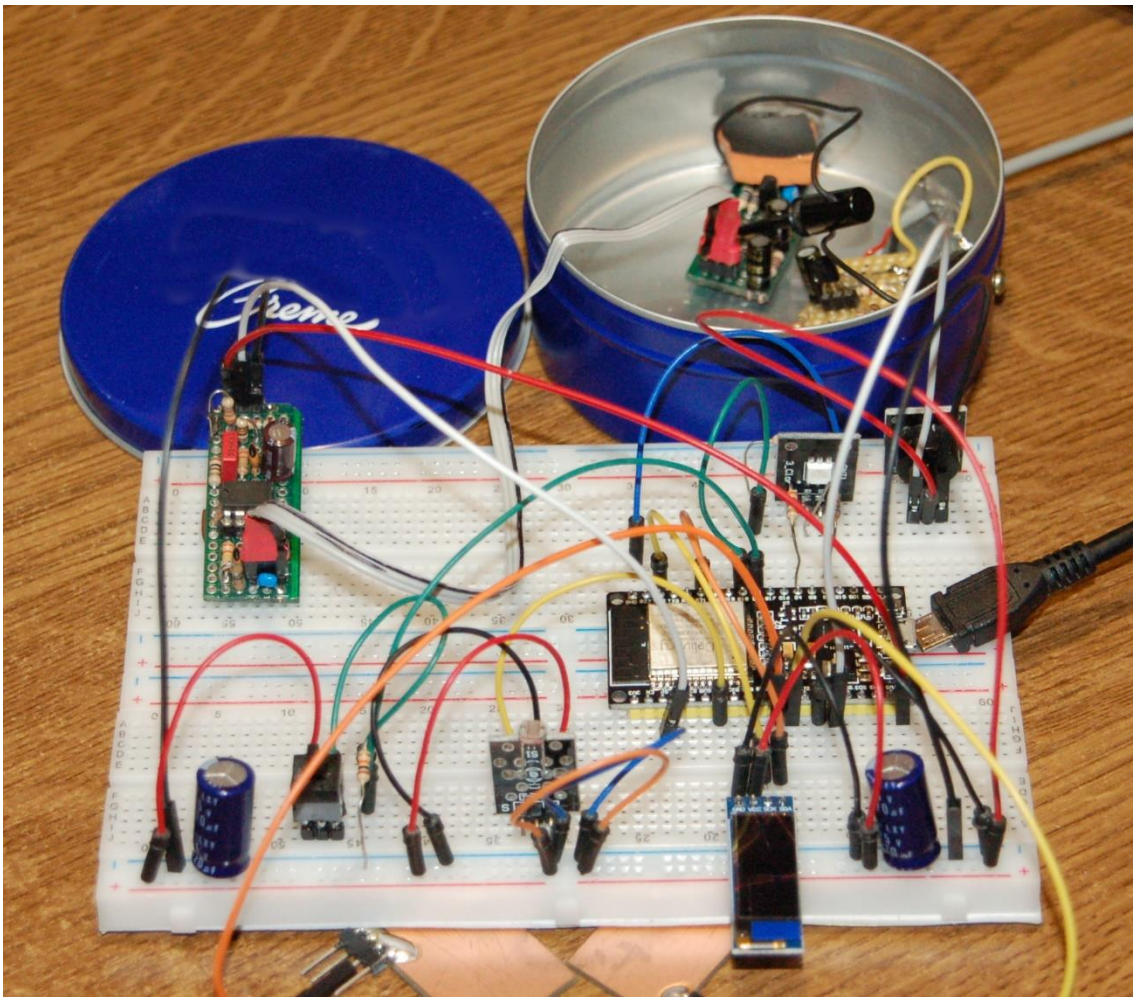


MicroPython mit dem ESP32/ESP8266 – Teil5

Heute wird das Geheimnis gelüftet, worum es nach den ganzen Vorbereitungen beim Projekt geht. Wir bauen eine Platine, um damit Kernstrahlung zu messen und arbeiten im Anschluss das Thema esptool.py auf. Die Besprechung der Hausaufgaben bringt wieder eine Menge von zusätzlichem Knowhow. Damit willkommen beim fünften Teil der Serie.

Vor über 100 Jahren veröffentlichte Marie Sklodowska-Curie ihre Doktorarbeit mit dem Titel "RECHERCHES SUR LES SUBSTANCES RADIOACTIVES" "Untersuchungen über die radioaktiven Substanzen". Seit 1903 haben sich die Methoden, Kernstrahlung zu messen enorm gewandelt und verbessert. Das von den Herren Geiger und Müller konstruierte Messgerät arbeitet mit einer speziellen Röhre und mit Spannungen zwischen 300V und 500V. Die Schaltung, die ich heute beschreibe, begnügt sich mit 5V.

Von den drei Strahlenarten Alpha-, Beta, und Gammastrahlung, die aus dem Inneren des Atomkerns kommen, kann unser Aufbau die letztere Sorte recht gut detektieren, weil es sich dabei um eine elektromagnetische Welle, ähnlich dem sichtbaren Licht, handelt. Was liegt also näher als eine Fotodiode als Sensor zu verwenden. Gamma-Strahlung kann die Kunststoffschicht unseres Sensors durchdringen wie Licht, was

den anderen beiden Arten versagt ist. Alphastrahlung besteht aus schnellen Heliumkernen und kann bereits durch ein Blatt Papier gestoppt werden. Die schnellen Elektronen der Betastrahlung bleiben in 1mm Alu oder Kunststoff stecken.

Unser Sensor ist eine PIN-Diode. Der 500 bis 1000nm dicke schwach negativ dotierte **I**ntrinsic-Bereich ist von der einen Seite her stark **p**ositiv, von der anderen Seite her stark **n**egativ dotiert. Durch die dünne p-dotierte Seite (ca. 35nm) kann Licht einfallen, das im breiten Intrinsic Bereich Ladungsträger freisetzen kann, welche im anliegenden elektrischen Feld beschleunigt werden, was zu einem Strompuls führt. Im Gegensatz zu normalen PN-Dioden, besteht durch den breiten Intrinsic-Bereich eine höhere Wahrscheinlichkeit, dass durch Lichtquanten eine Ladungstrennung hervorgerufen wird. Diese Aufgabe fällt in unserem Fall den Gammaquanten der Kernstrahlung zu.

Damit das viel stärkere umgebende Tageslicht nicht auch zum Zuge kommt und alle schwachen Gammasignale überlagert, muss die Messung im Dunklen stattfinden. Das ist der eine Grund, warum der Sensorteil der Schaltung in eine gut verschließbare Blechdose gesteckt werden muss. Zum anderen dient die Dose einfach als elektrische Abschirmung, damit der "normale" Elektrosmog um uns herum die empfindliche Schaltung nicht stören kann. Deshalb muss es auch eine Blechdose sein, Kunststoff schirmt zwar Licht ab aber keine elektrischen Felder. Dazu später mehr.

Den Sensor für die heutigen Experimente bauen wir aus Einzelteilen auf einer Lochrasterplatine auf. Dazu wird folgendes Material gebraucht.

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, Kohleschich
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	PCB Board Set Lochrasterplatte
1	Blechdose mit Blechdeckel ca. 9cm Ø und 4cm hoch, lichtdicht
1	Schraube M3 mit Beilage und Mutter
	etwas Schaltdraht

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

1	ESP32 NodeMCU Module WLAN WiFi Development Board oder
1	ESP-32 Dev Kit C V4 oder
1	NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F mit CH340
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel für Arduino und Raspberry Pi oder
1	0,96 Zoll OLED I2C Display 128 x 64 Pixel für Arduino und Raspberry Pi
1	KY-012 Buzzer Modul aktiv
2	LED (Farbe egal) und
2	Widerstand 330 Ohm für LED oder
1	KY-011 Bi-Color LED Modul 5mm und
2	Widerstand 560 Ohm für LED oder
1	KY-009 RGB LED SMD Modul und
1	Widerstand 330 Ohm für blaue LED
1	Widerstand 680 Ohm für rote LED
1	Widerstand 3,9k Ohm für grüne LED
1	keypad-ttp224-14-kapazitiv oder
1	KY-004 Taster Modul
2	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
2	Blechstücke ca. 20 x 20 mm (nicht Aluminium!) oder Platinenreste
	einige Steckstifte 0,6x0,6x12mm

Folgendes Werkzeug ist hilfreich.
Lötkolben oder Lötstation, feine Spitze, Lötzinn
kleiner Seitenschneider
kleiner Schraubenzieher
3mm-Bohrer (Handbetrieb oder Minimot)
Handsäge mit Metallblatt
evtl. "Dritte Hand" zum Halten der Platine beim Löten
evtl. Digitalvoltmeter
kleine Flachfeile oder ein Stück Schleifpapier Korn120-150

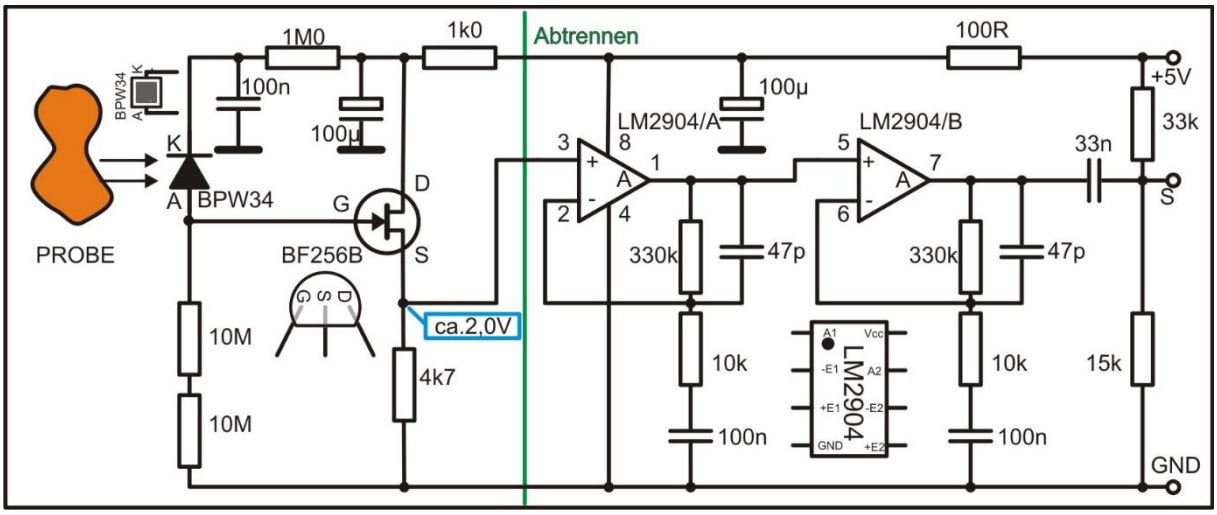
Aufbau der Sensor- und Verstärkerplatine

Für meinen Aufbau habe ich eine Lochrasterplatine von diesem Set PCB Board Set Lochrasterplatte benutzt. Wenn Sie nicht in Serie gehen wollen genügt ein Lochrasteraufbau völlig.

Die Spalten und Zeilen der Platine sind mit Buchstaben und Nummern versehen, sodass man die Position der Bauteilanschlüsse leicht nachvollziehen und kontrollieren kann. Ich wollte es wissen und habe es tatsächlich geschafft, beide Schaltungsteile auf einen der schmalen Streifen des Sets aufzubauen. Wenn Sie im Löten noch nicht so sicher sind, verwenden Sie lieber 2 Streifen, vielleicht sogar die breiteren. Freilich stimmt dann leider die Vorlagezeichnung nicht mehr.

Die Abbildung zeigt die Oberseite der Platine. Die Buchstaben laufen dann in alphabetischer Reihenfolge von links nach rechts, die Nummern von unten nach oben. Die Rückseite ist analog beschriftet, sodass zum Beispiel die Bohrung A01, von beiden Seiten betrachtet, dieselbe ist.

Im Internet gibt es verschiedene Ansätze, die letztlich alle dasselbe Ziel haben, die kurzen und schwachen Spannungssignale der PIN-Diode auf eine, durch Microcontroller auswertbare Zeitdauer und Größenordnung zu bringen. Das kann rein digital geschehen, indem das eingefangene Zerfallsprodukt in einen digitalen Puls von definierter Länge übersetzt wird. Die Grundlage für die hier verwendete Schaltung habe ich vor Jahren in Elektor gefunden. Dem Ausgang des Verstärkerteils habe ich noch einen Hochpassfilter spendiert, der niederfrequentes Rauschen bis 300Hz unterdrückt und das Nutzsignal deutlich verbessert. Der 33kΩ-Widerstand dient der Pegelanhebung, damit am ESP32/ESP8266-Analogeingang keine negativen Spannungen auftreten können. Hier ist das Schema der Schaltung.

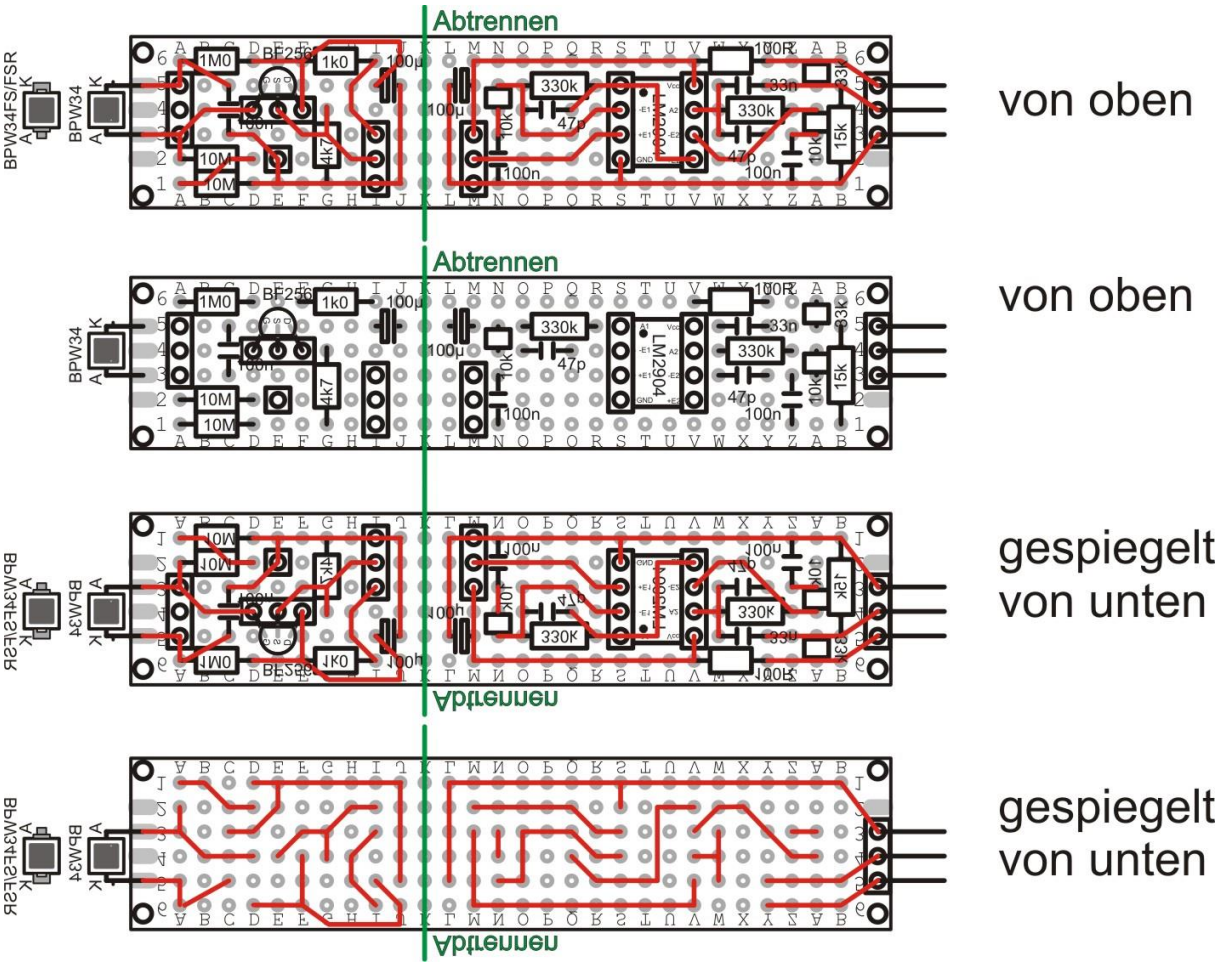


Der linke Teil bis zur grünen Linie ist der Sensorteil, der in die Dose verbannt wird. Drei Leitungen führen heraus und werden an der Verstärkerplatine angeschlossen, die den rechten Teil des Schemas ausmacht.

Die Umsetzung der Schaltung auf der Platine zeigt die nächste Abbildung. Beim obersten Bild schaut man von oben auf Bauteile und Platine. Die Leiterbahnen liegen so, als wäre die Platine durchsichtig. Im zweiten Bild, gleiche Blickrichtung, sind die Leiterbahnen weggelassen, damit man die Beschriftung der Bauteile besser lesen kann.

Im dritten Bild ist die Platine um die Längsseite gekippt, und Sie schauen von unten auf die Leiterbahnen. So etwa muss das aussehen, wenn Sie mit Löten fertig sind. Die Leitungen werden einfach durch Umbiegen der Bauteilbedrahtung und schrittweises Verlöten hergestellt. Achten Sie aber darauf, dass in manchem Verlauf mehrere Anschlüsse auftreten. Löten Sie erst dann, wenn alle Drähtchen ein Loch gefunden haben. Nachträglich wieder eine Bohrung von Lötzinn zu befreien ist schwierig!

Die Bestückung erfolgt stets von oben. Kontrollieren Sie gewissenhaft Position und Ausrichtung der Bauteile. Fotodiode BPW34, Transistor BF256 und die Elkos (Elektrolytkondensatoren) müssen gemäß ihrer Polung richtig herum eingesetzt werden. Wenn Sie von links beginnen, können Sie Ihre Fortschritte auch schön am Schaltplan verfolgen. Die Koordinaten mit Buchstaben und Ziffern helfen Ihnen dabei.



Für die Fotodiode, den Transistor und das IC LM2904=LM358 werden Teile der Buchsenleiste als Fassung verwendet, das erleichtert einen eventuellen Austausch. Für die Steckverbindungen von Platine zu Platine sowie zum ESP32 werden Teile der Stiftleiste verwendet. Die Verdrahtung zum Verstärkerteil selbst können Sie dann mit Jumperkabeln erledigen.

Die folgenden Abbildungen sollen Ihnen helfen, die Bauteile und deren Lage zu identifizieren.

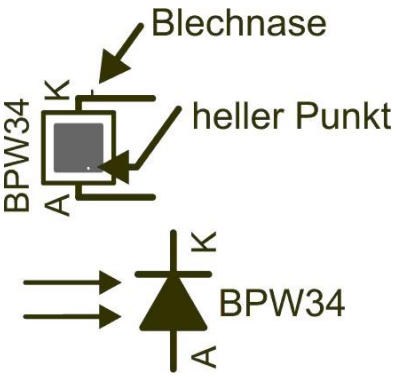
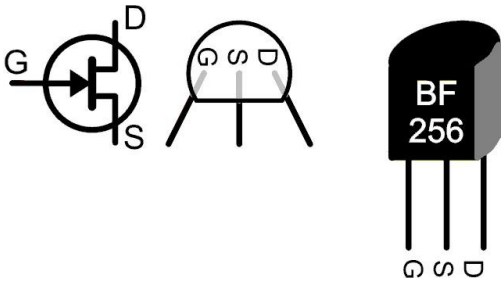
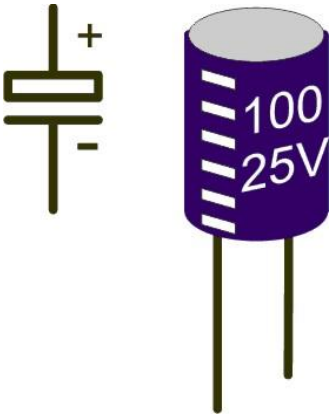


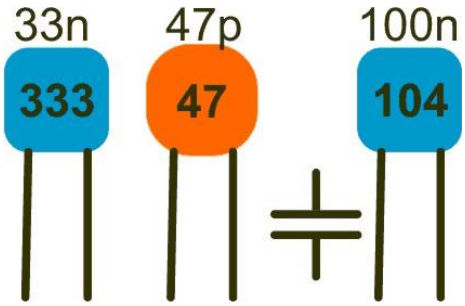
Foto Diode BPW34



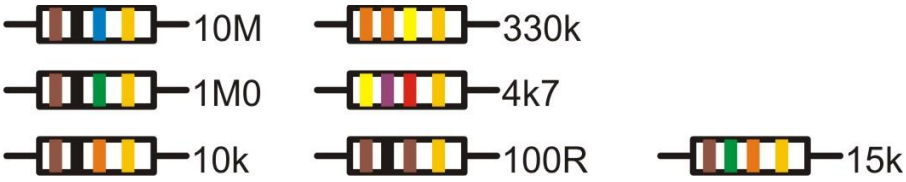
Transistor BF256



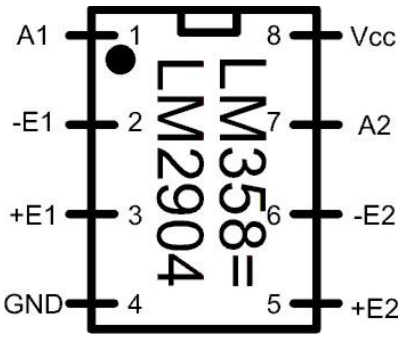
Elektrolytkondensator (Elko)



Keramikkondensatoren

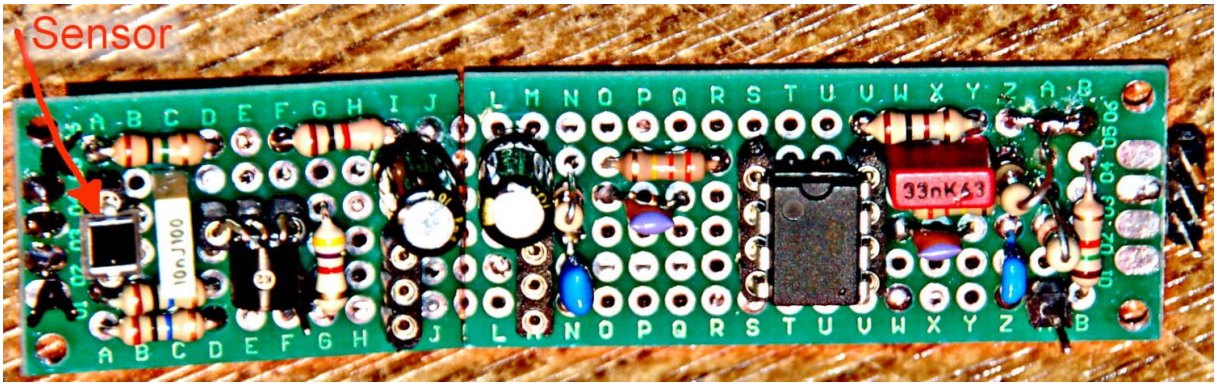


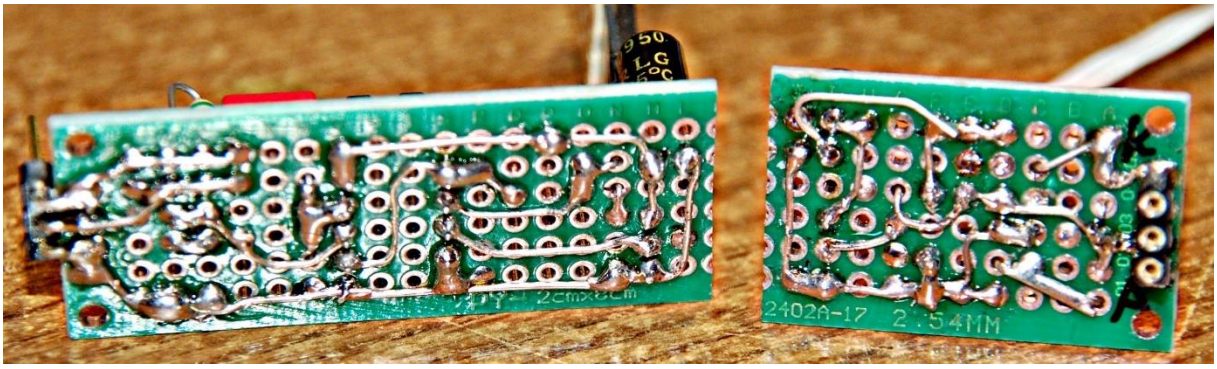
Widerstände, der 33kΩ-Widerstand mit or,or,or,gold ist nicht abgebildet.



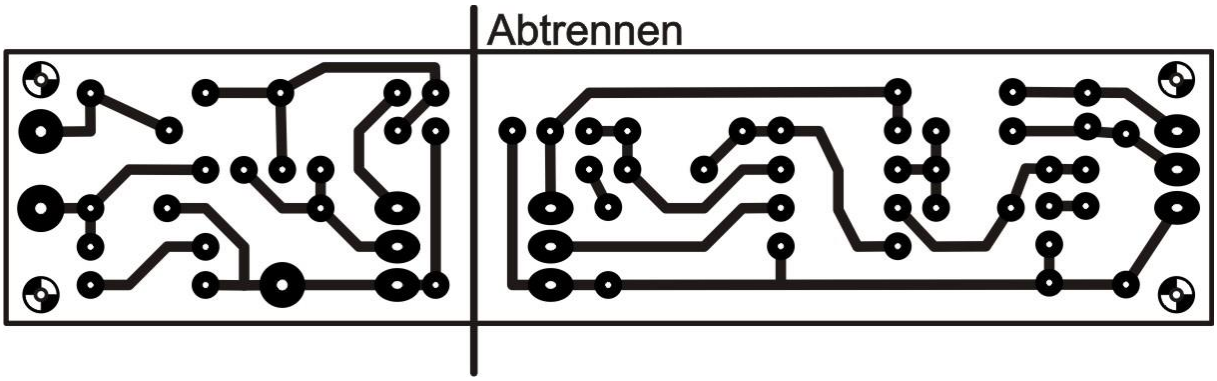
Zweifach-Operationsverstärker LM358 alternativ LM2904

Nach kompletter Bestückung wird die Platine an der Spalte K getrennt. Danach versäubern Sie die Kanten mit Hilfe des Schleifpapiers. Das Ganze sollte dann etwa so aussehen. (Die Buchsenreihen, bei I und M im Bild, wurden später durch Stiftreihen ersetzt. Der Transistor ist nach unten gebogen.)





Für Experten, die selbst Platinen herstellen können, habe ich auch noch dieses Layout, das mit Bestückungsplan als PDF-Datei heruntergeladen werden kann. Die Herstellung von Platinen durch die Bügelmethode habe ich hier genauer beschrieben.



Die fertige Schaltung braucht keinen Abgleich und sollte von Anfang an in der Dose ihren Dienst tun. Damit sie das fehlerfrei kann, muss die Dose innen mit einem nichtleitenden Material ausgekleidet werden, um Kurzschlüsse zu vermeiden. Ich habe dazu dicke Klarsichtfolie verwendet und mit Klebepads an Boden und Wand befestigt.

Einen Pin der Stiftleiste habe ich mit etwas Schaltdraht verlötet und diesen dann mit der Schraube fest an der Dosenwand montiert. Mittels Jumperkabel wird die Verbindung mit der Sensorschaltung zum Pin an Position F1 hergestellt. Diese Masseverbindung ist **unbedingt nötig**, um die Schwingneigung der Schaltung zu unterdrücken. Die Platine wird nun so am Boden angebracht. dass man den Sensor gut an eine Strahlenquelle annähern kann. Was hierzu dienen kann, dazu komme ich gleich. Zum Anschluss an die Verstärkerplatine brauchen Sie drei Jumperkabel fm – fm für Vcc, Signal und GND. Wenn die Kabel zwischen Deckel und Dose herausgeführt werden können ist das gut, falls nicht, müssen sie durch ein Loch in der Seitenwand geführt werden. Natürlich muss man das Loch wieder lichtdicht verkleben.

Proben für den Strahlungsmesser

Kommen wir zu den Proben. Nachdem kaum jemand einen Kastor im Keller hat, an dem er Strahlung messen kann, müssen wir uns nach einfachen Alltagsgegenständen umsehen, die Kernstrahlung emittieren.

In diesem Zusammenhang weise ich darauf hin, dass ein Bezug der Messwerte zum Größensystem des SI in keiner Weise gegeben ist. Es ist also nicht möglich, mit diesem einfachen Gerät Aussagen über die Energiedosis D in Gray (Gy) oder die Äqivalentdosis H = q • D in Sievert (Sv) zu machen, wie dies im Elektorartikel behauptet wird. Selbst eine Angabe in Becquerel (Bq) ist nicht möglich, weil dafür die Masse und die Art des Strahlers bekannt sein müssten. Bei den hier aufgeführten Proben kann man beides nicht festlegen. Dieser Artikel beschränkt sich daher auf die Angabe **cpm** = **c**ounts **p**er **m**inute = Zerfallsakte pro Minute und die Darstellung **relativer** Energiewerte.

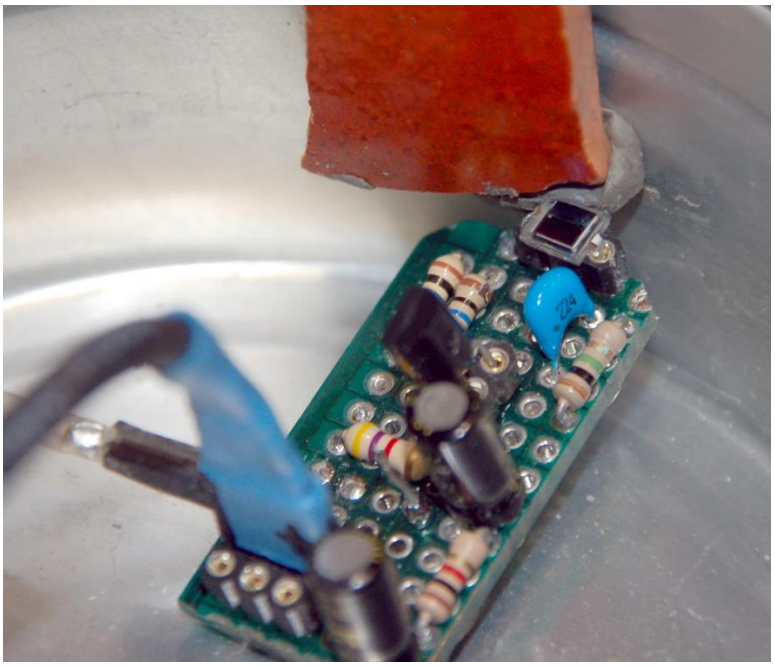
Hier nun ein paar Fotos von Gegenständen, die in Frage kommen. Eine umfangreichere Darstellung finden sie hier.

Die orange Uranglasur der Dose ist ein guter Teststrahler, ebenso der Schwarzumdruck der Tasse oder die Unterglasurfarbe der Katze. Wesentlich weniger

effektiv ist Uranglas, das meist in grün oder gelb auftritt, aber durchaus auch andere Farben aufweisen kann. Eine wesentliche Eigenschaft von Uranglas ist es, in UV-Beleuchtung grün zu fluoreszieren. Der Kaliumgehalt der Glimmerminerale bringt nur schwache Gammasignale. Dagegen ist der Radiumgehalt der Leuchtfarbe auf Zeigern und Zifferblättern alter Uhren gut nachweisbar. **Wenn man Teile davon herausnimmt sollte man diese zum Experimentieren in eine luftdichte Kunststoffhülle verpacken, damit Staub oder kleine Partikel davon nicht eingeatmet oder verschluckt werden können. Deswegen sollte man während der Experimente mit den Teilen von Uhren auch nicht essen, trinken oder rauchen. Und danach Hände waschen.**

Die folgenden Dinge findet man häufig auf Trödelmärkten.

Wer nichts davon daheim hat, kann es mit **Pottasche** (Kaliumcarbonat) oder **Diätsalz** (Kaliumchlorid) aus dem nächsten Supermarkt versuchen. Natürliches Kalium enthält zu 0,0117% das radioaktive Isotop Kalium40 (K40). Dieses wandelt sich zu ca. 90% unter Betazerfall in Calcium40 um. Die restlichen ca. 10% der Zerfallsereignisse beruhen auf dem Einfang eines Elektrons aus der K-Schale durch den Atomkern und nachfolgender Emission eines Gammaquants. Dabei entsteht Argon40. Unser Sensor erfasst davon ca. alle ein bis zwei Minuten einen solchen Vorgang, das liegt auch an der sehr kleinen Sensor-Fläche von gerade mal ca. 7mm². Das Ergebnis ist ein Spannungspuls von bis zu 300mV am Verstärkerausgang, der deutlich über das Grundrauschen hinausgeht. Uranglasuren liefern Pulse die auch mehr als doppelt so hoch sein können. Der Uranglasurscherben im Foto wird zur Messung auf den Sensor heruntergeklappt.

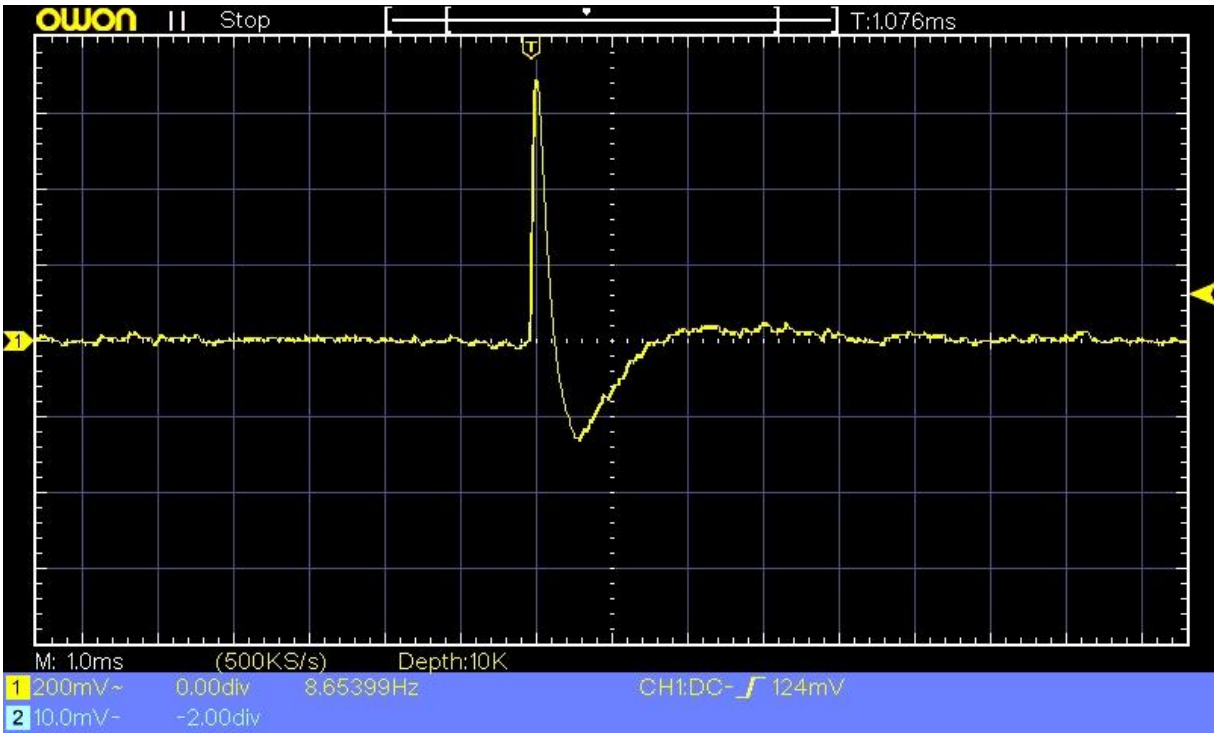


Grundbedingungen bei den Messungen sind

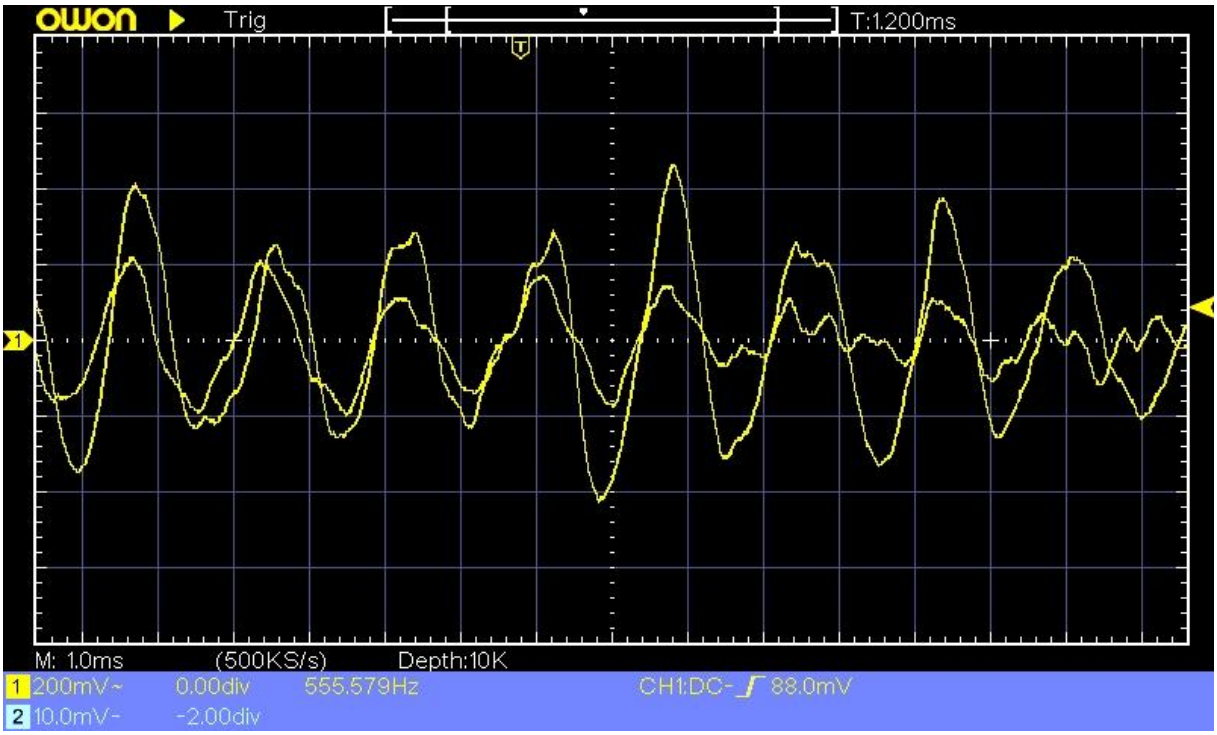
- möglichst störungsfreie Spannungsversorgung von 5V
- abschotten des Sensors von Umgebungslicht in der Dose
- Schließen des Deckels mit leitender Verbindung zur Dose
- Stille – jedes noch so kleine Kratzen am Tisch oder normales Sprechen führt zu Störungen am Verstärkerausgang.

Den letzten Punkt übersieht man gerne. Die Dose wirkt nämlich im Nebeneffekt als extrem empfindliches Kondensatormikrofon. Leichtes Kratzen am Tisch ergibt bereits Signale am Ausgang, die sogar drei- bis viermal höher sein können als das Nutzsignal.

Ideal ist es natürlich, wenn man ein Oszilloskop zur Verfügung hat, weil man damit die Signalsituation am Verstärkerausgang bestens kontrollieren kann. Der sehr kräftige Puls mit 700mV auf dem nächsten Bild stammt von einem Stück Uranglasur unmittelbar vor dem Sensorfenster. Etwas mickriger fallen die Signale aus, wenn man die Probe statt unmittelbar vor dem Sensor in der Dose, außen auf den Dosendeckel legt. Es ist dann das Aluminium zu durchdringen und die ca. 3cm Luft in der Dose bis zum Sensor. Aber der Sensor spricht trotzdem immer noch an.



Kratzen am Tisch sieht dagegen so aus.


OWON
Trig
T:1.200ms
M: 1.0ms
(500KS/s)
Depth:10K
1 200mV~ 0.00div 555.579Hz CH1:DC- 88.0mV
2 10.0mV~ -2.00div

Programmieren und Messen

Kommen wir jetzt zu den Messungen. Was wir brauchen, ist ein Programm das den Pegel am Verstärkerausgang überwacht, um beim Auftreten eines Spannungspulses möglichst genau dessen Maximum zu ermitteln. Der maximale Spannungswert ist für uns ein relatives Maß für die in der Intrinsic-Zone abgegebene Energie des auslösenden Gammaquants. Weil, wegen der dennoch dünnen Schicht, nicht sichergestellt ist, dass die gesamte Gammaenergie zur Ladungstrennung absorbiert wurde, spiegelt sich in der Größe des Spannungspulses auch nicht unbedingt die Gesamtenergie des Gammaquants, sondern eben nur absorbierte Anteil wider.

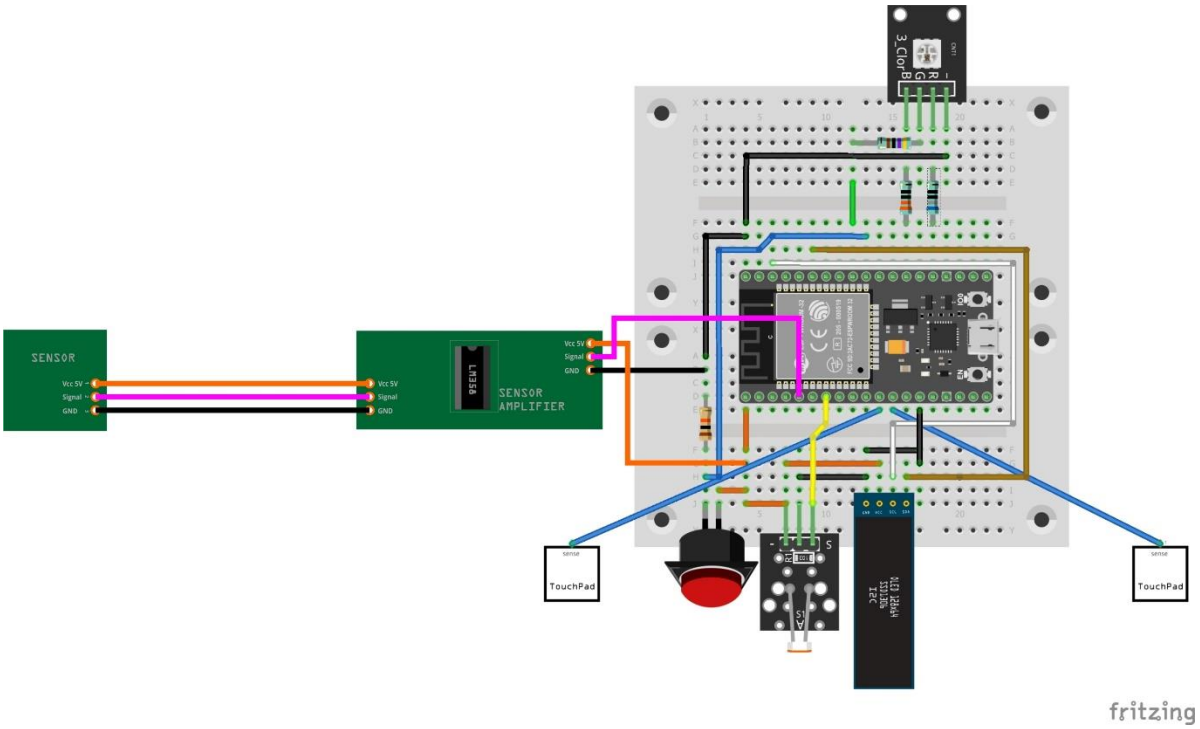
Absolute Energiewerte können wir mit dem Gerät sowieso nicht messen, weil es nicht kalibriert ist, uns fehlt einfach der zuverlässige Vergleich zwischen der Ursache (Gammaenergie) und der Wirkung (Spannungspuls). Was wir aber machen können, ist eine Aufschlüsselung der Häufigkeit, mit der bestimmte absorbierte Energieportionen auftreten. Mit Hilfe des OLED-Displays können wir ein "Gammaspektrum" der Strahlungsquelle erstellen. Aha, sagen Sie jetzt sicher, dazu brauchten wir die Balkendiagramme, und damit haben Sie recht. Wahrscheinlich haben Sie auch in Sachen Geschwindigkeitsmessung beim ADC einen Verdacht. Den kann ich ebenfalls bestätigen, denn damit fangen wir an.

Aus dem Pulsdiagramm oben können Sie ablesen, dass der positive Puls etwa 0,3 bis 0,4 Millisekunden dauert. Da muss sich der ESP32 mit ca. 13 Messungen pro Millisekunde ranhalten, um den Spitzenwert mitzubekommen. Für einen ESP8266 ist es mit 4 bis 5 Messungen in dieser Zeit eher ein Zufall, wenn er genau den Spitzenwert erwischt. Solcherart Überlegungen sind für Auswahl eines Controllers für zeitkritische Aufgaben sehr wichtig und das spricht hier eindeutig eher für den ESP32. Versuche haben aber gezeigt, dass der ESP8266-12F dennoch gute Ergebnisse bringt.

Nächster Punkt, wann soll die Vermessung eines Pulses beginnen? Klare Antwort, dann, wenn der Spannungspegel das Grundrauschen übersteigt. Diesen Wert, **noise**, des Basisrauschens müssen wir bestimmen, ebenso den Ruhepegel **ruhePegel** und den maximal zu erwartenden Wert **cntMax**. Ich habe weiter unten eine Grafik eingefügt, die das Zusammenspiel der verschiedenen Pegel zeigt.

Die Schaltungen für ESP32 und ESP8266 sehen unterschiedlich aus. Ich beginne mit dem ESP32.

Der ESP32 als Messknecht



Die Spannung am Verstärkerausgang der Sensoreinheit lege ich an Pin 34 des ESP32. Die Bitbreite des ADC stelle ich auf 10bit und als Spannungsbereich wähle ich 3,3V für den Anfang. Alles zusammen packe ich in das Testprogramm **ruhepegel**.py. Die Probe wird aus der Dose entfernt, dann starte ich die Messung, die während 10 Sekunden den durchschnittlichen ADC-Pegel **ruhePegel** am Verstärkerausgang bestimmt. Notieren Sie diesen Wert, wir brauchen ihn noch.

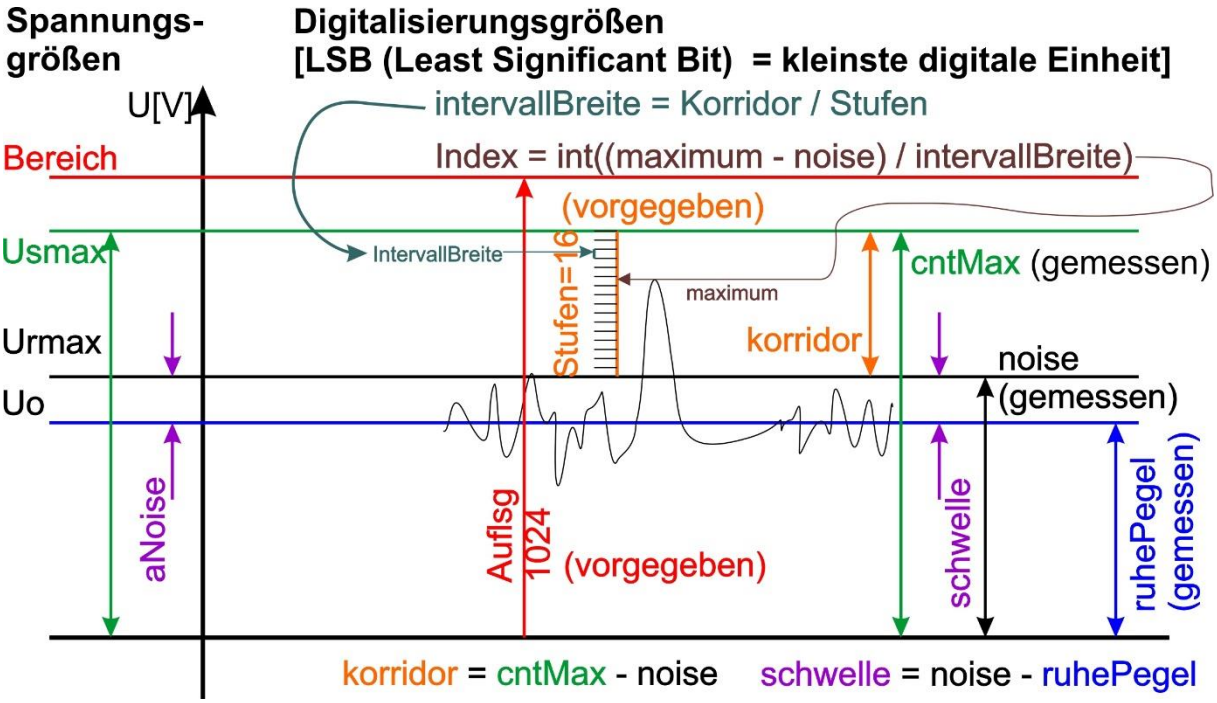
Als nächstes ist die Amplitude des Rauschsignals wichtig. Das Programm **maximum**.py bestimmt dazu den absoluten Spitzenwert auf der Signalleitung. Wenn **keine Probe in der Dose** ist, liefert das den maximalen Wandlerwert **noise** der Störspitzen, der durch das Rauschen hervorgerufen wird. Den zugehörigen ADC-Wert **maximum** notieren wir uns wieder. Das maximale Rauschsignal in ADC-Einheiten (LSB) ist damit **noise = maximum**.

Mit dem gleichen Programm **maximum**.py bestimmen wir jetzt noch den maximalen ADC-Wert, **cntMax**, der durch die Pulse vom Strahlungssensor entsteht. Das sagt uns, welchen Spannungsbereich wir für die Messungen einstellen müssen. Wir platzieren die Probe in der Dose am Sensor und starten die Messung für mindestens 60 Sekunden. Den ADC-Wert **cntMax** notieren wir uns auch.

Meine Werte waren **ruhePegel** = 449, **noise** = 500 und **cntMax** = 713. Das entspricht Uo = 449 / 1023 * 3,3V = 1,45V Ruhepegel. Die **Rauschamplitude**, **noise – ruhePegel**, ist 51 LSB, das entspricht 164 mV. Alles was darüber hinausgeht können wir als Nutzsignal werten. Mit 713 LSB berechnen wir den maximalen Spannungspegel zu 2,3V. Das sind mehr als 2,0V und das heißt, dass wir bei dieser Probe mit dem bereits eingestellten Bereichswert 11DB weiterarbeiten müssen, um den ADC-Eingang nicht zu überlasten.

Die folgende Grafik verdeutlicht die Zusammenhänge. Die Spannungen interessieren nur für die Abschätzung des Eingangsspannungsbereichs des ESP32. Ansonsten beschränken wir uns auf die Wandlerwerte, weil es hier nur um relative ADC-Werte und nicht um absolute Spannungsgrößen geht.

Die drei bislang ermittelten Werte **ruhePegel**, **noise** und **cntMax** werden jetzt in das eigentliche Messprogramm **messen**.py übertragen, um dort fürs Erste als Defaultwerte zu dienen. Einen Überblick über die diversen Größen liefert die folgende Grafik.



Werfen wir jetzt einen Blick auf die Programme. Der Import der nötigen Module, die Instanzierung und Belegung der Startwerte, die Vorbereitung der Zeitsteuerung sowie die Auswertung der Messungen stellen den größten Umfang in **ruhepegel**.py und **maximum**.py dar. Das Wesentliche passiert aber in den while-Schleifen mit einer beziehungsweise zwei Befehlszeilen .

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)))
```

Für den Ruhepegel werden alle Messwerte aufaddiert und zum Schluss für die Anzeige durch die Anzahl Messungen dividiert. Nach den Regeln der Statistik ergibt das bei sehr vielen Messungen den Ruhepegel als Mittelwert.

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 der Schleife von **maxmum**.py wird einfach fortlaufend der Wert **pegel** eingelesen, der den Wert **maximum** immer dann ersetzt, wenn er den bisherigen Wert von maximum übersteigt. Das wird einmal für die Störspannungsspitzen ohne Probe gemacht und ein zweites Mal für die Messung der höchsten Spannungspulse mit eingelegter Probe.

Die grundsätzliche Funktion des eigentlichen Messprogramms **messen**.py habe ich oben bereits kurz umrissen. Jetzt schauen wir genauer hin. Die Zeilen, auf die ich näher eingehe, sind im Listing fett formatiert.

Download: 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:{} CPM".format(int(activity)),0,0)
d.writeAt("{} EVENTS IN".format(events),0,1)
print(spektrum)
```

Wie arbeitet das Programm?

Neben den üblichen Vorbereitungsarbeiten werden unsere drei Werte eingebaut und drei weitere berechnet. Es folgen bereits bekannte Einstellungen.

Einige Zeilen weiter unten erzeugen wir ein OLED-Objekt, legen die Messdauer fest, löschen den Wert für die Aktivität und erzeugen eine leere Liste, die wir sogleich mit Nullen füllen, für jede Stufe brauchen wir einen Eintrag. Zerfallszähler auf Null, Rotlicht an für Messbeginn. Anzeige löschen, Zeitsteuerung einrichten, ausgeben und los geht's.

Die erste while-Schleife ist der Messzeit-Manager, das kennen wir auch schon.

Maximalwert auf Rauschpegelobergrenze, Pegel einlesen und fortlaufend überprüfen, ob wir uns noch innerhalb des Rauschbereichs befinden. Die Schleife wird verlassen, wenn das nicht mehr der Fall ist, der Pegel also im Nutzsignalkorridor liegt oder die Messzeit überschritten ist. Sind wir in der Zeit, wurde also ein Zerfall detektiert, lösen wir einen Beep mit LED-Blitz aus und setzen das Maximum auf den neuen Pegelwert. Unabhängig vom Verlauf des Hauptprogramms wird der Beep nach 5 ms durch die Callbackfunktion des Timerinterrups gestoppt.

Die nächste while-Schleife wartet so lange, bis der ADC-Pegel wieder im Rauschbereich liegt, vergleicht aber in dieser Zeit Pegel und Maximum. Liegt der Pegelwert höher, ersetzt er das bisherige Maximum.

Nach dem Verlassen der Schleife erhöhen wir den Ereigniszähler und geben gegebenenfalls das Maximum zur Kontrolle aus. Letzteres kann im Stand-Alone-Betrieb wegfallen, da dann kein Terminal angeschlossen ist. Das gilt auch für alle anderen Printbefehle.

In seltenen Fällen kann der eben festgestellte Maximalwert über cntMax liegen. Weil dafür aber kein Stufenspeicher vorgesehen ist, wird der Wert auf ein LSB unter **cntMax** gesetzt. Das ist nötig, um im Anschluss Indizierungsfehler in die Liste **spektrum** zu vermeiden. Würde nämlich aufgrund des Werts in maxCount ein Index berechnet, der größer als 15 wäre, dann stiege das Programm mit einer Fehlermeldung aus, weil für diesen Index kein Listenelement in spektrum existiert. Wenn dieser Schritt notwendig war, wird der Zähler für die Bereichsüberschreitung erhöht.

Weil Zerfallsereignisse mit Werten innerhalb des Rauschbereichs zwar sehr wahrscheinlich stattfinden, aber nicht detektiert werden können, sind dafür auch keine Zählerspeicher vorgesehen. Deswegen wird der Rauschpegel vom Maximum abgezogen, aus diesem Wert dann der Index in die Liste spektrum berechnet und dann der entsprechende Levelzähler erhöht. Abschließend wird die Messzeit aktualisiert.

Nach dem Verlassen der Messzeitschleife machen wir das Rotlicht aus und informieren im Display über die Ergebnisse. Nun können Sie schon einmal nach Belieben mit Ihren Proben experimentieren, aber es warten auch noch Erweiterungen auf Sie.

Autostart und der Stand-Alone-Betrieb

Für erste Tests sind die drei einzelnen Programme gut geeignet, weil sie in überschaubarer Weise die Funktion unserer Schaltung zusammen mit dem ESP32 und seiner Peripherie demonstrieren. Für einen Stand-Alone-Betrieb ist das aber zu wenig, denn wir haben in diesem Fall ja kein Terminal zum Start der Programme zur Verfügung. Das Programm muss in diesem Fall von selbst starten, die gewünschten Aktionen ausführen und die Ergebnisse mitteilen.

MicroPython hat für den Autostart einen ähnlichen Mechanismus zur Verfügung wie die **Arduino-IDE** mit ihren **setup**- und **loop-Proceduren**. Das Setup hießt in MicroPython **boot** und ist eine eigene Datei mit diesem Namen, also **boot.py**. Auch loop wird durch eine eigenständige Datei mit dem Namen **main.py** ersetzt. Dieser Programmteil **muss im Gegensatz zum loop-Konstrukt der Arduino-IDE, eine Endlosschleife in der Form while True enthalten**, sonst bricht nach einem Durchlauf das Programm im Nirvana ab.

Warnung!

> **Es ist schwierig, einen ESP32 nach einem Autostart ohne definiertes Programmende anzuhalten, um Änderungen am Programm vorzunehmen. Bauen Sie unbedingt nach dem Start in boot.py eine zeitgesteuerte Warteschleife ein, während der ein Abbruch über Strg+C von Thonny oder µPyCraft aus via USB möglich ist. Um abzubrechen drücken Sie Strg+C.**

Die Notbremse ist das Programm Putty. Zuerst wird die Verbindung zum CP2102 hergestellt, Baudrate 115200. Dann startet man den ESP32 mit der Resettaste und gibt über das Puttyfenster Strg+C ein. Das sollte den Controller stoppen. Benennen Sie dann als Erstes die Datei boot.py um, der neue Dateiname ist egal.

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

Nach erneutem Neustart des ESP32 sollten Sie ihn auch wieder in µPyCraft oder Thonny kontakten können.

Geht das alles schief, hilft nur der Selbstzerstörungsknopf und der heißt esptool.py. Löschen sie den Speicher wie unten beschrieben und flashen Sie MicroPython neu.

Für den Autostart habe ich die drei bisherigen Programme in Funktionen umgewandelt und das Ganze noch um vier weitere Funktonen aufgestockt. Das alles wurde der Datei bootpart.py hinzugefügt, die für erste Tests dient. Wenn alles tadellos funktioniert, kopieren Sie den gesamten Inhalt dieser Datei in die Zwischenablage und fügen alles dem bisherigen Inhalt von boot.py hinter der letzten Zeile hinzu. Nach dem Abspeichern wird die neue boot.py ins Device hochgeladen.

Zwei von den neuen Programmen dienen der bequemeren LED-Steuerung, das neue **jaNein**() fragt mit zusätzlicher Zeitsteuerung die Touchpads ab und **report**(), gibt die Werte der **spektrum**-Liste als Balkengrafik aus, weil ja jetzt wohl kein Terminal mehr dafür vorhanden ist – aber natürlich dennoch weiterhin angeschlossen sein darf.

Die Touchpadabfrage mit Timeout kennen Sie bereits von früheren Hausaufgaben. Eine Änderung habe ich in **touch**.py an der Methode **getTouch**() vorgenommen. Die gibt jetzt statt ADC-Wert oder Millisekunden die Werte **True**, **False** oder **None** zurück, entsprechend **berührt**, **nicht berührt** oder **Timeout**.

jaNein() nimmt die optionalen Parameter **meldung** und **laufZeit**. Der String in meldung wird in Zeile 1 am Display ausgegeben, laufZeit stellt einen Timoutwert dar, nach welchem die Funktion beendet wird, falls keine Touchaktion stattgefunden hat. Der Rückgabewert von jaNein() richtet sich nach dieser Tabelle.

laufZeit	Aktion ja	Aktion nein	Rückgabewert
0	-	-	- (endlos warten)
0	x	-	2
0	-	x	1
0	x	x	3
>0	x	-	2
>0	-	x	1
>0	x	x	3
>0	-	-	0 (Timeout)

Rückgabewerte auf Touchpad-Aktionen

Die Rückgabewerte sind in den Konstanten JA=2, NEIN=1, BEIDE=3 und TIMEOUT=None codiert.

Die Funktion **report**() nimmt als optionalen Parameter eine der Konstanten **Fcut**, **Fprop** oder **Flog**, um die Höhe der Säulen an die Werte in **spektrum** und an die Höhe des Displays (in d.HEIGHT) anzupassen. Die Tabelle erleichtert die Auswahl.

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(spektrum) =d.HEIGHT herunter gerechnet.
Fprop	<= d.HEIGHT	alle Werte werden proportional zu max(spektrum) = d.HEIGHT gestreckt.
Flog		alle Werte werden als log(spektrum[i]) logarithmisch dargestellt, wobei log(max(spektrum)) = d.HEIGHT.

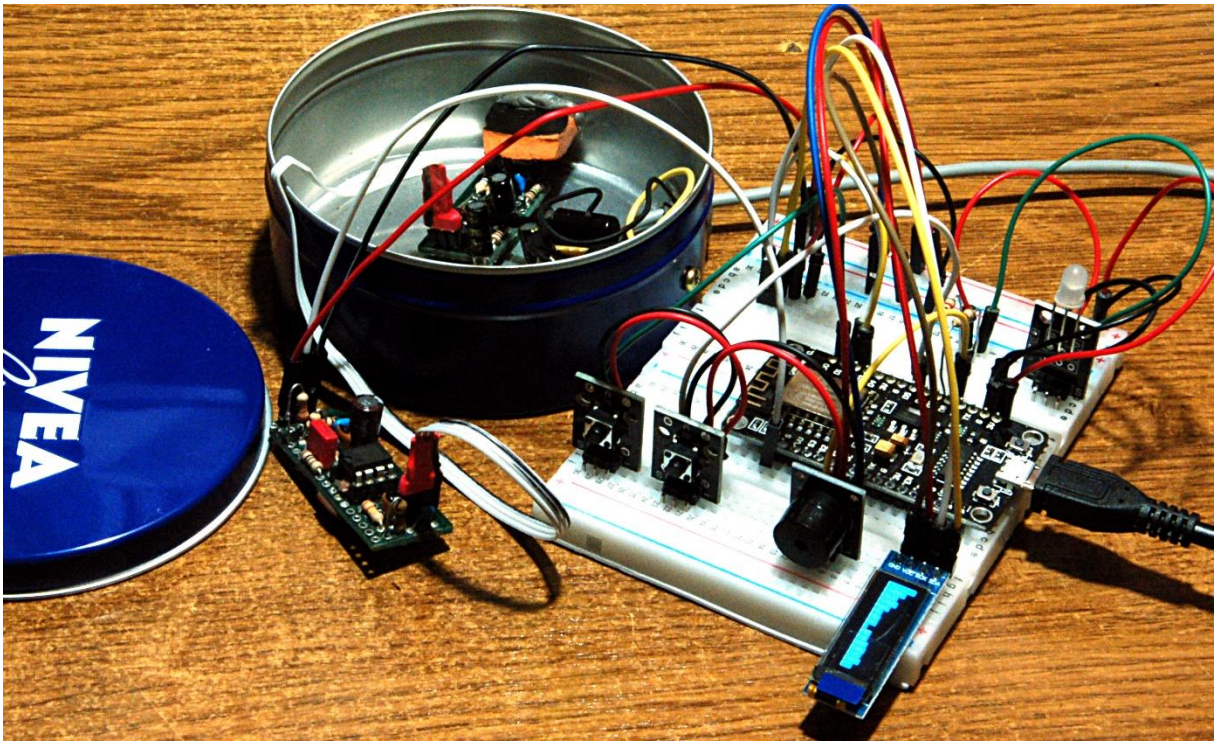
Nach dem Start können die Werte für ruhePegel, noise und cntMax neu erfasst werden, falls die Touchpadabfrage innerhalb von 10 Sekunden mit "ja" beantwortet wird. Bei cntMax gibt es kein Zeitlimit, weil die Probe in Ruhe eingesetzt werden muss. Wenn Sie möchten, können Sie das ja ändern.

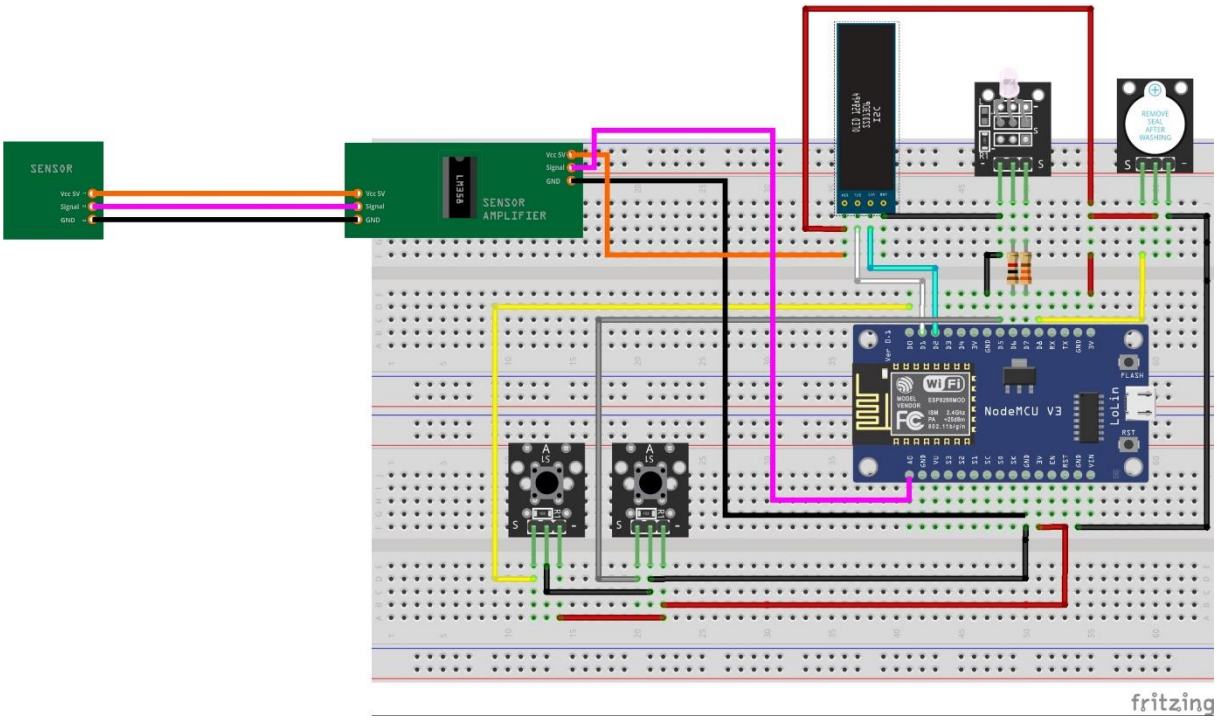
Die Werte werden für 3 Sekunden angezeigt, dann wird boot.py beendet und automatisch main.py gestartet, das in einer Endlosschleife über Touchpad gesteuert die Messfunktion aufruft und die Ergebnisse am Display darstellt. Die Lösung zu Hausaufgabe 6 kann Ihnen als Anregung dazu dienen, zwischen der Werte- und Grafikanzeige fortlaufend umzuschalten, bis ein Touchpad berührt wird, um eine neue Messung zu starten. Auf die Wiedergabe der fertigen bootpart.py und boot.py

sowie mainpart.py und main.py habe ich hier verzichtet, weil sie nur Wiederholungen und Zusammenfassungen von bereits bekannten Inhalten darstellen. Aber Sie können ja die Downloads zum Studium der Dateien nutzen.

Kann der ESP8266 das auch alles?

Ja, das kann er, mit gewissen Anpassungen, deren Umfang sich in Grenzen hält. Diese Änderungen im Vergleich zum ESP32 betreffen die Touchpads, die durch Taster ersetzt werden, zusammen mit der Treiberdatei touch8266.py sowie die Initialisierung des analogen Eingangs. Bei den LEDs wird ein Duo-LED-Modul (rot an D6 und grün an D7) und die im ESP8266-12F eingebaute blaue LED an Pin GPIO2 = D4 verwendet. Diese LED ist LOW-aktiv, weshalb die entsprechenden Stellen in den Funktionen **ledOn**() und **ledOff**() sowie im Modul **beep** angepasst werden müssen. Hier kommen ein Foto vom Aufbau und die Schaltskizze von Fritzing.





Der Strahlungssensor wird mit Vcc an **Vin = 5V** des ESP8266 gelegt, GND an GND und das Signal an A0. Achten Sie bitte darauf, dass die Bezeichnungen der Anschlüsse des OLED-Displays bei Ihrem Exemplar von denen im Schaubild abweichen können.

Die Programme für den ESP8266 sind in der folgenden Liste verlinkt. Die Bedienung des ESP8266-Boards folgt ansonsten den Beschreibungen zum ESP32.

beep.py
boot.py
bootpart.py
main.py
mainpart.py
maximum.py
messen.py
oled.py
ruhepegel.py
ssd1306.py
touch8266.py

In der nächsten Folge, es ist zum Thema Kernstrahlungsmessung die letzte, werden wir zur Bedienung des ESP32/ESP8266 eine Webseite benutzen, auf der dann auch die Ergebnisse der Einstellungen und Messungen übersichtlich dargestellt werden. Dafür sorgt der Webserver aus der zweiten Folge, den wir für diesen Job kannibalisch aufmuffen werden. Einen kurzen Abstecher gibt es dann auch noch in Richtung Vererbung bei Klassen.

Diesen Teil können Sie natürlich auch als PDF herunterladen.

Hier folgt im Anschluss noch die Beschreibung, wie man esptool.py zum Flashen der MicroPython-Firmware hernehmen kann.

Der Einsatz von esptool.py beim ESP32/ESP8266

Mit der Installation von Thonny wird das Verzeichnis **X:\Benutzer\<username>\AppData\Local\Programs\Thonny\Lib\site-packages** angelegt, wenn die Installation als normaler Benutzer erfolgte. Wenn Thonny als Administrator installiert wurde, ist es das Verzeichnis **X:\Program Files (x86)\Thonny\Lib\site-packages**. In diesem Verzeichnis befindet sich neben anderen .py-Files die Datei **esptool**.py, die zum Flashen von Firmware auf den ESP32 benutzt werden kann.

esptool.py ist ein Python-Programm, das von der Kommandozeile aus gestartet werden muss. Dazu wird die von Thonny mitgebrachte Pythoninstallation genutzt. Starten Sie jetzt die Eingabeaufforderung und wechseln Sie in das entsprechende Verzeichnis, das Ihrer Installation entspricht. Ich habe Thonny als Administrator installiert und wähle daher die zweite Variante. Listen Sie die Dateien auf, esptool.py sollte mit aufgeführt sein.

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

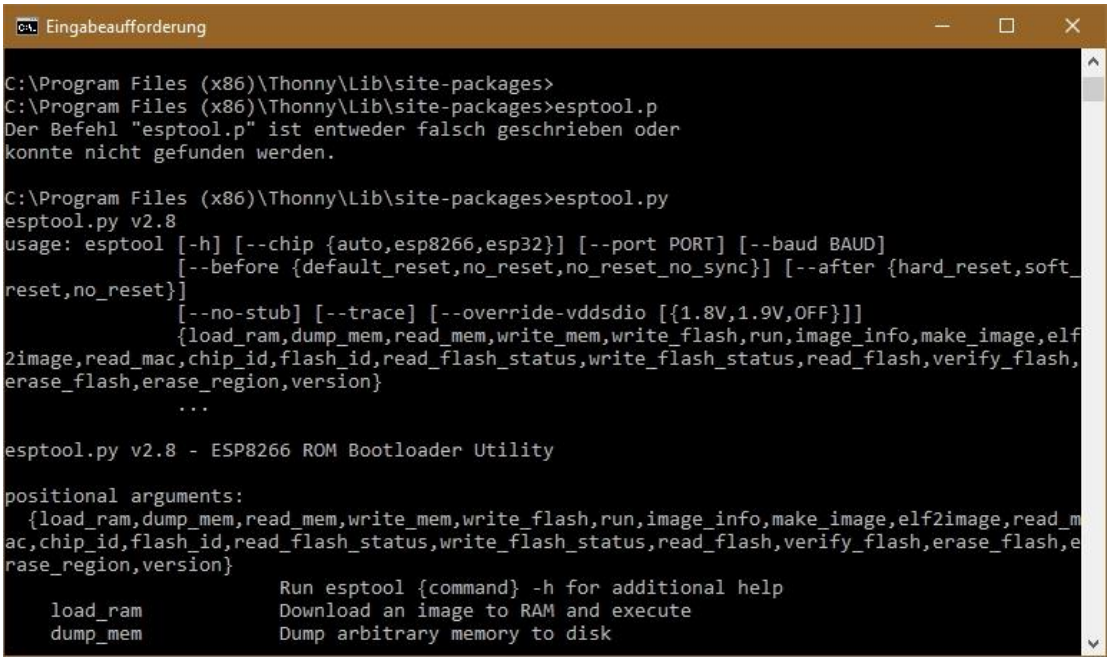
C:\Program Files (x86)\Thonny\Lib\site-packages>dir esptool.py

Verzeichnis von C:\Program Files (x86)\Thonny\Lib\site-packages

23.10.2019 05:07 143.643 esptool.py
1 Datei(en), 143.643 Bytes

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

Wenn die Bildschirmausgabe dann etwa so aussieht, ist das schon mal gut.



Bei Fehlermeldungen versuchen Sie einen der folgenden Aufrufe.

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

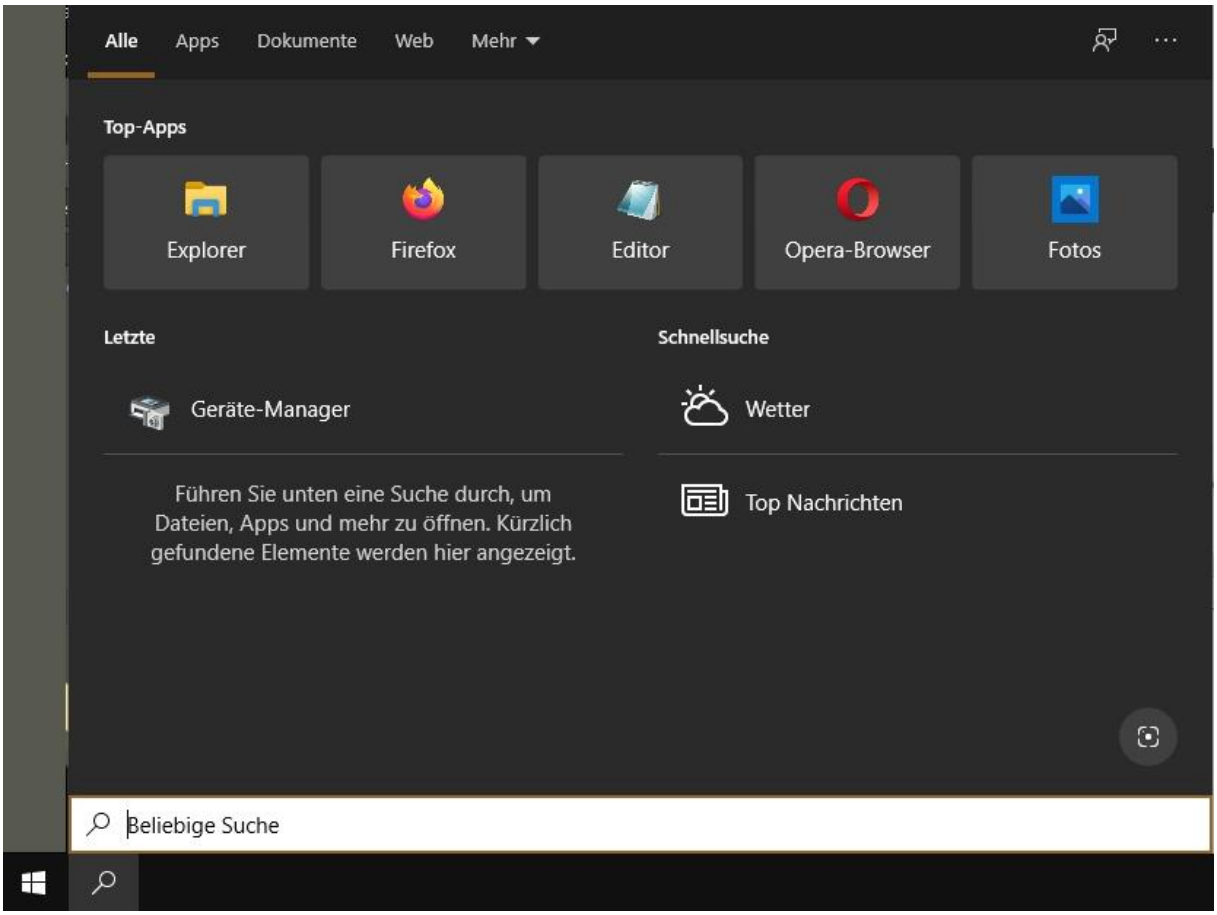
```
C:\Program Files (x86)\Thonny\Lib\site-packages>python3 esptool.py
```

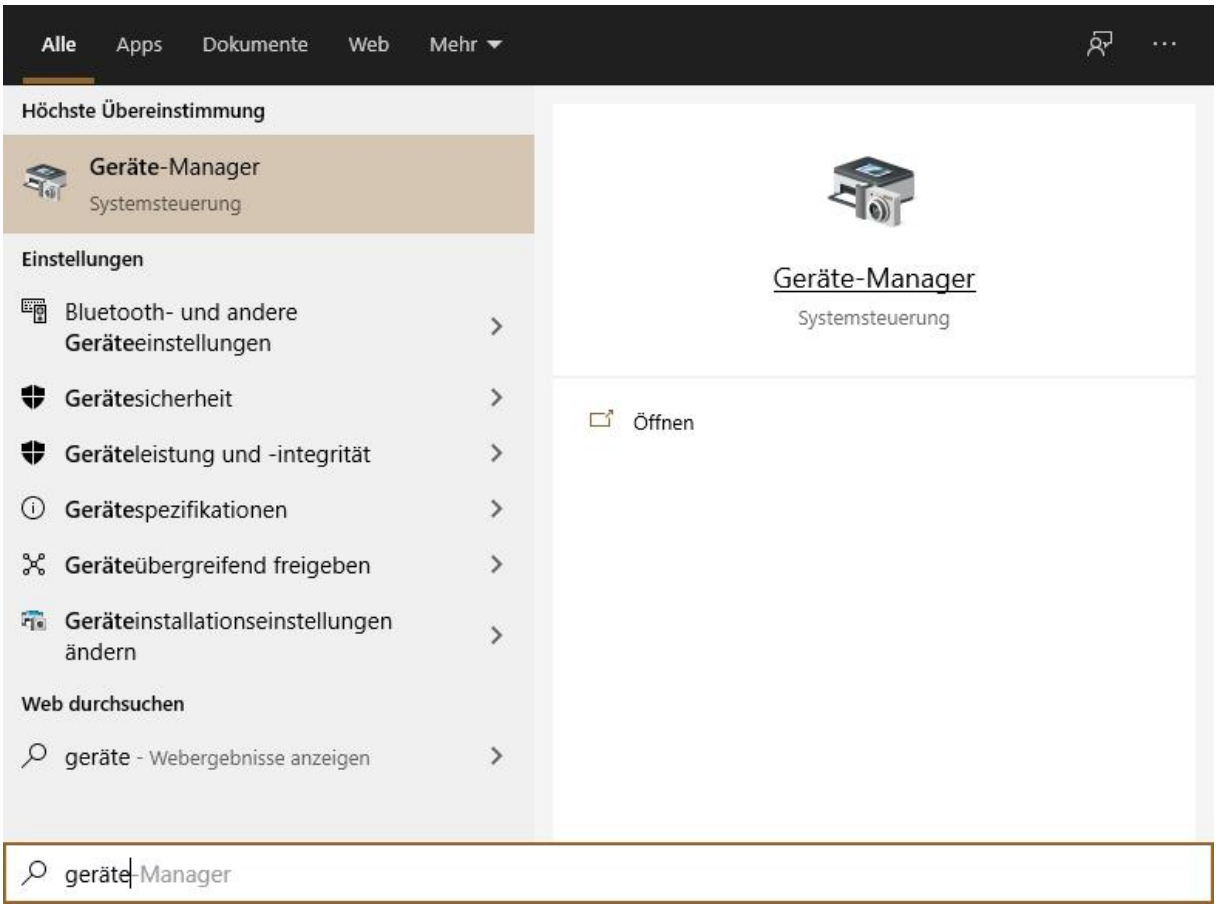
Zur bequemeren Nutzung sollte jetzt noch eine Vorbereitung getroffen werden, die mit der Art des Aufrufs von Programmen zu tun hat. Das betrifft Windows und in ähnlicher Weise auch Linux.

Damit ein Programm aus einem beliebigen Fenster der Eingabeaufforderung von der Kommandozeile gestartet werden kann, muss eine der folgenden Bedingungen erfüllt sein.

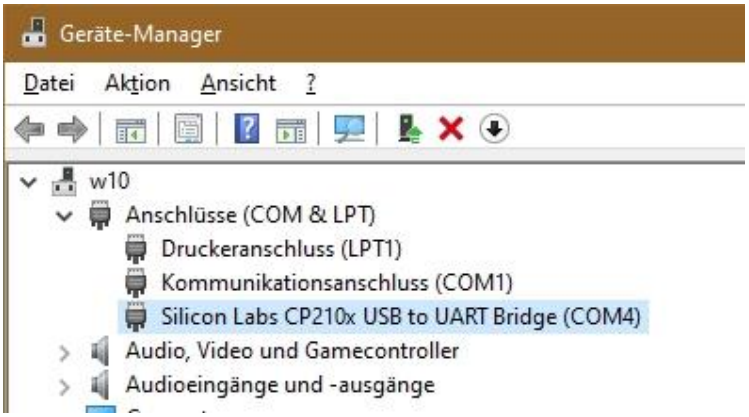
1. Die aufzurufende Datei muss sich in dem aktuellen Verzeichnis befinden, das im Eingabefenster gerade eingestellt ist.
2. Dem Dateinamen wird die relative oder absolute Pfadangabe zur zu startenden Datei vorangestellt.
3. Der Pfad zur aufzurufenden Datei muss in der Umgebungsvariable Path von Windows (oder Linux) angegeben sein.

Ich stelle jetzt das Verfahren für die drei Fälle vor. Schließen Sie ihren ESP32 am USB-Bus an und stellen Sie die Comportnummer des Anschlusses fest. Geben Sie hierzu in die Windowssuche links unten "Gerätemanager" ein.





Klicken Sie 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 Nummer sein.

Ausgangsszenario1:
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**

Die Firmware liegt also in einem ganz anderen Verzeichnis wie esptool.py. Wenn wir esptool.py von seinem Programmverzeichnis aus aufrufen, muss der Pfad zur bin-Datei mit im Aufruf angegeben werden. Beachten Sie, dass die Eingabe jeweils in ein

und derselben Zeile erfolgen muss. Es dürfen auch keine Leerzeichen in der Pfadangabe enthalten sein. Lösch- und Brennvorgang werden beide ohne Tastendruck auf dem ESP32 durch dessen Flash-Boot-Automatik gestartet.

Wir befinden uns **im Verzeichnis der Programm-Datei esptool.py**. Zuerst muss der Flashspeicher gelöscht werden. Der Aufruf dafür ist relativ kurz. Das abwechselnde Punkt-Strich-Muster kennen Sie evtl. von der Arduino-IDE, denn die benutzt ebenfalls esptool.py zum Flashen. Vielleicht sind Ihnen aber auch schon einmal die ellenlangen Aufrufe beim Flashen der Sketche in der Arduino-IDE aufgefallen. Also jetzt:

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

```
C:\Program Files (x86)\Thonny\Lib\site-packages>esptool.py  --chip esp32  --port COM4  erase_flash
esptool.py v2.8
Serial port COM4
Connecting........_
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
Crystal is 40MHz
MAC: b4:e6:2d:ac:56:b1
Uploading stub...
Running stub...
Stub running...
Erasing flash (this may take a while)...
Chip erase completed successfully in 8.9s
Hard resetting via RTS pin...

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

Als Nächstes erfolgt der Brennvorgang der Firmware und der sieht schon sehr viel hässlicher aus. Alles in eine Zeile, keine Leerzeichen in Pfadnamen.

esptool.py --chip esp32 --port COM4 write_flash -z 0x1000 F:\micropython\ESP32\firmware\latest\esp32-idf3-20200902-v1.13.bin

Wird esptool.py nicht gestartet, führt eine der folgenden Versionen zum Erfolg, was das Ganze nicht weniger hässlich macht.

python esptool.py --chip esp32 --port COM4 write_flash -z 0x1000 F:\micropython\ESP32\firmware\latest\esp32-idf3-20200902-v1.13.bin

python3 esptool.py --chip esp32 --port COM4 write_flash -z 0x1000 F:\micropython\ESP32\firmware\latest\esp32-idf3-20200902-v1.13.bin

Wenn's geklappt hat, sollte das auf dem Bildschirm so ähnlich aussehen.

```
C:\Program Files (x86)\Thonny\Lib\site-packages>esptool.py  --chip esp32 --port COM4 write_flash -z
 0x1000 F:\micropython\ESP32\firmware\latest\esp32-idf3-20200902-v1.13.bin
esptool.py v2.8
Serial port COM4
Connecting........_
Chip is ESP32D0WDQ6 (revision 1)
Features: WiFi, BT, Dual Core, 240MHz, VRef calibration in efuse, Coding Scheme None
Crystal is 40MHz
MAC: b4:e6:2d:ac:56:b1
Uploading stub...
Running stub...
Stub running...
Configuring flash size...
Auto-detected Flash size: 4MB
Compressed 1448768 bytes to 926007...
Wrote 1448768 bytes (926007 compressed) at 0x00001000 in 82.5 seconds (effective 140.4 kbit/s)...
Hash of data verified.

Leaving...
Hard resetting via RTS pin...

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

Der ESP ist jetzt bereit, um mit Thonny oder µPyCraft oder einer weiteren IDE die Programmierung in MicroPython durchzuführen.

Eine Abkürzung der elend langen Befehlszeile zum Flashen der .bin-Datei ist nur möglich, wenn man diese ins Programmverzeichnis kopiert. Dann würde die Pfadangabe wegfallen, aber das Programmverzeichnis würde nach und nach vollgekleistert. Keine gute Lösung!

Ausgangsszenario2:
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**

Wir öffnen ein Fenster der Eingabeaufforderung und **wechseln ins Verzeichnis der Firmware**. Alle Eingaben müssen in einer Zeile erfolgen. Pfadangaben dürfen keine Leerzeichen enthalten.

C:\Program Files (x86)\Thonny\Lib\site-packages>F:

F:\ >cd F:\micropython\ESP32\firmware\latest\

F:\micropython\ESP32\firmware\latest>

Der Aufruf des esptools ist jetzt schwieriger geworden, dafür ist die Referenz auf die bin-Datei kürzer, weil wir uns im Verzeichnis von **esp32-idf3-20200902-v1.13.bin** befinden. Versuchen wir, esptool.py ohne Parameter zu starten wie ganz am Anfang, bekommen wir die Meldung, dass die Datei nicht gefunden wurde. Also setzen wir den Pfad zu esptool.py mit Backslash vor den Namen.

F:\micropython\ESP32\firmware\latest>C:\Program Files (x86)\Thonny\Lib\site-packages\esptool.py

Antwort von Windows:
Der Befehl "C:\Program" ist entweder falsch geschrieben oder

konnte nicht gefunden werden.

Das Problem ist, dass im Programmpfad zwei Leerzeichen sind. Somit wird **Program** als Dateiname und der Rest als Parameter angesehen. Abhilfe schafft der Trick, den Teil des Pfades mit den Leerzeichen in Anführungszeichen zu schreiben. Mit folgender Variante klappt es schließlich.

```
F:\micropython\ESP32\firmware\latest>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,elf2imag
e,read_mac,chip_id,flash_id,read_flash_status,write_flash_status,read_flash,verify_flash,erase_flas
h,erase_region,version}
               ...

esptool.py v2.8 - ESP8266 ROM Bootloader Utility
```

Die beiden Aufrufe zum Löschen und Brennen des Flashspeichers müssen also jetzt wie folgt aussehen. Der erste Aufruf wurde hässlicher und der zweite nicht schöner.

C:\"Program Files (x86)"\Thonny\Lib\site-packages\esptool.py --chip esp32 --port COM4 erase_flash

C:\"Program Files (x86)"\Thonny\Lib\site-packages\esptool.py --chip esp32 --port COM4 write_flash -z 0x1000 esp32-idf3-20200902-v1.13.bin

Auch hier gibt es lange bis sehr lange Befehlszeilen. Wenn jetzt auch noch der Programmpfad in den privaten Verzeichnissen liegt, wie anfangs angedeutet, werden die Aufrufe noch hässlicher. Nur zur Abschreckung:

C:\Benutzer\<username>\AppData\Local\Programs\Thonny\Lib\site-packages\esptool.py --chip esp32 --port COM4 write_flash -z 0x1000 esp32-idf3-20200902-v1.13.bin

Bei der eben durchgeführten zweiten Variante kann man ohne Eingriff in das Programmverzeichnis die Befehlszeilen kürzen, wenn man der nächsten Beschreibung folgt. Sie erfordert ein einziges Mal einen höheren Einsatz, erleichtert die Arbeit dann aber wesentlich.

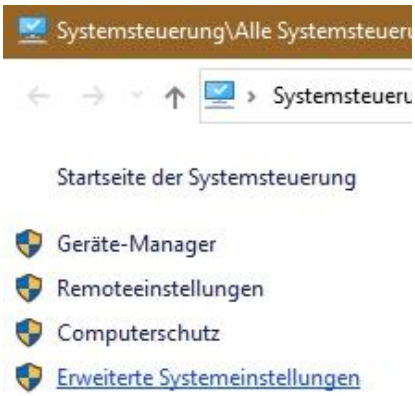
Ausgangsszenario3:
esptool.py: **C:\Program Files (x86)\Thonny\Lib\site-packages**
Firmware: **F:\micropython\ESP32\firmware\latest\esp32-idf3-20200902-v1.13.bin**
COM-Port: **COM4**

Die Pfadangabe zum Programm esptool.py kann man sich sparen, wenn man den Programmpfad in die Umgebungsvariable Path des Windowssystems aufnimmt. Das geht so.

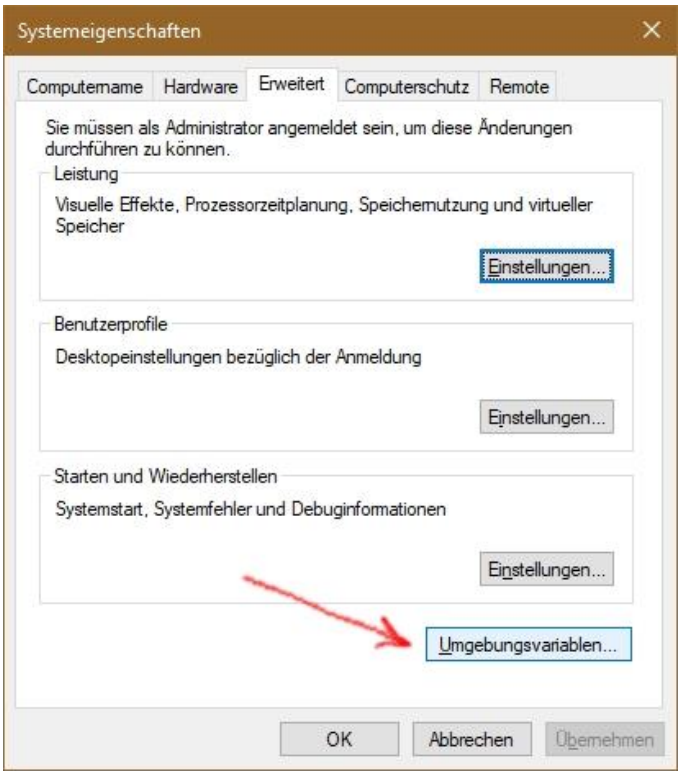
Starten Sie mit einem Rechtsklick auf das Windowssymbol in der linken unteren Bildschirmecke und klicken Sie auf **System**.

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

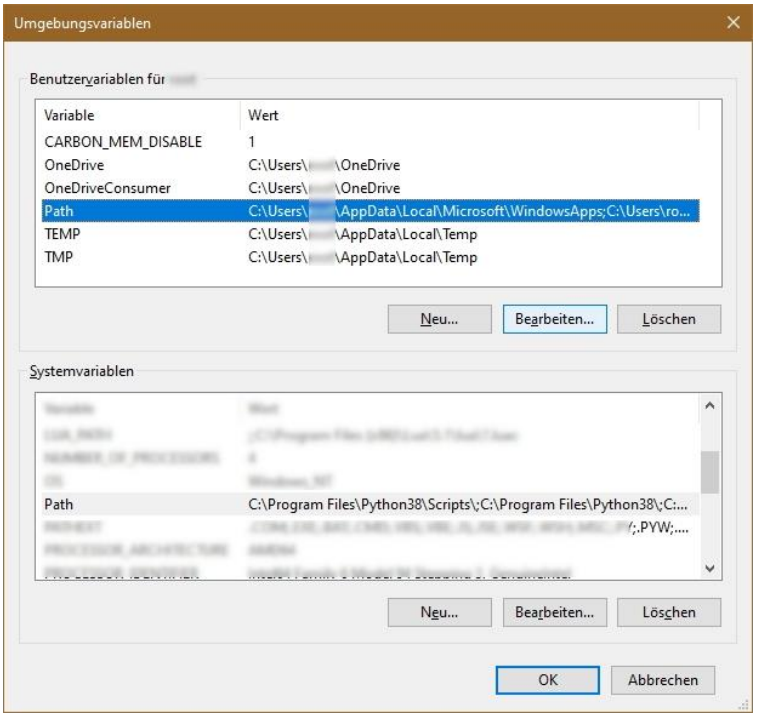
Im folgenden Fenster gehen Sie auf **Erweiterte Systemeinstellungen**.

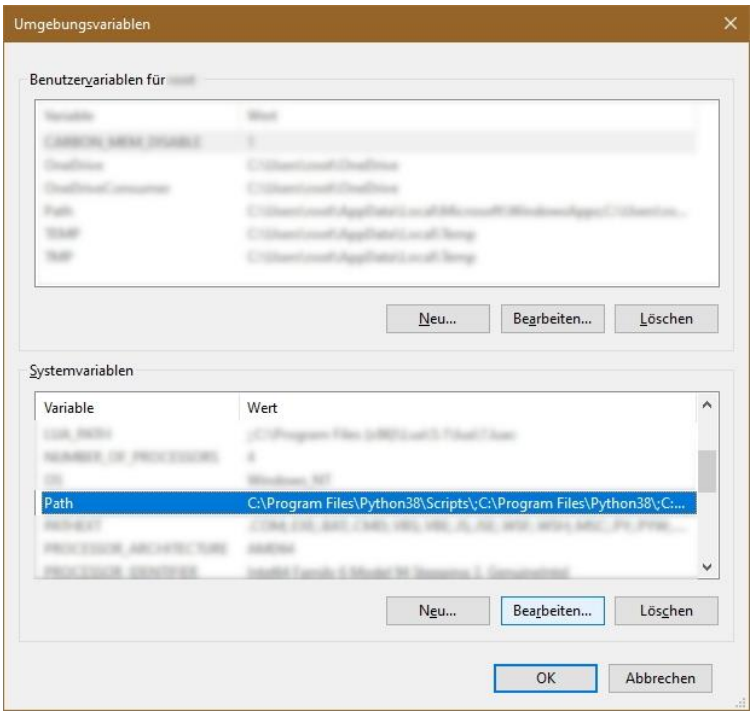


Ganz unten finden Sie das Feld **Umgebungsvariablen** -- Klick.



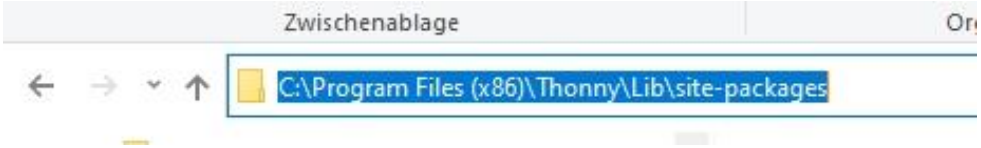
Im oberen Teil stellen Sie die Variable Path nur für sich selbst ein, im unteren Teil betrifft eine Änderung alle Benutzer systemweit. Meine Installation von Thonny erfolgte als Administrator, also wähle ich die untere Variante.



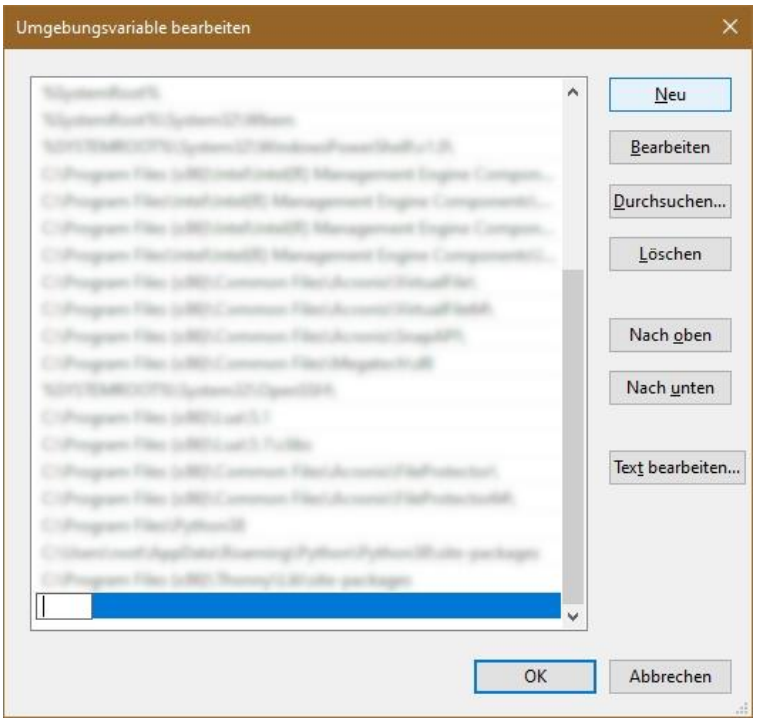


Um Einträge in die Pfadvariable aufzunehmen, klicken Sie zunächst auf **Bearbeiten.**

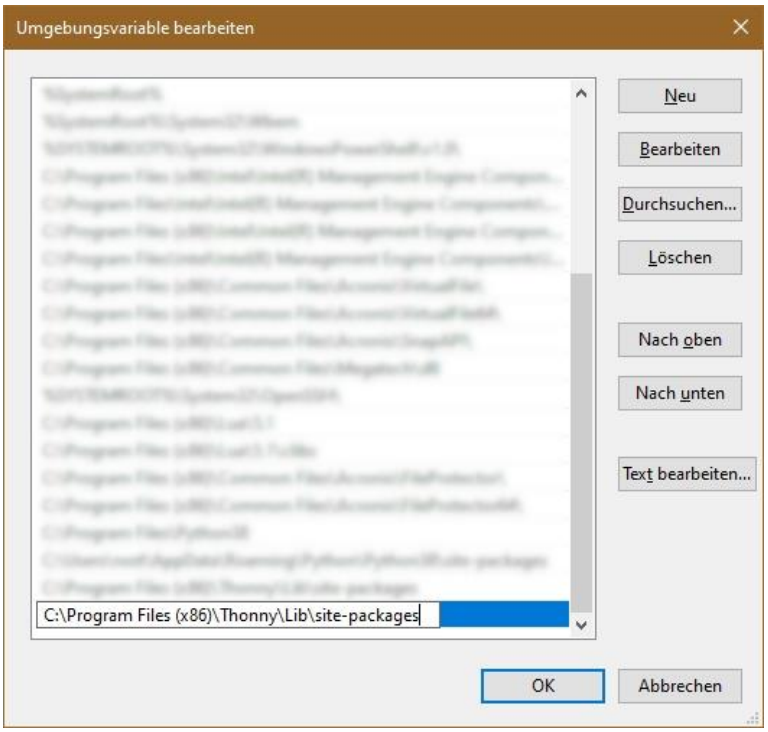
Bei längeren Pfadangaben ist es besser, diese aus dem Windowsexplorer zu übernehmen. Navigieren Sie im Windowsexplorer zum Programmverzeichnis von esptool.py und klicken Sie, dort angekommen, hinten in die Zeile mit der gesamten Pfadangabe. Mit **Strg+C** kopieren Sie die markierte Pfadangabe in die Zwischenablage. Gehen Sie zurück zum Fenster **Umgebungsvariable bearbeiten**.

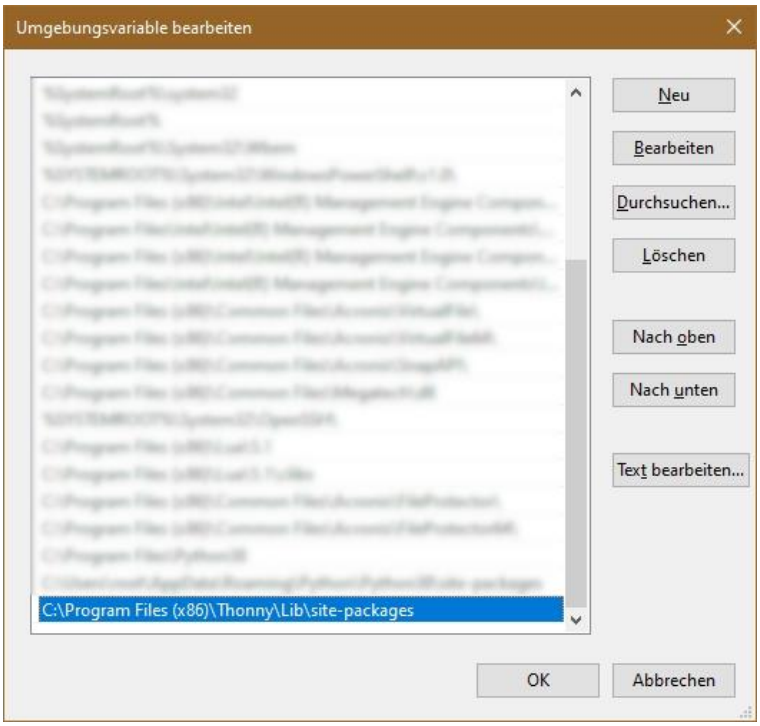


Klicken Sie auf Neu, um der Path-Variablen den kopierten Pfad hinzuzufügen.



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





Schließen Sie nun alle Fenster im Rückwärtsgang mit **OK**.

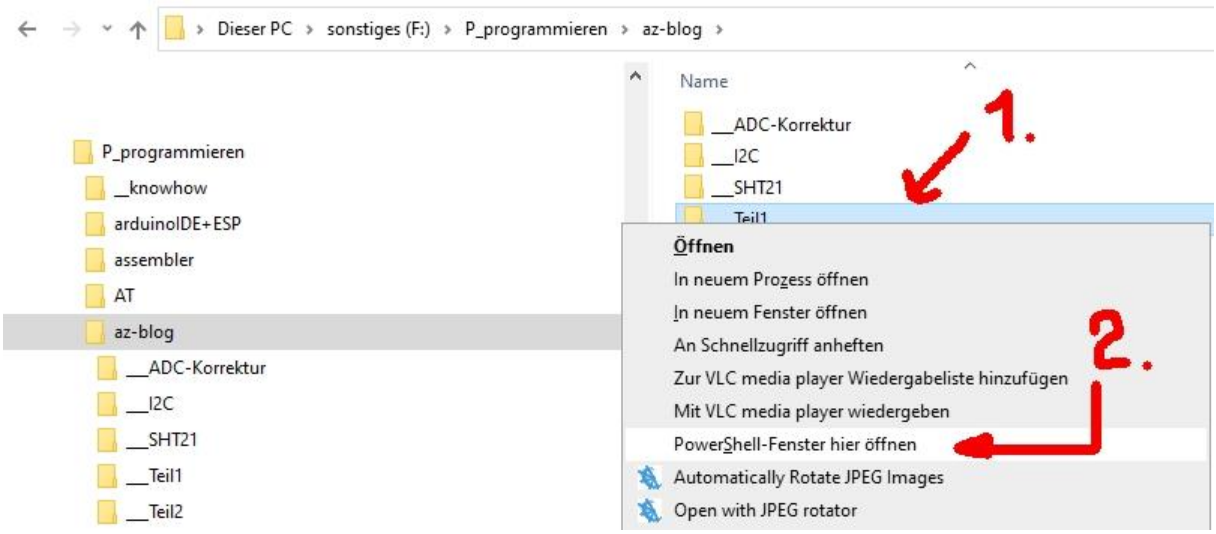
Jetzt dürfen Sie, vom Firmwareverzeichnis aus, die Aufrufe für einen ESP32 wie folgt ausführen .Beachten Sie bitte, dass für diesen Chip die Kennung explizit angegeben werden muss, während sie für den ESP8266 optional ist. Die Chips erfordern auch unterschiedliche Startadressen für die Firmware, beim ESP8266 ist es 0x0000 und beim ESP32 müssen Sie 0x1000 angeben.

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

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

Das sieht doch viel besser aus und hat überdies den Vorteil, dass der Aufruf von esptool.py nun von **jedem beliebigen Verzeichnis** aus erfolgen kann. Wenn Sie eine Node-MCU-Firmware flashen wollen, wechseln Sie eben in das entsprechende Firmware-Verzeichnis und starten von dort den Flashvorgang. Die Programmaufrufe sehen dann bis auf den Namen der Firmwaredatei und den verwendeten Chip genauso aus wie hier. **esp32** ersetzen Sie gegebenenfalls durch **esp8266**. Der Standardwert ist esp8266 und muss nicht einmal eingegeben werden, ich habe das oben schon erwähnt. Der Dateiname kann vom Verzeichnis aus durch Eingabe der ersten Zeichen des Dateinamens und die Verwendung der Tab-Taste schneller angegeben werden.

Hier noch ein Trick zum Aufrufen der Powershell im richtigen Zielverzeichnis. Navigieren Sie im Explorer in das Verzeichnis, welches das Verzeichnis mit den .bin-Dateien enthält (1). Führen Sie einen Rechtsklick auf das Zielverzeichnis aus und wählen Sie aus dem Kontextmenü **PoweShell-Fenster hier öffnen** (2). Damit landen Sie im Powershellfenster im richtigen Verzeichnis auf der Kommandozeile und können sofort mit den Eingaben beginnen.



Es folgen nun noch einige **Befehle für den ESP8266**, die häufig gebraucht werden.

esptool.py flash_id
Bringt eine ganze Reihe an Daten zum Chip unter anderem MAC-Adresse und Flashgröße. Der COM-Port wird automatisch gesucht und verbunden.

esptool.py erase_flash
Löscht den Flashspeicher bervor er neu beschrieben wird. Der COM-Port wird automatisch gesucht und verbunden.

esptool.py -p COM5 -b 460800 write_flash -fm dio -fs 4MB 0x00000 esp8266-1m-20200902-v1.13.bin
Beschreibt einen ESP8266 an Port COM5 mit Baudrate 460800 im Modus dio mit dem Inhalt der Datei esp8266-1m-20200902-v1.13.bin ab Adresse 0x0000. Die Speichergröße ist bei einem D1 mini mit den tatsächlich vorhandenen 4MB anzugeben, auch wenn von MicroPython nur 1MB genutzt wird.

esptool.py --port COM3 write_flash --flash_mode dio -flash_size 1MB 0x0000 esp8266-1m-20200902-v1.13.bin
Flashen eines ESP8266-01 mit 1MB Flash. Danach bleiben 374KB für Programme übrig.

Es gab beim Verfassen dieses Beitrags durchaus auch überraschende Momente. Da war einmal die Feststellung, dass ohne die Angabe der Flashgröße und des Modus ein ESP8266 D1 mini mit esptool.py zwar geflasht werden konnte, danach aber sofort ein wildes Blinken der eingebauten LED begann und der Chip nicht ansprechbar war. Das gleiche Verhalten zeigte ein ESP8266-01, den ich mit µPyCraft geflasht hatte. mit esptool.py und den richtigen Parametern funktionierten am Ende beide Boards. Wesentlich ist für **-fs** oder **–flash_size** die Angabe des tatsächlich vorhandenen Flashspeichers, was bei den ESP8266-ern nicht unbedingt mit der Größe im Filenamen der bin-Datei übereinstimmen muss. Der Befehl
esptool.py --port COM3 write_flash --flash_mode dio -flash_size 1MB 0x0000 esp8266-1m-20200902-v1.13.bin
passt also für den ESP8266-01 und für den ESP8266 D1 mini muss es heißen
esptool.py --port COM3 write_flash --flash_mode dio -flash_size 4MB 0x0000 esp8266-1m-20200902-v1.13.bin.

Lösungen der Hausaufgaben aus Teil 4

1. **Nehmen wir an, Sie führen folgende Eingaben über die Kommandozeile aus:**
   >>> from beep import BEEP
   >>> BEEP.dauer = 5000

   >>> BEEP.dauer
   5000

   >>> b=BEEP(2,13)
   ???
   ???
   >>> b.beep()
   ???
   Wie reagiert der ESP auf die beiden letzten Befehle? Können Sie das Verhalten begründen?

Die Ausgabe des Konstruktors ist:
Konstruktor von BEEP
LED:2, Buzz:13, dauer=15

Der Ton erklingt 15ms lang.

Schauen wir den Beginn der Klassendefinition von BEEP an

```
DAUER   =     const(15)

class BEEP:
  dauer   =     DAUER
```

dauer erhält einen Verweis auf den Speicherplatz der Konstante 15

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

Beim **Import** läuft der Interpreter über diese Zeile des Konstruktors und baut den Verweis auf die 15 fest in das interpretierte (=übersetzte) Programm ein. Wird im weiteren Verlauf dauer auf einen neuen Wert gesetzt, dann ändert das aber nichts mehr an der Zuordnung in der Parameterliste des Konstruktors. Dazu müsste das Programm neu interpretiert werden. Der Interpreter übersetzt das Programm nur einmal beim Import. Instanzen können dann beliebig viele durch Ausführen der bereits übersetzten Konstruktormethode erzeugt werden, mit der festen Referenz auf den Wert 15.

Weil duration ein optionaler Parameter ist, kann die Referenz auf die Konstante 15 dadurch überschrieben werden, dass für duration bei der **Instanzierung** eines Objekts ein Argument angegeben wird, als Zahl oder Namen-Wert-Paar. Wird kein Argument angegeben, dann erfolgt der Zugriff über die vorgegebene Referenz auf die Konstante 15.

Die Mitteilung des Konstruktors und die Tondauer sind somit stimmig.

2. **Lassen Sie die Ausgaben in der Datei touchtest.py auf dem OLED-Display ausgeben, statt am Terminal. Wann macht das Sinn, wann nicht?**

Das Beispielprogramm können Sie downloaden. Terminal und Display haben Vor- und Nachteile. Das Terminal kann auch längere, umfangreiche Texte problemlos darstellen aber man braucht halt einen Rechner dazu. Das Display arbeitet "stand alone", kann aber zum einen nur kurze Texte wiedergeben und das auch nur in einer beschränkten Anzahl von Zeilen. Während man im Terminal zurückrollen kann, werden am Display alte Ausgaben von neuen überschrieben.

3. **Messen Sie die Geschwindigkeit, die mit der Ausgabe am Terminal und am OLED-Display erreichbar ist.**

Das Testprogramm entspricht im Wesentlichen dem Geschwindigkeitstest für den ADC. Sie können es auch herunterladen.

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

n = 0
now=time()
dauer = 5  # Programmlaufzeit ca. 3 Sekunden
then=now+dauer
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/dauer,"Bloecke pro Sekunde")
```

Konstruktor von OLED
SDA:21, SCL:22, Size:128x32
48.8 Bloecke pro Sekunde

Eine Zeile von 16 Zeichen kann ca. 50-mal pro Sekunde ausgegeben werden. Das ist weitgehend unabhängig von der Anzahl der Zeichen. Mit 4 Zeichen messe ich 50 Blöcke/s und mit nur einem Zeichen 52 Blöcke/s. Drei Zeilen mit 16 Zeichen bringen es dementsprechend auf 16,6 Blöcke/s.

Für die Terminalausgabe ersetzen Sie im Programm die Zeile

```
  d.writeAt("0123456789012345",0,0)
```

durch

```
  print("0123456789012345")
```

Mit 1 Zeile zu 16 Zeichen ist das Terminal mit 628,4 Blöcken/s gut 12-mal schneller, bei 4 Zeichen gar über 40-mal.

Das liegt daran, dass beim Display stets der gesamte Framebuffer des ESP zur Anzeige geschickt werden muss, auch wenn nur ein Zeichen verändert wurde. Was bremst ist, die **show**()-Methode.

Im Gegensatz dazu überträgt die serielle Leitung zum Terminal nur so viele Zeichen, wie gesendet werden müssen. Dass der ESP32 oder ESP8266 bedeutend schneller arbeitet wie die serielle Leitung äußert sich darin, dass man auf das Ergebnis länger warten muss als die 5-Sekundenvorgabe des Programms. Die Meldung kommt erst, wenn der Sendebuffer komplett übertragen worden ist.

4. **Macht es für die Geschwindigkeit einen Unterschied, wie viele Zeichen am Display und am Terminal pro Durchgang ausgegeben werden?**

Beim Display nein, beim Terminal ja.

5. **Erzeugen Sie mit dem pillar()-Befehl und dem Zufallszahlengenerator aus der Hausaufgabenlösung ein Säulendiagramm aus 10 Balken, das die Anzeigefläche eines Displays möglichst voll nutzt. Denken Sie dran, dass es Displays mit verschiedener Breite gibt.**

Die Breitenangabe sollt variabel gehalten sein und nicht, zum Beispiel, als 64 vorgegeben werden.Die Information stammt hier aus dem Instanzattribut width des Objekts d.

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

d.clearAll()

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

d.xAxis()
d.yAxis()
for i in range(10):
  hoehe = os.urandom(1)[0] // hTeiler
  d.pillar(i*breite,breite-1,hoehe)
```

6. **Bereiten Sie zwei verschiedene Ausgaben für das OLED-Display vor, zum Beispiel einen Textschirm und ein Säulendiagramm. Lassen Sie beides im Wechsel von 3 Sekunden anzeigen, während (also gleichzeitig!) im Terminal die Ausgabe der Zahlen von 1 bis 30 nach folgendem Schema im Dauerlauf erfolgt.**
   1
   12

123
1234
12345
…

Das Beispiel soll demonstrieren, dass während der normalen Ausführung eines Hauptprogramms interruptgesteuerte Aktionen (relativ-) intime möglich sind. Letztlich hängt das genaue Timing davon ab, wie die Firmware die Priorität der Interrupts einstuft und behandelt.

Das hier verlinkte Programm gibt eine Beispiellösung der Aufgabe.

Die Lösung erfolgt über einen Timer, der alle 3 Sekunden eine der beiden Funktionen texte() oder pillars() aufruft. Der Wechsel erfolgt über das Hochzählen und das bitweise Undieren der Variable n.

Die Zahlenausgabe geschieht über zwei For-Schleifen, wobei die äußere bis 31 und die innere daher bis 30 läuft. Damit die serielle Schnittstelle mitkommt, ist eine kurze Schlafpause eingebaut.

7. **Bauen Sie den Reaktionstester aus der Hausaufgabenlösung so um, dass bei jedem Durchgang eine andere LED scharf ist. Natürlich müssen sie dem Spieler über das OLED sagen, welche Farbe das jeweils ist.**

Ins Programm hausi4b.py wurden einige Zeilen eingefügt. Das geänderte Programm ist auch zum Download verfügbar.
Gleich zu Beginn wird die Klasse OLED importiert, das Displayobjekt d erzeugt und das Display gelöscht.
Passend zur Liste der LED-Objekte wird eine Liste der Farben definiert. Die feste Zuweisung einer Nummer an testLed wird entfernt.

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

Die Ziel-LED wird wie gehabt gewürfelt und dann die Farbe am Display ausgegeben.

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

Schließlich würfeln wir eine Test-LED und weisen das entsprechende Objekt dem Laufattribut ledx zu. Der Rest der Hausaufgabenlösung muss nicht geändert werden.

```
  testLed = os.urandom(1)[0] & 0x03
  testLed = (testLed if testLed != 3 else 2)
  #TestLED zuweisen, weiter wie gehabt
  ledx =ledList[testLed]
```

Programm starten und los!

8. **Können Sie es einrichten, dass in einem neuen Modul touch die Variante ESP32 und die Variante ESP8266 vorgehalten werden und je nach Controllertyp beim Import automatisch die richtige zum Einsatz kommt?**

Die Datei touch.py enthält die Lösung, die ganz einfach ist. Beide originalen Teile wurden in eine if-elif-else-Struktur verpackt, jeweils um eine Stufe weiter eingerückt.

Der Import von Pin und TouchPad wurde ebenfalls in die if-Struktur verschoben. Der Grenzwert für die ESP8266-Variante wurde auf 0,5 gelegt, damit Vergleiche wie bei der 32-er Variante möglich sind.