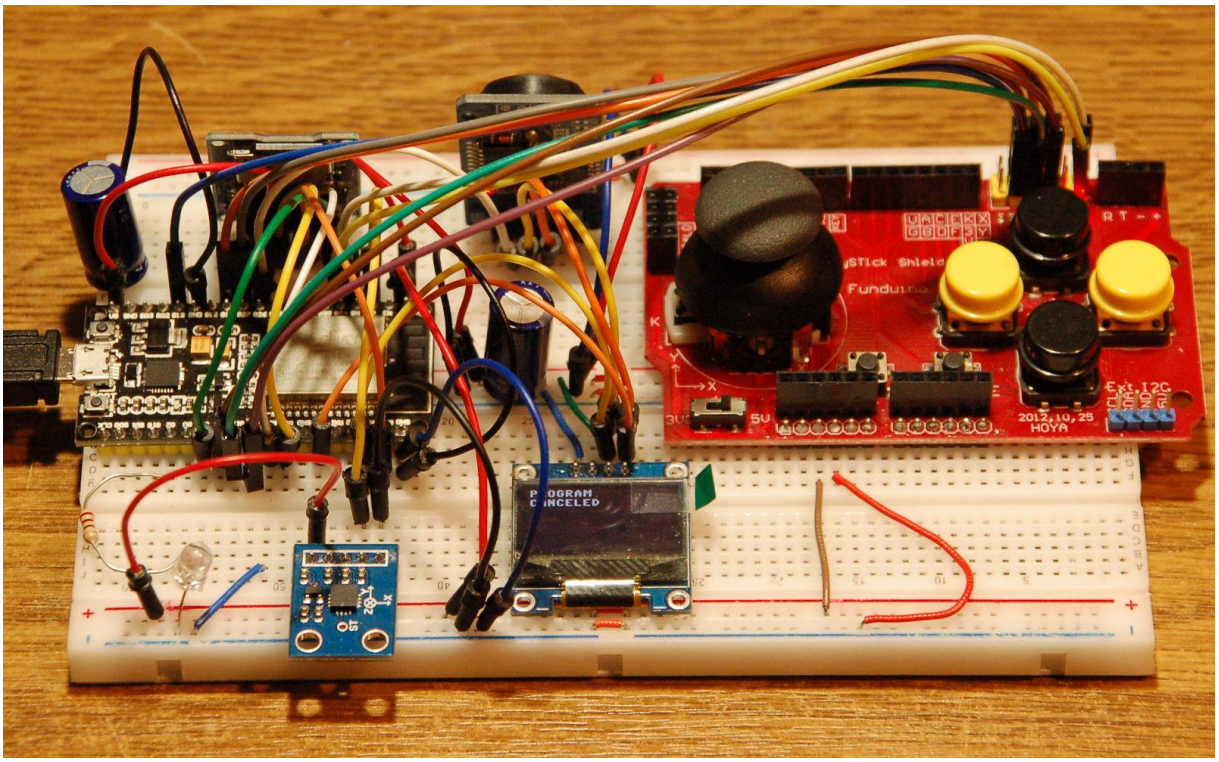

Seismometer - zweite Ausbaustufe

Diesen Beitrag gibt es auch als PDF-Dokument.

Direkteingaben im Terminal von Thonny (REPL) sind im Text **fett** formatiert und am Prompt **>>>** zu erkennen. Antworten vom ESP32 sind ***kursiv*** gesetzt.

Heute funkt es, oder besser funkt er. Der ESP32 bekommt heute ein RTC-BOB spendiert, mit dessen Hilfe er die Dateinamen bei der Erfassung von Bodenerschütterungen zeitlich dingfest machen kann. Das Board mit einem DS3231 weist eine sehr gute Ganggenauigkeit auf und wird zudem jeden Tag zu einer bestimmten Zeit, die Sie nach Ihren Wünschen festlegen können, durch Kontakt mit einem Time-Server im Web synchronisiert. Das geschieht auf der Grundlage des NTP (Network Time Protocol). Der DS3231 selbst gibt via Interrupt den Auftrag, die Synchronisation durchzuführen. Das neue Format der Dateinamen Datum_Zeit_data.csv macht die bisherige, einfache Durchnummerierung obsolet. Das alles erfordert den Umbau einiger Funktionen, bringt neue und macht eine überflüssig. Folgen Sie mir auf eine neue Tour durch

MicroPython auf dem ESP32 und ESP8266

heute

Bodenerschütterungen messen und darstellen – Teil 3 die Funkübertragung

Gleich zu Beginn will ich Ihnen die beiden Störenfriede des Projekts offenbaren. Es begann bereits mit dem Einführen des Card-Readers in der letzten Folge. Hin und wieder streikte das Mounten, also das Einhängen der Karte in den Verzeichnisbaum des Dateisystems, meistens funktionierte es und die Karte war ansprechbar, manchmal aber eben nicht. Da kam mir dann wieder die Belegung der GPIO-Pins beim Booten in den Sinn.

- GPIO 0 (LOW zum Booten)
- GPIO 2 (offen oder LOW zum Booten)
- GPIO 5 (HIGH zum Booten)
- GPIO 12 (LOW zum Booten)
- GPIO 15 (HIGH zum Booten)

Tatsächlich hatte ich die CS-Leitung an GPIO5 liegen, so wie ich es in einer früheren Schaltung gemacht hatte. Nach dem Umzug auf GPIO4 klappte es zuverlässig, die Karte einzuhängen, bis – ja, bis zum Beginn der zweiten Erweiterung.

Ich hatte alle Programmbausteine aus meiner Sammlung zusammengetragen und angepasst. Das Einhängen der SD-Karte wurde gemeldet, die anderen Objekte wurde ohne Fehler deklariert, doch als die gesamten Vorbereitungen durchgelaufen waren und dich die Karte ansprechen wollte, war sie verschwunden. Eine erneute Verlegung der CS-Leitung half nichts. Ich tauschte den Card-Reader, die Speicherkarte – nada! Also holte ich mir das e-Book zum Modul. Leider wieder kein Hinweis.

Da hilft nur strategisches Vorgehen. Zum Debuggen gibt es ein prima Hilfsmittel: **sys.exit**(). Mit dieser Funktion kann man den Programmlauf an genau definierten Stellen abbrechen. Wir haben das ja schon etliche Male praktiziert. Ich überlasse dem Programm die Aufgabe, Objekt einzurichten und Variablen zu initiieren. Alles was hier über drei Zeilen hinausgeht ist mir händisch zu umständlich. Nachdem das Programm gestoppt hat, kann ich meine Tests und Abfragen durchführen. Das brachte mich hier bis an die Stelle an der **connect**() aufgerufen wird, um zum WLAN-Router zu verbinden. Bis hierher war die SD-Karte ansprechbar, nach dem Aufruf nicht mehr. **connect**() war also der Ganove, der mir die Karte geklaut hatte. Und zwar verschwand die Karte in dem Moment, als sich der ESP32 mit

```
nic.connect(mySSID, myPass)
```

beim Router anzumelden versuchte. Klick, machte es bei mir, das ist die Ursache. Klar! Die Vcc-Leitung des Card-Raders hatte ich an den 3,3V-Pin des ESP32 gelegt. Für den Normalbetrieb reichte das auch aus. Aber sobald der Controller zu funken begann, fiel durch den erhöhten Strombedarf des ESP32 die Spannung so weit ab, dass der Kartenleser einen Neustart machte und die Karte somit nicht mehr gemountet war. Seit dem liegt Vcc am **Vin**-Pin mit 5V und der Card-Reader schnurrt wie eine zufriedene Mietzekatze. Ich hoffe, Ihnen, mit solchen Geschichten zur Fehlersuche und Beseitigung, immer wieder ein paar Tipps zum Vorgehen in eigenen Problemlagen geben zu können.

Hardware

Die Teileliste umfasst auch die Hardware von der vorangegangenen Folge. Ergänzt habe ich nur das RTC-Modul. Letzteres ist am I2C-Bus anzuschließen und verträgt sich gut mit dem Display, weil alle drei andere Hardwareadressen (HWADR) haben. Ja, Sie haben schon richtig gelesen, und ich kann auch richtig zählen, drei. Auf dem Board befindet sich neben dem DS3231 nämlich noch ein 4-MB-Flash-EEPROM. Im MicroPython-Modul ds3231.py wohnt neben der Klasse DS3231 noch deren Nachbar AT24C32. Mit den Methoden aus dieser Klasse können Sie Integerzahlen als Word (=2 Byte), strings und Blobs (=Bytesfolgen und Bytearrays) speichern und abrufen.

Der Anschluss SQW des DS3231 ist als IRQ-Pin zu gebrauchen. Ich verbinde diesen Ausgang mit dem GPIO26 und kann dadurch Programmunterbrechungen beim ESP32 auslösen.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	GY-61 ADXL335 Beschleunigungssensor 3-Axis Neigungswinkel 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
1	Mehrgang rotary Potentiometer mit Schutzwiderstand 3590S 10K Ohm
1	PS2 Joystick Shield Game Pad Keypad V2.0
1	SPI Reader Micro Speicher SD TF Karte Memory Card Shield Modul
1	Micro-SD-Card, 4GB – 32GB
1	Real Time Clock RTC DS3231 I2C Echtzeituhr
1	LED
1	Widerstand 270 Ohm
diverse	Jumperkabel
	Digital-Voltmeter (DVM) für die Kalibrierung

Die Anordnung der Teile zeigt Abbildung 1. Der Card-Reader steckt links oben, daneben der DS3231.

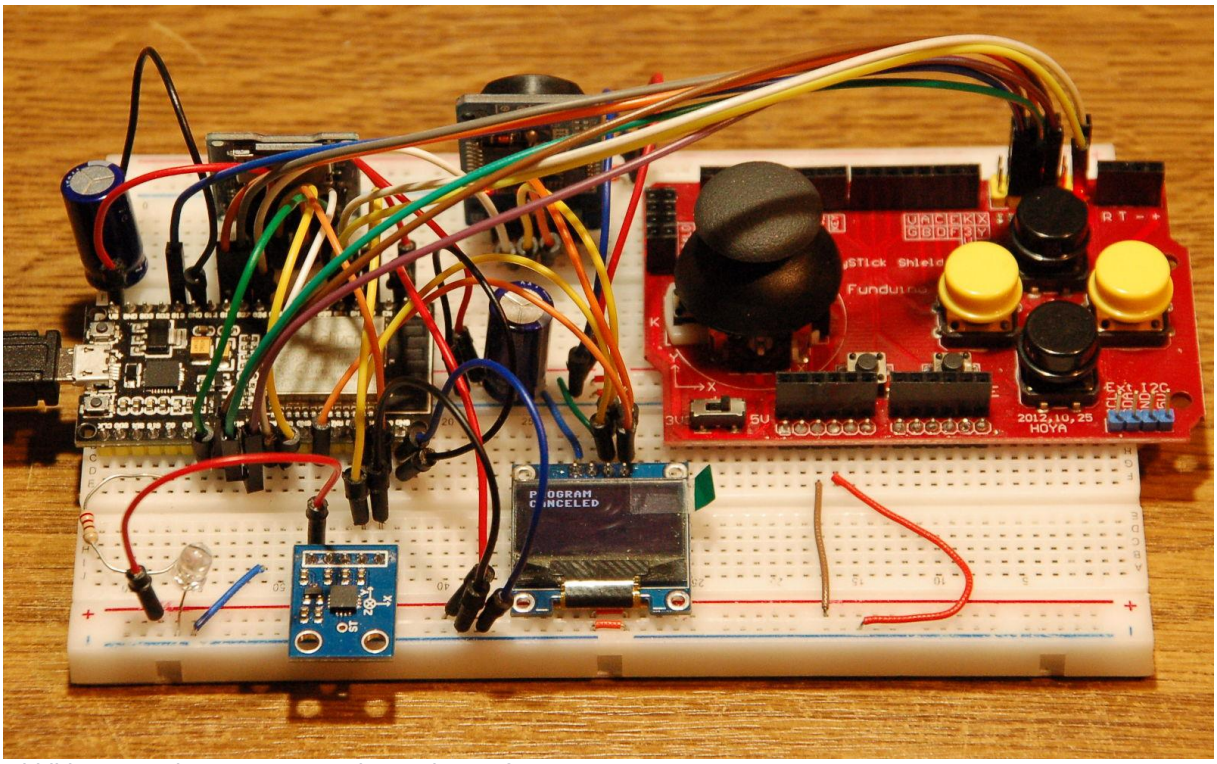

Abbildung 1:Seismometer - zweite Ausbaustufe

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
Python 3.8 oder höher incl. Idle IDE für Python auf dem PC

Verwendete Firmware für den ESP32:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für OLED-Displays
sdcard.py Treiber für das SD-Reader-Modul
buttons.py API für den Betrieb von Tasten
earthquake_sd.py Betriebsprogramm

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 05.02.2022) 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.

Die Schaltung

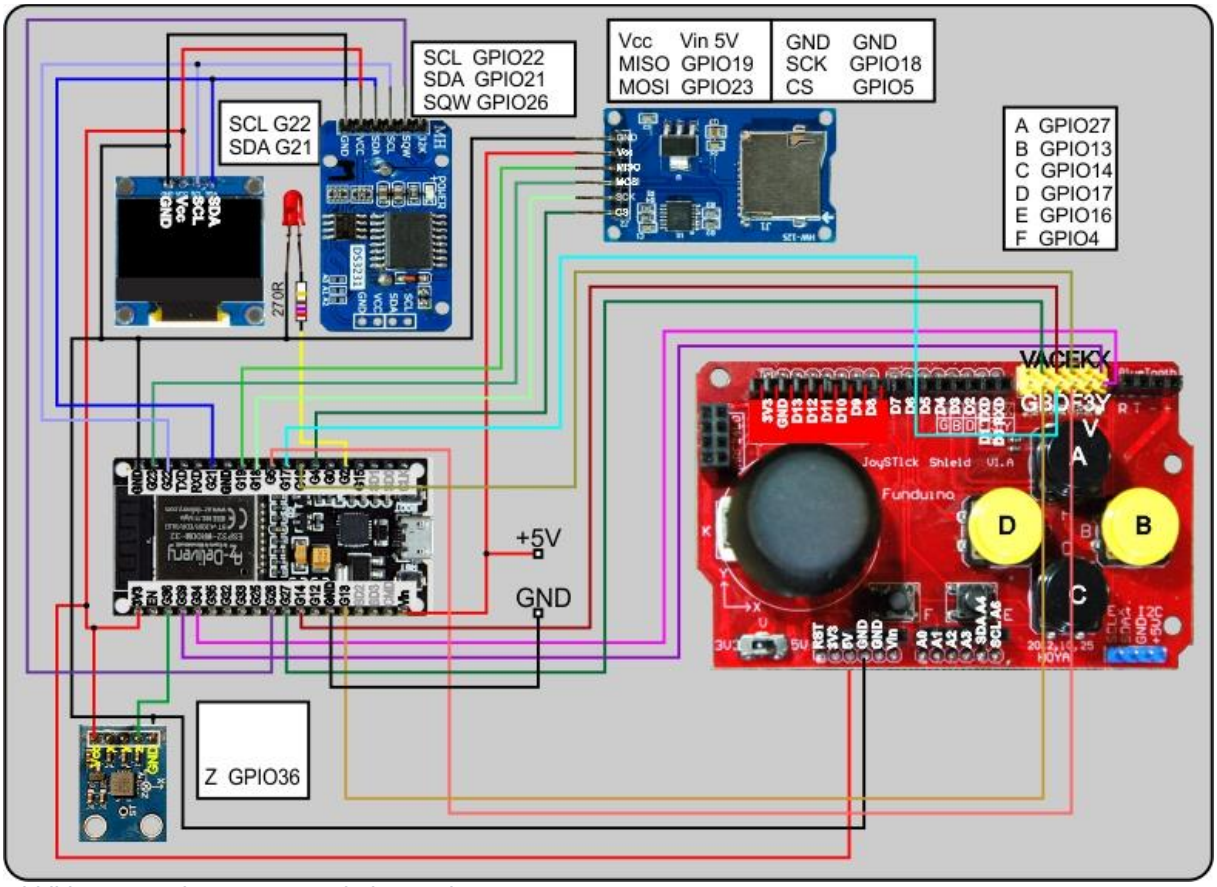

Abbildung 2: Seismometer - Schaltung mit RTC

Auf dem I2C-Bus findet ein Scan in dieser Schaltung drei Hardware-Adressen.

```
>>> from machine import Pin, SoftI2C
>>> i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=400000)
>>> i2c.scan()
[60, 87, 104]
```

Die 60 = 0x3C gehört dem Display, die 87= 0x57 dem EEPROM und der RTC-Chip wohnt auf Hausnummer 104=0x68.

Programmänderungen

Um den Post im üblichen Rahmen zu halten werde ich hier nur die Teile des Programms besprechen, die geändert wurden oder neu hinzugekommen sind. Die beiden bisher erschienen Beiträge, auf denen der aktuelle aufbaut, finden Sie hier (Teil1) und hier (Teil2).

Die erweiterte Infrastruktur macht natürlich Ergänzungen und Änderungen an der Software nötig. Das beginnt beim Importgeschäft mit den fett formatierten Zeilen. Wir haben ein paar zeitliche Klimmzüge vor, **localtime**, **ds3231** und **ntptime** helfen uns dabei. Für die Netzwerkverbindung sorgen **network** und **socket**.

```
from machine import Pin, ADC, SoftI2C, freq, SPI
from time import sleep,ticks_ms, localtime
import os, sys, sdcard
from oled import OLED
from esp32 import NVS
from buttons import *
import network, socket
import ntptime as ntp
from ds3231 import DS3231
```

Damit der Router den ESP32 reinlässt, geben Sie hier bitte Ihre eigenen Credentials ein. In **receiver** steht an erster Stelle die IP-Adresse des Geräts, mit dem Sie die Daten vom ESP32 empfangen wollen. Bei mir ist das ein Windows-Rechner, auf dem Python 3.8 läuft. Die IDE Idle ist darauf automatisch mit eingerichtet. Das brauche ich später zum Editieren des UDP-Empfangsprogramms. An der zweiten Position des Tupels steht eine frei wählbare Portnummer, die auch im Empfangsprogramm angegeben werden muss.

```
receiver=("10.0.1.10",9091)
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySSID = 'EMPIRE_OF_ANTS'
myPass = 'nightingale'
```

Der DS3231 ist darauf getrimmt, eine Zeitsynchronisation anzufordern, wenn sein Alarm-Timer 2 abgelaufen ist. Das geschieht über dessen Ausgangspin **SQW**, das in Ruhezustand auf HIGH-Pegel liegt und das dann nach LOW wechselt. Damit der ESP32 darauf reagieren kann, muss einer der GPIO-Pins als interruptfähiger Eingang programmiert werden und bei fallender Flanke eine Interrupt-Service-Routine (ISR) aktivieren. Der print-Befehl in **isr**() kann auch wegfallen. Er sagt mir in der Entwicklungsphase nur, dass die Routine ausgeführt wurde. Die Methode **ClearAlarm**() aus der Klasse DS3231 setzt das IRQ-Bit für den Alarm 2 zurück, damit geht **SQW** wieder auf HIGH. **synchronize**() kontaktiert dann den NTP-Server **pool.ntp.org** und holt die Anzahl der Sekunden seit Anfang der Epoche, dem 01.01.2000 0:0:0. Das war ein Samstag und der erste Tag im Jahr. Die folgende Anweisung zeigt das.

>>> localtime(0)
(2000, 1, 1, 0, 0, 0, 5, 1)

```
def isr(pin):
    print("Pin:",pin)
    ds.ClearAlarm(2)
    synchronize()

ds=DS3231(i2c)
rtcirq=Pin(26,Pin.IN)
rtcirq.irq(handler=isr,trigger=Pin.IRQ_FALLING)
```

Dem Konstruktor der Klasse DS3231 übergebe ich das I2C-Objekt. GPIO26 wird als Eingang geschaltet, damit der IRQ durch den Pegel auf der SQW-Leitung reagieren

kann. Würde ich den Pin als Ausgang schalten, dann würde der IRQ durch einen Pegelwechsel des Ausgangspuffers ausgelöst. Die letzte Zeile macht die Unterbrechungsanforderung scharf. In **handler** übergebe ich die Referenz auf die ISR.

Die Zeilen 33 bis 173 in **earthquake_rtc+wlan.py** wurden nicht geändert, ich überspringe sie daher.

Die nächste Änderung erfolgte in der Routine **readData**() in Bezug auf die neuen Dateinamen. Die Funktion liest eine Datei ein und füllt damit die global definierte Liste **s**. In der neuen Dateistruktur wird auch der jeweilige Ruhewert vom ADXL335 als erster Eintrag mit abgelegt. Damit er nach dem Einlesen auch global verfügbar ist, sind **s** und **S0** global deklariert. Nach dem Leeren der Liste s setze ich den Pfad zur Datei zusammen, deren Name an den Parameter n übergeben wird.

```
def readData(n):
    global s,s0
    s=[]
    name="/sd/"+n
    try:
        with open(name,"r") as f:
            line=f.readline()
            zeile=line.strip()
            s0=int(zeile)
            for line in f:
                zeile=line.strip()
                if  zeile.find(";") != -1:
                    nr,s1=zeile.split(";")
                    val=int(s1)
                    s.append(val)
    except OSError as e:
        print("Datei nicht gefunden",e)
        d.writeAt("data not found",0,5)
        sleep(3)
        d.clearAll()
```

try fängt Fehler ab, die zum Beispiel eine nicht vorhandene Datei verursachen könnte. Ich öffne die Datei zum Lesen. Über das Handle **f** habe ich jetzt Zugriff auf den Inhalt. Ein Dateihandle ist von der Funktion her vergleichbar mit einem Socket im Funkverkehr. Beides kann als Schleusentor zu einem (Informations-) Kanal verstanden werden.

Die erste Zeile, wird gelesen und vom LF (= Linefeed = \n") befreit. Danach muss der String in eine Zahl umgewandelt werden, damit man damit rechnen kann. Die for-schleife holt jetzt die restlichen Zeilen aus der Datei. LF entfernen und dann prüfen wir auf einen Strichpunkt, der in der Zeile die Nummer vom Wert trennt. Wird er nicht gefunden, dann ist der Rückgabewert von **find**() -1 und die Zeile wird übersprungen. Sonst gibt find die Position des Strichpunkts zurück, die wir aber nicht verwenden, denn es gibt eine komfortablere Methode für die Trennung, **split**(). split() liefert eine Liste, die sogleich in **nr** und **s1** entpacken. **s1** bauen wir in eine Zahl um und **append**() hängt diese an die Liste s an.

Eine nicht auffindbare Datei verursacht eine **OSError-Exception**, die wir mit **except** abfangen und dann Meldung im Terminal und im Display machen.

Im gleichen Zug mit dem Ändern der Lese-Routine muss auch die Funktion zum Speichern von Werten angepasst werden. Als erstes bestimmen wir die Länge der an **v** übergebenen Liste. Das kann, wie in diesem Programm die gesamte Liste s sein, oder auch nur ein Slice, also ein Teil davon. **len**() bestimmt die Länge der Liste, also die Anzahl an Werten. Die Funktion **name**() bastelt aus dem aktuellen Timestamp den Dateinamen.

>>> name()
'2023-02-27_10-09-15_data.csv'

Den teilen wir an den Unterstrichen auf. Die ersten beiden Listenelemente enthalten Datum und Uhrzeit, das letzte Element schicken wir ins Nirwana. Die Liste entpacken wir sofort nach **datum** und **zeit**. Das Nirwana wird vertreten durch den **"_"**. Der Unterstrich ist quasi die notwendige dritte Variable

>>> name().split("_")
['2023-02-27', '10-12-07', 'data.csv']

Der Dateipfad wird zusammengesetzt und der Basiswert der aktuellen Session aus dem nichtflüchtigen Speicher geholt. Im Display und im Terminal wird der Vorgang zur Kenntnis gebracht. Der Backslash kann lange Zeilen überall dort umbrechen, wo auch ein Leerzeichen stehen darf.

```
def save2sd(v):
    aw=len(v)
    dn=name()
    datum,zeit,_=dn.split("_")
    file="/sd/"+dn
    S0=nvs.get_i32("s0")
    d.clearAll()
    d.writeAt("SAVING file",0,2)
    d.writeAt(datum,0,3)
    d.writeAt(zeit,0,4)
    print("***** Speichere {} Werte auf SDCard *****".\
          format(aw))
    with open(file,"w") as f:
        f.write("{}\n".format(S0))
        for i in range(aw):
            f.write("{};{}\n".format(i,v[i]))
    print("============= FERTIG ================")
    d.writeAt("DONE!",0,5)
    sleep(3)
    d.clearFT(0,2)
```

Wir öffnen die Datei zum schreiben und sichern erst einmal S0, LF, "\n", nicht vergessen, sonst kann man den Bandwurm beim Einlesen nicht mehr trennen. In eine Textdatei können auch nur Strings geschrieben werden. Die Umcodierung von

Zahl nach String erledigen hier der Formatstring **"{}\n"** und die String-Methode **format**(). Folgende Anweisungen sind wirkungsgleich.

>>> "{}\n".format(S0)

>>> Str(S0)+"\n"

Die **for-Schleife** schreibt die Nummer-Wert-Paare als String mit einem LF am Ende in die Datei. Vollzugsmeldung ins Terminal und Display, drei Sekunden zum Lesen, dann löschen wir im OLED-Display alles von Zeile 2, Spalte 0 bis Zeile 5, Spalte 15.

Mit Formatstrings habe ich aber weitere Möglichkeiten, das Ausgabebild zu beeinflussen. Ein paar Beispiele:

>>> a=3.141593
>>> str(a)
'3.141593'
>>> "pi = {0:.2f}".format(a)
'pi = 3.14'

>>> b=0xe43c
>>> b
58428
>>> "{0:#X}".format(b)
'0XE43C'
>> "0b{0:024b}".format(b)
'0b000000001110010000111100'

>>> c=31
>>> "0b{0:024b}".format(c)
'0b000000000000000000011111'

Im Menü ergänzen wir einen Programmpunkt zum Speichern der Werteliste auf der SD-Karte.

```
def Menu():
    while 1:
        d.clearAll(False)
        d.writeAt("MENU",0,0,False)
        d.writeAt("D Dateien",0,1,False)
        d.writeAt("A LISTE SENDEN",0,2,False)
        d.writeAt("C EXIT",0,5)
        waitForRelease(keyA,2)
        t=waitForAnyKey((keyD,keyA,keyC),5000)
        if t==0:
            waitForRelease(keyD,2)
            file=files()
        elif t==1:
            try:
                print(file)
            except:
```

```
                file=name()
            sendList(s,file)
        elif t==2:
            d.clearAll()
            return
```

Trickreich prüfe ich mit **try**, ob bereits ein Dateiname in **file** existiert. Das ist der Fall, wenn der **print**-Befehl keine Exception wirft, **except** wird dann übersprungen. Existiert noch keine Variable file, weil ich zum Beispiel vorher keine Datei via Taste D eingelesen habe, sondern direkt von einer Messung komme, dann muss ich einen Namen aus dem aktuellen Timestamp durch **name**() generieren lassen. Denn würde **file** nicht existieren, dann gibt das einen Fehler, wenn ich file als Argument beim Aufruf von **sendList**() referenziere. **file** existiert somit in jedem Fall und hat den Timestamp einer eingelesenen Liste oder den der aktuellen Messung.

Dateinamen spielen auch eine Rolle in **files**(). Auch hier stehen demnach Änderungen an. Der äußere Ablauf bleibt derselbe, Die Anzeige der Dateinamen muss wegen deren Länge wieder auf zwei Zeilen im Display aufgeteilt werden. Das passiert in der schon bekannten Weise.

```
def files():
    d.clearAll()
    dateien=os.listdir("/sd")[1:]
    n=len(dateien)
    dateien.sort()
    i=0
    while 1:
        dn=dateien[i]
        datum,zeit,_=dn.split("_")
        file="/sd/"+dn
        d.writeAt(datum,0,0)
        d.writeAt(zeit,0,1)
        x,y=joystick(5)
```

Im Bereich **E Datei löschen** habe ich noch einen Bug erkannt und beseitigt. Wenn keine Datei mehr existiert (n ist 0), weil alle gelöscht wurden, macht es auch keinen Sinn, einen Namen anzuzeigen oder gar die Datei zu laden. Also, go back to caller, zurück zum aufrufenden Programm.

Sonst war es offenbar möglich, die noch angezeigte Datei zu entfernen, wir melden also, "FILE KILLED". Der Index **i** in die Liste der Dateinamen **dateien** hat auch nach dem Löschen immer noch denselben Wert, zeigt aber jetzt nach dem Entfernen eines Eintrags (del dateien[i]) auf den nächsten. Wie geht das? Nun, wenn Sie auf ein Buch in einem Stapel zeigen und dieses herausziehen, rutscht das darüberliegende nach unten, Ihr Finger zeigt immer noch auf dieselbe Stelle, an der jetzt ein anderes Buch liegt.

```
        if getTouch(confirm): # E Datei loeschen
            os.remove(file)
            del dateien[i]
            n=len(dateien)
            if n == 0 :
                d.writeAt("NO FILES LEFT ",0,5)
                sleep(3)
                d.clearAll()
                return
            else:
                d.writeAt("FILE KILLED ",0,5)
            if n==i and i>0:
                i-=1
            else:
                return
            sleep(3)
            d.clearFT(0,5)
```

Da ist aber noch ein zweites Grenzwertproblem. Was passiert, wenn sie auf das oberste Buch im Stapel zeigen und dieses herausziehen? Dann zeigt Ihr Finger ins Leere. Der Interpreter von MicroPython meldet dann **Index out of range**.

>>> w=[1,2,3]
>>> w[2]
3
>>> w[3]
Traceback (most recent call last):
* File "<stdin>", line 1, in <module>*
IndexError: list index out of range

Wenn weiterhin auf ein Buch zeigen wollen, müssen Sie jetzt wohl oder übel Ihren Finger um eine Position absenken, i-=1 oder i = i – 1. Das geht natürlich nur, wenn nach unten noch Bücher liegen, also wenn i größer 0 ist. Falls i bereits 0 ist, haben Sie von oben kommend eben das letzte Buch entfernt. Es gibt nix mehr zu holen und auch nicht anzuzeigen, also, get back to sender, return.

Jetzt kommen lauter neue Funktionen. Beginnen wir mit dem Aufbau einer Verbindung zum WLAN-Router. Die Funktion **connect**() tötet erst einmal das Accesspoint-Interface, wenn es denn existiert. Das ist zwar beim ESP32 in der Regel nicht aktiv und daher kein Problem, verursacht dieses aber beim ESP8266 sehr häufig.

Dann wird das Station-Interface-Objekt erzeugt und aktiviert. Wir lassen uns von **config**() die MAC-Adresse flüstern und wandeln das, meistens kryptische, bytes-Objekt durch die Funktion **hexMac**() in normale Hexadezimal-Ziffern 0-9 und a bis f.

STATION MAC: fc-f5-c4-27-9-10

```
def connect():
    # ************** Zum Router verbinden ******************
    nic=network.WLAN(network.AP_IF)
    nic.active(False)

    nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt
    nic.active(True)                    # nic einschalten
    MAC = nic.config('mac')# binaere MAC-Adresse abrufen und
    myMac=hexMac(MAC)      # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
```

Wir sollten jetzt dem Interface noch etwas Zeit geben, sich zu konstituieren. Interne WLAN-Errors können sonst die Folge sein. Dann starten wir mit **nic.connect**() und den Credentials den Verbindungsversuch, wenn noch keine Connection besteht. In der while-Schleife prüfen wir den Status der Verbindung. Ist der Rückgabewert von **status**() noch nicht 1010, müssen wir noch warten, denn das bedeutet, dass uns vom DHCP-Server im Netz noch keine IP-Adresse zugewiesen wurde. In der Regel wird der Vergabeservice für IP-Adressen auf dem WLAN-Router laufen. Ohne diesen Dienst kann unser Programm nicht funktionieren, es sei denn, wir vergeben mit **ifconfig**() selbst eine IP.

nic.ifconfig(('192.168.0.4', '255.255.255.0', '192.168.0.1', '8.8.8.8'))

Während de ESP32 auf den Abschluss der Verbindung wartet, wird im Sekundenabstand im Display und im Terminal ein punkt ausgegeben.

```
    sleep(1)
    if not nic.isconnected():
      nic.connect(mySSID, myPass)
      d.writeAt("WLAN connecting",0,1)
      points="..........."
      n=1
      while nic.status() != network.STAT_GOT_IP:
        print(".",end='')
        d.writeAt(points[0:n],0,2)
        n+=1
        sleep(1)
```

Wir lassen uns Verbindungsstatus mitteilen, der natürlich an dieser Stelle nur lauten kann und bekommen IP, Netzwerkmaske und Gateway mitgeteilt: Damit wir auch händisch Versuche mit der Netzwerkverbindung starten können, lassen wir uns eine Referenz auf die WLAN-Instanz zurückgeben.

Status: STAT_GOT_IP
STA-IP: 10.0.1.220
STA-NETMASK: 255.255.255.0
STA-GATEWAY: 10.0.1.20

```
    print("\nStatus: ",connectStatus[nic.status()])
    d.clearAll()
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    print()
    d.writeAt(STAconf[0],0,0)
    d.writeAt(STAconf[1],0,1)
    d.writeAt(STAconf[2],0,2)
    return nic
```

Ein Aufruf von connect() sieht dann etwa so aus.

>>> sta=connect()
STATION MAC: fc-f5-c4-27-9-10

Status: STAT_GOT_IP
STA-IP: 10.0.1.220
STA-NETMASK: 255.255.255.0
STA-GATEWAY: 10.0.1.20

>>> sta
<WLAN>
>>> sta.status()
1010
>>> sta.ifconfig()
('10.0.1.220', '255.255.255.0', '10.0.1.20', '10.0.1.100')

sta verweist auf die Instanz nic und ist unser Informationskanal. Damit Informationen fließen können, brauchen wir noch ein Tor, das wir zum richtigen Zeitpunkt öffnen können, einen Socket. Das Protokoll meiner Wahl ist UDP (User Datagram Protokol). Es ist schnell, arbeitet ohne Handshakes beim Verbindungsaufbau allerdings auch auf niedrigem Absicherungsniveau was Datenintegrität und Datenverlust angeht. Damit kann man in unserem Anwendungsfall leben.

```
def setSocket():
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    sock.bind(('', 9003))
    print("sending on port 9003...")
    sock.settimeout(0.1)
    return sock
```

Ein socket-Objekt **sock** aus der Familie ipv4 wird für den Datagram-Transport instanziiert. SO_REUSEADDRESS lässt uns, ohne den ESP32 neu booten zu müssen, dieselbe IP-Adresse wiederverwenden. Dann binden wir die Portnummer 9003, oder eine andere Nummer größer als 1024 an die uns zugewiesene IP-Adresse.

Der durch **settimeout**() gesetzte Zeitraum von 0,1 Sekunden beendet das Lauschen der Empfangs- oder Sendschleife und verhindert so eine Blockade der Mainloop, die beides bedienen muss. Damit wir für das Senden von Daten auf unseren Socket zurückgreifen können, muss **setSocket**() das Objekt **sock** zurückgeben.

Mit der nächsten Funktion wird nun endlich gesendet, **sendList**(). Speicher aufräumen ist wichtig, denn es werden einige kB umgeschaufelt. Dann öffnen wir den Socket und lassen uns sagen, was grade abgeht. Wir holen den aktuellen Basiswert der letzten Messung, s0, aus dem NVS und bereiten daraus einen String mit führendem ";". Dieser String S0 wird an jede Zeile aus Nummer und Wert angehängt und liefert später, in der grafischen Auswertung von EXCEL, die Nulllinie. Als Erstes senden wir den Dateinamen, damit das Empfangsprogramm die Datei unter derselben Bezeichnung wie auf der SD-Karte ablegen kann. Die for-Schleife räumt bei jedem Durchlauf auf, setzt dann die Zeile zusammen und sendet sie über den Socket. Die Zeile mit dem Inhalt ende sagt dem Empfängerprogramm, dass die Datei auf dem PC geschlossen werden kann. Danach machen wir das Tor zum Kanal zu, sock.close(). Das ist in dieser Anwendung wichtig, weil sich die Dateiübertragung und der NTP-Zugriff das nic-Interface teilen müssen. Auf ein und denselben Parkplatz kann ja auch immer nur ein Auto abgestellt werden.

```
def sendList(v,datei):
    gc.collect()
    sock=setSocket()
    d.writeAt("SENDE LISTE",0,4)
    print(len(v), "Werte")
    s0=getS0()
    S0=";"+str(s0)
    sock.sendto((datei+"\r").encode(),receiver)
    for n in range(len(v)):
        gc.collect()
        line=(str(n)+";"+str(v[n])+S0+"\r").encode()
        sock.sendto(line,receiver)
    sock.sendto("ende",receiver)
    sock.close()
    d.writeAt("DONE   ",0,5)
    sleep(3)
    d.clearFT(0,4)
```

Und so wird die RTC synchonisiert. Wir versuchen mit ntp.time() einen Zugriff auf den Zeitserver, der die Weltzeit in Sekunden seit Epochenbeginn liefert. Diesen Wert müssen wir für die Ortszeit unter Berücksichtigung der Zeitzone korrigieren, timezone*3600. Localtime() macht aus den Sekunden ein Siebener-Tupel der Form

(Jahr, Monat, Monatstag, Stunden, Minuten, Sekunden, Wochentag, Tag im Jahr)

Ds.DateTime() synchronisiert damit die RTC, indem die übernommenen Werte in die Register des DS3231-Chips geschrieben werden. Hat die Verbindung nicht geklappt, passiert gar nix, wir kriegen nur eine Nachricht im Terminal, oder im Display, wenn Sie eine programmieren mögen.

```
def synchronize():
    try:
        ds.DateTime(localtime(ntp.time()+timeZone*3600))
        print("sychronized")
        with open("lastSync.txt","w") as f:
            f.write(name()[:-9]+"\n")
        return ds.DateTime()
    except:
        print("Nicht synchronisiert")
```

Zu guter Letzt, die Namensgebung für die Dateien. Das Ganze schaut ziemlich mystisch aus, ist aber bei genauerem Hingucken nur die Wiederholung einer kurzen Sequenz. Aber der Reihe nach. Wenn keine Datums-Zeit-Liste übergeben wird, holt sich **name**() selbst eine vom DS3231. Nun muss ein String daraus gebastelt werden, der stets dieselbe Länge haben muss, wegen der Sortierung der Liste von Dateinamen in **files**(). Um das zu erreichen, müssen einstellige Datums- oder Zeitwerte mit einer führenden 0 versehen werden. Genau das macht folgender Term gemäß der Vorschrift, wandle die Zahl in einen String um und setze eine "0" davor. Davon nimm ab dem vorletzten Zeichen (Index -2) alles bis zum Ende.

```
("0"+str(dt[1]))[-2:]
```

Aus dt[1] = 4 wird: "04", das vorletzte Zeichen ist 0, bis zum Ende also "04"
Aus dt[1] = 15 wird: "015", das vorletzte Zeichen ist 1, bis zum Ende also "15"

Zwischen die Datums und Zeitwerte werden "-"-Zeichen eingebaut, die Blöcke trennt jeweils ein "_". Abschließend wird noch "data.csv" angehängt.

```
def name(dt=None):
    if dt is None:
        dt=ds.DateTime()
    fn=str(dt[0])+"-"+("0"+str(dt[1]))[-2:]+"-"+ \
          ("0"+str(dt[2]))[-2:]+"_"+("0"+str(dt[3]))[-2:]
    fn=fn+"-"+("0"+str(dt[4]))[-2:]+"-"+ \
              ("0"+str(dt[5]))[-2:]+"_"
    return fn+"data.csv"
```

Nun können wir auf die Früchte unserer Arbeit zugreifen. Wir ernten ein WLAN-Verbindungs-Objekt in sta. Die in der Funktion an das Display gelieferten Verbindungsdaten können 3 Sekunden lang gelesen werden. Messreihenspeicher deklarieren, Messreihenzähler auf 0, RTC synchronisieren und den Alarm-Timer für die Synchronisation stellen, jeden Tag zur elften Stunde.

```
sta=connect()
sleep(3)

s=[]
n=0
synchronize()
ds.Alarm2(11,0,0,DS3231.StundenAlarm)
```

Dann sind wir auch schon in der Hauptschleife. Speicher putzen, Messwert holen und Abweichung vom Basiswert berechnen. Der absolute Betrag davon entscheidet gleich darüber, ob eine Sequenz aufgezeichnet werden soll.

```
while 1:
    gc.collect()
    s1=getAdc(3)
    trigger=s1-s0
    deltaS=abs(trigger)
```

Das ist der Fall, wenn nämlich **deltaS** größer ist als das Grundrauschen **dsc**.

```
    if deltaS > dsc :
        bussy.on()
        s=[]
        over=TimeOut(1000)
        while not over():
            s.append(s1)
            s1=getAdc(5)
            sleep(0.001)
            if getTouch(stop): break
        bussy.off()
        print("{}. Trigger: {}".format(n,deltaS))
        d.writeAt("{}. Trig:{}".format(n,trigger),0,2)
        d.clearFT(0,4)
        save2sd(s)
        n+=1
```

Die LED geht an, die Liste s wird geleert und der Timer für die Messzeit auf eine Sekunde gestellt.

Solange der Timer noch nicht abgelaufen ist, kommt der zuletzt eingeholte Wert vom ADXL335 als neues Element in die Liste ein neuer Wert wird geholt, kurzes Rasten und wenn die F-Taste nicht gedrückt ist, Ring frei zur nächsten Runde.

Die Schleife wird spätestens durch den Timer **over**() beendet. Wenn Sie Closure **TimeOut**() suchen sollten, die versteckt sich heute in der Klasse Buttons. Durch den Rundumschlag beim Import, kann ich darauf genau so zugreifen, als wäre sie als Funktion im Hauptprogramm deklariert.

```
from buttons import *
```

Wir erhalten Meldungen zur Messung, löschen dann die unteren beiden Zeilen im Display und schreiben die Liste in eine Datei, die in save2sd() als Dateinamen den aktuellen Zeitstempel erhält. Nun noch den Messreihenzähler erhöhen.

Wenn kein Erschütterungsmoment den Schleifendurchlauf unterbrochen hat, geht es mit gedrückter A-Taste ins Menü, oder mit der Flash-Taste zum Beenden des Programms.

Das Python-Programm auf dem PC

In der Regel verfügt der PC bereits über eine Netzwerkverbindung, via Kabel oder WLAN, darum müssen wir uns nicht kümmern. Aber eine Tor zum Kanal müssen wir öffnen, einen – richtig, Socket.

```
import socket

myPort=9091
```

Dem ESP32 hatten wir anfangs gesagt, dass de PC unter der Hausnummer 9091 zu erreichen ist.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
s.settimeout(0.1)
```

Die Socket-Instanziierung und die Einstellungen sind dieselben wie beim ESP32.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
#s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
s.settimeout(0.1)
```

In der Hauptschleife startet sofort eine zweite while-Schleife, die auf den Eingang des Dateinamens wartet. Solange der Empfangspuffer nach Ablauf des Timeouts leer ist, wird eine Exception geworfen, die mit except abgefangen wird. Es muss nichts getan werden, als weiter zu warten, also – pass. Wurden Bytes empfangen, landen sie in **rec**, die Adresse des Senders in **adr**. Das bytes-Objekt wird in einen String decodiert und davon der Zeilenvorschub abgezwickt. Der Dateiname erscheint im Terminal, dann wird die Schleife mit break abgebrochen.

```
while 1:
    while 1:
        try:
            rec,adr=s.recvfrom(150)
            name=(rec.decode()).strip()
            print(name)
            break
        except OSError:
            pass
```

Mit dem Dateinamen wird jetzt eine Datei zum Schreiben geöffnet. Wieder gibt es eine Empfangsschleife, in der, abgesichert durch try, die Messwerte vom ESP32 eingelesen werden. Jede Zeile wird decodiert und im Terminal ausgegeben. Wenn "ende" erkannt wird, kann die Datei geschlossen und die Schleife verlassen werden. Sonst landet die Zeile in der Datei.

Andere Fehler außer dem, durch **recvfrom**() bei leerem Empfangspuffer verursachten **OSError**, führen zu einem Programmabbruch.

Nach der Meldung "Waiting for data." Geht es zurück an den Anfang der Hauptschleife.

Natürlich können Sie nicht nur den ADXL335 als Sensor im Zusammenhang mit den dargestellten Programmen nutzen. Viele weitere Sensoren warten nur darauf, entdeckt und eingesetzt zu werden.