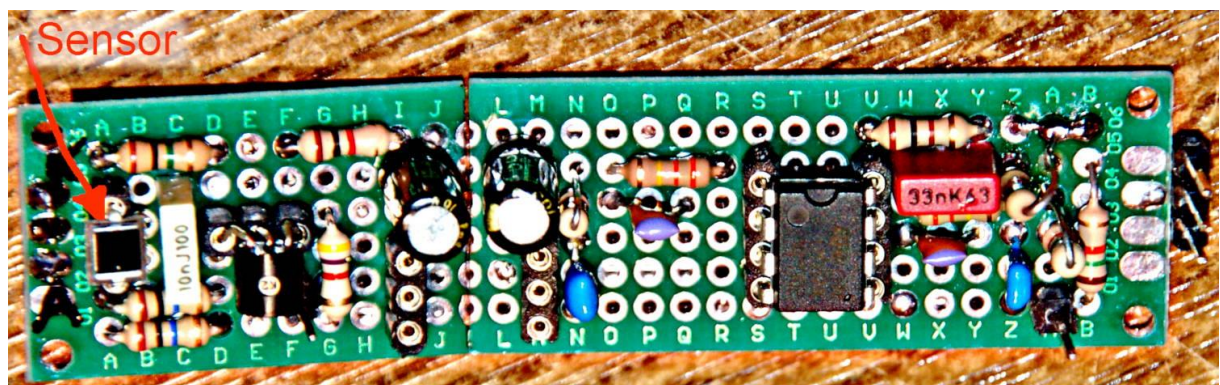
The following illustrations should help you to identify the components and their location.



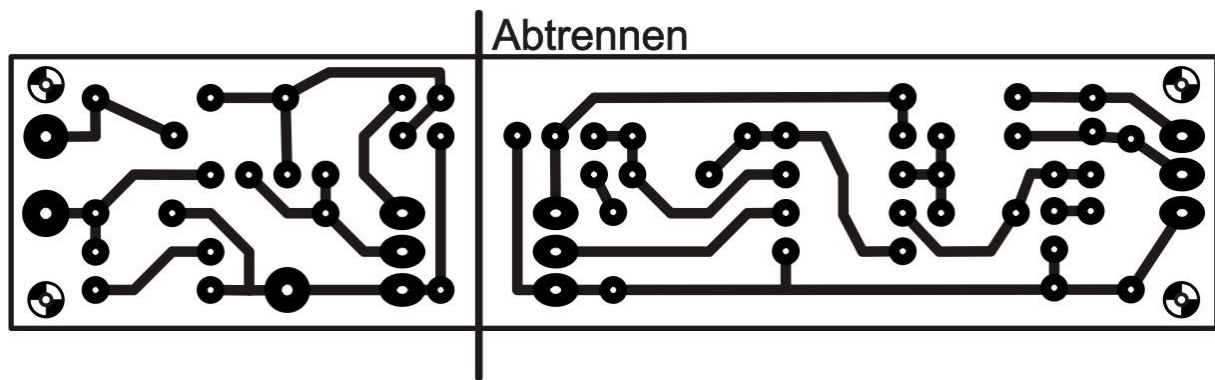




After the complete assembly, the board is separated at column K. Then clean the edges with the help of the sandpaper. It should look something like this. (The rows of sockets at I and M were later replaced by rows of pins. The transistor is bent down.)



For experts who can produce circuit boards themselves, I also have this layout, which can be [downloaded as a PDF file](#) with the assembly plan. I have described the production of circuit boards by the ironing method [in more detail here](#).



The finished circuit does not need to be adjusted and should do its job right from the start. In order for it to do that, the inside of the can must be lined with a non-conductive material to prevent short circuits. I used thick cling film and attached it to the floor and wall with adhesive pads.

I soldered a pin of the pin header with some jumper wire and then fixed it to the wall of the socket with the screw. The connection with the sensor circuit to the pin at position F1 is established using a jumper cable. This ground connection is absolutely necessary to suppress the circuit's tendency to oscillate. The board is now attached to the floor, that the sensor can be easily approached to a radiation source. I will come to what can serve this purpose in a moment. For the connection to the amplifier board you need three jumper cables fm - fm for Vcc, signal and GND. If the cables can be led out between the lid and the can, this is good, if not, they have to be fed through a hole in the side wall. Of course you have to glue the hole light-tight again.

## Samples for the radiation meter

Let's get to the rehearsals. Since hardly anyone has a castor in their basement that they can use to measure radiation, we have to look around for simple everyday objects that emit nuclear radiation.

In this context I would like to point out that the measured values are in no way related to the SI system of sizes. It is therefore not possible to make statements about the absorbed dose  $D$  in Gray (Gy) or the equivalent dose  $H = q \cdot D$  in Sievert (Sv) with this simple device, as is claimed in the electronic article. Even a specification in Becquerel (Bq) is not possible because the mass and type of the emitter would have to be known. With the samples listed here, neither can be determined. This article is therefore limited to the specification cpm = counts per minute = decay rates per minute and the representation of relative energy values.

Here are a few photos of items that could be used. You can find a more [comprehensive presentation here](#)

The orange uranium glaze on the can is a good test source, as is the black print on the cup or the underglaze color of the cat. Much less effective is uranium glass, which usually comes in green or yellow, but can also have other colors. One of the main properties of uranium glass is that it fluoresces green when exposed to UV light. The potassium content of the mica minerals brings only weak gamma signals. On the other hand, the radium content of the luminous paint on the hands and dials of old



clocks can be easily traced. If you take parts out of it, you should pack them in an airtight plastic sleeve for experimentation so that dust or small particles cannot be inhaled or swallowed. Therefore, one should not eat, drink or smoke during the experiments with the parts of clocks. And then wash your hands!

The following things can often be found at flea markets.

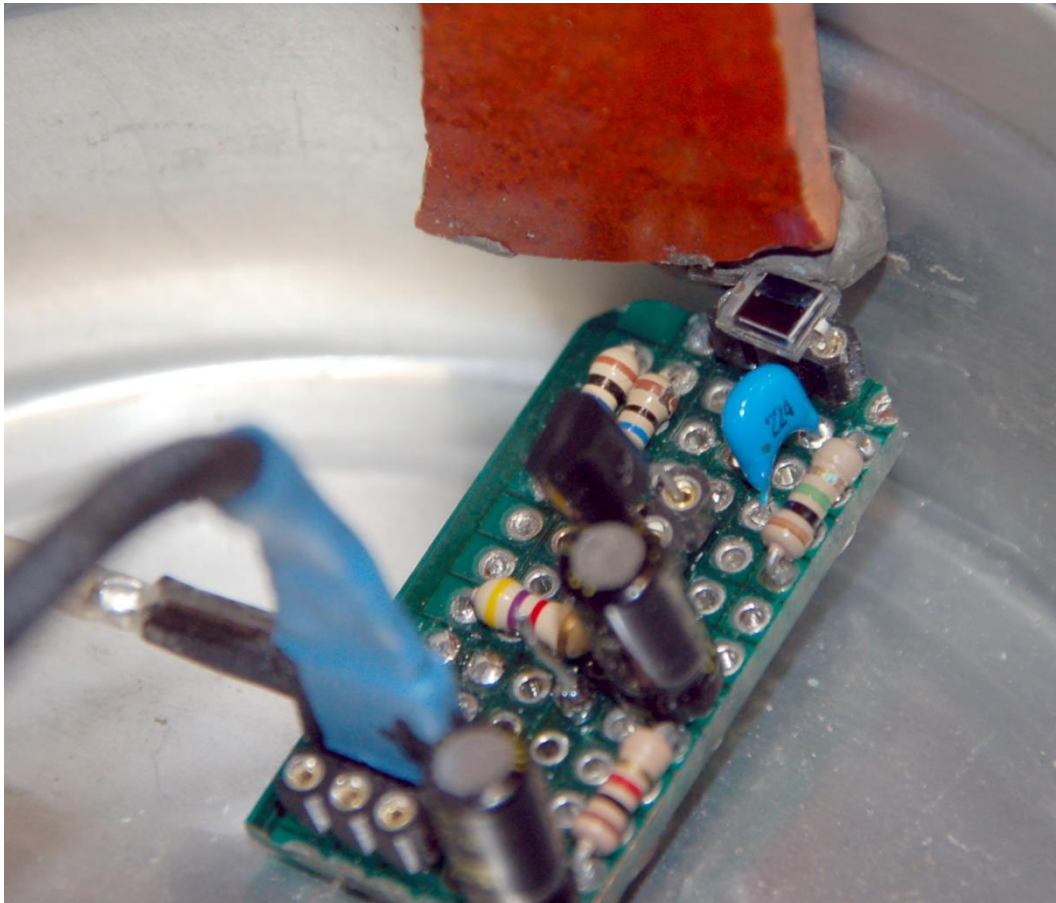




If you don't have any of these at home, you can try potash (potassium carbonate) or diet salt (potassium chloride) from the nearest supermarket. Natural potassium contains 0.0117% of the radioactive isotope potassium-40 ( $K^{40}$ ). About 90% of this is converted into calcium-40 with beta decay. The remaining approx. 10% of the decay



events are based on the capture of an electron from the K shell by the atomic nucleus and the subsequent emission of a gamma quantum. This creates argon40. Our sensor detects such a process approximately every one to two minutes, which is also due to the very small sensor area of just 7mm<sup>2</sup>. The result is a voltage pulse of up to 300mV at the amplifier output, which goes well above the background noise. Uranium glazes deliver pulses that can be more than twice as high. The uranium glaze fragment in the photo is folded down onto the sensor for measurement.



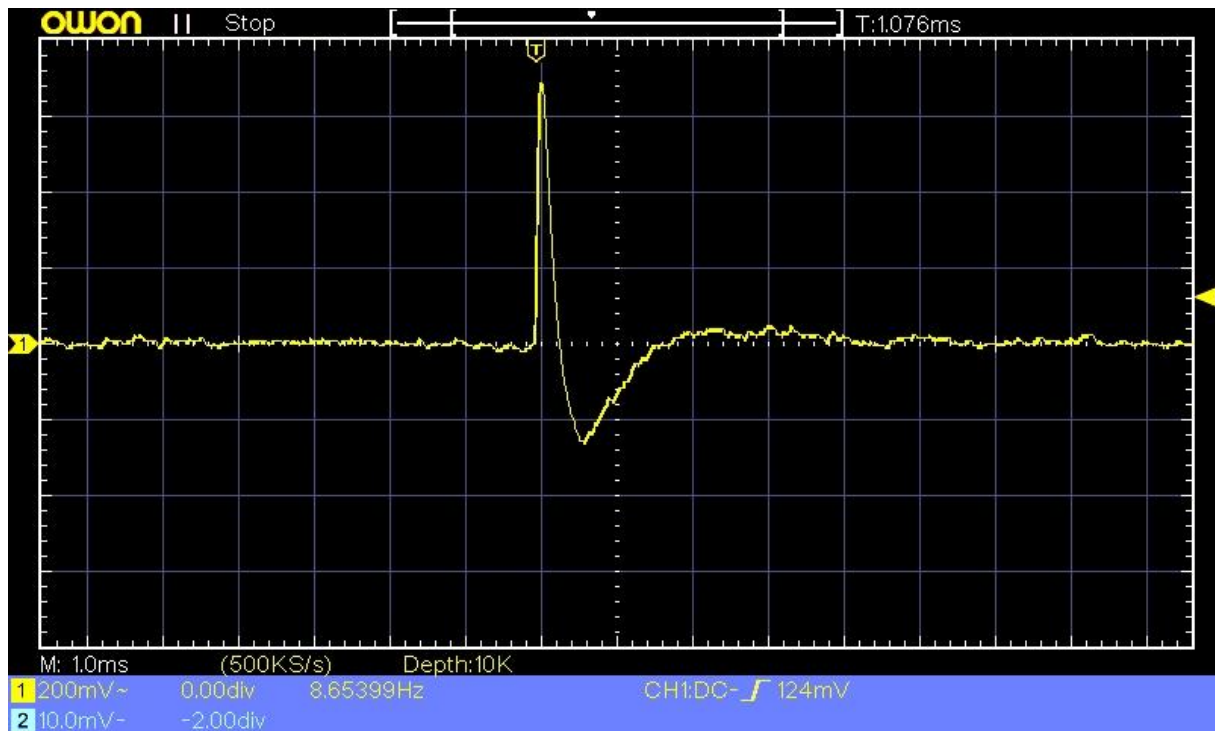
Basic conditions for the measurements are

- If possible, interference-free power supply of 5V
- Isolate the sensor from ambient light in the can
- Closing the lid with a conductive connection to the socket
- Silence - every little scratching on the table or normal speaking leads to interference at the amplifier output.

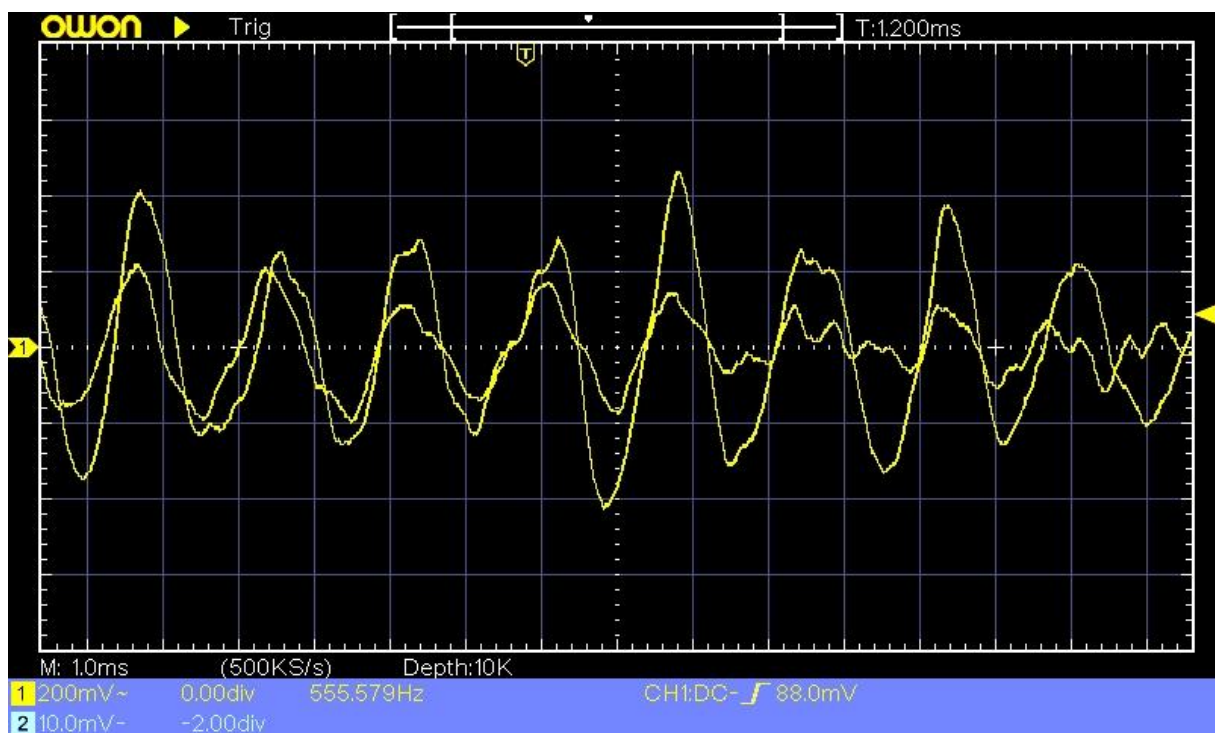
The last point is easy to overlook. As a side effect, the can acts as an extremely sensitive condenser microphone. Slightly scratching the table results in signals at the output that can be three to four times higher than the useful signal.

It is of course ideal if you have an oscilloscope at hand, because you can use it to control the signal situation at the amplifier output. The very strong pulse with 700mV in the next picture comes from a piece of uranium glaze directly in front of the sensor window. The signals are a little less if the sample is placed on the outside of the can lid instead of directly in front of the sensor in the can. Then the aluminum has to be penetrated and the approx. 3 cm air in the can up to the sensor. But the sensor still responds.





Scratching the table looks like this.



## Programming and measuring

Now we come to the measurements. What we need is a program that monitors the level at the amplifier output in order to determine its maximum as precisely as possible when a voltage pulse occurs. For us, the maximum voltage value is a

relative measure of the energy of the triggering gamma quantum released in the intrinsic zone. Because, because of the thin layer, it cannot be guaranteed that the entire gamma energy for charge separation was absorbed, the size of the voltage pulse does not necessarily reflect the total energy of the gamma quantum, but only the absorbed portion.

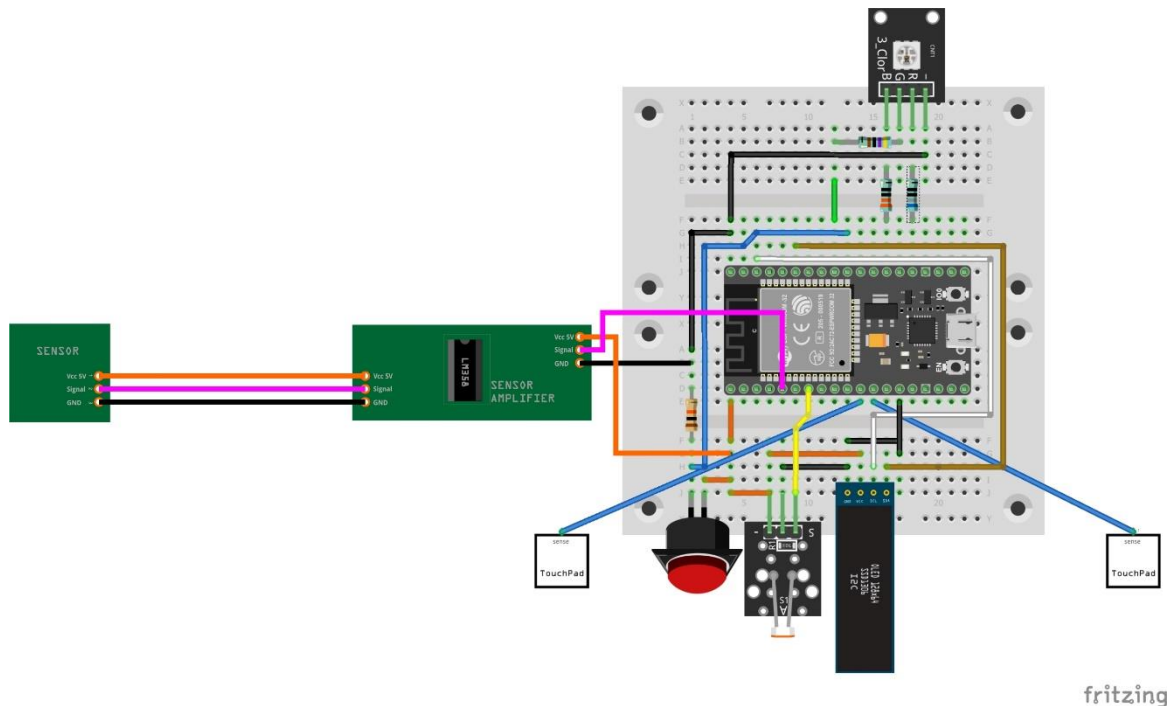
We cannot measure absolute energy values with the device anyway, because it is not calibrated, we simply lack the reliable comparison between the cause (gamma energy) and the effect (voltage pulse). What we can do, however, is a breakdown of the frequency with which certain absorbed portions of energy occur. With the help of the OLED display we can create a "gamma spectrum" of the radiation source. Ah, now you can say for sure, we needed the bar charts for that, and you are right. You probably have suspicions when it comes to speed measurement with the ADC. I can confirm that too, because that's where we start.

From the pulse diagram above, you can see that the positive pulse lasts about 0.3 to 0.4 milliseconds. The ESP32 has to keep up with about 13 measurements per millisecond to get the peak value. For an ESP8266, with 4 to 5 measurements in this time, it is more of a coincidence if it gets exactly the peak value. Such considerations are very important for the selection of a controller for time-critical tasks and that clearly speaks in favor of the ESP32. Tests have shown, however, that the ESP8266-12F still produces good results.

Next point, when should the measurement of a pulse begin? Clear answer, when the voltage level exceeds the noise floor. We have to determine this value, noise, of the basic noise, as well as the quiescent level  $r_{\text{uhePegel}}$  and the maximum expected value  $\text{cntMax}$ . I have added a graphic below that shows the interaction of the various levels.

The circuits for ESP32 and ESP8266 look different. I'll start with the ESP32.

## The ESP32 as a measurement servant



I apply the voltage at the amplifier output of the sensor unit to pin 34 of the ESP32. I set the bit width of the ADC to 10bit and as the voltage range I choose 3.3V to start with. I put everything together in the test program [ruhepegel.py](#). The sample is removed from the can, then I start the measurement, which determines the average ADC level at the amplifier output for 10 seconds. Make a note of this value, we still need it.

Next, the amplitude of the noise signal is important. The [maximum.py](#) program determines the absolute peak value on the signal line. If there is no sample in the can, this provides the maximum transducer value of the noise of the glitches caused by the noise. We note down the associated ADC value maximum. The maximum noise signal in ADC units (LSB) is therefore  $\text{noise} = \text{maximum}$ .

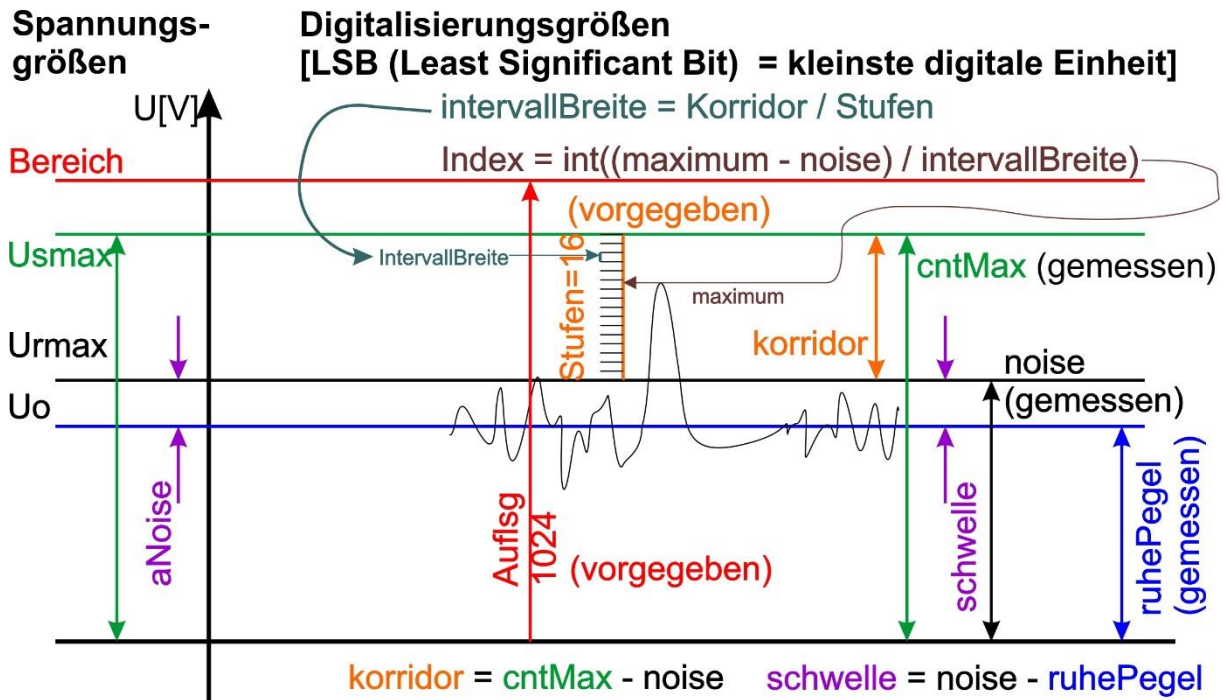
With the same program [maximum.py](#) we now determine the maximum ADC value, `cntMax`, which is created by the pulses from the radiation sensor. That tells us which voltage range we need to set for the measurements. We place the sample in the can on the sensor and start the measurement for at least 60 seconds. We also note the ADC value `cntMax`.

My values were quiet level = 449, noise = 500 and `cntMax` = 713. This corresponds to  $U_0 = 449/1023 * 3.3V = 1.45V$  quiet level. The noise amplitude, noise - quiet level, is 51 LSB, which corresponds to 164 mV. Everything that goes beyond that can be seen as a useful signal. With 713 LSB we calculate the maximum voltage level to be 2.3V. That is more than 2.0V and that means that we have to continue to work with the already set range value 11dB in order not to overload the ADC input.



The following graphic illustrates the relationships. The voltages are only of interest for estimating the input voltage range of the ESP32. Otherwise we limit ourselves to the converter values, because we are only concerned with relative ADC values and not with absolute voltage values.

The three values that have been determined so far - restPegel, noise and cntMax - are now transferred to the actual measuring program, measure.py, in order to serve there as default values for the moment. The following graphic provides an overview of the various values.



Now let's take a look at the programs. The import of the necessary modules, the instantiation and assignment of the start values, the preparation of the time control as well as the evaluation of the measurements represent the greatest extent in ruhepegel.py and maximum.py. The essentials happen in the while loops with one or two command lines .

Download [ruhepegel.py](#)

```
# ruhepegel.py
from machine import ADC, Pin
from time import time, sleep
ksPin = 34
Auflsg = ADC.WIDTH_10BIT
Bereich = ADC.ATTN_11DB

led = Pin(2, Pin.OUT)
sleep(3)
led.on()
rad = ADC(Pin(ksPin))
rad.atten(Bereich) # Messbereich: 3.3V
rad.width(Auflsg) # Aufloesung 512 Bit
ruhe = 0
n = 0
laufZeit = 10
current = time()
start = current
end = current + laufZeit
while current <= end:
    ruhe = ruhe + rad.read()
    current = time()
    n += 1
led.off()
print ("Ruhepegel =", ruhe/n)
print ("Messungen:", n)
print ("Messungen {} / ms".format(n/(laufZeit*1000)))
```

For the quiescent level, all measured values are added up and then divided by the number of measurements for the display. According to the rules of statistics, if there are a large number of measurements, this results in the rest level as the mean value.

Download [maximum.py](#)

```
# maximum
from machine import ADC, Pin
from time import time, sleep
ksPin = 34
Auflsg = ADC.WIDTH_10BIT
Bereich = ADC.ATTN_11DB

led = Pin(2,Pin.OUT)
sleep(3)
led.on()
rad = ADC(Pin(ksPin))
rad.atten(Bereich) # Messbereich: 3.3V
rad.width(Auflsg) # Aufloesung 512 Bit
ruhe = 449
maximum = 0
n = 0
laufZeit = 60
current = time()
start = current
end = current+laufZeit
while current <= end:
    pegel=rad.read()
    if pegel > maximum: maximum = pegel
    current = time()
    n += 1
led.off()
print ("maximaler Pegel =",maximum)
print ("Messungen:",n)
print ("Messungen {} / ms".format(n/(laufZeit*1000)))
```

In the loop of maxmum.py the value level is read in continuously, which replaces the value maximum whenever it exceeds the previous value of maximum. This is done once for the interference voltage peaks without a sample and a second time for the measurement of the highest voltage pulses with an inserted sample.

I have already briefly outlined the basic function of the actual measurement program measure.py above. Now let's take a closer look. The lines I am going to explain are formatted in bold in the listing.



Download: [messen.py](https://messen.py)

```
from machine import Pin, ADC, Timer
from time import sleep, time
from beep import BEEP
from oled import OLED
from touch import TP
import math

stufen = 16                # Anzahl Energiestufen
cntMax = 713               # hoechster LSB-Wert mit Probe
ruhePegel = 449            # Durchschnitt für 60 Sekunden
noise = 500                # hoechster Rauschpegel
schwelle = noise - ruhePegel # Rauschamplitude
korridor = cntMax - noise   # nutzbares ADC-Intervall
intervallBreite = korridor / stufen # Breite einer Energiestufe
extended = 0                # Anzahl Bereichsueberschreitungen

LedPin = const(4)          # Pin fuer LED-Ausgang HIGH-aktiv
BuzzPin = const(13)         # Pin fuer Piezoelement
tweet = BEEP(LedPin,BuzzPin,5)
red = Pin(2,Pin.OUT)

TPin1 = TP(27)              # Touchpads zur
TPin2 = TP(14)              # Ablaufsteuerung

rad = ADC(Pin(34))          # Detektor-Pin
rad.atten(3)                # Messbereich: 3.3V
rad.width(1)

d=OLED()

messDauer=300
activity=0
spektrum=[]
for i in range( stufen):
    spektrum.extend([0])
events = 0
red.on()
d.clearAll()
d.writeAt("DAUER: {}s".format(messDauer),0,2)
jetzt = time()
ende = jetzt + messDauer
print ("Beginn:",jetzt, " Ende:",ende)
aktuell = jetzt

while aktuell <= ende:
    maximum = noise
    pegel = rad.read()
    while pegel <= noise and time()<=ende:
        if time()>ende: break
        pegel = rad.read()
```

```

tweet.beep(5) # Rauschpegel ueberschritten
if pegel > maximum: maximum = pegel

while pegel > noise and time()<=ende:
    if time()>ende: break
    pegel = rad.read()
    if pegel > maximum: maximum = pegel
events += 1
print(maximum)
if maximum >= cntMax:
    maximum = cntMax - 1
    extended += 1
maximum -= noise
index = int(maximum / intervallBreite)
spektrum[index] += 1
aktuell = time()
# Ende der Messschleife
red.off()
activity= events/( messDauer/60)
d.writeAt("ACTIVITY:{0} CPM".format(int(activity)),0,0)
d.writeAt("{0} EVENTS IN".format(events),0,1)
print(spektrum)

```

How does the program work?

In addition to the usual preparatory work, our three values are incorporated and three more are calculated. Known settings follow.

A few lines below we create an OLED object, define the measurement duration, delete the value for the activity and create an empty list, which we immediately fill with zeros; we need an entry for each level. Decay counter to zero, red light on to start measuring. Delete the display, set up time control, display the time and off you go.

The first while loop is the measurement time manager, we already know that.

Maximum value on noise level upper limit, read in level and continuously check whether we are still within the noise range. The loop is left when this is no longer the case, i.e. the level is in the useful signal corridor or the measurement time has been exceeded. If we are on time, i.e. a decay has been detected, we trigger a beep with an LED flash and set the maximum to the new level value. Regardless of the course of the main program, the beep is stopped after 5 ms by the callback function of the timer interrupt.

The next while loop waits until the ADC level is again in the noise range, but compares the level and maximum during this time. If the level value is higher, it replaces the previous maximum.

After leaving the loop, we increase the event counter and, if necessary, output the maximum for control. The latter can be omitted in stand-alone operation, since no terminal is then connected. This also applies to all other print commands.

In rare cases, the maximum value just determined can be above cntMax. However, since no step memory is provided for this, the value is set to an LSB under cntMax. This is necessary to avoid indexing errors in the spectrum list afterwards. If, on the basis of the value in maxCount, an index were calculated that was greater than 15, the program would exit with an error message because there is no list element in spectrum for this index. If this step was necessary, the over-range counter is incremented.

Since decay events with values within the noise range are very likely to take place, but cannot be detected, no counter memories are provided for this. For this reason, the noise level is subtracted from the maximum, the index in the spectrum list is calculated from this value and the corresponding level counter is then increased. The measurement time is then updated.

After leaving the measurement time loop, we turn off the red light and inform about the results in the display. You can now experiment with your samples as you wish, but there are still extensions waiting for you.

## **Autostart and Stand-Alone**

The three individual programs are well suited for initial tests because they clearly demonstrate the function of our circuit together with the ESP32 and its peripherals. But that's not enough for stand-alone operation, because in this case we don't have a terminal available to start the programs. In this case, the program must start by itself, carry out the desired actions and report the results.

MicroPython has a similar mechanism available for autostart as the Arduino IDE with its setup and loop procedures. The setup is called boot in MicroPython and is a separate file with this name, i.e. boot.py. Loop is also replaced by a separate file called main.py. In contrast to the loop construct of the Arduino IDE, this part of the program must contain an endless loop in the form while True, otherwise the program will abort in nirvana after one run.

### **Warning!**

It can be difficult to stop an ESP32 after an autostart without a defined program end in order to make changes to the program. It is imperative that you install a time-controlled waiting loop in boot.py after the start, during which Thonny or µPyCraft can be aborted via USB. To cancel, press Ctrl + C.

If that is not possible from the IDE due to USB connection problems, there is an emergency brake and a self-destruct button. The emergency brake is the Putty program. First the connection to the CP2102 is established, baud rate 115200. Then you start the ESP32 with the reset button and enter Ctrl + C in the putty window. That should stop the controller. Then first rename the boot.py file, the new file name does not matter.



```
os.rename ("boot.py", "bootpy.org")
```

After restarting the ESP32 you should be able to contact it again in µPyCraft or Thonny.

If that all goes wrong, the only thing that helps is the self-destruct button called esptool.py. Clear the memory as described below and reflash MicroPython.

For the autostart, I converted the three previous programs into functions and added four more functions to the whole thing. All of this has been added to the bootpart.py file, which is used for initial tests. If everything works fine, copy the entire contents of this file to the clipboard and add everything to the previous contents of boot.py after the last line. After saving, the new boot.py is uploaded to the device.

Two of the new programs are used for more convenient LED control, the new jaNein () queries the touchpads with additional time control and report () outputs the values of the spectrum list as a bar graph, because there is no longer a terminal for it - but of course it can still be connected.

You already know the touchpad query with timeout from previous homework. I made a change in touch.py to the getTouch () method. Instead of the ADC value or milliseconds, it now returns the values True, False or None, accordingly touched, not touched or timeout.

yes no () takes the optional parameters message and runtime. The string in message is output in line 1 on the display, running time represents a timeout value after which the function is ended if no touch action has taken place. The return value of yesNo () is based on this table.

laufZeit	Aktion ja	Aktion nein	Return value
0	-	-	- (endles waiting)
0	x	-	2
0	-	x	1
0	x	x	3
>0	x	-	2
>0	-	x	1
>0	x	x	3
>0	-	-	0 (Timeout)

The return values are also coded in the constants JA = 2, NEIN = 1, BOTH = 3 and TIMEOUT = None.

The function report () takes one of the constants Fcut, Fprop or Flog as an optional parameter to adapt the height of the columns to the values in spectrum and to the height of the display (in d.HEIGHT). The table makes the selection easier.

Parameter	max(spektrum)	Bemerkung
Fcut	> d.HEIGHT	alle Werte in spektrum werden, falls > d.HEIGHT auf diesen Wert gekappt, kleinere Werte werden in ihrer Originalhöhe dargestellt.
Fcut	<= d.HEIGHT	alle Werte werden in ihrer Originalhöhe dargestellt.

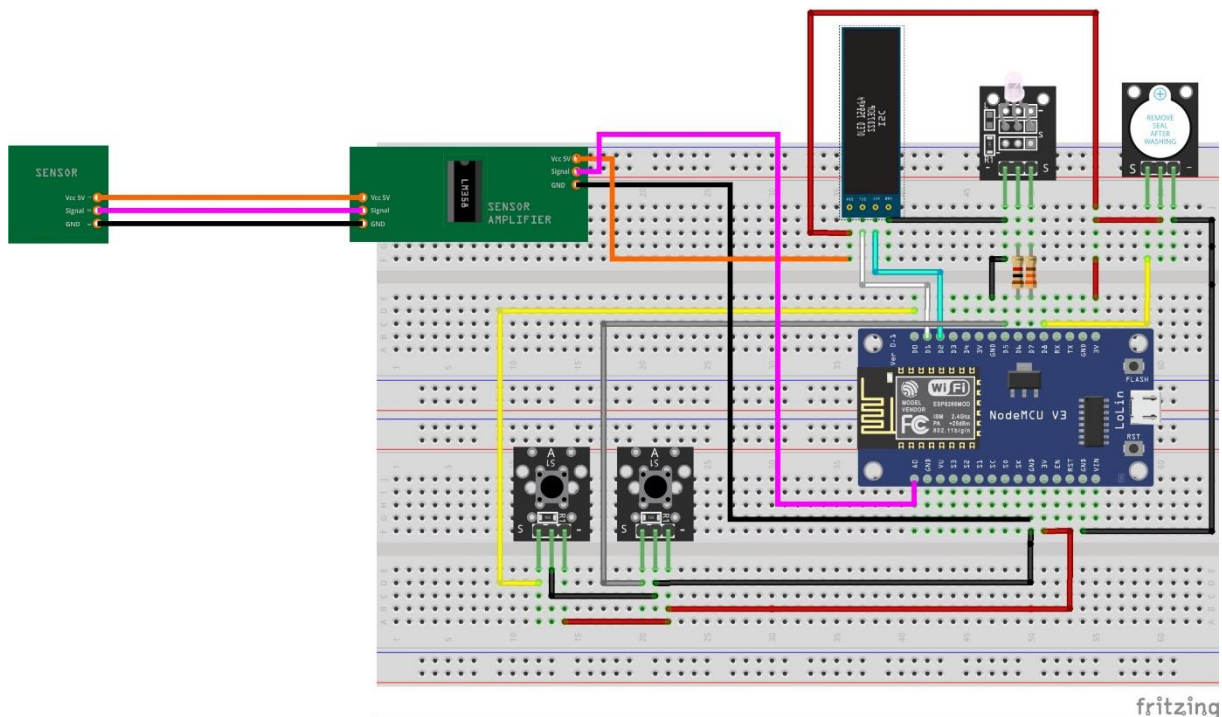
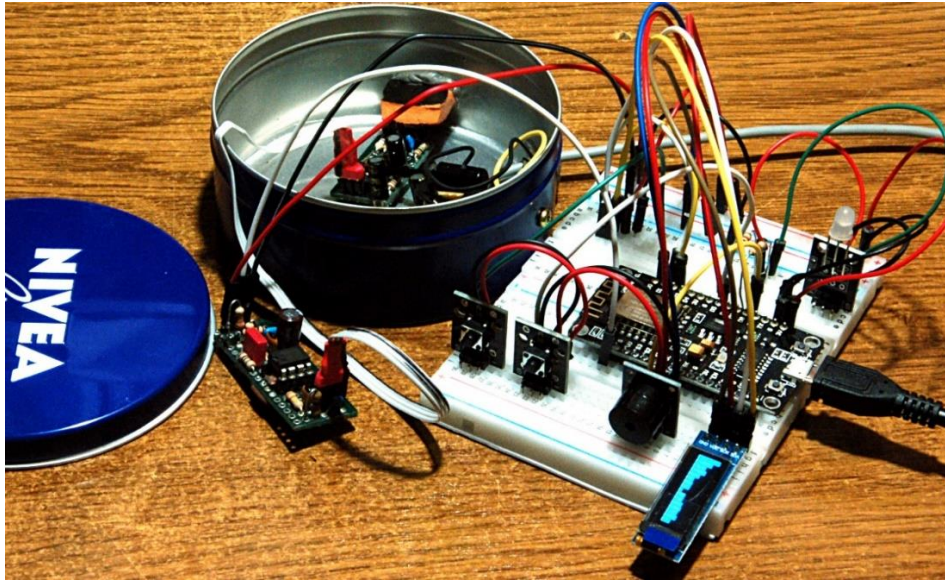
Fprop	> d.HEIGHT	alle Werte werden proportional zu $\max(\text{spektrum}) = d.\text{HEIGHT}$ herunter gerechnet.
Fprop	$\leq d.\text{HEIGHT}$	alle Werte werden proportional zu $\max(\text{spektrum}) = d.\text{HEIGHT}$ gestreckt.
Flog		alle Werte werden als $\log(\text{spektrum}[i])$ logarithmisch dargestellt, wobei $\log(\max(\text{spektrum})) = d.\text{HEIGHT}$ .

After the start, the values for quiet level, noise and cntMax can be recorded again if the touchpad query is answered with "yes" within 10 seconds. With cntMax there is no time limit because the sample has to be inserted at rest. You can change that if you want.

The values are displayed for 3 seconds, then boot.py is terminated and main.py is started automatically, which calls up the measurement function in an endless loop controlled via the touchpad and shows the results on the display. The solution to homework 6 can serve as a suggestion for you to switch continuously between the display of values and graphics until a touchpad is touched to start a new measurement. I have not shown the finished [bootpart.py](#) and [boot.py](#) as well as [mainpart.py](#) and [main.py](#) here, because they only represent repetitions and summaries of already known contents. But you can use the downloads to study the files.

## Can the ESP8266 do all of this?

Yes, it can, with certain adjustments, the scope of which is limited. These changes compared to the ESP32 concern the touchpads, which are replaced by buttons, together with the driver file touch8266.py and the initialization of the analog input. A duo-LED module (red on D6 and green on D7) and the blue LED built into the ESP8266-12F on pin GPIO2 = D4 are used for the LEDs. This LED is LOW-active, which is why the corresponding positions in the functions ledOn () and ledOff () as well as in the beep module must be adapted. Here is a photo of the structure and the circuit diagram from Fritzing.



The radiation sensor is connected with Vcc to Vin = 5V of the ESP8266, GND to GND and the signal to A0. Please note that the names of the connections of the OLED display on your copy may differ from those in the diagram.

The programs for the ESP8266 are linked in the following list. The operation of the ESP8266 board otherwise follows the descriptions for the ESP32.

[beep.py](#)  
[boot.py](#)  
[bootpart.py](#)  
[main.py](#)  
[mainpart.py](#)  
[maximum.py](#)  
[messen.py](#)  
[oled.py](#)  
[ruhepegel.py](#)  
[ssd1306.py](#)  
[touch8266.py](#)

In the next episode, the last one on the subject of nuclear radiation measurements, we will use a website to operate the ESP32 / ESP8266, on which the results of the settings and measurements are clearly displayed. This is ensured by the web server from the second episode, which we will cannibalically call up for this job. There is also a short detour in the direction of inheritance in classes.

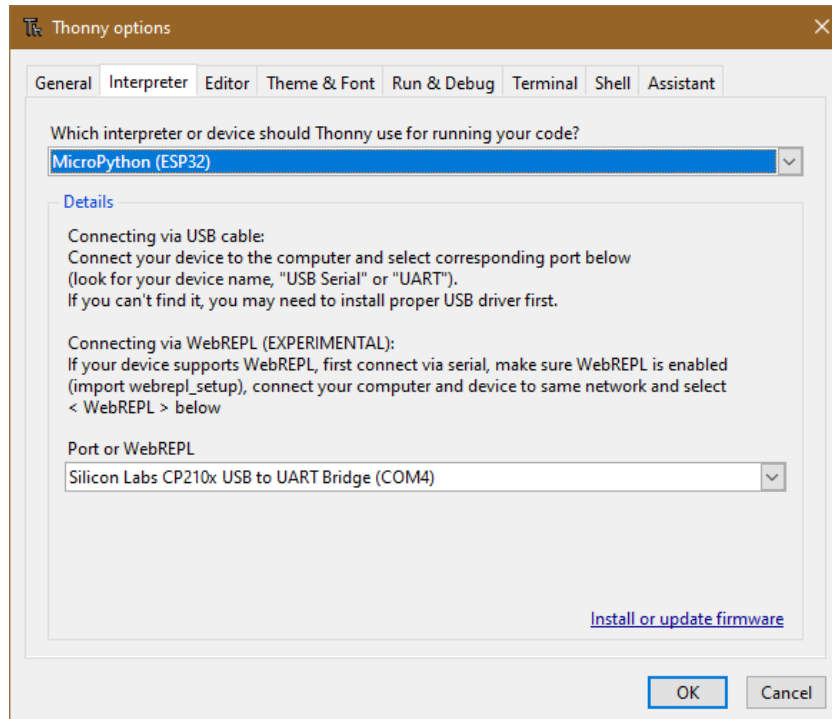
You can of course also download this part as a [PDF](#) in German.

This is followed by the description of how you can use esptool.py to flash the MicroPython firmware.

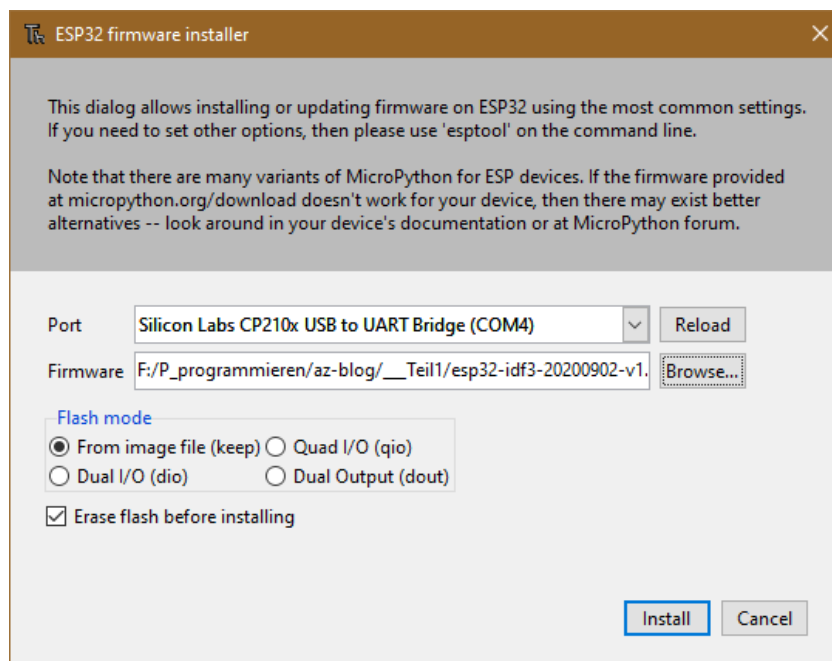


# The use of esptool.py with the ESP32 / ESP8266

In the third part of the blog I wrote that Thonny cannot flash the firmware on the ESP. Thanks to a comment from a reader, I'm a lot smarter now. In fact, there is a very small hidden link hidden in the Interpreter window, which can be opened via the Run - select Interpreter... menu: Install or update firmware.



Left-click to land in the settings window.



And left click on Install starts the process. Many thanks to Rudolf R. for the tip.

But because I promised to describe the flash process for esptool.py, with this chapter I am fulfilling the promise that has already been postponed twice.

When Thonny is installed, the directory X: \ Users \ <username> \ AppData \ Local \ Programs \ Thonny \ Lib \ site-packages is created if the installation was carried out as a normal user. If Thonny was installed as administrator, it is the directory X: \ Program Files (x86) \ Thonny \ Lib \ site-packages. Please replace the 'X' with your drive letter. This directory contains, along with other .py files, the esptool.py file, which can be used to flash firmware on the ESP32.

esptool.py is a Python program that must be started from the command line. The Python installation brought by Thonny is used for this. Now start the command prompt and change to the appropriate directory that corresponds to your installation. I installed Thonny as an administrator and therefore choose the second variant. List the files, esptool.py should be listed.

```
C: \ Users \ root> cd C: \ Program Files (x86) \ Thonny \ Lib \ site-packages
```

```
C: \ Program Files (x86) \ Thonny \ Lib \ site-packages> dir esptool.py
```

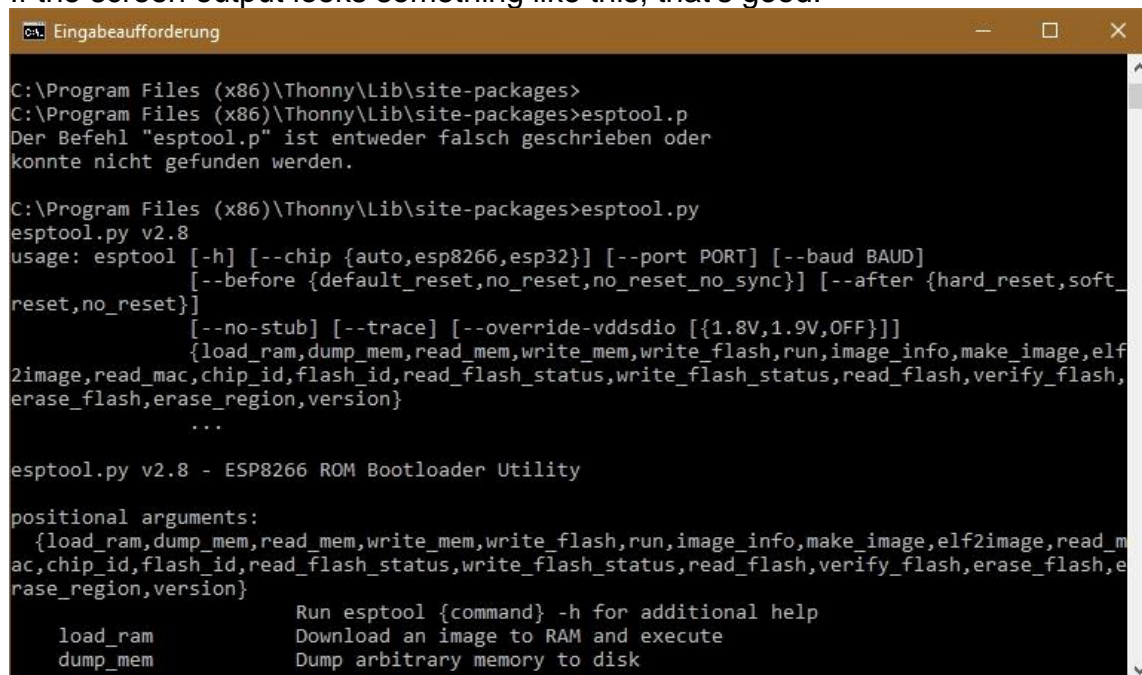
Directory of C: \ Program Files (x86) \ Thonny \ Lib \ site-packages

```
10/23/2019 5:07 AM 143.643 esptool.py
                1 file (s), 143,643 bytes
```

Now start esptool.py

```
C: \ Program Files (x86) \ Thonny \ Lib \ site-packages> esptool.py
```

If the screen output looks something like this, that's good.



```
C:\Program Files (x86)\Thonny\Lib\site-packages>
C:\Program Files (x86)\Thonny\Lib\site-packages>esptool.p
Der Befehl "esptool.p" ist entweder falsch geschrieben oder
konnte nicht gefunden werden.

C:\Program Files (x86)\Thonny\Lib\site-packages>esptool.py
esptool.py v2.8
usage: esptool [-h] [--chip {auto,esp8266,esp32}] [--port PORT] [--baud BAUD]
               [--before {default_reset,no_reset,no_reset_no_sync}] [--after {hard_reset,soft_
reset,no_reset}]
               [--no-stub] [--trace] [--override-vddsdio [{1.8V,1.9V,OFF}]]
               {load_ram,dump_mem,read_mem,write_mem,write_flash,run,image_info,make_image,elf
2image,read_mac,chip_id,flash_id,read_flash_status,write_flash_status,read_flash,verify_flash,
erase_flash,erase_region,version}
               ...

esptool.py v2.8 - ESP8266 ROM Bootloader Utility

positional arguments:
  {load_ram,dump_mem,read_mem,write_mem,write_flash,run,image_info,make_image,elf2image,read_m
ac,chip_id,flash_id,read_flash_status,write_flash_status,read_flash,verify_flash,erase_flash,e
rase_region,version}
    load_ram            Run esptool {command} -h for additional help
    dump_mem            Download an image to RAM and execute
                       Dump arbitrary memory to disk
```

If you get error messages, try one of the following calls.

```
C:\Program Files (x86)\Thonny\Lib\site-packages> python esptool.py
```

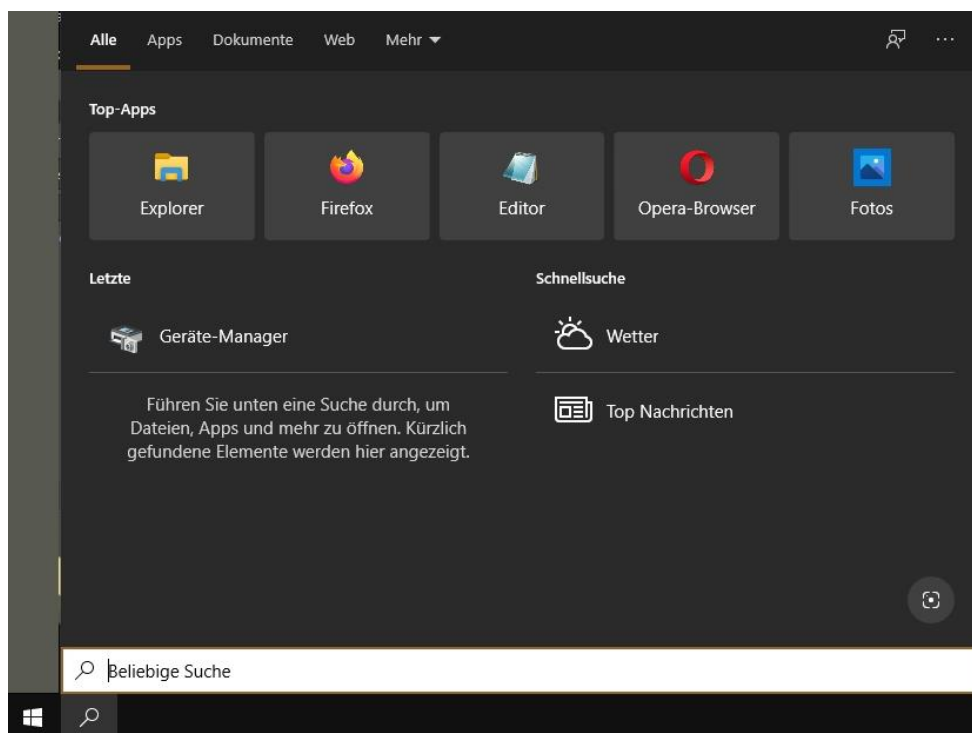
```
C:\Program Files (x86)\Thonny\Lib\site-packages> python3 esptool.py
```

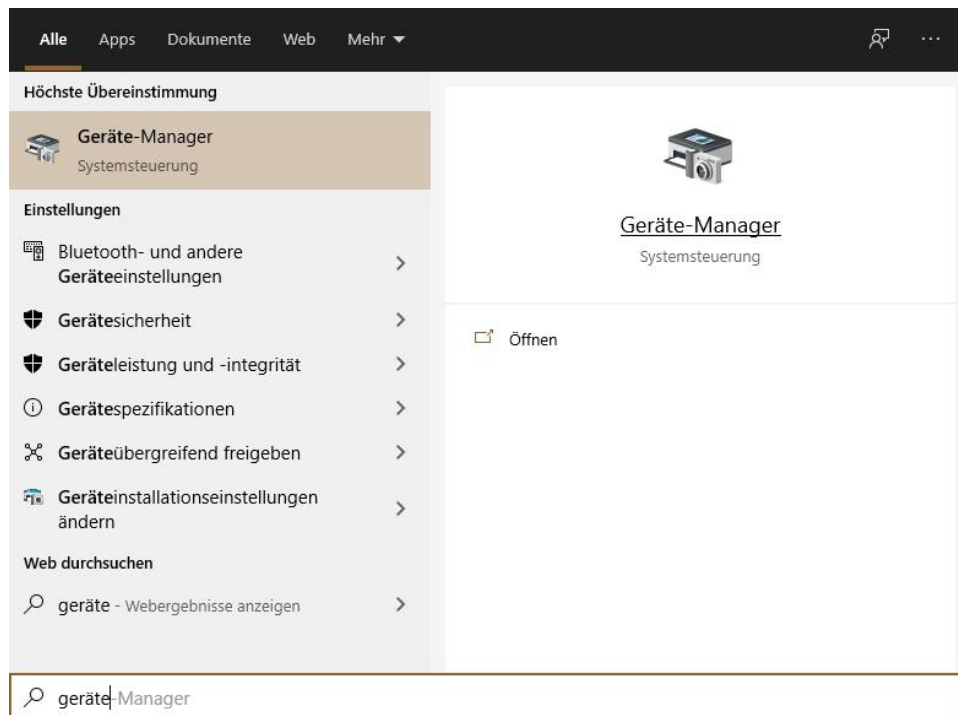
For more convenient use, a preparation should now be made that has to do with the type of program call. This affects Windows and, in a similar way, Linux.

One of the following conditions must be met so that a program can be started from any window in the command prompt.

1. In the input window you are in the directory of esptool.py. Esptool.py is started without specifying a path. The bin file must be specified with its full path name. Alternatively, the firmware file can also be copied into the program directory.
2. You are in the directory of the bin file. The file name esptool.py is preceded by the relative or absolute path to the program directory. The bin file can be specified without a path.
3. The path to the file to be called up is contained in the Windows (or Linux) PATH environment variable. We are in the directory of the bin file. Path information is not required either to start esptool.py or for the file name of the firmware.

I am now introducing the procedure for the third case because the other methods can result in very long command lines. Connect your ESP32 to the USB bus and determine the COM port number of the connection. To do this, enter "Device Manager" in the Windows search at the bottom left.





Klicken Sie rechts auf **Geräte-Manager** und öffnen Sie dann den **Ordner Anschlüsse (COM & LPT)**.



Hier ist der CP210x des ESP32 als COM4 eingetragen. Das kann bei Ihnen eine andere COM-Nummer sein.

In Linux helfen für das ganze Prozedere die folgenden Befehle

```
lsusb
```

```
Bus 002 Device 004: ID 067b:2303 Prolific Technology, Inc. PL2303 Serial Port
Bus 002 Device 001: ID 1d6b:0001 Linux Foundation 1.1 root hub
```

```
dmesg | grep tty
```

```
[3977499.045783] usb 2-1: pl2303 converter now attached to ttyUSB0
```

Die Portangabe lautet damit **/dev/ttyUSB0**



Um die PATH-Variable in Linux zu erweitern wird in der Datei ~/.bashrc folgender Eintrag hinzugefügt.

```
PATH=$PATH:/pfad/zu/esptool.py
export PATH
```

What is very easy in Linux with two commands and two lines of text requires more effort in Windows, but it is also worthwhile. The addition of the PATH variables in Windows is "a bit" more complex. Here as there, the eternally long path specifications are the problem because they result in even longer command lines.

Initial scenario:

esptool.py: C: \ Program Files (x86) \ Thonny \ Lib \ site-packages

Firmware: F: \ micropython \ ESP32 \ firmware \ latest \ esp32-idf3-20200902-v1.13.bin

COM port: COM4

You can save yourself the path to the esptool.py program if you include the program path in the Windows system's Path environment variable. And you save the path to the firmware directory by changing to this directory via Explorer. Long file names can be entered quickly using the automatic input completion. So that this can be done, we now add the string of the program path from esptool.py to the path variable PATH of the Windows system. It works like this.

Start with a right click on the Windows symbol in the lower left corner of the screen and click on System.



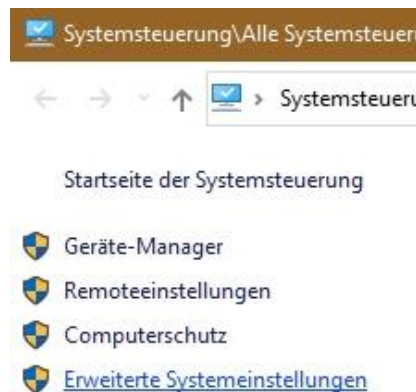
Im rechten Fenster rollen sie nach unten bis **Verwandte Einstellungen**. Klicken Sie auf **Systeminfo**.

## Verwandte Einstellungen

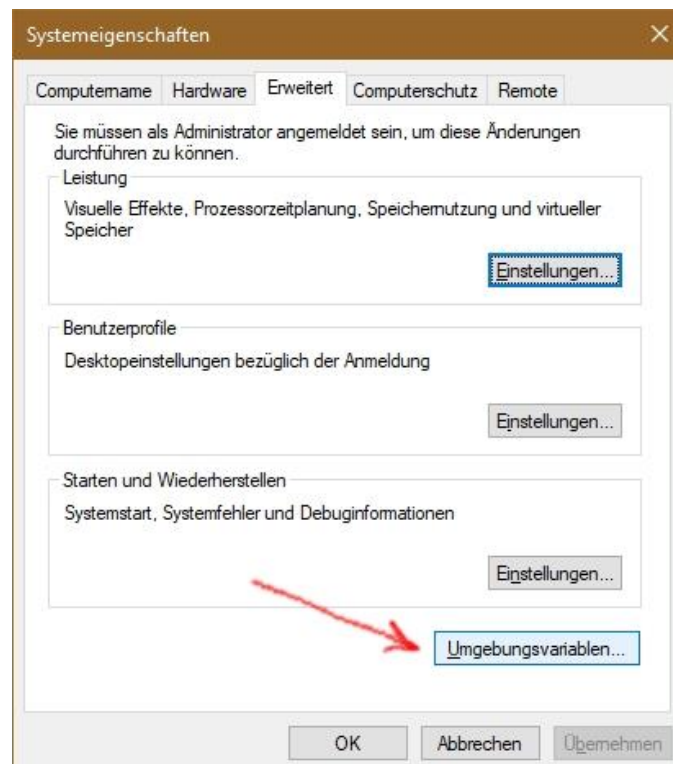
BitLocker-Einstellungen

Systeminfo

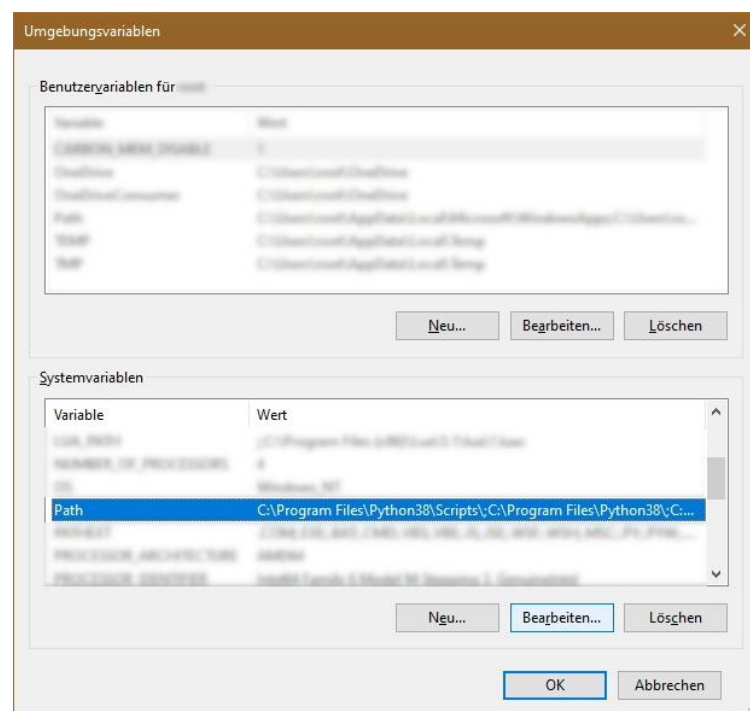
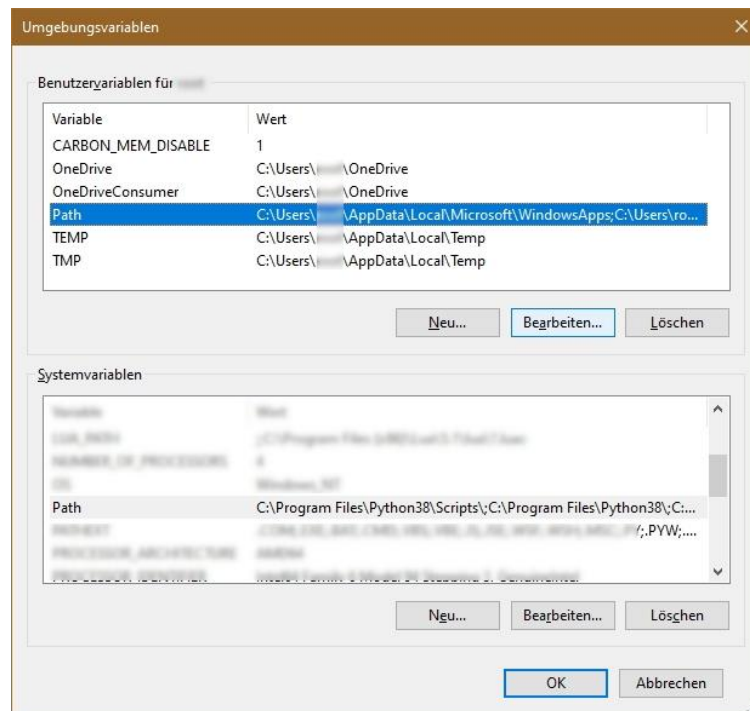
Im folgenden Fenster gehen Sie auf **Erweiterte Systemeinstellungen**.



Ganz unten finden Sie das Feld **Umgebungsvariablen** -- Klick.

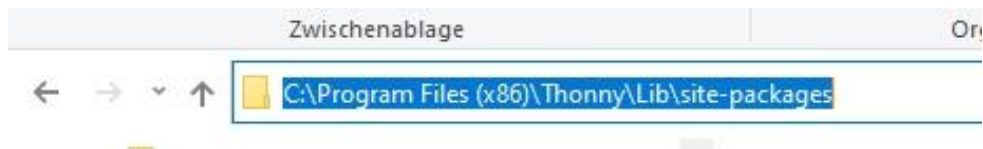


In the upper part you set the variable Path only for yourself, in the lower part a change affects all users system-wide. My installation of Thonny was done as an administrator, so I choose the variant below.

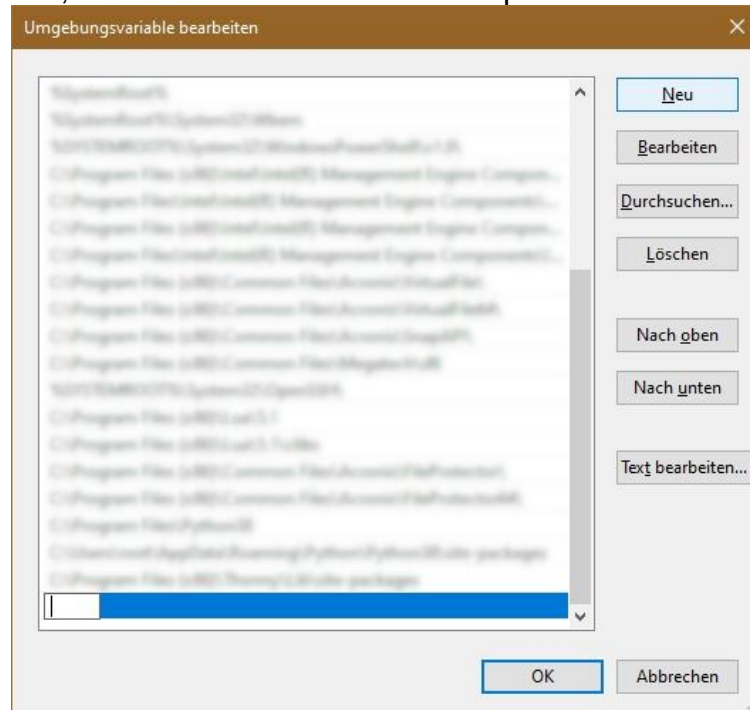


To add entries to the path variable, first click on Edit.

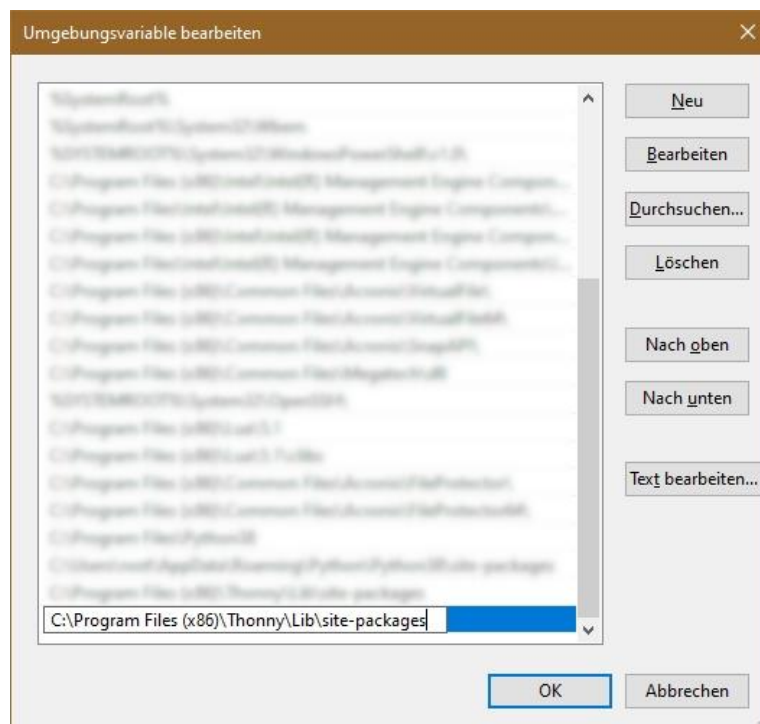
If the path is longer, it is better to use it from the Windows Explorer. In Windows Explorer, navigate to the program directory of esptool.py and, when you arrive there, click in the back of the line with the entire path. Use Ctrl + C to copy the marked path information to the clipboard. Go back to the Edit Environment Variable window.



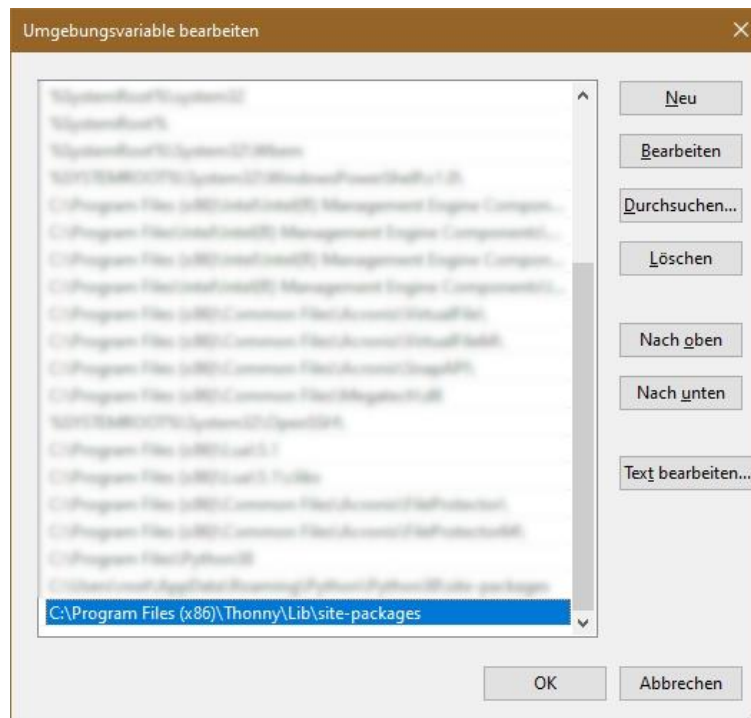
Klicken Sie auf Neu, um der Path-Variablen den kopierten Pfad hinzuzufügen.



Fügen Sie den Pfad aus der Zwischenablage ein und bestätigen Sie mit **OK**.







Now close all windows in reverse with OK.

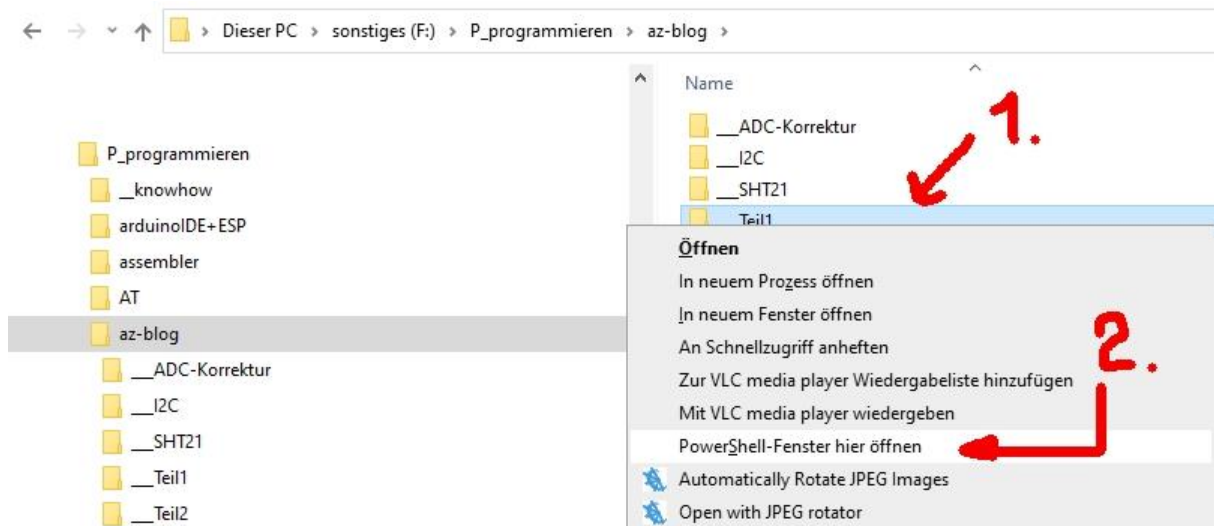
Now, from the firmware directory, you can execute the calls for an ESP32 as follows. Please note that the identifier must be specified explicitly for this chip, while it is optional for the ESP8266. The chips also require different start addresses for the firmware, with the ESP8266 it is 0x0000 and with the ESP32 you have to specify 0x1000.

```
esptool.py --chip esp32 --port COM4 erase_flash
```

```
esptool.py --chip esp32 --port COM4 write_flash -z 0x1000 esp32-idf3-20200902-v1.13.bin
```

That looks good and also has the advantage that esptool.py can now be called from any directory. If you want to flash a Node MCU firmware, switch to the corresponding firmware directory and start the flash process from there. The program calls then look exactly like here, except for the name of the firmware file and the chip used. If necessary, replace esp32 with esp8266. The default value is esp8266 and doesn't even need to be entered, I mentioned that above. The file name can be specified more quickly from the directory by entering the first characters of the file name and using the Tab key.

Here is another trick to call the Powershell in the correct target directory. In Explorer, navigate to the directory that contains the directory with the .bin files (1). Right-click on the target directory and select Open PowerShell window here from the context menu (2). This will land you in the Powershell window in the correct directory on the command line and you can start entering data immediately.



A few more commands for the ESP8266, which are often used, now follow.

`esptool.py flash_id`

Brings a whole range of data to the chip, including MAC address and flash size. The COM port is searched for and connected automatically.

`esptool.py erase_flash`

Erases the flash memory before it is rewritten. The COM port is searched for and connected automatically.

`esptool.py -p COM5 -b 460800 write_flash -fm dio -fs 4MB 0x00000 esp8266-1m-20200902-v1.13.bin`

Writes an ESP8266 to port COM5 with baud rate 460800 in dio mode with the content of the file `esp8266-1m-20200902-v1.13.bin` from address 0x0000. For a D1 mini, the memory size must be specified with the actually available 4MB, even if only 1MB is used by MicroPython.

`esptool.py --port COM3 write_flash --flash_mode dio -flash_size 1MB 0x0000 esp8266-1m-20200902-v1.13.bin`

Flashing an ESP8266-01 with 1MB Flash. This leaves 374KB for programs.

There were also surprising moments when writing this article. Once there was the finding that without specifying the flash size and mode, an ESP8266 D1 mini could be flashed with `esptool.py`, but then the built-in LED immediately began to flash wildly and the chip was not addressable. The same behavior was shown by an ESP8266-01 that I had flashed with `µPyCraft`. With `esptool.py` and the right parameters, both boards worked in the end. What is essential for `-fs` or `-flash_size` is the specification of the actually available flash memory, which in the case of the ESP8266 does not necessarily have to match the size in the file name of the bin file. The command `esptool.py --port COM3 write_flash --flash_mode dio -flash_size 1MB 0x0000 esp8266-1m-20200902-v1.13.bin` so it fits for the ESP8266-01 and for the ESP8266 D1 mini it has to be called `esptool.py --port COM3 write_flash --flash_mode dio -flash_size 4MB 0x0000 esp8266-1m-20200902-v1.13.bin`.

## Solutions to the homework from part 4

1. Let us assume that you make the following entries on the command line:

```
>>> from beep import BEEP
```

```
>>> BEEP. Duration = 5000
```

```
>>> BEEP. Duration
```

```
5000
```

```
>>> b = BEEP (2.13)
```

```
???
```

```
???
```

```
>>> b.beep ()
```

```
???
```

How does the ESP react to the last two commands? Can you justify the behavior?

The output of the constructor is:

BEEP constructor

LED: 2, Buzz: 13, duration = 15

The tone sounds for 15 ms.

Let's look at the beginning of the BEEP class definition

```
DURATION = const (15)
```

```
class BEEP:
```

```
    duration = DURATION
```

duration receives a reference to the storage space of the constant 15

```
def __init__ (self, led, buzz, duration = duration):
```

During the import, the interpreter runs over this line of the constructor and builds the reference to 15 into the interpreted (= translated) program. If duration is set to a new value in the further course, this does not change anything in the assignment in the parameter list of the constructor. To do this, the program would have to be reinterpreted. The interpreter only translates the program once during the import. Any number of instances can then be created by executing the already translated constructor method, with the fixed reference to the value 15.

Because duration is an optional parameter, the reference to the constant 15 can be overwritten by specifying an argument for duration when an object is instantiated, as a number or a name-value pair. If no argument is given, access takes place via the specified reference to the constant 15.

The message from the constructor and the duration of the sound are therefore consistent.

2. Display the output in the touchtest.py file on the OLED display instead of on the terminal. When does that make sense, when not?

You can download the [sample program](#). Terminal and display have advantages and disadvantages. The terminal can also display longer, extensive texts without any problems, but you need a computer. The display works "stand alone", but on the one hand can only reproduce short texts and only in a limited number of lines. While you can scroll back in the terminal, old issues are overwritten by new ones on the display.

3. Measure the speed that can be achieved with the output on the terminal and on the OLED display.

The test program essentially corresponds to the speed test for the ADC. You can also [download it](#).

```
from time import sleep, time
from oled import OLED
d = OLED ()

n = 0
now = time ()
duration = 5 # program run time approx. 3 seconds
then = now + duration
actual = now
while actual <= then:
    d.writeAt ("01234567890123456", 0.0)
    # d.writeAt ("01234567890123456", 0.1)
    # d.writeAt ("01234567890123456", 0.2)
    #print ("0123")
    n += 1
    actual = time ()
    # print (ldr_value)
    # sleep (0.1)
print (n / duration, "blocks per second")
```

OLED constructor  
SDA: 21, SCL: 22, Size: 128x32  
48.8 blocks per second

A line of 16 characters can be output approx. 50 times per second. This is largely independent of the number of characters. With 4 characters I measure 50 blocks / s and with only one character 52 blocks / s. Three lines with 16 characters bring it accordingly to 16.6 blocks / s.

For the terminal output, replace the line in the program  
d.writeAt ("0123456789012345", 0.0)  
by  
print ("0123456789012345")



With 1 line of 16 characters, the terminal is a good 12 times faster with 628.4 blocks / s, with 4 characters even over 40 times.

This is due to the fact that the entire ESP frame buffer always has to be sent to the display, even if only one character has been changed. What slows down is the show () method.

In contrast to this, the serial line to the terminal only transmits as many characters as need to be sent. The fact that the ESP32 or ESP8266 works significantly faster than the serial line is expressed in the fact that you have to wait longer for the result than the 5-second specification of the program. The message only comes when the send buffer has been completely transferred.

4. Does it make a difference to the speed how many characters are output on the display and on the terminal per pass?

No for the display, yes for the terminal.

5. Use the pillar () command and the random number generator from the homework solution to create a bar chart made up of 10 bars that uses the full display area of a display. Remember that there are displays of different widths.

The width specification should be kept variable and not, for example, given as 64. The information here comes from the instance attribute width of the object d.

```
import os
from oled import OLED
d = OLED ()
```

```
d.clearAll ()
```

```
width = d.width // 10
hTeiler = 256 // d.height
```

```
d.xAxis ()
d.yAxis ()
for i in range (10):
    height = os.urandom (1) [0] // h divider
    d.pillar (i * width, width-1, height)
```

6. Prepare two different outputs for the OLED display, for example a text screen and a bar chart. Have both displayed alternately for 3 seconds, while (at the same time!) The numbers from 1 to 30 are output continuously in the terminal according to the following scheme.

```
1
12th
123
1234
12345
...
```

The example is intended to demonstrate that interrupt-controlled actions (relatively) intimate are possible during the normal execution of a main program. Ultimately, the exact timing depends on how the firmware classifies and handles the priority of the interrupts.

The [program linked](#) here gives an example of the solution to the task.

The solution is via a timer that calls one of the two functions `texte ()` or `pillars ()` every 3 seconds. The change takes place via the counting up and the bitwise undating of the variable `n`.

The numbers are output via two for loops, the outer one running to 31 and the inner one therefore running to 30. A short sleep break is built in so that the serial interface can keep up.

7. Modify the reaction tester from the homework solution in such a way that a different LED is sharp with each round. Of course you have to tell the player via the OLED which color it is.

A few lines have been added to the [hausi4b.py](#) program. The [modified program](#) is also available for download.

Right at the beginning, the OLED class is imported, the display object `d` is created and the display is deleted.

A list of colors is defined to match the list of LED objects. The fixed assignment of a number to `testLed` is removed.

```
ledList = [ledG, ledR, ledB]
ledColList = ["green", "red", "blue"]
```

The target LED is rolled as usual and then the color is shown on the display.

```
d.writeAt (ledColList [led], 0,1)
```

Finally, we roll a test LED and assign the corresponding object to the `run` attribute `ledx`. The rest of the homework solution doesn't need to be changed.

```
testLed = os.urandom (1) [0] & 0x03
testLed = (testLed if testLed != 3 else 2)
Assign #TestLED, continue as usual
ledx = ledList [testLed]
```

Start the program and go!

8. Can you set it up so that the ESP32 and ESP8266 variants are kept in a new touch module and, depending on the controller type, the correct one is automatically used during import?

The [touch.py](#) file contains the solution, which is very simple. Both original parts were packaged in an if-elif-else structure, each indented one level further. The import of `Pin` and `TouchPad` has also been moved to the if structure. The limit value for the ESP8266 variant was set to 0.5 so that comparisons can be made as with the 32 variant.