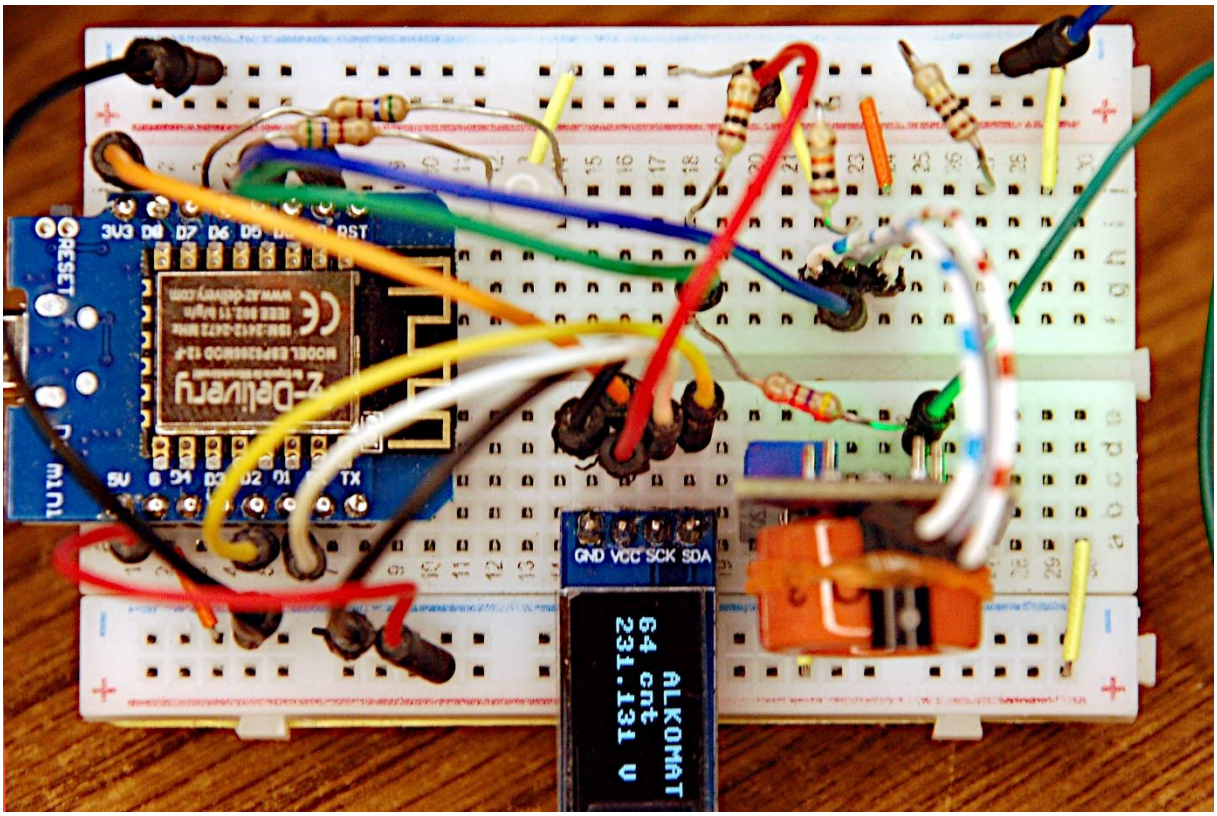

MQ-3 + ESP8266 = Alkomat

Diese Blogfolge gibt es auch als PDF-Dokument.

Die Faschingszeit naht und da wird – die Pandemie ist vorüber, sagt man – sicher hier und da eine Fete angesagt sein. Und wo gefeiert wird, wird auch getrunken, sicher nicht nur Brause und Saft, sondern auch ganz andere Sachen. Aber bevor man sich hinters Steuer setzt, sollte man vorsichtshalber den Blutalkoholspiegel im Visier behalten, andernfalls hat die Gaudi schnell ein Loch, wenn der Bußgeldbescheid ins Haus flattert, wegen Trunkenheit im Verkehr. Wie Sie messtechnisch Ihren Alkoholgenuss überprüfen können, das zeigt dieser Beitrag aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Ein Alkometer mit dem ESP8266 und einem MQ-3-Sensor

In dieser Folge werden wir uns zunächst einmal mit den technischen Hintergründen beschäftigen, bevor es in Folge 2 dann ums Eingemachte geht. Wie eiche ich den ADC eines ESP8266 oder eines ESP32? Wie stelle ich fest, ob der MQ-3 seine Arbeitstemperatur erreicht hat? Diese Fragen beantwortet der vorliegende Post. Im Folgebeitrag wird es dann um die Programmierung des Alkotesters speziell gehen.

Zugegeben, eine genaue Messung, die auch einem Vergleich mit dem amtlichen Alkomat der Polizei standhält, lässt sich mit den einfachen Mitteln, die uns zur Verfügung stehen, nicht durchführen. Das beginnt schon einmal bei der Eichung des Alkohol- und Gassensors MQ-3. Nach mehrmaligem Durchforsten des Datenblatts und des Internets, habe ich kein brauchbares Verfahren dafür gefunden. Dennoch ist mit der einfachen Schaltung einiges möglich.

Aber beginnen wir erst einmal mit den Grundlagen zu den Messungen. Das MQ-3-BOB (Break Out Board) liefert ein digitales und ein analoges Ausgangssignal. Letzteres besteht aus einer Gleichspannung, deren Wert umso mehr zunimmt, je höher die Alkoholkonzentration in der Atemluft ist. Ein Diagramm (Figure 2) im Datenblatt zum MQ-3 verrät nun leider keinen absoluten Zusammenhang zwischen Konzentration und Spannungswert, sondern es stellt nur das **Widerstandsverhältnis** zwischen dem Sensorwiderstand Rs zum Messwert Ro bei einer Konzentration von 0,4mg Alkohol pro Liter Luft für verschiedene Konzentrationen dar. Außerdem hat der Lastwiderstand RL auf dem Modul den Wert 1 kΩ, statt einen Wert zwischen 100kΩ und 500kΩ, wie es das Datenblatt vorgibt. Da ist also "Jugend forscht" gefragt.

Den im Beitrag verwendeten Teil der Schaltung des Moduls habe ich in Abbildung 1 dargestellt. Den Digitalteil mit dem Komparator LM393 habe ich weggelassen.

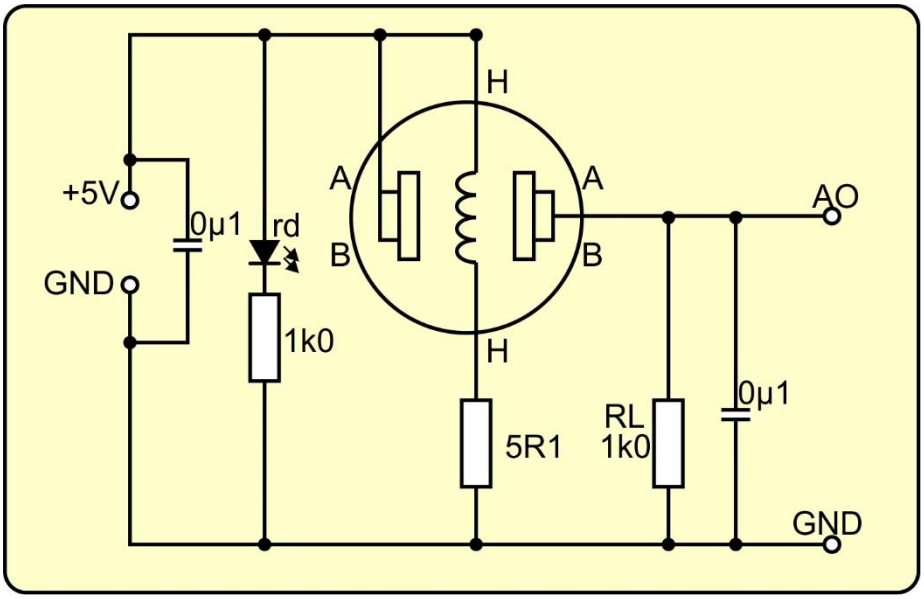

Abbildung 1: Sensorschaltung

Der Sensorwiderstand Rs zwischen den Punkten AB – AB kann berechnet werden, wenn man die Betriebsspannung von 5V und die Spannung Uao am analogen Ausgang AO, sowie den Wert des Lastwiderstands RL kennt. Rs und RL bilden einen Spannungsteiler. An Rs liegt die Spannung 5V – Uao, an RL liegt die Ausgangsspannung Uao. Die Teilspannungen verhalten sich so wie die Teilwiderstände.

$$\frac{5V - Uao}{Uao} = \frac{Rs}{RL}$$

$$\frac{(5V - Uao) \cdot RL}{Uao} = Rs$$

Abbildung 2: Berechnung des Sensorwiderstands Rs

Das wäre also nicht das Problem. Aber die Bestimmung des Widerstandswerts Rs bei einer Konzentration von 0,4mg Alkohol in 1 Liter Luft für den Wert Ro stellt ein Problem dar. Allein schon 0,4mg Alkohol, das sind 0,5mm³, mit haushaltstechnischen Hilfsmitteln abzumessen ist unmöglich.

Es bleibt daher nur der Selbstversuch. Nach einem Blick in den Bußgeldkatalog könnte das so aussehen. Verdoppelt man den Wert der Atemluftkonzentration, erhält man den Blutalkoholspiegel. Andersherum kriege ich die Atemluftkonzentration durch Halbieren des Blutwerts. Letzteren kann man berechnen. Ein Beispiel:

Ein Mann mit 80kg Körpermasse besitzt 80kg • 0,7 = 56kg Körperflüssigkeit. Der Reduzierfaktor bei Frauen ist 0,6. Er trinkt 1L Bier = eine Maß mit einem Alkoholgehalt von 5,5 % Vol. In der Maß sind also 1000cm³ • 5,5 / 100 = 55cm³ reiner Alkohol. Reiner Alkohol hat eine Dichte von 0,79g/cm³, damit haben die 55cm³ eine Masse von 55cm³ • 0,79g/cm³ = 43,45g. Bezogen auf 56kg Körperflüssigkeit sind das 43,45g / 56kg = 0,78 Promille. Und das würde einer Atemluftkonzentration von knapp 0,4 Promille oder 0,4mg / Liter ergeben. Allerdings müsste die Maß möglichst in einem Zug getrunken werden, denn die Leber baut pro Stunde 0,15 Promille ab. Ob Sie jedoch nach dem Genuss von 1Liter Bier noch in der Lage sind, Ihren Aufbau zu calibrieren, können nur Sie selbst entscheiden, Autofahren sollten Sie dann besser nicht mehr.

Hardware

Als Controller habe ich einen ESP8266 gewählt, weil der eine einigermaßen lineare ADC-Kennlinie hat. Grundsätzlich wäre auch ein ESP32 brauchbar, aber der verursacht einen höheren Aufwand für das Geraderichten der Kennlinie. Ich komme später darauf zurück.

Die ersten drei ESP8266-Modelle in der Teileliste haben den Vorteil, dass am Anschluss VIN / 5V die Spannung des USB-Anschlusses verfügbar ist. Das spart während der Entwicklung ein zusätzliches Netzteil. Für den späteren Betrieb als Stand-Alone-Gerät ist ein Netzteil empfehlenswert, denn Batterien oder ein Akku ist schnell leergelutscht, immerhin zieht die Heizung des MQ-3 150 mA und der ESP8266 noch einmal 20 mA.

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder D1 Mini V3 NodeMCU mit ESP8266-12F oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Alkohol Gas Sensor DC 5V MQ-3 mit Signalausgang
1	Mehrgang rotary Potentiometer mit Schutzwiderstand 3590S 10K Ohm
1	KY-011 Bi-Color LED Modul 5mm
1	DS18B20 digitaler Temperatursensor TO92
1	Mini Breadboard 400 Pin mit 4 Stromschienen
	Diverse Jumperkabel
1	Widerstand 4k7
2	Widerstand 10k
2	Widerstand 390 Ohm
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266:

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

Verwendete Firmware für den ESP32:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
calibrate.py Programm zum Calibrieren des ADC
aufheizen.py Programm zum Ermitteln der Vorheizdauer des MQ-3

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit

der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Calibrieren des ADC

Bevor es um den Einsatz des MQ-3-Sensors geht, müssen wir uns mit der ADC-Kennlinie des ESP8266 beschäftigen, denn die beeinflusst die Genauigkeit der Spannungsmessung am Gassensor. Der ADC (Analog Digital Converter) wandelt Spannungen im Bereich von 0V bis 3,3V in ganzzahlige Werte von 0 bis 1023 um. Es geht erst einmal schlicht darum, festzustellen wie sich der Zusammenhang zwischen den ADC-Werten und den Ablesungen an einem DVM (Digital-Voltmeter) darstellt. Das Ziel ist dann, eine Methode zu entwickeln, die dazu führt, dass beide Werte (nahezu) gleich sind. Die Schaltung für die Untersuchung zeigt Abbildung 3.

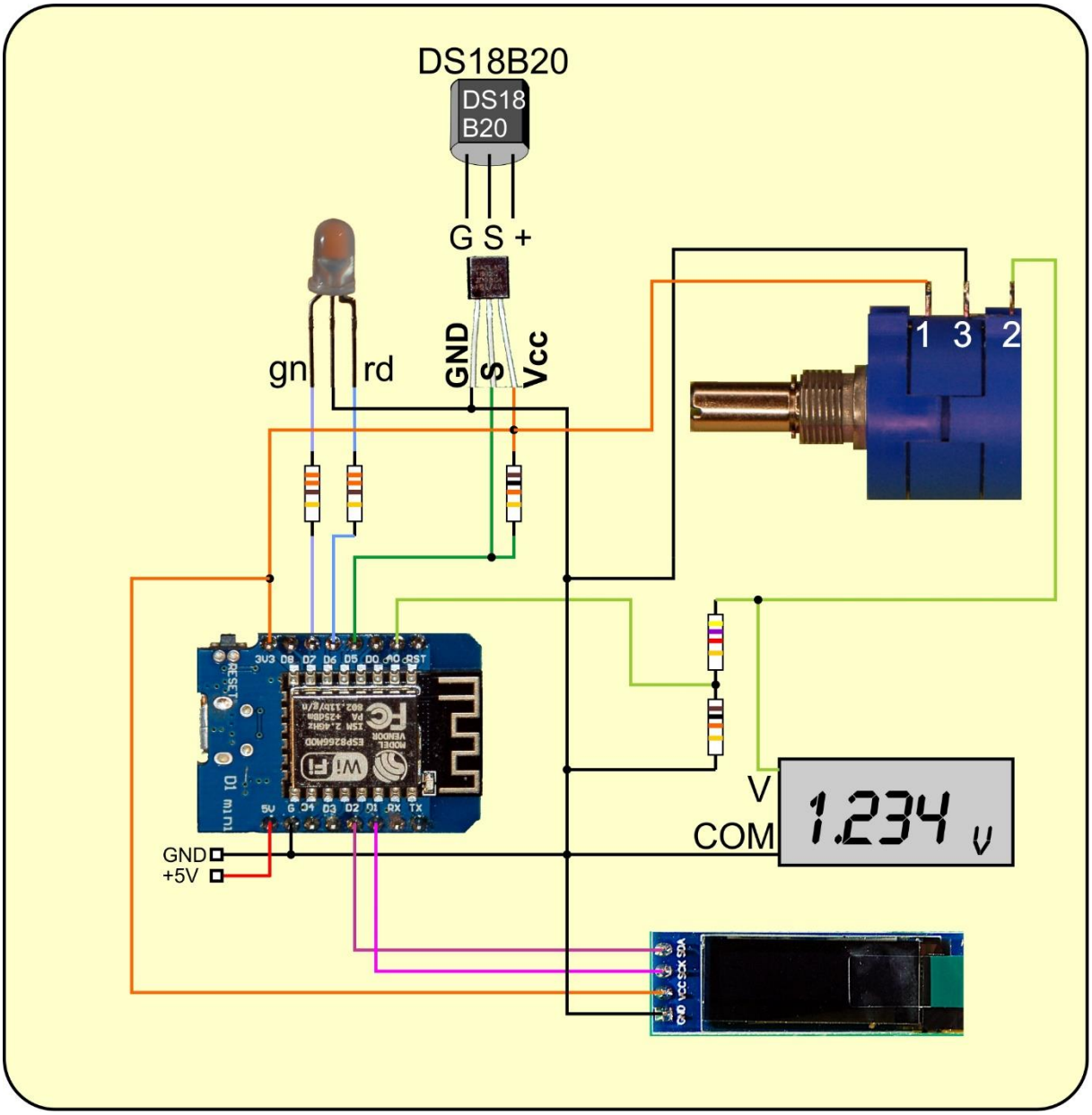

Abbildung 3: Alkomat - Calibrieren des ADC

Mit dem 10-Gang-Poti gebe ich eine genau einstellbare Spannung vor (2), die über den Spannungsteiler aus den Widerständen 4k7 und 10k an den Analogeingang A0 des ESP8266 geführt wird. Der Anschluss 1 liegt an 3,3V, Anschluss 3 an GND. Der V-Eingang des DVM wird ebenfalls mit dem Schleifkontakt (2) des Potis verbunden,

der COM-Anschluss liegt an GND. Die Pin-Belegung des Potis ist gewöhnungsbedürftig, denn der Abgriff (2) ist nicht der mittlere Anschluss!

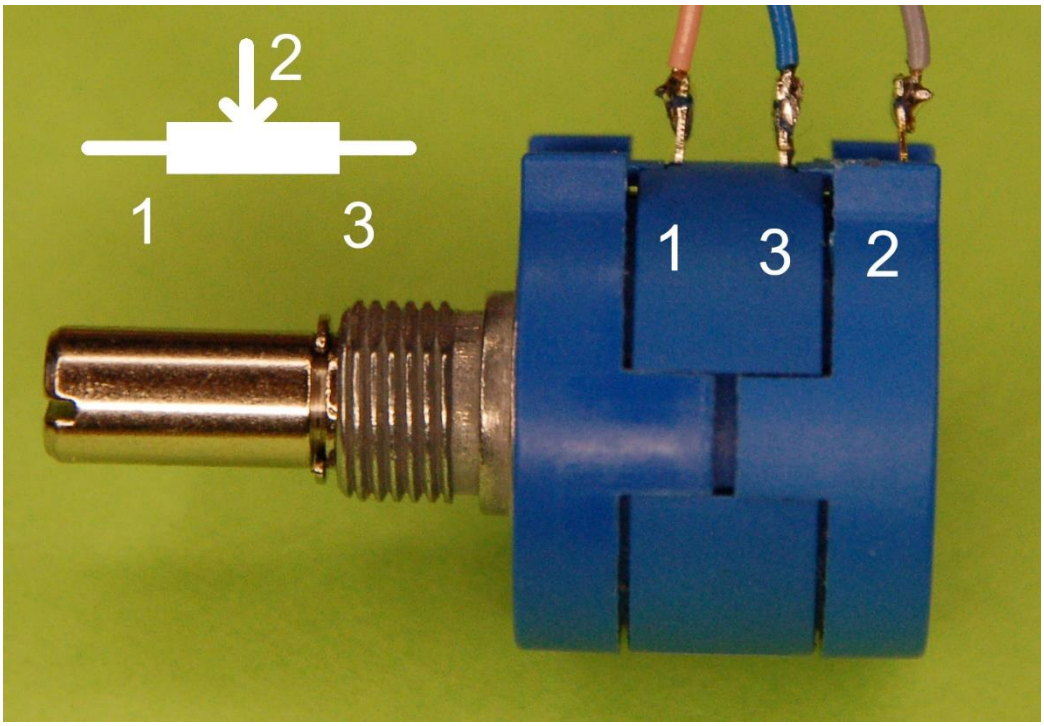

Abbildung 4: Schaltung des Zehngang-Potentiometers

Jetzt stelle ich eine Spannung ein, notiere den Wert vom DVM, und schreibe den ADC-Wert (counts) und die daraus berechnete Spannung dazu. Den Vorgang wiederhole ich in kleinen Schritten, bis ich bei 3,2V angekommen bin. Dazu benutze ich das Programm **calibrate**.py. Das Ergebnis der Messungen stelle ich nach der Besprechung des Programms vor.

```
# calibrate.py

# Nach dem Flashen der Firmware auf dem ESP8266:
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!

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

from machine import Pin, ADC, SoftI2C
from time import sleep,ticks_ms, sleep_ms
from oled import OLED
import sys
from onewire import OneWire
from ds18x20 import DS18X20
```

Die Module **oled**.py und **ssd1306**.py müssen auf den ESP8266 hochgeladen worden sein, bevor man sie importieren kann. Alle anderen sind Bestandteil des MicroPython-Kernels.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
    adc=ADC(0)
    maxCnt=1023
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
    adc=ADC(Pin(36))
    adc.atten(ADC.ATTN_11DB)
    adc.width(ADC.WIDTH_12BIT)
    maxCnt=4095
else:
    raise RuntimeError("Unknown Port")
```

Das if-Konstrukt prüft auf die unterstützten Ports, ESP32 und ESP8266. Andere Boards führen zu einer Runtime-Exception und damit zum Programmabbruch. Controllerspezifisch wird der I2C-Bus initialisiert, ebenso der ADC nebst Einstellungen.

```
ds_pin = Pin(14)  # D5 @ ESP8266
ds = DS18X20(OneWire(ds_pin))
chip = ds.scan()[0]
print(chip)
```

Ich erzeuge ein One-Wire-Objekt an Pin 14 (D5) und übergebe es direkt an den Konstruktor der Klasse DS18X20. **ds.scan**() sucht auf dem Bus nach einem DS18B20-Chip und gibt dessen ROM-Kennung in Form einer Liste zurück.

```
R1=4600 # Ohm
R2=10000 # Ohm
U0=3.2  # Volt am Sensor
RL=1000 # Ohm Lastwiderstand
scale=(R1+R2)/R2
```

Die Spannung des MQ-3-Sensors wird später ebenso, wie jetzt der Abgriff (2) des Potis an den Spannungsteiler gelegt. Die Werte sollten mittels DMM (Digital-Multimeter) ausgemessen und für R1 und R2 eingetragen werden. Der Skalierungsfaktor **scale** erlaubt die Hochrechnung der vom ADC gemessenen counts auf die Spannung am Eingang des Spannungsteilers.

```
d=OLED(i2c,heightw=32)
d.clearAll()
d.writeAt("ALKOMAT 1.0",3,0)
```

Ein OLED-Objekt wird erzeugt. Ich übergebe das I2C-Objekt und die Höhe des Displays in Pixeln. Die Defaultbreite ist 128 und muss nicht angegeben werden. Display löschen und die Überschrift ausgeben.

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

Die Closure **TimeOut**() produziert beim Aufruf eine Methode, die einen nicht blockierenden Softwaretimer darstellt. Während dieser abläuft, können vom Programm andere Dinge erledigt werden.

```
def waitHeating(val=90000):
    hot=TimeOut(val)
    d.writeAt("BITTE WARTEN",0,1)
    rest=int(val/1000)
    while not hot():
        d.writeAt("NOCH {} SEKUNDEN ".format(rest),0,2)
        sleep(1)
        rest-=1
    d.clearFT(0,1)
```

Die Funktion **waitHeating**() nimmt einen Wert in Millisekunden. Hiermit stelle ich mittels **TimeOut**() einen Timer her, der während des Ablaufs die Restdauer in Sekunden am Display ausgibt. Der Timer wird über die Funktion **hot**() abgefragt, welche **TimeOut**() zurückgegeben hat. **hot**() ist nichts anderes als ein Alias für die Funktion **compare**(), die innerhalb **TimeOut**() definiert wird.

```
def getVal(n):
    sum=0
    for i in range(n):
        sum = sum + adc.read()
    avg = sum/n
    volt=U0/maxCnt*avg*scale
#    volt=(volt*1000-63.5567)/0.99
    return (avg, volt)
```

An die Funktion **getVal**() übergebe ich mit **n** die Anzahl der zu machenden Einzelmessungen des ADC. Aus der Summe der Messwerte wird der Mittelwert als wahrscheinlichstes Ergebnis ermittelt. Diese Vorgehensweise eliminiert das Rauschen auf der Leitung weitgehend. Es folgt die Hochrechnung auf den Eingangspegel des Spannungsteilers. Der Quotient aus der maximalen Spannung **U0** = 3,2V und dem maximalen Wandlerwert **maxCnt**=1023 liefert das kleinste Spannungsquant, das

einem count des ADC entspricht. Multipliziert mit dem Zählwert des ADC erhalte ich die Spannung am Anschluss A0. Dieser Wert muss jetzt noch auf den Eingang des Spannungsteilers durch Multiplikation mit dem Skalierungsfaktor **scale** hochgerechnet werden. Die auskommentierte vorletzte Zeile korrigiert später die Messfehler des ESP8266. Die Erklärung dazu folgt weiter unten.

```
def getTemperature(c):
    ds.convert_temp()
    sleep_ms(750)
    return ds.read_temp(c)
```

Die Temperatur des MQ-3-Gehäuses ist ein Maß für die Brauchbarkeit des Messwerts am Gassensor. Sie muss mindestens 25°C betragen. Der Funktion **getTemperature**() übergebe ich den ROM-Code des DS18B20. Eine Messung wird getriggert. Nach 750 Millisekunden liegt der Messwert vor und kann abgerufen und zurückgegeben werden.

```
def getRS(u):
    return ((U0*1000 - u) * RL ) / u
```

Aus der Ausgangsspannung **u** des Sensors kann die Funktion **getRS**(), gemäß Abbildung 2, den Sensorwiderstand **Rs** berechnen, dessen Wert sie zurückgibt. U0 wurde eingangs in Volt angegeben, u wird aber in Millivolt berechnet. Daher muss U0 mit 1000 multipliziert werden um mV zu erhalten.

Bis hier ging es um Programmteile, die auch für andere Programme rund um den MQ-3 gebraucht werden. Das folgende sehr kurze Hauptprogramm liefert im Display den ADC-Rohwert in Counts und den hochgerechneten Wert am Eingang des Spannungsteilers. **getVal**() liefert beide Werte als Tupel zurück, das in die Variablen **cnt** und **volt** entpackt wird.

```
d.clearFT(0,1)

while 1:
    cnt,volt=getVal(50)
    d.writeAt("{} cnt           ".format(int(cnt)),0,1)
    d.writeAt("{0:2.3f} mV       ".format(volt),0,2)
    sleep(1)
```

Hier nun das Ergebnis der Messreihe.

DVM mV	ESP8266 cnt	ESP8266 mV
102	36	169
149	46	210
200	57	260
249	67	310
309	80	365
406	101	463
499	122	557
609	146	671
750	177	809
906	210	962
1114	255	1169
1317	300	1374
1552	350	1601
1998	449	2051
2472	551	2518
3019	667	3047
3189	706	3227

Alle Messwerte des ESP8266 sind eindeutig zu groß. Den Zusammenhang liefert ein Diagramm, das ich in Libre Office Calc aus der Spalte **DVM mV** und **ESP8266 mV** erstellt habe.

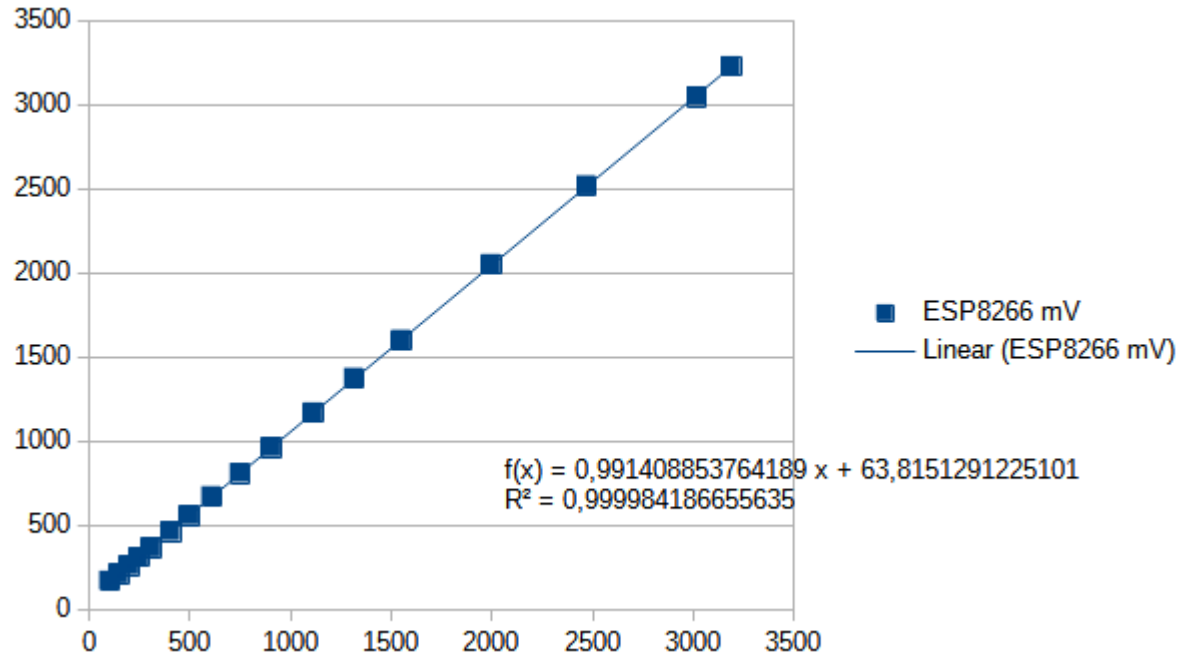

Abbildung 5: DVM-ESP8266-Diagramm

Der Relationskoeffizient R^2 bestätigt mit einem Wert von quasi 1, dass die Messpunkte sehr genau auf der Trendlinie (Gerade) liegen. Es besteht also ein linearer Zusammenhang zwischen den DVM- und den ADC-Werten. Allerdings geht

die Gerade nicht durch den Ursprung des Koordinatensystems. Das sagt auch der Funktionsterm f(x) aus.

Um aus dem hochgerechneten ADC-Spannungswert = Funktionswert f(x), den DVM-Wert x zu erhalten muss ich die Gleichung nach x auflösen.

$$f(x) = 0{,}99140 \cdot x + 63{,}81513$$
$$f(x) - 63{,}81513 = 0{,}99140 \cdot x$$
$$\frac{(f(x) - 63{,}81513)}{0{,}99140} = x$$

Abbildung 6: ADC-Fehlerkorrektur

Mit dieser Formel lasse ich aus den ESP8266 mV-Werten die korrigierten Werte berechnen. Die Tabelle zeigt die Wirkung. Mit Ausnahme von zwei Werten liegen alle korrigierten Werte innerhalb einer Fehlergrenze von +/- 1%. Diese Formel findet sich auch in der Funktion **getVal**() wieder. Wenn ich das Kommentarzeichen jetzt entferne und das Programm erneut starte, sind DVM- und ESP8266 mV-Wert annähernd gleich. Wegen der Exemplarstreuung sollten Sie die Calibrierung in jedem Fall selbst durchführen.

DVM mV	**Korrigiert mV**	**Delta mV**	**Delta %**
102	107	5	4,4
149	148	-1	-0,7
200	198	-2	-0,8
249	249	0	0,0
309	304	-5	-1,5
406	403	-3	-0,6
499	498	-1	-0,1
609	613	4	0,7
750	753	3	0,4
906	908	2	0,2
1114	1117	3	0,2
1317	1324	7	0,5
1552	1553	1	0,1
1998	2008	10	0,5
2472	2479	7	0,3
3019	3014	-5	-0,2
3189	3195	6	0,2

MQ-3 vorheizen – wie lange dauert das?

Der MQ-3 arbeitet nur dann zuverlässig, wenn der Sensor seine Arbeitstemperatur erreicht hat. Wann das eintritt findet das Programm **aufeizen**.py heraus.

Der einzige Unterschied zu **calibrate**.py besteht im Hauptprogramm.

```
d.clearAll()
zeit = 0
delta=2
while 1:
    cnt,volt=getVal(50)
    temp=getTemperature(chip)
    d.writeAt("{} sec           ".format(zeit),0,0)
    d.writeAt("{0:2.2f} *C       ".format(temp),0,1)
    d.writeAt("{0:2.3f} mV       ".format(volt),0,2)
    print(zeit,temp,volt)
    zeit+=delta
    sleep(delta-1)
```

Ich hole ADC- und Spannungswert sowie die Temperatur am MQ-3-Gehäuse. Den Temperaturwert liefert der DS18B20, den ich mit einem Spanngummi auf dem Gehäuse befestigt habe.

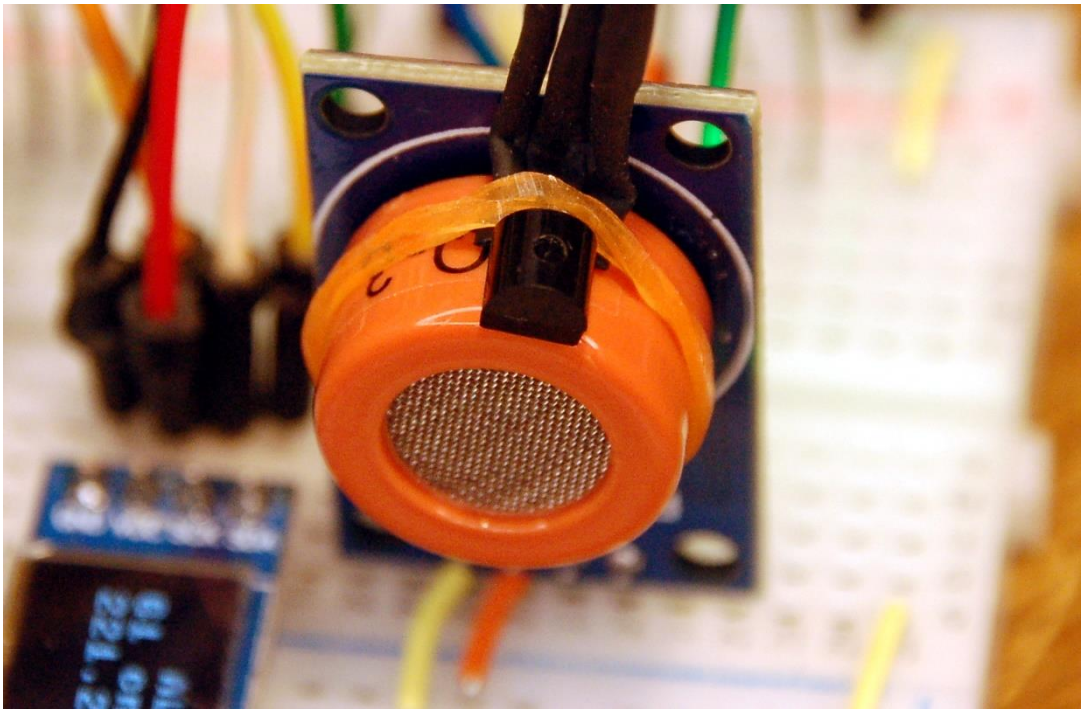

Abbildung 7: DS18B20 am MQ-3-Gehäuse

Die Datenleitung wird mit einem Pullup-Widerstand von 10kΩ auf die Betriebsspannung von 3,3V gezogen. Statt dem Potentiometer liegt jetzt der Ausgang AO des MQ-3 am Spannungsteiler. Abbildung 8 zeigt das Schaltbild.

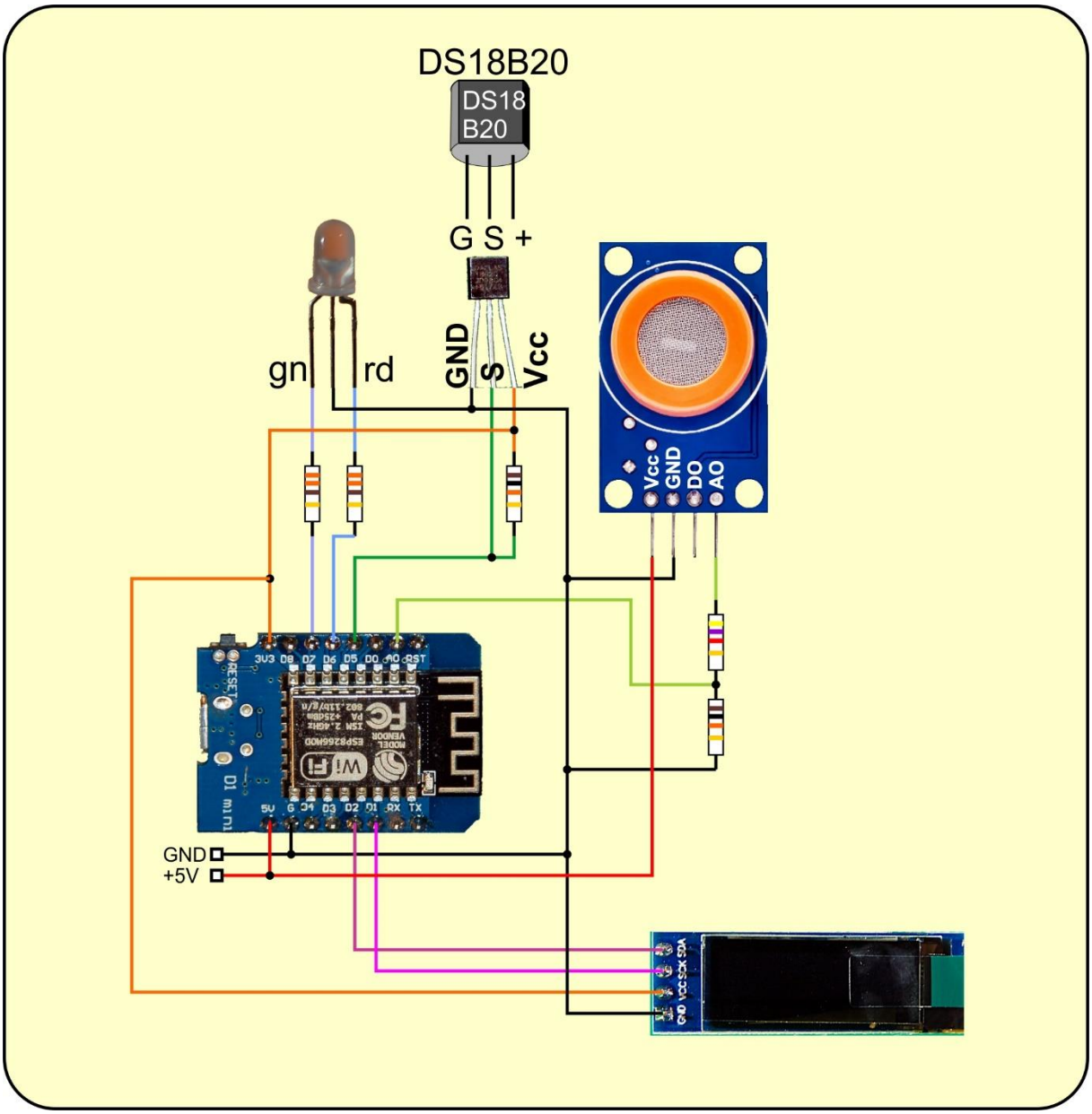

Abbildung 8: Alkomat - Schaltung

Da der Sensor an +5V liegt, könnten theoretisch Spannungen von mehr als 3,3V am Ausgang AO auftreten. Aus diesem Grund habe ich den Spannungsteiler eingesetzt. Er liefert am Mittelabgriff zwei Drittel der Eingangsspannung. Bei 5V Eingangsspannung sind das 3,33V, und wir haben die Schaltung in trockenen Tüchern.

Das Programm listet nun im Terminal in gleichen zeitlichen Abständen Zeit-Temperatur- und Spannungswert auf. Wenn sich Letzterer vom vorhergehenden Wert nicht mehr wesentlich unterscheidet, ist der Zeitpunkt erreicht. Die dazu angegebene Zeit kann dann an die Funktion **waitHeating**() übergeben werden, die beim Start eines Anwenderprogramms aufgerufen wird. Bei meinen Vorversuchen hat sich eine Zeit von ca. 5 Minuten ergeben, die vor dem ersten Testpusten gewartet werden sollten. Zum Anblasen des Sensors hat sich übrigens ein dickerer Trinkhalm oder ein 10 bis 15 cm langes, dünnes Schlauchstück als günstig erwiesen.

Dieses statische Vorgehen mit einer festen Wartezeit ist für einen absoluten Kaltstart sinnvoll. Im Dauerbetrieb ist eine dynamisch arbeitende Methode sinnvoller. Die stelle ich mit dem Programm **alktest**.py in der nächsten Folge vor.

Bleiben Sie dran!