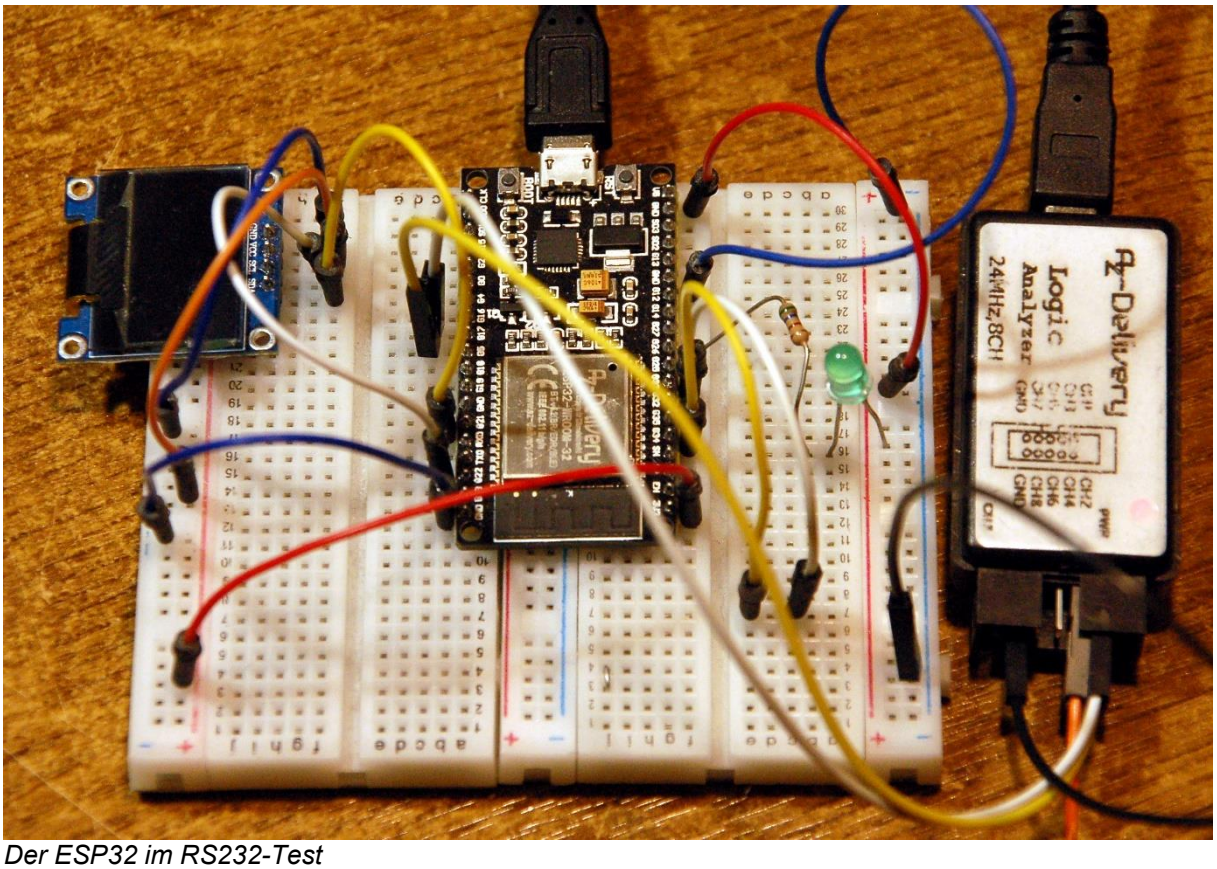

Der ESP32 im RS232-Test

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

In der letzten Folge haben wir uns mit grundlegenden Eigenschaften einer seriellen Verbindung nach dem RS232-Standard beschäftigt. Dabei wurde ein ESP8266 dazu gebracht, Messwerte über seinen UART1 an ein Terminal-Programm auf dem PC zu senden. Bemängelt hatten wir, dass der ESP8266 auf dieser Schnittstelle keine Daten empfangen kann. Beim ESP32, den wir heute untersuchen, werden wir drei deutliche Verbesserungen feststellen. Außerdem betrachten wir den Zusammenhang zwischen erzielbarer Baudrate und Leitungslänge. Willkommen bei einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 und RS232 auf dem PC

Pegelwandler

Vielleicht hat Ihr PC ja noch eine serielle Schnittstelle nach antikem Muster mit 9-poliger SubD-Steckverbindung. Wenn Sie in die Geräteverwaltung schauen, findet sich dort dann ein COM1- oder / und vielleicht auch ein COM2-Eintrag. Diese Schnittstelle arbeitet dann sicher mit +/-12V-Pegeln. Prinzipiell lässt sich dort auch ein Selbstbaugerät mit TTL-Pegeln wie beim Arduino oder dem ESPs anschließen – nicht direkt, aber, wie meine Wetterstation, über einen fliegenden Pegel-Converter

wie in Abbildung 1. Er setzt die V.24 -12V in TTL 5V und V.24 +12 in TTL 0V um. Die Elektronik dafür befindet sich in dem blauen Stecker.

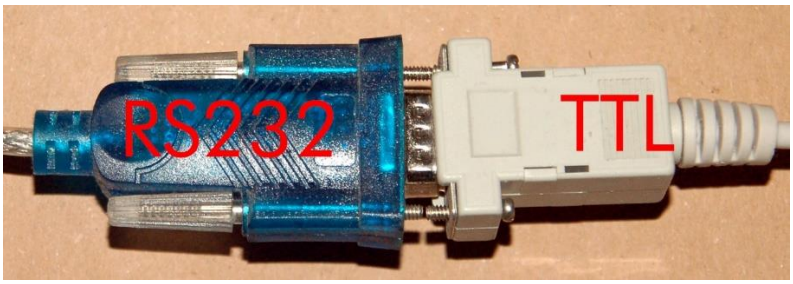

Abbildung 1: Fliegender RS232-TTL-Converter

So einen Wandler kann man auch gut selber bauen. Man nimmt dazu am besten einen MAX232-Chip von MAXIM, der kann zwei Paare RXD/TXD von +/-12V auf 0V/5V umsetzen und leicht in eine Eigenbauschaltung integriert werden.

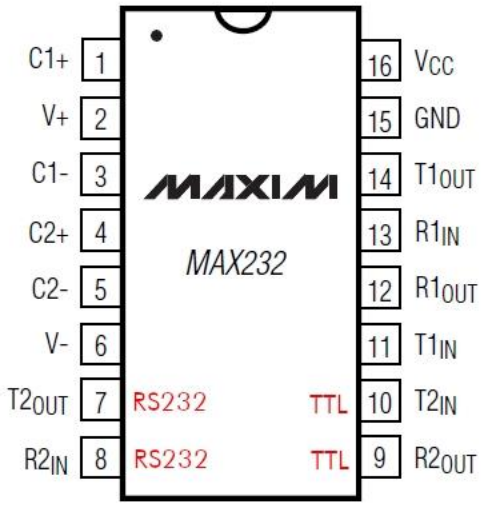

Abbildung 2: RS232-Schnittstellen-Pegelwandler MAX232

Die Schaltung dazu liefert das Datenblatt.

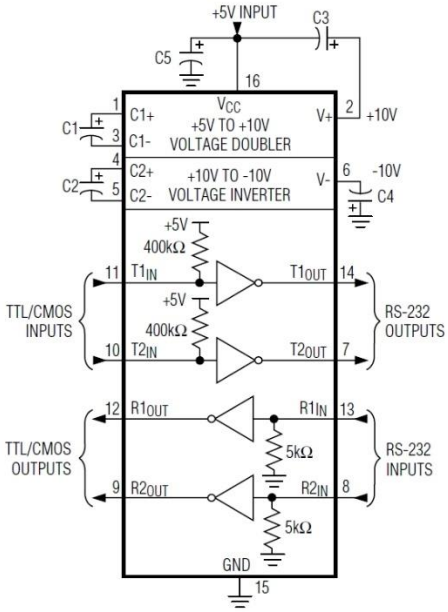

Abbildung 3: RS232-Schnittstellen-Pegelwandler MAX232

Die Realisierung der Schaltung auf einem PCB sehen Sie in Abbildung 4, das Layout und die Bestückung zeigt Abbildung 5. Alle Elkos sind 1µF/16V-Typen. Der Chip generiert die +/-10V aus der Betriebsspannung 5V selbst.

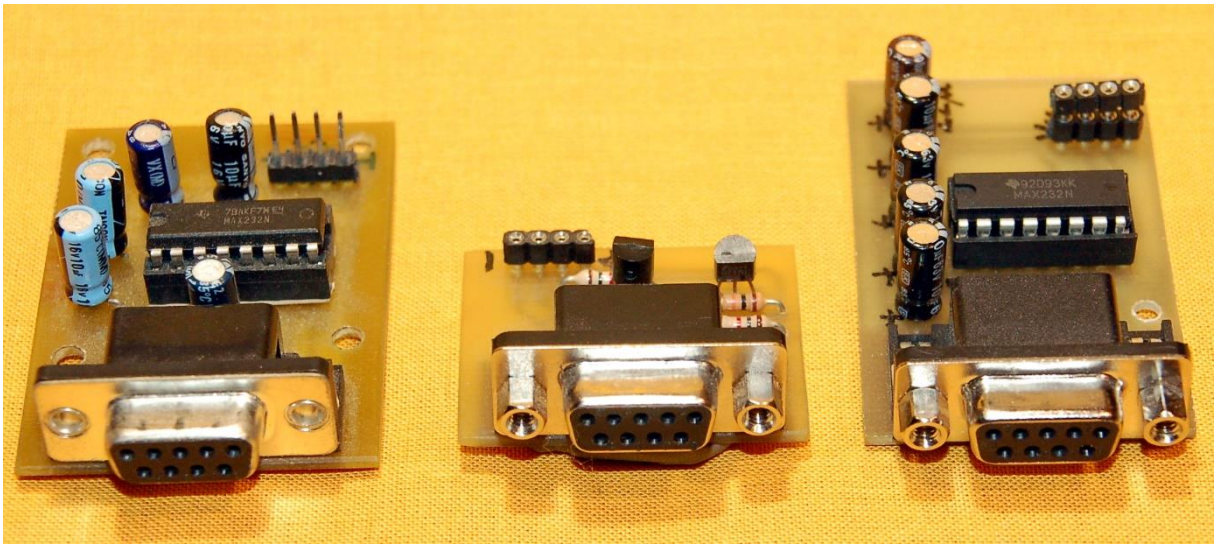

Abbildung 4: Schnittstellen-Module mit MAX232 und Transistoren

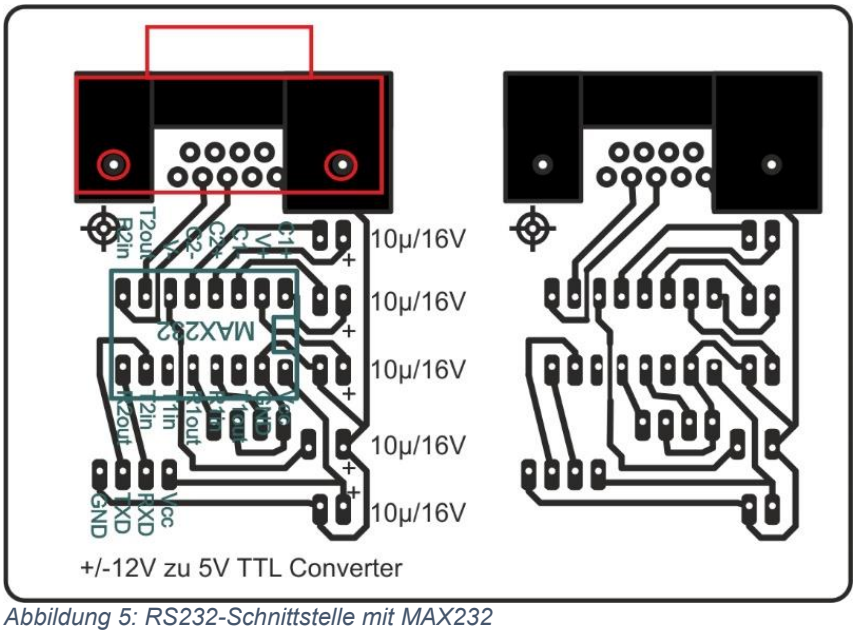

Abbildung 5: RS232-Schnittstelle mit MAX232

Das Layout können Sie als PDF-Datei herunterladen.

Möchte man zum Beispiel einen ESP32 mit seiner 3,3V-Bordspannung an eine alte COM1 oder COM2 über so einen Adapter anschließen, braucht man in der Empfangsleitung einen Spannungsteiler, der die 5V von TXout des MAX232 auf verträgliche 3V an RX des ESP32 herunterdrückt. Die 3,3V an TX des ESP32 reichen aus, um am Ausgang T2out des MAX232 -10V zu erhalten, wenn TX auf 3,3V liegt.

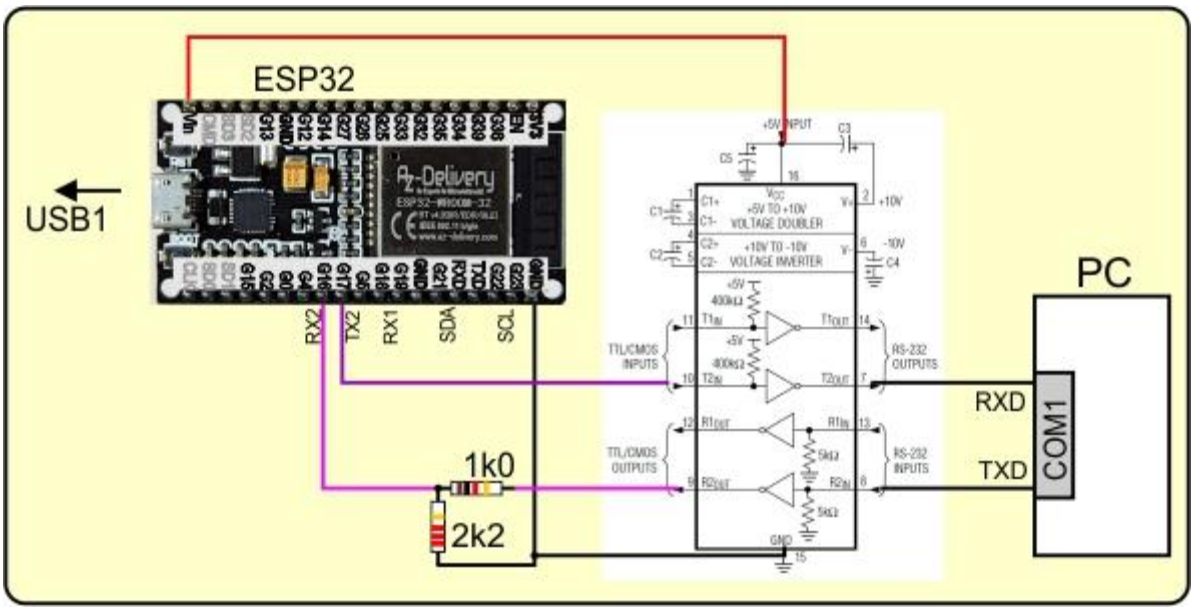

Abbildung 6: ESP32 an herkömmlicher RS232-Schnittstelle

Leitungslänge versus Baudrate

Weshalb man für die RS232-Übertragung Spannungen zwischen +/-3V und +/-15V vorgeschrieben hat, liegt an der besseren Störsicherheit der Signale. Je höher die Amplitude der Signalpegel, desto geringer ist der Einfluss von Störsignalen bei der Übertragung. Immerhin kann man mit RS232-Verbindungen Entfernungen bis knapp einem Kilometer aufbauen. Eine Tabelle nach den Angaben von TI (Texas Instruments) gibt einen Überblick. Maßgeblich beteiligt an Leitungsstörungen ist aber auch der Umstand, dass die Leitung an der Empfängerseite nicht durch den Wellenwiderstand des Kabels abgeschlossen ist und daher Reflexionen auf der Leitung die ankommenden Impulse überlagern können.

Anmerkung:
Durch Darstellung eines gesendeten und reflektierten Impulses am Anfang einer Leitung auf dem Oszilloskop kann man aus der Laufzeit des Impulses, hin und zurück, die Länge des Kabels oder die Position eines Kabelbruchs oder Kurzschlusses berechnen.

Baudrate	Entfernung
2400	900m
4800	300m
9600	152m
19200	15m
57600	5m
115200	2m

Abbildung 7: Entfernung versus Baudrate

Nach dieser Tabelle kann man im Haus locker Kabellängen bis ca. 20m – 25m einplanen, um mit 9,6kBaud noch sicher übertragen zu können. Das ist dann hilfreich, wenn ein WLAN wegen massiver Stahlbetondecken etc. nicht eingesetzt werden kann.

RS232 auf dem ESP32

Den ESP32 zeichnen drei vollwertige UARTs aus. UART0 ist für REPL im Einsatz und wie beim ESP8266 für uns nicht verfügbar, solange vom Terminal auf den Controller zugegriffen werden muss. UART1 ist an GPIOs herausgeführt, die aber standardmäßig, ebenso wie beim ESP8266, für die Unterhaltung des Controllers mit dem Flash-Speicher belegt sind. ABER! Anders als beim kleinen Bruder kann man dem UART1 andere GPIOs, statt den standardmäßig zugeordneten Pins, zuweisen. Und das macht diese Schnittstelle für uns ebenfalls nutzbar. Darüber hinaus gibt es auch noch einen UART2 mit TX an GPIO17 und RX an GPIO16. Die beiden GPIOs haben keine weitere Sonderfunktion, was UART2 ohne Einschränkung frei verfügbar macht.

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft
Betriebs-Software Logic 2 von SALEAE
PuTTY ein Terminalprogramm

Verwendete Firmware für einen ESP32:

Micropython Firmware Download
v1.19.1 (2022-06-18) .bin

Verwendete Firmware für einen ESP8266:

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

Die MicroPython-Programme zum Projekt:

rs232_thermometer.py: Demoprogramm aus dem vorigen Blogpost
keller.py: Datensender
display.py: Empfangsstation
ssd1306.py: Hardwaretreiber für das Display
oled.py: API für das Display

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 07.01.2024) 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.

Starten wir doch gleich mit ein paar Experimenten. Was wir dazu brauchen sagt uns die Hardwareliste.

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI Wifi Development Board mit CP2102 oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul kompatibel mit Arduino oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI Wifi Development Board mit CH340 oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI WLAN unverlötet mit CP2102
1	KY-015 DHT 11 Temperatursensor Modul
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
diverse	Jumper Wire Kabel 3 x 40 STK
1	FT232-AZ USB zu TTL Serial Adapter für 3,3V und 5V
1	Logic Analyzer

Die erste Schaltung dient der Untersuchung vor UART1 und UART2 mit dem Logic Analyzer. Dessen Installation und Einrichtung habe ich im ersten Teil zu diesem Thema genau beschrieben. Hier das Schaltbild für den ESP32.

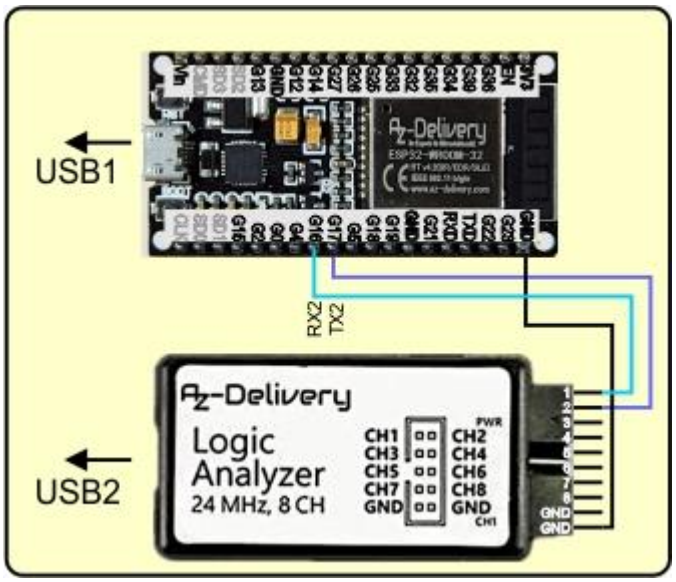

Abbildung 8: ESP32 under Test

In REPL geben wir folgende Anweisungen ein.

>>> from machine import UART
>>> u = UART(2, baudrate=115200)

Die nächste Zeile geben wir nur ein, ohne sie abzuschicken. Wir starten Logic2. Mit der Taste "R" leiten wir einen Scan ein und wechseln sofort zu Thonny, wo wir den letzten Befehl an den ESP32 schicken. Zurück nach Logic2 – "R" stoppt die Aufzeichnung. Mit der "7" teilt uns MicroPython mit, dass sieben Zeichen gesendet wurden.

>>> u.write(b"HALLO\r\n")
7

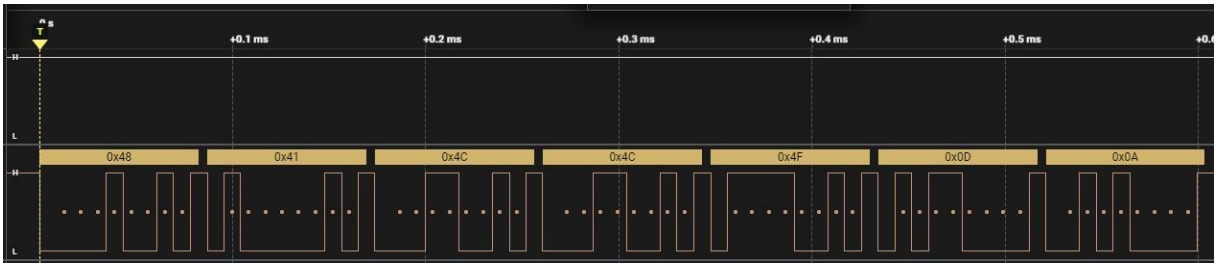

Abbildung 9: HALLO nur gesendet

Nun stecken wir eine Brücke von GPIO16 zu GPIO17 und verbinden somit TX2 mit RX2. Wir haben ein Null-Modem gebaut. Der UART2 empfängt jetzt alles, was er über TX2 sendet, wieder (gleichzeitig) an RX2. Wir müssen nur noch das Prozedere der Abtastung wiederholen, um den Scan von Abbildung 10 zu erhalten.

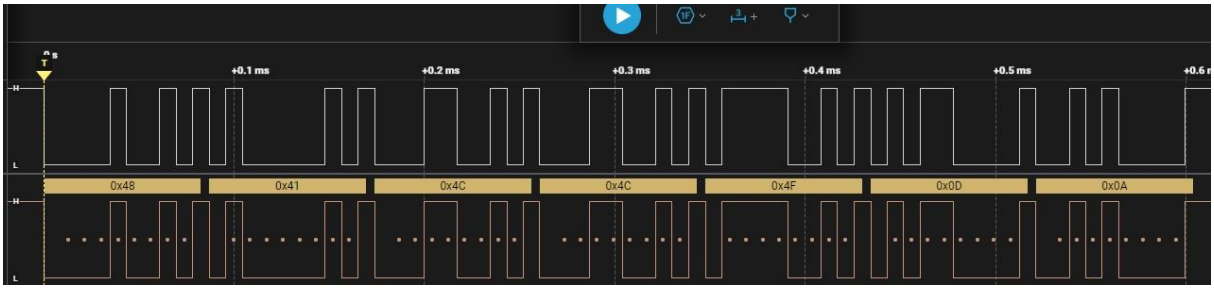

Abbildung 10: HALLO gesendet und empfangen mit Null-Modem-Verbindung

Den Empfang bestätigen wir durch folgende Anweisung.

>>> u.read()
b'HALLO\r\n'

Wir entfernen jetzt die Brücke und stecken sie neu von GPIO16 nach GPIO19. An GPIO19 stecken wir auch den Kanal 3 des Logic Analyzers an. Die Anschlüsse sind von 1 bis 8 durchnummeriert, die Kanäle in Logic2 aber von 0 bis 7, im Prinzip nicht schlimm, man muss es nur wissen.

Weiter geht es mit folgenden Anweisungen:
>>> u2 = UART(2, baudrate=115200)
>>> u1 = UART(1, baudrate=115200,tx=19,rx=18)

Nur eingeben, aber noch nicht abschicken, Aufzeichnung starten …
>>> u1.write(b"HALLO\r\n") jetzt abschicken …
7

>>> u2.read()
b'HALLO\r\n'

Fröhliches Händegeklapper – es hat funktioniert! Wir haben Zugriff auf UART1, ohne den Chat des Controllers mit dem externen Flash zu stören.

Abbildung 11: UART1 an UART2

Die Lösung liegt in der freien Zuordnung der UART-Anschlüsse zu beliebigen GPIO-Pins. Der ESP8266 bietet diese Möglichkeit leider nicht. Beachten muss man lediglich, dass die Pins 34, 35, 36, und 39 nur als Eingänge fungieren können, und dass manche Pins den Bootvorgang beeinflussen.

Pin - Zustand beim normalen Booten
0 – HIGH; LOW zum Flashen
2 – HIGH; LOW zum Flashen
3 – HIGH
12 - LOW

ESP32 und serielle Peripherie

Wir simulieren jetzt ein Peripheriegerät, das über RS232 mit dem ESP32 verbunden wird. Nehmen wir an, die Temperatur in einem Kellerraum soll überwacht werden. Eine Übermittlung der Daten via WLAN hat nicht funktioniert aber eine alte, nicht mehr verwendete Telefonleitung (4 Adern) führt in den 3.Stock.

Im Keller stationieren wir die ESP8266-Einheit vom letzten Beitrag. Den USB-RS232-Converter klemmen wir ab. Stattdessen verbinden wir zwei Adern des "Telefonkabels" mit GND (schwarz) und D4 (GPIO2) (weiß). An der ESP32-Station kommt die schwarze Leitung auch GND und das weiße Kabel wird mit GPIO18 = RX1 verbunden.

Im Schaltbild schaut das so aus wie in Abbildung 12.

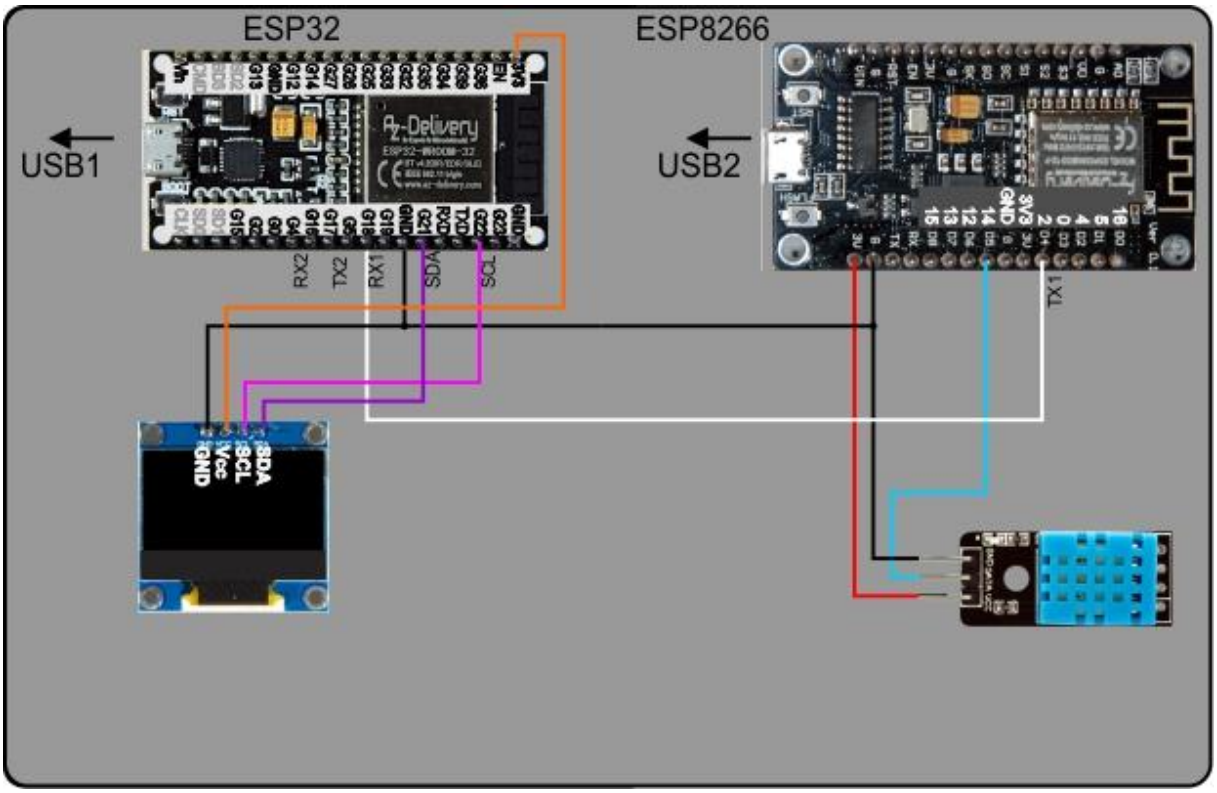

Abbildung 12: ESP32 und ESP8266 über RS232 gekoppelt

Und das ist der Aufbau, die LED hat sich verirrt, sie wird hier nicht verwendet.

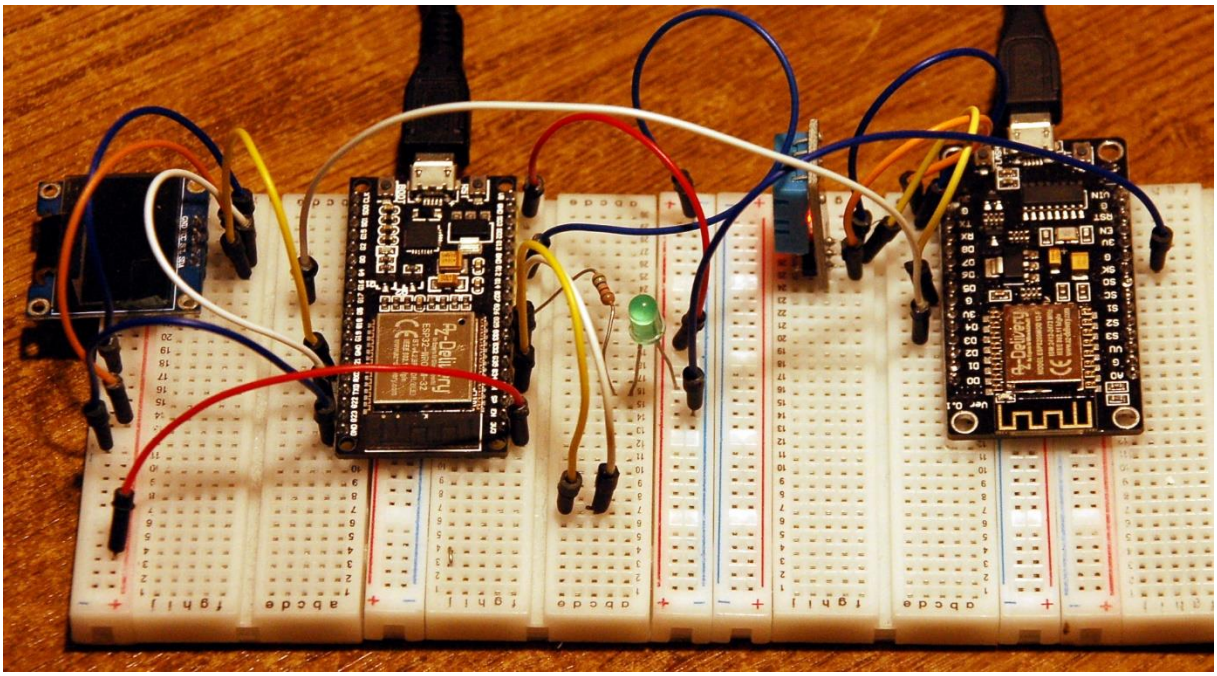

Abbildung 13: ESP32 (links) und ESP8266

Damit wir beide Programme gleichzeitig editieren und auch testen können, bedarf es einer kleinen Änderung der Thonny-Einstellungen. Im Folder **General** der **Thonny Options** nehmen wir den Haken bei dem roten Pfeil heraus und quittieren den Dialog mit **OK**.

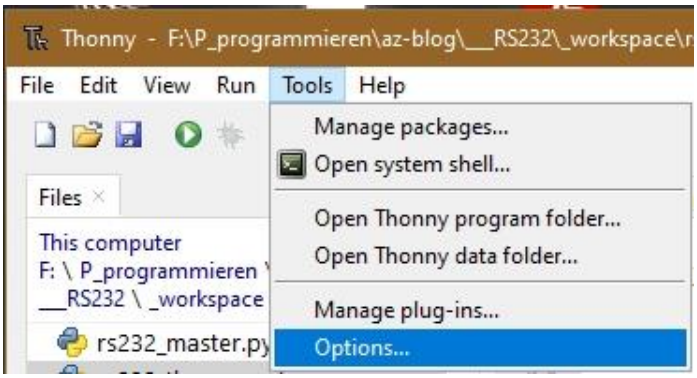

Abbildung 14: Einstellungen von Thonny ändern

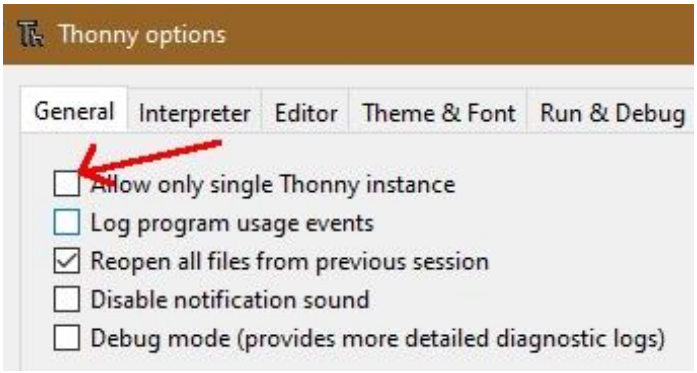

Abbildung 15: Den Haken hier entfernen

Danach muss Thonny neu gestartet werden. Über **Run – Select Interpreter** verbinden wir uns mit dem ESP32 an COM4.

Abbildung 16; Thonny-Options aufrufen

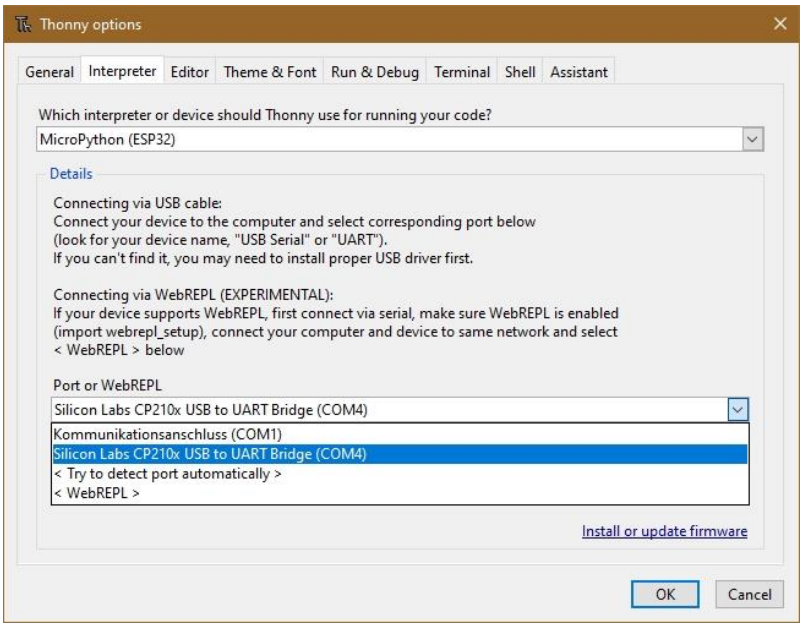

Abbildung 17: Anschluss des ESP32 festlegen

Jetzt stecken wir den ESP8266 an den USB und starten Thonny ein zweites Mal. Wir bekommen eine Fehlermeldung, die uns aber gar nicht meinen kann:

Unable to connect to COM4: could not open port 'COM4': PermissionError(13, 'Zugriff verweigert', None, 5)

Wir rufen nämlich wieder die Interpreterauswahl auf, stellen den Controllertyp auf ESP8266 und verbinden uns zum neu erschienen COM5-Port - **OK**.

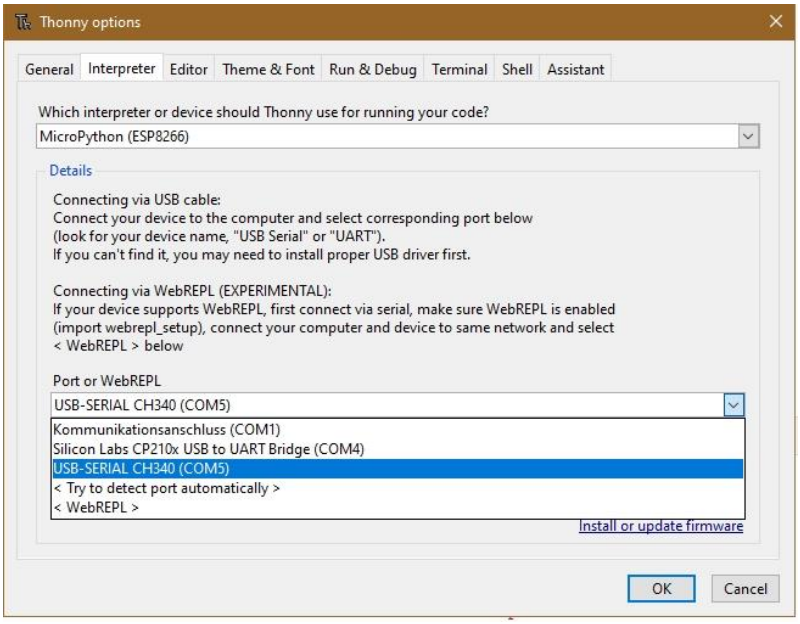

Abbildung 18: Verbindung mit dem ESP8266

Das Programm rs232_thermometer.py aus der vorangegangenen Folge greifen wir wieder auf und bringen ein paar kleine Änderungen an. Wir ändern den Namen des Programms und wegen des "langen" Kabels stellen wir die Baudrate auf **9600**. Den passiven Schönheitsschlaf mit **sleep**(), ersetzen wir durch einen aktiven Rundlauf mit der Funktion warten. Das Sportgerät dazu finden wir im Modul **timeout**, von dem wir die Methode **TimeOut**() importieren. Die setzen wir in der Funktion **warten**() als nicht blockierenden Softwaretimer ein.

In **TimeOut**() wohnt eine lokale Funktion namens **compare**(), sie ist eine sogenannte Closure. Das Listing finden Sie am Schluss dieses Beitrags. Der Aufruf der Routine **TimeOut**() gibt die Referenz auf die innere Funktion **compare**() zurück, die der lokalen Variablen **genugGewartet** zugewiesen wird. Wir haben auf diese Weise Zugriff auf die in **TimeOut**() deklarierte Funktion **compare**() und auf die übergebene Verzögerungszeit, auch wenn **TimeOut**() bereits verlassen wurde. **genugGewartet()** gibt **True** zurück, wenn die Verzögerungszeit abgelaufen ist. In der Zwischenzeit fragen wir fleißig die Flash-Taste des ESP8266 ab, was mit **sleep**() nicht möglich ist. Wurde die Taste gedrückt, schließen wir UART1 und beenden das Programm.

Dann speichern wir das Programm unter **keller.py** ab.

```
# keller.py
# Pintranslator fuer ESP8266-Boards
#LUA-Pins       D0 D1 D2 D3 D4 D5 D6 D7 D8 RX TX
#GPIO-32 Pins   16  5  4  0  2 14 12 13 15  3  1
#Achtung        hi sc sd hi hi          lo hi hi
#Verwendung              TA RS DH

from dht import DHT11
from machine import Pin,UART
# from time import sleep,ticks_us   <<< löschen
from sys import exit
from timeout import TimeOut

taste= Pin(0,Pin.IN,Pin.PULL_UP)

dht=DHT11(Pin(14))
u = UART(1, baudrate=9600)
delay = 5

def warten(sekunden):
    genugGewartet=TimeOut(sekunden)
    while not genugGewartet():
        if taste.value() == 0:
            u.deinit()
            print("Programm beendet")
            exit()

while 1:
    dht.measure()
    warten(2)
    temp=dht.temperature()
    hum=dht.humidity()
    print("Temperatur: {}; Rel. Luftfeuchte: {}". \
          format(temp,hum))
#     u.write((str(temp)+";"+str(hum)+"\n").encode())
    u.write(("{};{}\n".format(temp,hum)).encode())
    warten(delay-2)
```

Weil an zwei Stellen gewartet werden muss, wurde die Sequenz als Funktion deklariert. Während mit **sleep**() reichlich lange gewartet werden muss, bis die Tastenbetätigung Wirkung zeigt (delay = 300 entspricht zum Beispiel bis zu 5 Minuten) erfolgt der Abbruch über die Funktion **warten**() jetzt umgehend.

Weil der Empfänger jetzt ein MicroPython-Client ist, und keine Windowsanwendung, entfernen wir das cr = \r = 0x0D in der Ausgabezeile.

Jetzt wechseln wir ins erste Thonny-Fenster, um das Programm für die Anzeige der übermittelten Werte zu erstellen. Ein paar Tests zeigen die Vorgehensweise. Damit alles funktioniert, müssen die beiden Dateien **ssd1306.py** und **oled.py** in den Flash des ESP32 hochgeladen werden.

>>> from oled import OLED
>>> from machine import Pin,UART,SoftI2C
>>> u = UART(1, baudrate=9600, rx=18, tx=19)
>>> u
*UART(1, baudrate=9600, bits=8, parity=None, stop=1, tx=19, rx=18, rts=-1, cts=-1, txbuf=256, **rxbuf=256**, timeout=0, timeout_char=2)*

>>> i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=100000)
>>> d = OLED(i2c,heightw=64)

Der Aufruf der Instanz **u** verrät uns, dass der Empfangspuffer der Schnittstelle 256 Zeichen aufnehmen kann. Das ist gut, denn so müssen wir uns nicht selbst um das Empfangsmanagement kümmern. Werden von UART am Eingang RX1 ankommende Zeichen registriert, wandern sie automatisch in den Puffer. Von dort können wir die Bytes abholen, wenn uns danach ist. Lassen wir doch den ESP8266 mal eine Zeil lang werkeln. – Flashtaste – Programmstopp.

ESP8266-Fenster: Taste F5
>>> %Run -c $EDITOR_CONTENT
Temperatur: 20; Rel. Luftfeuchte: 34
Temperatur: 20; Rel. Luftfeuchte: 32
Temperatur: 20; Rel. Luftfeuchte: 34
Temperatur: 20; Rel. Luftfeuchte: 34
Temperatur: 20; Rel. Luftfeuchte: 34
Programm beendet

ESP32-Fenster:
>>> u.any()
30

Jeder der fünf Datensätze wurde mit einem Zeilenvorschub beendet. Das erlaubt uns, jeweils einen Datensatz mit **u.readline**() aus dem Puffer zu lesen.

>>> u.readline()
b'20;34\n'

Zur Ermittlung der einzelnen Werte muss das bytes-Objekt zuerst in einen String umgewandelt werden.

>>> b'20;34\n'.decode()
'20;34\n'

Dann entfernen wir das Linefeed-Zeichen \n durch die Methode **strip**() des Stringobjekts.

>>> '20;34\n'.strip("\n")
'20;34'

Und schließlich teilen wir den String am ";" auf. Das Ergebnis von **split**() ist eine Liste, die wir durch die Zuweisung auf die beiden Variablen auch gleich entpacken.

```
>>> '20;34'.split(";")
['20', '34']


>>> t,h = ['20', '34']
>>> t;h
'20'
'34'

>>> t,h='20;34'.split(";")
>>> t;h
'20'
'34'
```

So und jetzt das Ganze in einem Abwasch:

```
>>> t,h=u.readline().decode().strip("\n").split(";")
>>> t;h
'20'
'32'
```

Für die Ausgabe der Werte im Display erzeugen wir zwei Formatstrings.

```
>>> Hum= "Feuchte: {} %".format(str(h))
>>> Temp="Temp.  : {} *C".format(str(t))
>>> d.writeAt(Temp,0,1,False)
14
>>> d.writeAt(Hum,0,2)
13
```

Das erste **writeAt**() schreibt die Zeichen nur in den Display-Pufferspeicher, die zweite Anweisung ergänzt den Feuchtewert und sendet den Puffer zum Display, weil der optionale Schlüsselwort-Parameter **show** der Methode per default mit True vorbelegt ist.

Die besprochenen Beispiele bilden zusammen das Programm display.py.

```
# display.py
# Demoprogramm zur Nutzung des UART1 auf dem ESP32
#
from machine import Pin,UART,SoftI2C
from sys import exit
from oled import OLED

i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=100000)

taste= Pin(0,Pin.IN,Pin.PULL_UP)

d = OLED(i2c,heightw=64)
u = UART(1, baudrate=9600, rx=18, tx=19)
u.read() # Puffer leeren
```

```
d.writeAt("WERTE KELLER",2,0)
d.writeAt("============",2,1)
while 1:
    zeichen=u.any()
    if zeichen !=0:
        t,h=u.readline().decode().strip("\n").split(";")
        Temp="Temp.   : {} *C".format(str(t))
        Hum= "Feuchte: {} %".format(str(h))
        d.clearFT(0,3,15,5,False)
        d.writeAt(Temp,0,3,False)
        d.writeAt(Hum,0,4)
    if taste.value() == 0:
        d.writeAt("!beendet!",2,5)
        exit()
```

Die Funktionalität habe ich abschließend mit 20m Kleinsignal-Kabel 4x 0,6mm² erfolgreich überprüft.

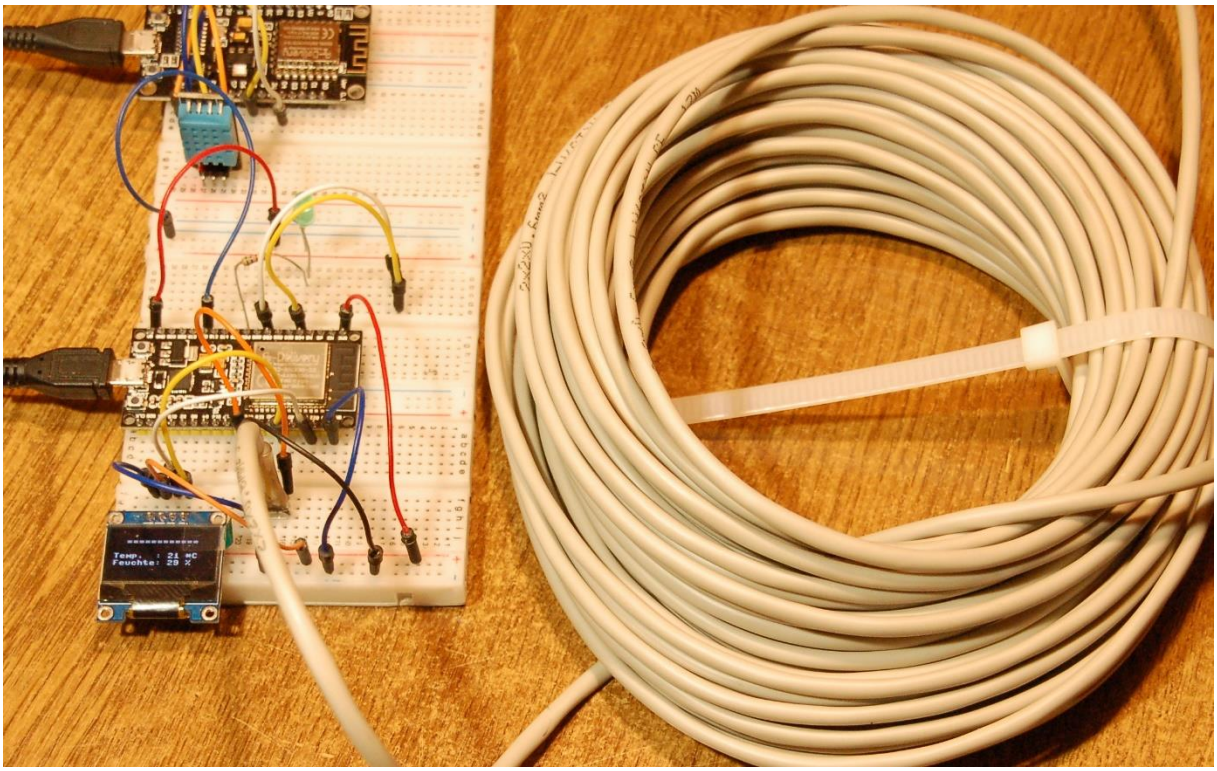

Abbildung 19: Test mit 20 Meter Kabel 4x0,6qmm

In der nächsten Folge werden wir den PC mit dem großen Bruder von MicroPython bestücken, CPython oder kurz Python. Es hat einen viel größeren Umfang an Fähigkeiten, Funktionen und Modulen, kann es doch auf wesentlich mehr Ressourcen zurückgreifen. Damit sich der PC auch mit unserem ESP32 unterhalten kann, braucht er pySerial. Auch das werden wir installieren. Außerdem untersuchen wir die UART-Eigenschaften des Raspberry Pi pico und lassen ihn als neues Mitglied unserer RS232-Familie mit dem PC plaudern.

Bis dann!

Anhang:
Hier kommt noch das versprochene Listing der Funktion **TimeOut**(). Näheres zu deren Funktion erfahren Sie in dem PDF-Dokument Closures und Decorators.pdf

```
def TimeOut(t):
    start=time()

    def compare():
        nonlocal start
        if t==0:
            return False
        else:
            return int(time()-start) >= t

    return compare
```