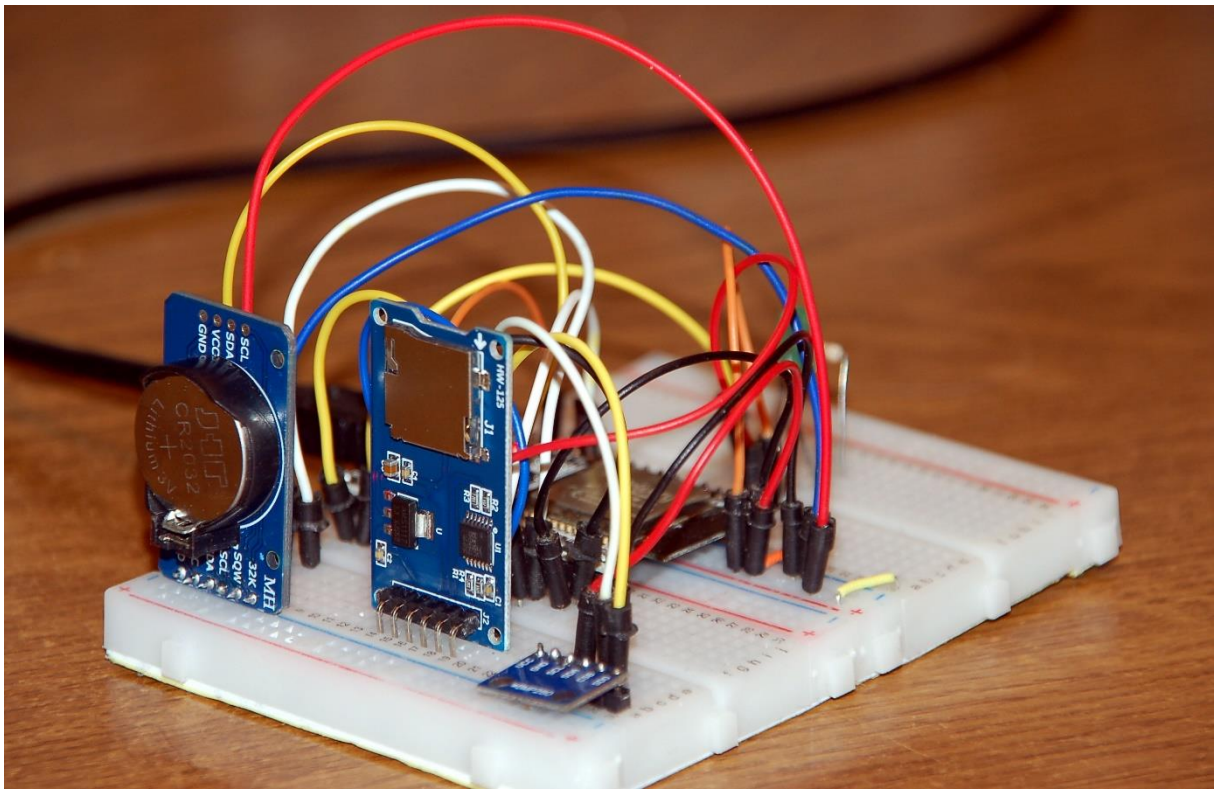

Abbildung 1: Datenlogger mit RTC, SD-Card und BMP280

Wie waren denn die Temperatur- und Luftdruckwerte im letzten Monat? Vielleicht interessieren auch noch die Werte von Beleuchtung, Sonnenscheindauer und relativer Feuchte im Treibhaus. Wie oft hat über Nacht das Licht im Keller gebrannt und wie oft hat die Heizung wie lang gefeuert? Fragen, deren Beantwortung nicht unmittelbar auf den Nägeln brennt. Dennoch sind die Ergebnisse derartiger Langzeitmessungen für eine pragmatische Lösung gewisser Missstände ganz hilfreich. Was tun, wenn kein Server in greifbarer Nähe ist, an den ein Messknecht der Kategorie ESP32 Daten senden könnte. Ganz einfach, man gibt dem ESP32 den Auftrag, die Daten in einer Datei zu speichern. Welche Möglichkeiten man dazu hat und wie das geht, das verrät Ihnen der heutige Beitrag. Herzlich willkommen zum Blog mit dem Thema

MicroPython und ESP32/ESP8266 als Datenlogger

Datenlogger sind Anwendungen, die Daten von Sensoren sammeln und zusammen mit einem Zeitstempel in einer Datei ablegen. ESP8266 und ESP32 sind Meister im Anbinden von Sensoren aller Art. Dafür stehen diverse Schnittstellen vom einfachen GPIO-Pin über analoge Spannungseingänge bis hin zu Bussystemen wie One Wire, I2C und SPI zur Verfügung. Die Frage: "Wohin mit den Daten?" ist zunächst nicht so einfach zu beantworten, denn ein ESP32 ist doch kein PC auf dem man mal eben eine Datei aufmacht, um die Daten reinzuschieben.

So seltsam es auch klingen mag, aber auf dem ESP32 geht das genauso wie unter Windows oder Linux oder… Das "Wie" klären wir sofort nach der Liste der benötigten Hardware. Trotzt der drei Möglichkeiten der Speicherung gibt es nur zwei zugeordnete Teilelisten.

Die Hardware

Dateien im Flash des ESP32/ESP8266
oder im EEPROM auf dem DS3231-Board ablegen:

1	ESP32 LOLIN32 oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F
1	BMP280
1	DS3231-Modul (Real-Time-Clock aka RTC)
1	LED, Farbe egal
1	Widerstand 1k0
1	Netzteil oder Batterie (mit Regler) auf 3,3V oder 5V

Dateien auf einer SD-Speicherkarte ablegen:

1	ESP32 LOLIN32
1	SPI Reader Micro Speicher SD TF Karte
	Rest (ohne Controller) wie oben

MicroPython bietet die Möglichkeit, den Flash-Speicher auch als Träger eines Dateisystems zu nutzen. Die dazu nötigen Methoden ähneln sehr stark denen der Systeme LUA und Linux, dürften aber auch den Anhängern von Windows nicht ganz unbekannt sein. Die Dialekte weisen halt gewisse Unterschiede in der Syntax auf. Hier wie dort gibt es Befehle, die den Umgang mit Speicher- und Lesevorgängen erlauben und im Grundwortschatz enthalten sind und solche, die eher das Betriebssystem (aka Operating System oder kurz OS) betreffen und dem Anlegen, Verwalten und Entfernen von Verzeichnissen und Dateien dienen. Mit der ersten Gattung werden wir uns heute beschäftigen. Zuvor aber noch ein paar Informationen zur Software.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

24cxx.py
bmp280.py
datalogger.py
datalogger_ds.py
datalogger_eep.py
datalogger_flash.py
datalogger8266.py
datalogger8266_ds.py
ds3231.py
ds3231.py
readEEPROM.py
sdcard.py

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung wie die MicropythonFirmware 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, bevor 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 compilieren 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 Macro 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 compiliertes 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.

1. Dateien auf dem ESP32 / ESP8266

Weil in den meisten Fällen der Flash-Bereich des ESP32 durch Programme bei weitem nicht ausgereizt wird, bleibt dort (reichlich) Platz für die Ablage von Dateien. Das Handling ist ganz einfach, wir öffnen ein Dateiobjekt (aka Handle) zum Schreiben, Anhängen oder Lesen, schreiben in die Datei oder lesen aus ihr und schließen das Objekt am Ende wieder. So einfach wie dieser Satz, ist auch die Handhabung, wie das folgende Programmschnippsel zeigt. Die einzelnen Befehle kann man auch nacheinander über die REPL-Kommandozeile eingeben.

```
f=open("testdatei.txt","w")
f.write("Hallo World\n")
f.close()
```

```
12
```

Nach dem Durchführen dieser Sequenz befindet sich im Root-Verzeichnis des ESP32/8266 die Datei **testdatei.txt**. MicroPython sagt uns, dass 12 Bytes geschrieben wurden. Wir können uns davon überzeugen, indem wir einen Refresh auf das Device-Directory des ESP32 in Thonny durchführen.

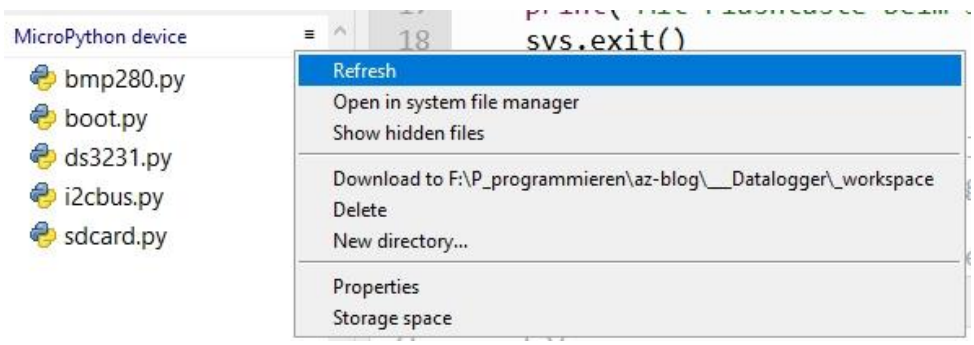

Abbildung 2: Refresh des Device Directories

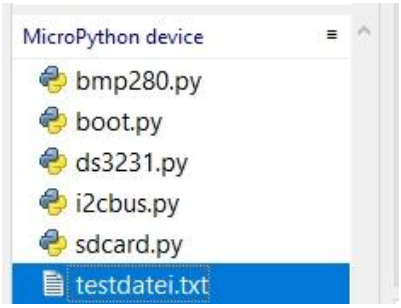

Abbildung 3: Anzeige der Testdatei

Durch Doppelklick auf den Eintrag können wir die Datei auch öffnen.

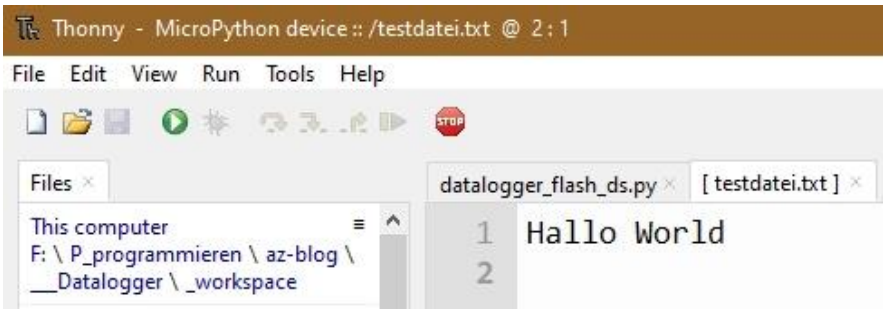

Abbildung 4: Hallo World

Natürlich kann MicroPython die Datei ebenfalls zum Lesen öffnen. Das "r" steht für lesen, das "w" für (neu) schreiben und das "a" für schreibend anhängen.

```
f=open("testdatei.txt","r")
print(f.readline())
f.close()
```

```
Hallo world
```

Sehr wahrscheinlich wird unser Datenlogger die meiste Zeit vor sich hin duseln und auf den nächsten Messjob warten. Dabei zieht er zur Selbsterhaltung ca. 45 mA. Energiesparender ist es, wenn wir wenigstens den ESP32 in dieser Zeit einfach zum Schlafen schicken, dann braucht er im Schnitt nur rund ein Drittel der Stromstärke verglichen mit dem Wachzustand. Zusammen mit der Peripherie sind das ca. 15 mA. Das Programm, das diese Betriebsart ermöglicht, hat keine 90 Zeilen.

Zum Tiefschlafmodus muss man folgendes wissen. ESP8266 und ESP32 handhaben den Deepsleep-Mode sehr unterschiedlich. Ich beschreibe hier das sehr einfache Vorgehen für den ESP32. Wie das beim ESP8266 aussieht, schauen wir uns gegen Ende dieses Beitrags an.

Die Methode **deepsleep**() wohnt im Modul **machine**. Wir müssen sie daher importieren. Die Methode verlangt einen Parameter der die Ruhezeit in Millisekunden angibt. Sobald deepsleep() aufgerufen wird, begibt sich der ESP32 auf das Sofa und ist für die angegebene Zeit durch nichts und niemand mehr ansprechbar. Ist die Ruhezeit abgelaufen, erwacht der ESP32 und startet **komplett neu** durch, so als würde er gerade eingeschaltet. Das dauert natürlich ein paar Sekunden, bis auch alle

Subsysteme erneut laufen. Wir werden das durch den Wert in der Variablen **Startzeit** berücksichtigen.

Damit wir uns nicht komplett aussperren, müssen wir eine Notbremse einbauen, das ist die **Flashtaste** des ESP32. Wird die zur rechten Zeit gedrückt und gehalten, bricht das Programm ab und wir haben wieder Zugriff auf den ESP32 via REPL. Drücken der RST-Taste hilft nicht, denn danach startet der Controller so durch, wie nach dem Erwachen. Ohne Abbruchmöglichkeit kämen wir also nicht mehr ins System. Zustände signalisiert uns der ESP32 durch eine LOW-aktive LED am Pin GPIO2. Man kann die Taste loslassen, wenn die LED ausgegangen ist.

```
# datalogger_flash_ds.py
# Autor: J. Grzesina
# Rev.: 1.0
# Date: 27.08.2021
# ********************************************************
import esp
esp.osdebug(None)
import os
import gc               # Platz fuer Variablen schaffen
gc.collect()
from machine import Pin,I2C,deepsleep
from time import sleep
from bmp280 import BMP280
import os, sys
from ds3231 import DS3231

taste=Pin(0,Pin.IN,Pin.PULL_UP)
led=Pin(2,Pin.OUT,value=0)
sleep(2)
led.value(1)
if taste.value()==0:
    print("Mit Flashtaste beim Start abgebrochen")
    sys.exit()
# ****************** Filesystem Info   ******************
def df():
    s = os.statvfs('//')
    return ('{0} KB'.format((s[0]*s[3])/1024))
```

```
# ********************** I2C+ Sensors **********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
b=BMP280(i2c)
rtc=DS3231(i2c)
d2="{:0>2}."
h3="{:0>2}:"
y5="{:>4} "
s3="{:0>2};"
p4="{:0>4}\n"
t6="{:> 5.1f};"
Y,M,D,h,m,s,flag=0,0,0,0,0,0,0

i2cDevices=i2c.scan()
```

```
print("Found I2C-Devices @:",end="")
for i in i2cDevices:
    print(hex(i),end=";")
    # 0x57: DS3231.EEPROM 32Kb
    # 0x68: DS3231.RTC
    # 0x76: BMP280
print("")
# ********************* MAIN PART **********************
Minutenabstand=10
Startzeit=5800 # ms
duration=1000*60*Minutenabstand - Startzeit
duration=10000-Startzeit
# ********************* MAIN PART **********************
led.value(0)
Y,M,D,_,h,m,s=rtc.DateTime()
T=b.calcTemperature()
P=int(b.calcPressureNN(465,T))
th=t6.format(T).replace(".",",")
dtwString=d2.format(D)+d2.format(M)+y5.format(Y)+\
          h3.format(h)+h3.format(m)+s3.format(s)+\
          th+p4.format(P)
print(dtwString)
f=open("logging.txt","a")
f.write(dtwString)
f.close()
sleep(3)
led.value(1)
if taste.value()==0:
    print("Mit Flashtaste abgebrochen")
    sys.exit()
deepsleep(duration)
```

Damit wir den BMP280 und die RTC im DS3231 ansprechen können, brauchen wir die entsprechenden Treiber-Klassen aus den Modulen **bmp280**.py und **ds3231**.py. Diese beiden Dateien und das Modul **i2cbus**.py müssen ins Device-Directory hochgeladen werden. Wir erzeugen ein I2C-Objekt und instanziieren damit ein BMP280- und ein DS3231-Objekt durch Aufruf der Konstruktoren.

Es folgen Definitionen für Formatstrings, die eine einheitliche Ausgabe der Messdaten gewährleisten. Dann deklarieren wir die Variablen für die Datums-, Zeit- und Messgrößen und vergewissern uns, dass auch alle Schnittstellen der Sensoren ansprechbar sind. Während der Testphase setzen wir die Ruhezeit auf 10 Sekunden fest, danach sind Zeiten zwischen 10 Minuten und einigen Stunden sinnvoll.

Dann betreten wir den Hauptteil des Programms. Die heiße Phase wird nach außen durch die eingeschaltete LED kenntlich gemacht. Wir holen die Datums-, Zeit-, Temperatur- und Druckwerte und setzen daraus mit Hilfe unserer Formatstrings den String für die Ausgabe zusammen. Die Trennzeichen sind so gewählt, dass die Datei später direkt als CSV-Datei importiert werden kann. Die Ausgabe erfolgt im Terminalbereich von Thonny und in die Datei **logging**.txt durch Anhängen. Die Datei muss explizit geschlossen werden.

Erfolgt während der nächsten 3 Sekunden kein Tastendruck, dann macht der ESP32 ein Nickerchen von der eingestellten Dauer. Danach beginnt die Story vom roten Pferd wieder von neuem, falls – und das habe ich noch nicht erwähnt – ja, falls die Zeilen des obigen Listings sich in der Datei **boot**.py im Root Directory auf dem ESP32 befinden. Denn nur dann kann nach einem Neustart die automatische Ausführung des Programms erfolgen.

Während die LED leuchtet, sollte man den ESP32 nicht ausschalten, weil dann schlimmstenfalls das Dateisystem, es ist ein VfsLittle2, beschädigt werden kann und die Aufzeichnung der Messdaten für die Katz war. Wenn die LED gerade nicht leuchtet kann man die Messwertaufnahme durch Ausschalten beenden. Das ist hilfreich, wenn die Schlafenszeit mehr als ein paar Sekunden dauert.

Zum Auslesen der Logdatei wird auf dem PC Thonny gestartet und der ESP32 über USB-Kabel mit dem Rechner verbunden. Sobald die LED das erste Mal aufleuchtet wird mit der Flashtaste am ESP32 der Autostart abgebrochen. Das kann mehrmals hintereinander nötig sein. Dann können wir mit Rechtsklick auf **logging**.txt die Datei in den Workspace auf dem PC laden, um sie von dort aus weiterzuverarbeiten.

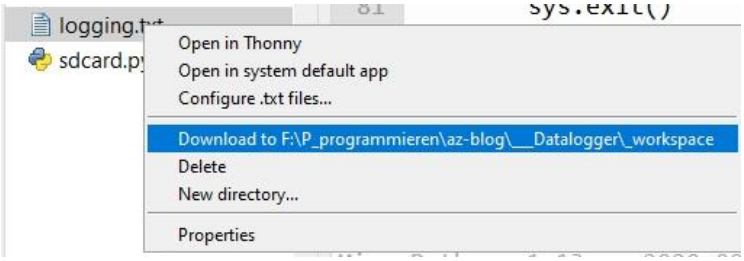

Abbildung 5: Logdatei auslesen

Jetzt würden Sie das alles sicher gerne ausprobieren. Hier ist die minimalistische Schaltung dazu.

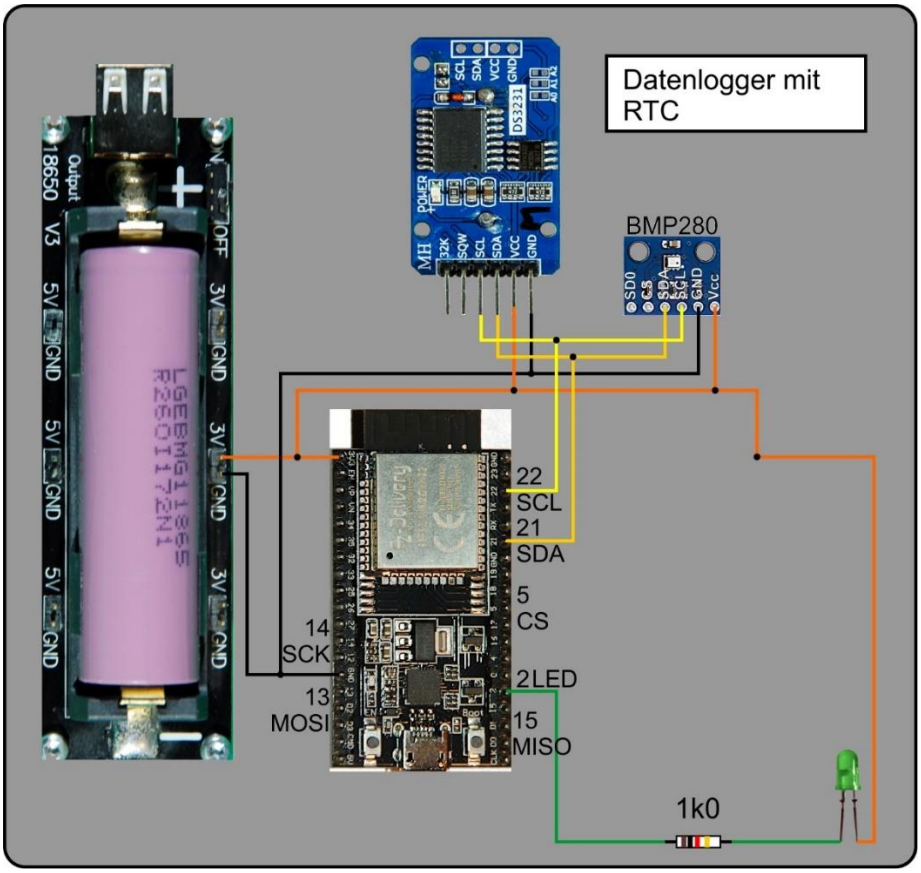

Abbildung 6: Datalogger_Schaltung ohne SD-Card Reader

Die Funktion df() liefert den verfügbaren Speicherplatz auf dem VfsLittle2-Dateisystem. Dass es sich um diese Art Filesystem handelt, sagt uns der Aufruf von os im Terminal, wenn der Object inspector eingeschaltet ist.

Abbildung 7: Object inspector einschalten

```
MicroPython v1.13 on 2020-09-02; ESP module (1M) with ESP8266
Type "help()" for more information.
>>> os
<module 'uos'>
>>>
```

Abbildung 8: Der Befehl os

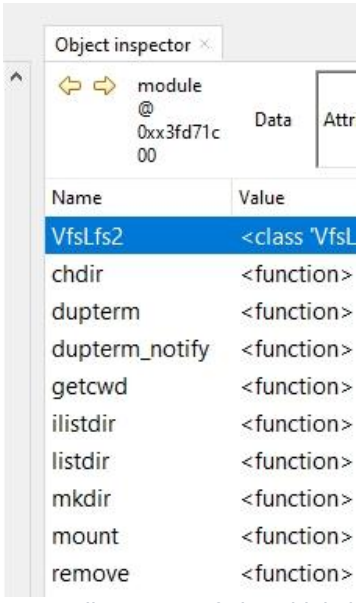

Abbildung 9: Filesystem-Info im Objekt inspetor

Mit einer Datensatzlänge von 31 Byte können wir leicht berechnen, wie viele Datensätze auf den ESP32 passen. Ähnlich wie bei einer Festplatte gibt es logische Sektoren von je 4096 Byte. Diesen Wert und die Anzahl freier Blöcke, hier 854, liefert der Befehl **os.statvfs**,zusammen mit anderen Werten in einem Tupel. Dieser Befehl wird auch in der Funktion df() verwendet.

```
>>> os.statvfs('//')
(4096, 4096, 866, 854, 854, 0, 0, 0, 0, 255)
```

In Abbildung 9 ist außerdem ersichtlich, wo sich die restlichen Befehle für die Datei- und Verzeichnisbehandlung verbergen, nämlich auch in der Klasse os.

2. Daten speichern in einem externen EEPROM

Der Vollständigkeit halber betrachten wir kurz die Datenspeicherung in einem externen EEPROM vom Typ AT24C32 o. ä. Wie die Typbezeichnung verrät, handelt es sich um einen nicht flüchtigen 32Kb-Speicher. In diese 4KB = 4096Byte passen 127 von unseren Datensätzen. Ein ESP32/ESP8266 bietet wie wir oben gesehen haben mehr als den 800-fachen Speicherplatz. Ich erwähne diese Speichermöglichkeit auch nur deshalb, weil auf dem RTC-Modul mit dem DS3231 ein solcher Speicherchip lebt. Er hat die Geräteadresse 0x57 und kann mit Hilfe der Methoden aus der Klasse ds3231.AT24C32 über den I2C-Bus angesteuert werden, nachdem ein entsprechendes Objekt instanziiert wurde. Das Beispielprogramm datalogger_eep.py können Sie gerne herunterladen und näher untersuchen. Wegen des geringen Speicherangebots ist es nicht für den Tiefschlafmodus konzipiert. Ein Timer übernimmt stattdessen die Zeitsteuerung. Über einen Index (Speicherplatznummer) wird die Speicheradresse berechnet und zugeordnet. Beim Erreichen der Obergrenze von 127 wird die Erfassung automatisch gestoppt. Wenn die LED in schnellem Rhythmus blinkt, ist dieser Zustand erreicht. Ein Druck auf die Flashtaste beendet den Programmlauf.

Das Programm readEEPROM.py kann das EEPROM auslesen und im Terminalfenster ausgeben. Über die Zwischenablage werden die Daten in eine einfache Textdatei zur weiteren Verarbeitung verfrachtet.

Das Modul 24cxx.py stellt die Klasse AT24CXX isoliert mit denselben Methoden zur Verfügung wie ds3231.AT24C32, kann aber auch größere EEPROMs mit 8KB ansteuern.

3. Eine SD-Card am ESP32

Genug mit den Peanuts, wenden wir uns dem Speicherriesen zu. Den Faktor 1000 beziehungsweise 1024, im Vergleich zum ESP32, ermöglichen uns die SD-Speicherkarten. Um sie zu nutzen brauchen wir ein Speicherkarten-Modul, eine Karte im Miniformat (mit Adapter zum Auslesen am PC) und die Klasse sdcard.py. Das Original habe ich von GitHub heruntergeladen und leicht modifiziert, damit es für meine Zwecke brauchbar wurde.

Wichtig ist die Angabe der entsprechenden GPIO-Pins für die Signale SCK, CS, MISO und MOSI. Vom Konstruktor werden keine Defaultwerte vorbelegt. Das im Original angegebene Pin GPIO12 für die MISO-Leitung konnte nicht verwendet werden, da sich der ESP32 bei jedem Neustart aufgehängt hat. Daher wurde der Anschluss ersatzweise an GPIO15 gelegt. Das SPI-Schnittstellen-Objekt definiere ich auch nicht in der Klasse sdcard.SDCard sondern außerhalb in datalogger.py und

übergebe es dann an den Konstruktor der SDCard-Klasse. Schauen wir uns aber erst einmal die erweiterte Schaltung an.

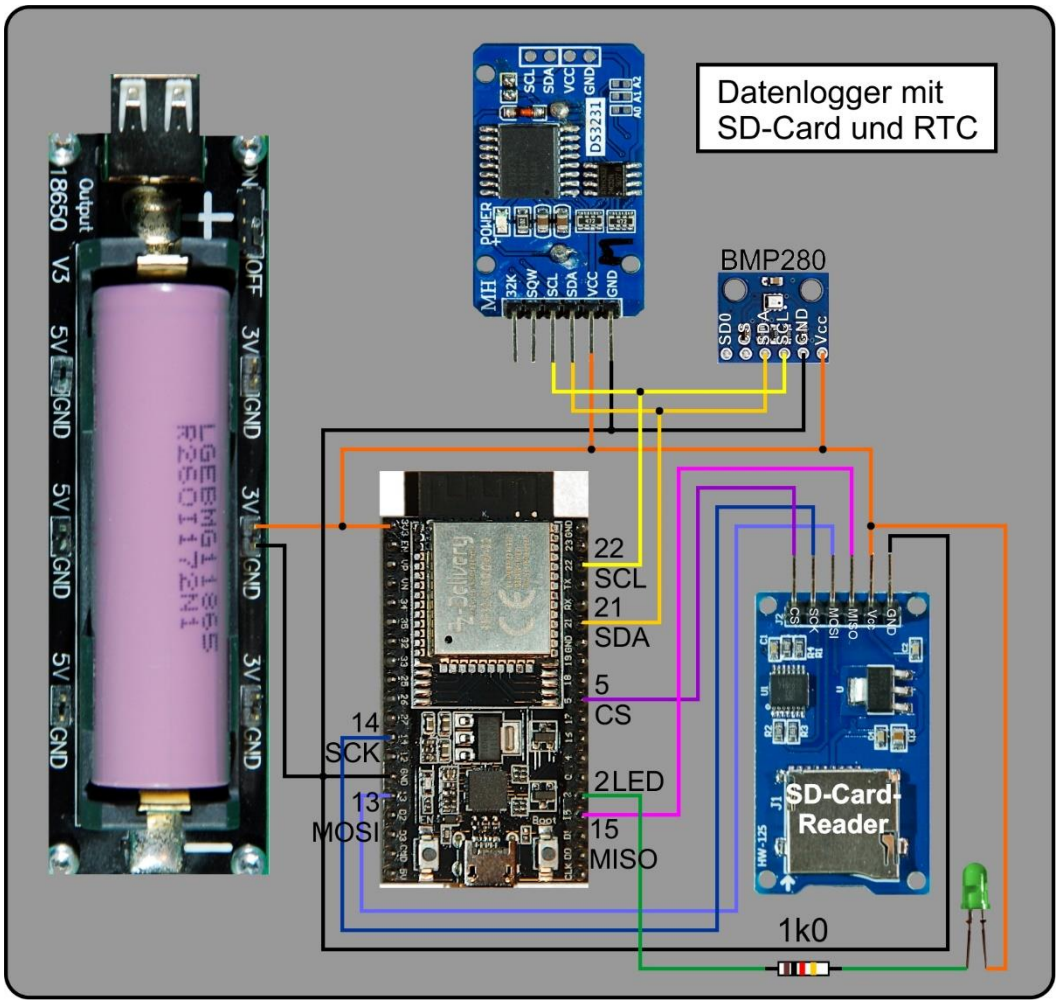

Abbildung 10: Datalogger_Schaltung mit SD-Card Reader

Der Kern des Programms bleibt derselbe. Die Vorbereitungen werden um den Import der SDCard-Klasse erweitert. Das erfordert auch erweiterte Vorbereitungen für den Aufbau des SD-Dateisystems. Bekannte Gefahrenstellen werden durch try-except-Konstrukte gefixt. Als Kontrast stelle ich die Programmvariante mit Timersteuerung vor. Deshalb wird die Timer-Klasse importiert und nach der Instanzierung des Timers 0 wird dieser auch gleich für den Dauerlauf-Modus konfiguriert und gestartet.

Die Timer-ISR (aka Interrupt Service Routine) **job()** nimmt den obligatorischen Parameter **tim**, holt mal schnell die Zeitdaten von der RTC und setzt für die weitere Verarbeitung das Flag **flag** auf 1.

In der Hauptschleife wird auf den Wert dieses Flags getestet und in schon bekannter Weise mit dem Aufbereiten der Daten für die Ausgabe reagiert. Der heiße Bereich ist wieder durch die leuchtende LED abgesichert. Zu jeder Zeit ist aber problemlos der Ausstieg durch Drücken der Flashtaste möglich. Sie muss bei heller LED nur gedrückt bleiben, bis die LED erlischt. Die while-Schleife wird sehr schnell durchlaufen, weshalb das Programm auch schnell auf die Abbruchtaste reagieren kann. Beim Beenden des Programms durch Tastendruck wird das Dateisystem der SD-Card ausgehängt, sodass man die Karte dann sicher entnehmen kann. Im

Terminalfenster von Thonny erscheint eine entsprechende Meldung. Bei laufendem Programm sollte der ESP32 nicht stromlos geschaltet werden. Die Stromaufnahme beträgt in Ruhe 45mA, während der kurzen Messphase ca. 65mA.

Die Syntax für die Speicherung auf SD-Card ist dieselbe wie beim Speichern in eine Datei im Flash des ESP32. Lediglich die Pfadangabe /sd muss ergänzt werden.

```
# datalogger.py
# Autor: J. Grzesina
# Rev.: 1.0
# Date: 26.08.2021
# ********************************************************
from machine import Pin,SPI,I2C,Timer
from bmp280 import BMP280
import os, sys, sdcard
from ds3231 import DS3231
taste=Pin(0,Pin.IN,Pin.PULL_UP)
led=(2,Pin.OUT)
# ******************* Create Filesystem *******************
spi=SPI(1,baudrate=100000,sck=Pin(14),mosi=Pin(13),\
        miso=Pin(15),polarity=0,phase=0)
sd = sdcard.SDCard(spi, Pin(5))
try:
    os.mount(sd, '/sd')
    print("SD-Card is mounted on /sd")
except OSError as e:
    print(e)
    print("SD-Card previously mounted")

# ********************** I2C+ Sensors **********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))

b=BMP280(i2c)
rtc=DS3231(i2c)
#eep=AT24C32(i2c)
d2="{:0>2}."
h3="{:0>2}:"
y5="{:>4} "
s3="{:0>2};"
p4="{:0>4}\n"
t6="{:> 5.1f};"
Y,M,D,h,m,s,flag=0,0,0,0,0,0,0

i2cDevices=i2c.scan()
print("Found I2C-Devices @:",end="")
for i in i2cDevices:
    print(hex(i),end=";")
    # 0x57: DS3231.EEPROM 32Kb
    # 0x68: DS3231.RTC
    # 0x76: BMP280
print("")
```

```
# ********************** Timer Stuff ********************
def job(tim):
    global Y,M,D,h,m,s,flag
    Y,M,D,_,h,m,s=rtc.DateTime()
    flag=1

t=Timer(0)
# Minutenabstand=10
# duration=1000*60*Minutenabstand
duration=5000
t.init(mode=Timer.PERIODIC,period= duration,callback=job)

# ********************** MAIN LOOP **********************
while 1:
    if flag==1:
        led.value(0)
        T=b.calcTemperature()
        P=int(b.calcPressureNN(465,T))
        th=t6.format(T).replace(".",",")
        dtwString=d2.format(D)+d2.format(M)+y5.format(Y)+\
                  h3.format(h)+h3.format(m)+s3.format(s)+\
                  th+p4.format(P)
        print(dtwString)
        f=open("/sd/logging.txt","a")
        f.write(dtwString)
        f.close()
        sleep(2)
        led.value(1)
        flag=0

    if taste.value()==0:
        t.deinit()
        os.umount("/sd")
        print("Mit Flashtaste abgebrochen\nSD-Card wurde
ausgehaengt und kann entnommen werden.")
        sys.exit()
```

Für längere Ruheintervalle kann das Programm natürlich auch wieder als Tiefschlafvariante datalogger_ds.py verfasst werden. Der Vorteil ist neben der Energieeinsparung (Ruhestrom 15mA) der, dass nach jedem Messeinsatz die Karte aus dem Dateisystem ausgehängt werden kann und somit der ESP32 bei dunkler LED jederzeit ausgeschaltet werden kann. Durch den Neustart wird ja alles wieder neu gebootet. Dabei wird auch die Speicherkarte wieder ins Filesystem eingehängt. Der Programmtext muss sich für den Autostart des Systems wieder in der Datei **boot**.py befinden.

4. Der ESP8266 als Datenlogger

Ja, auch der ESP8266 kann als Datenlogger eingesetzt werden. Für den I2C-Bus müssen aber die GPIO-Pins geändert werden. SCL wird GPIO5 = D1 und SDA kommt an GPIO4 = D2. Alles andere läuft genauso wie beim ESP32 bei der

Speicherung im Flash-Bereich. Hier können Sie das Timer-gesteuerte Programm für den ESP8266 herunterladen.

Und die SD-Card, geht die am ESP8266? Leider nein, für den Betrieb mit DS3231 und BMP280 zusammen - keine Chance, da fehlen 228 Bytes RAM trotz garbage collection. Dasselbe Spielchen hatte ich schon einmal vor einem Jahr, als ich versuchte, drei DS18B20 zusammen mit einer DS3231 RTC und einem TCP-Server auf einem D1 Mini zu betreiben. Damals ließ ich dann die RTC weg, weil die Zeitstempel vom Linux-Client geliefert werden konnten. Bei einem Stand-Alone-Datenlogger ist das halt nicht möglich. Geben wir uns deshalb mit dem Dateisystem im Flashbereich zufrieden, denn Messwerte ohne Zeitstempel sind für viele Fälle wenig aussagekräftig.

Gut, bliebe noch die Sache mit dem Energiesparen im Tiefschlaf beim ESP8266. Ich hatte schon angedeutet, dass das ganz anders funktioniert als beim ESP32. Beim kleinen Bruder läuft im Tiefschlaf erstens nur die interne RTC und kein extra Timer. Zweitens kann der IRQ, der den Controller aus dem Dornröschenschlaf holt, das nicht intern machen, sondern nur über den GPIO16-Pin, der mit dem RST-Pin zu verbinden ist. Der IRQ setzt den GPIO16 (=D0) auf GND-Potenzial und löst damit einen Hardware-Reset aus.

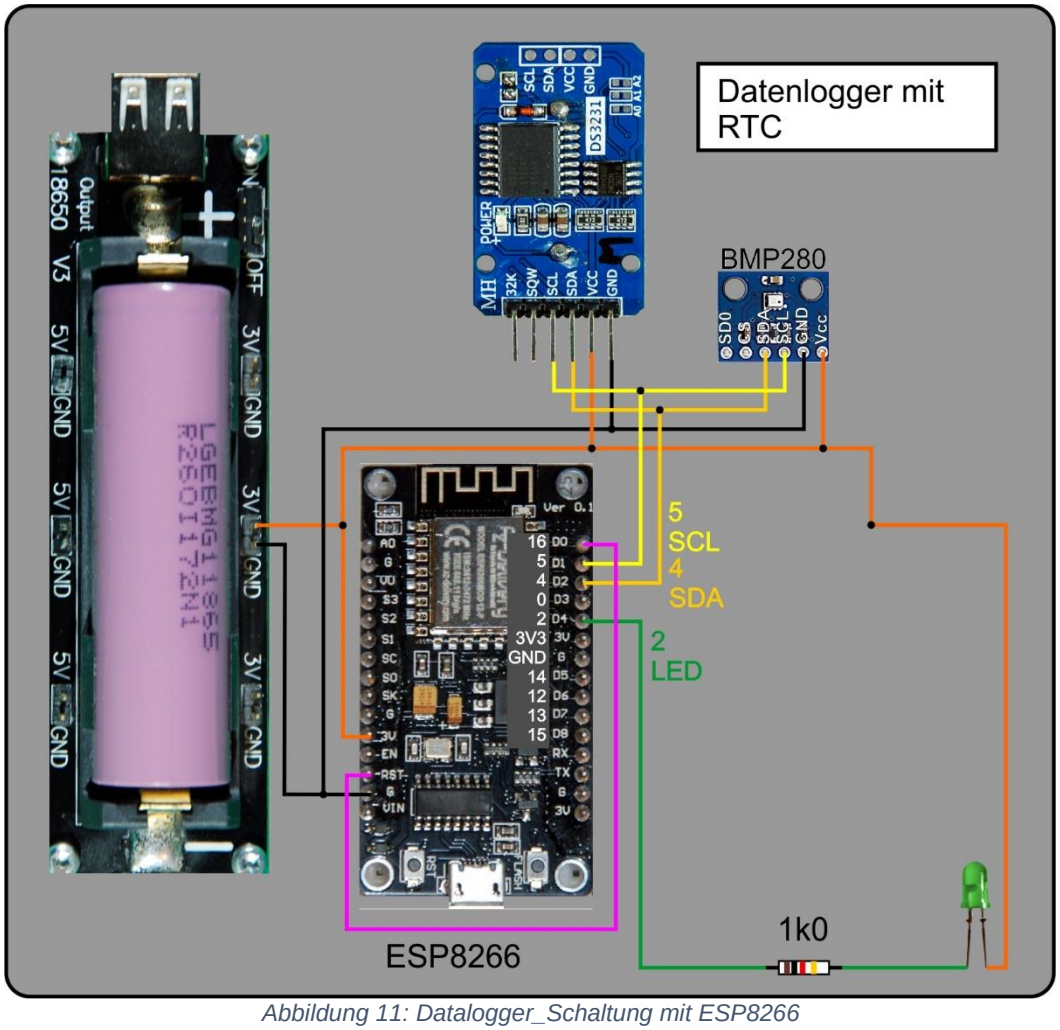

Abbildung 11: Datalogger_Schaltung mit ESP8266

Und hier ist das Programm dazu. Zum Testen läuft es mit einer Periodendauer von ca. 10 Sekunden.

```
# boot.py as datalogger8266_ds.py
# Autor: J. Grzesina
# Rev.: 1.0
# Date: 28.08.2021
# ********************************************************
import esp
esp.osdebug(None)
import os,sys
import gc               # Platz fuer Variablen schaffen
gc.collect()
from machine import Pin,I2C,Timer,RTC,DEEPSLEEP,deepsleep
from time import sleep
from bmp280 import BMP280
from ds3231 import DS3231
# 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
taste=Pin(0,Pin.IN,Pin.PULL_UP)
led=Pin(2,Pin.OUT,value=0)
sleep(2)
led.value(1)
if taste.value()==0:
    print("Mit Flashtaste beim Start abgebrochen")
    sys.exit()

# ************* On Board RTC configurieren ***************
dsl = RTC()
dsl.irq(trigger=dsl.ALARM0, wake=DEEPSLEEP)
# ******************* Filesystem Info   ******************
def df():
    s = os.statvfs('//sd')
    return ('{0} KB'.format((s[0]*s[3])/1024))
# ********************** I2C+ Sensors ********************
i2c=I2C(-1,scl=Pin(5),sda=Pin(4))
b=BMP280(i2c)
rtc=DS3231(i2c)
d2="{:0>2}."
h3="{:0>2}:"
y5="{:>4} "
s3="{:0>2};"
p4="{:0>4}\n"
t6="{:> 5.1f};"
Y,M,D,h,m,s,flag=0,0,0,0,0,0,0
i2cDevices=i2c.scan()
print("Found I2C-Devices @:",end="")
for i in i2cDevices:
    print(hex(i),end=";")
    # 0x57: DS3231.EEPROM 32Kb
```

```
    # 0x68: DS3231.RTC
    # 0x76: BMP280
print("")
# ********************* Timer Stuff *********************
Minutenabstand=10
StartUpTime=4300 #ms
duration=1000*60*Minutenabstand-StartUpTime
duration=10000 - StartUpTime
dsl.alarm(dsl.ALARM0, duration)
print("Freier Speicher:",df())
# ********************* MAIN LOOP ***********************
led.value(0)
Y,M,D,_,h,m,s=rtc.DateTime()
T=b.calcTemperature()
P=int(b.calcPressureNN(465,T))
th=t6.format(T).replace(".",",")
dtwString=d2.format(D)+d2.format(M)+y5.format(Y)+\
          h3.format(h)+h3.format(m)+s3.format(s)+\
          th+p4.format(P)
print(dtwString, end="")
f=open("logging.txt","a")
f.write(dtwString)
f.close()
flag=0
sleep(3)
led.value(1)
if taste.value()==0:
    t.deinit()
    print("Mit Flashtaste abgebrochen")
    print("Freier Speicher:",df())
    sys.exit()
deepsleep()
```

So, jetzt steht unseren Langzeit-Messreihen nichts mehr im Weg. Mit einer Lithium-Zelle mit 3600 mAh wie in den Schaltbildern, können wir über eine Woche Daten sammeln, wenn wir den Tiefschlafmodus wählen. Gehen wir von 4 Messungen pro Stunde aus, dann sind das summa summarum in 7 Tagen 672 läppische Datensätze. Und wenn MicroPython und der ESP32/ESP8266 nicht lügen, was die freie, verfügbare Speichergröße angeht, dann können wir 162 Wochen lang nach diesem Schema messen. Nur - drei Jahre hält sicher keine Akkuladung.

Viel Freude beim Ausprobieren und gute Messergebnisse wünsche ich Ihnen!