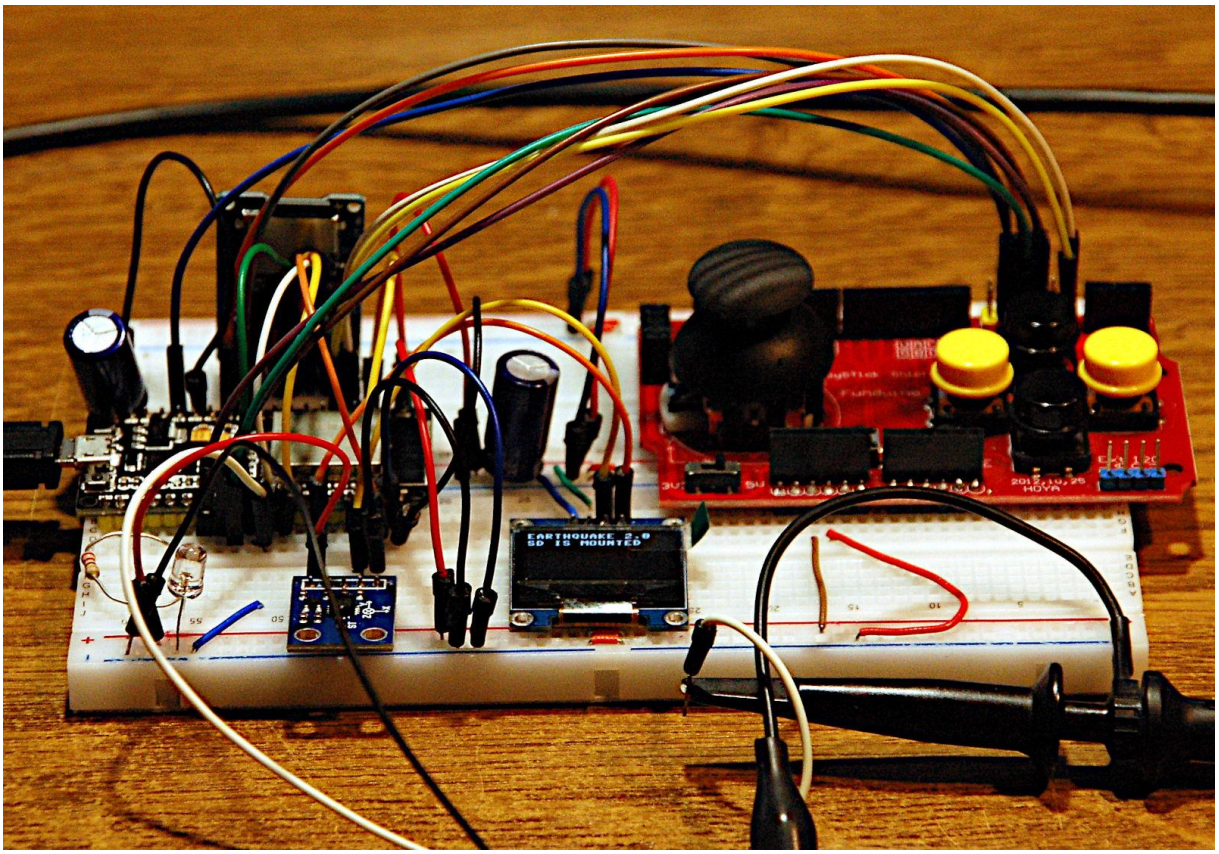

Seismometer - erste Erweiterung

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.

Es wird eng mit dem Speichern von Dateien auf dem ESP32, vor allem dann, wenn man die Aufzeichnungsdauer auf einige Sekunden ausdehnt. Mit fünf Einzelmessungen liefert die Methode **getAdc**(5) immerhin 600 Messpunkte pro Sekunde. Das könnte man noch steigern, wenn man die Pause von einer Millisekunde in der Messschleife herausnimmt. Pro 100 Messpunkte muss man mit einer Speichergröße von ca. 1KB rechnen. Wir brauchen also mehr Speicherplatz für den Dauerbetrieb. Den bekommen wir mit dem Einsatz einer SD-Speicherkarte. Bei 8GB Kapazität bringen wir gut 250000 Dateien zu je 30KB unter. Das entspricht 250000 Trigger-Events zu je fünf Sekunden Aufzeichnungsdauer. Also flugs ein SD-Karten-Modul besorgt und dazu eine micro-SD-Karte.

Für diese Aufzeichnungsdauer ist die bisherige grafische Anzeige zu träge. Außerdem fehlt die Möglichkeit im Graphen auch zurückzurudern. Für die ersten Versuche in diese Richtung hatte ich wegen der benötigten Tasten ein LCD-Keypad im Einsatz. Das hat aber nicht meine Erwartungen von Bedienungskomfort erfüllt, weswegen ich schließlich zu einem Joystickmodul gegriffen habe. Das besitzt auch 6 Tasten und mit dem Joystick ist es ein Vergnügen mit verschiedenen Geschwindigkeiten durch den Graphen zu scrollen und durch die Dateiliste zu blättern. Wie ich das alles umgesetzt habe, erfahren Sie in diesem Beitrag aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Bodenerschütterungen messen und darstellen – Teil 2

Hardware

Die Teileliste umfasst auch die Hardware von der vorangegangenen Folge. Ergänzt habe ich den SPI-Card-Reader und das Joystick-Shield. Letzteres ist eigentlich für den Arduino gedacht, funktioniert aber auch prächtig mit dem ESP32. Der kann nun gut seine Trümpfe ausspielen, denn selbst nach dem Verbinden der sechs Tasten auf dem Shield mit dem Controller bleiben immer noch freie GPIOs übrig.

Vier davon brauchen wir aber schon einmal für den SPI-Bus zum Card-Reader.

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	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.

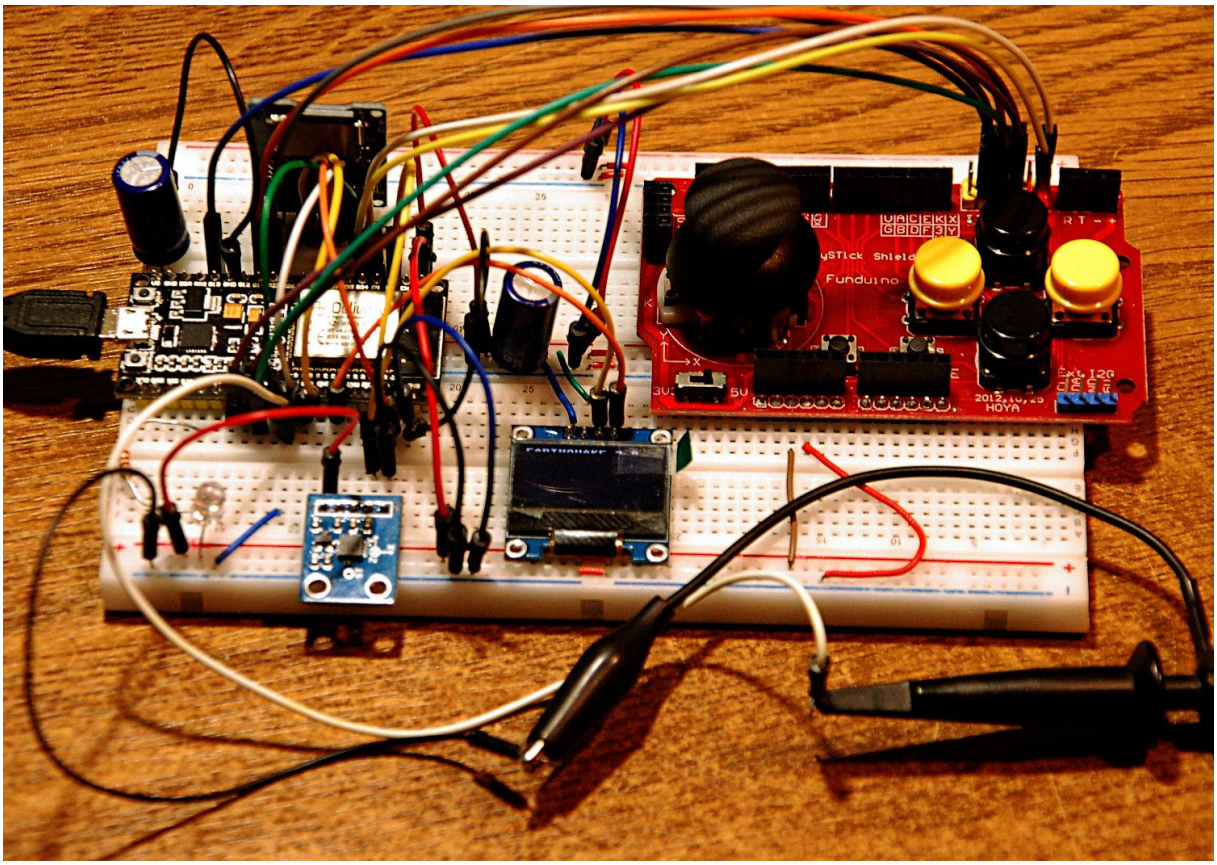

Abbildung 1: Seimograph mit SD-Karte und Joystickmodul

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

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

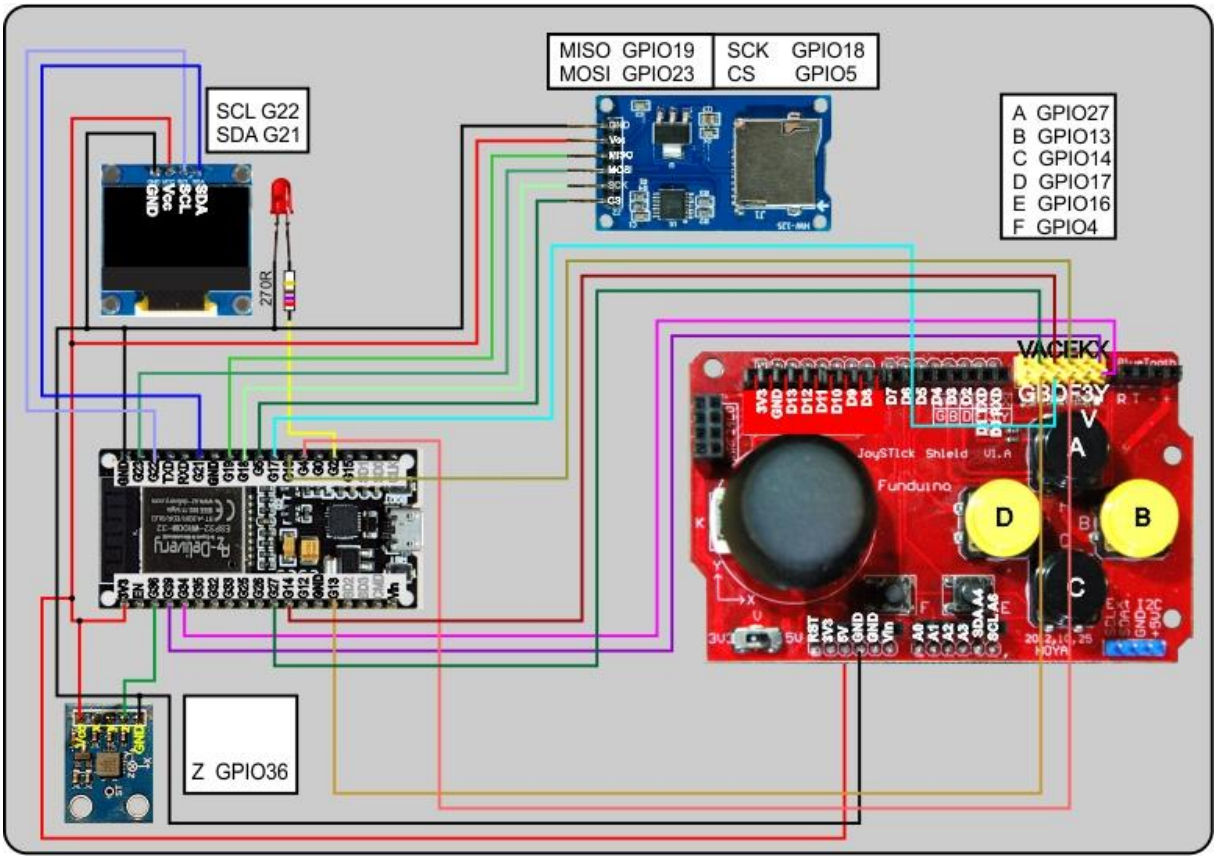

Abbildung 2: Seismometer - Erweiterte Schaltung

Ursprünglich hatte ich eine der Tasten an GPIO12 angeschlossen. Alles lief problemlos – bis zum nächsten Reset. Der ESP32 konnte keine Verbindung mit Thonny herstellen. Es stellte sich heraus, dass der ESP32, wie auch der ESP8266, gewisse Anschlüsse beim Booten auf einem bestimmten Pegel haben möchte. Folgende GPIOs sind davon betroffen.

- 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)

Durch den Pullup an der Taste A konnte der LOW-Zustand nicht erreicht werden, die Folge war der missglückte Start des Boards. Nach dem Umzug auf den Anschluss GPIO27 war das Problem behoben.

Der SPI-Bus

Mit dem SPI-Bus haben wir uns neben dem I2C-Bus eine weitere serielle Übertragungstechnik ins Boot geholt. Beide Systeme arbeiten bidirektional, und bei beiden Systemen ist der Controller der Chef. Beim I2C-Bus darf aber immer ein Partner nur zuhören, während der andere sendet. Beim SPI-Bus geht beides zur gleichen Zeit. Das liegt an der Anzahl von Leitungen, zwei sind es beim I2C-Bus, der

SPI-Bus braucht deren vier. Beiden Systemen gemeinsam ist eine Taktleitung SCL/SCK, die der Controller als Chef (Master) bedient. Beim I2C-Bus wandern die Anfragen vom Master über die SDA-Leitung zur Peripherie (Slave) und von dort kommt die Antwort auf derselben Leitung zurück zum Master. Das geht eben nicht gleichzeitig.

Beim SPI-Bus gibt es eine Sendeleitung (MOSI = Master Out Slave In) und eine Empfangsleitung (MISO = Master In Slave Out). Während der Master den nächsten Auftrag an den Slave sendet, kann dieser im Prinzip mit derselben Taktfolge auf den vorigen Auftrag antworten. Die Taktfrequenz auf SCK kann wesentlich höher sein als beim I2C-Bus, wird aber letztlich durch den oder die Slaves begrenzt.

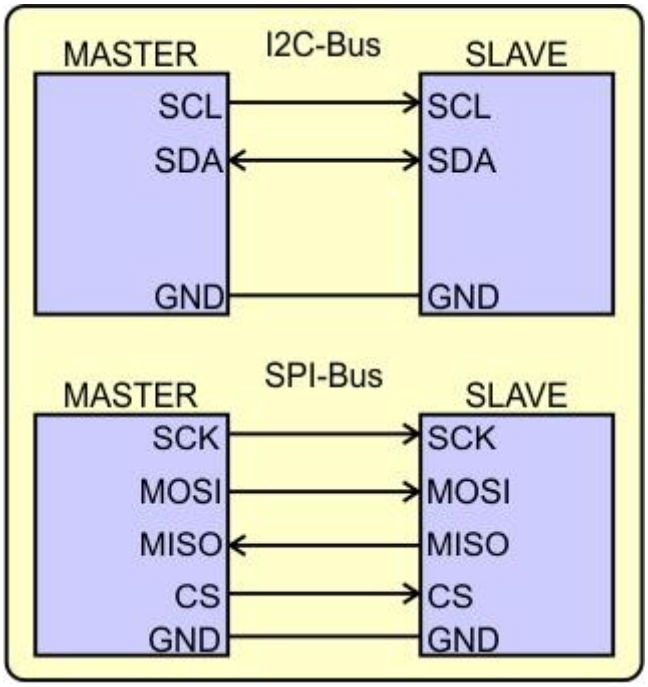

Abbildung 3: Bussysteme im Vergleich

Die Auswahl des Peripheriebausteins erfolgt beim I2C-Bus über eine sogenannte Hardwareadresse. Die ist beim SPI-Bus nicht erforderlich, der Baustein wird über eine spezielle Auswahlleitung (CS = Chip Select) angesprochen.

Für beide Bussysteme gibt es in MicroPython Klassen, I2C oder SoftI2C und SPI, die im Modul **machine** wohnen und Funktionen für die Abwicklung des Datenverkehrs bieten.

Wir testen das einmal. Mit dem Logic Analyzer zeichne ich die Signale auf den Leitungen SCK, MOSI und MISO auf. Der Card-Reader hat vorher noch keinen auswertbaren Befehl erhalten, die MISO-Leitung wird als HIGH eingelesen. Ich sende 0xA5 und lese gleichzeitig ein Byte über MISO ein, 0xFF. Befremdend mag erscheinen, dass mit einem **read**-Befehl gesendet wird, aber das liegt eben in der Natur des SPI-Transfers. Natürlich gibt es in der Klasse SPI auch einen reinen **write**-Befehl. Das erste Argument bei **read**(1,0xA5) bezeichnet die Anzahl einzulesender Bytes, das zweite Argument ist das zu sendende Byte. Daneben gibt es auch Funktionen, die **bytes**-Objekte oder **bytearrays** senden und empfangen. Eine Übersicht über die Objekte in der Klasse SPI erhalten Sie so:

>>> dir(SPI)
['__class__', '__name__', 'read', 'readinto', 'write', '__bases__', '__dict__', 'LSB', 'MSB', 'deinit', 'init', 'write_readinto']

Aber jetzt zum Test. Die Taktfrequenz von 100kHz wurde durch die Klasse SDCard vorgegeben.

>>> spi.read(1, 0xA5)
b'\xff'

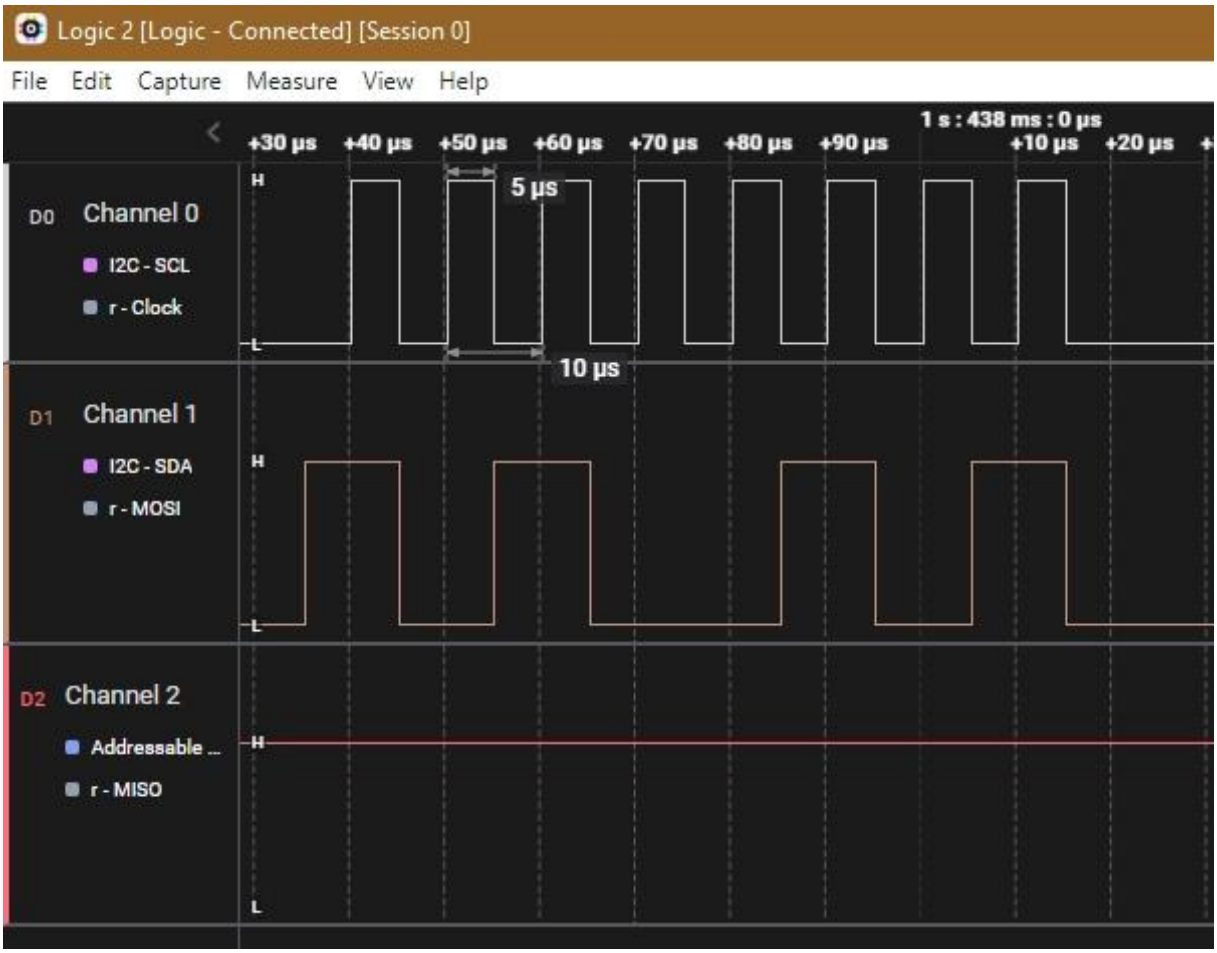

Abbildung 4: Senden 0xA5 - empfangen 0xFF

Die Parameter der SPI-Schnittstelle ruft folgender Befehl ab.

>>> spi
SPI(id=1, baudrate=100000, polarity=0, phase=0, bits=8, firstbit=0, sck=18, mosi=23, miso=19)

Das Protokoll ist in unserem Fall so eingestellt, dass der Master das Datenbit vor dem ersten Takt und bei fallender Flanke auf MOSI bereitstellt und der Slave dieses Bit mit der steigenden Flanke des Takts übernimmt. Der Controller sampelt die MISO-Leitung mit fallender Taktflanke.

Der Logic Analyzer sagte mir auch, dass bei reinen Lesebefehlen lauter 0xFF-Bytes gesendet werden. Das bedeutet, dass die MOSI-Leitung zum Beispiel beim Einlesen von der Speicherkarte permanent auf HIGH liegt. Wenden wir uns jetzt dem Programmzuwachs und den notwendigen Änderungen zu.

Das erweiterte Programm

Zur Konversation auf dem SPI-Bus brauchen wir die Klasse **SPI** aus **machine** und das externe Modul **sdcard**. Für die Behandlung von Tastenaktionen hole ich mir ferner das Modul **buttons**. Es erzeugt Tasten-Objekte und stellt eine Reihe von Funktionen zum Einlesen des Tastenzustands zur Verfügung. Der **Stern** holt alle Bezeichner aus **buttons**.py in den globalen Namensraum und erspart mir das Prefix Buttons bei jedem Aufruf einer Funktion. Das funktioniert genauso wie das selektive **from time import sleep, ticks_ms**.

```
from machine import Pin, ADC, SoftI2C, freq, SPI
from time import sleep,ticks_ms
import os, sys, sdcard
from oled import OLED
from esp32 import NVS
from buttons import *

freq(240000000)
```

Hier kommen auch gleich die sechs Tasten-Objekte, die alle LOW-aktiv sind. Ich erhaltet eine 1 bei gedrückter Taste, wenn ich **invert** auf True setze. Die 1 vereinfacht die Abfrage, weil eine 1 als True gewertet wird.

if getTouch(keyA):

Liest sich einfach wie "wenn die Taste A gedrückt ist…". Das ist intuitiver als

if keyA.value() == 0:

```
stop=Buttons(4,invert=True,pull=True,name="stop") # F stop
ende=Buttons(0,invert=True,pull=True,name="ende") # Flash
confirm=Buttons(16,invert=True,pull=True,name="confirm") # E
keyD=Buttons(17,invert=True,pull=True,name="Taste D") # Datei
keyB=Buttons(13,invert=True,pull=True,name="Taste B") # grafik
keyA=Buttons(27,invert=True,pull=True,name="Taste A") # menu
keyC=Buttons(14,invert=True,pull=True,name="Taste C") # menu
exit
```

Als SPI-Bus-Interface verwenden wir die Einheit 1 mit den Pins

GPIO18 – SCK
GPIO19 – MISO
GPIO23 – MOSI
GPIO4 - CS

```
spi=SPI(1,baudrate=100000,sck=Pin(18),mosi=Pin(23),\
        miso=Pin(19),polarity=0,phase=0)
```

Die Einrichtung der SD-Karte erfolgt in zwei Schritten. Als Erstes erzeugen wir eine SDCard-Instanz.

```
try:
    sd = sdcard.SDCard(spi, Pin(4))
except:
    print("SD-Card init failed")
    while 1:
        led.value(0)
        sleep(0.3)
        led.value(1)
        sleep(0.5)
        if getTouch(ende):
            led.value(0)
            print("Cancelled by flash key")
            print("restart with RST key")
            d.writeAt("SD init failed",0,1)
            sys.exit()
```

Kann die Speicherkarte aus irgend einem Grund nicht angesprochen werden, das prüft der Konstruktor über die Routine **SDCard.init_spi**(), dann bekommen wir eine Fehlermeldung im Terminal und eine blinkende LED. Die Programmsequenz habe ich als Baustein aus meinem Fundus übernommen, da heißt der LED-Ausgang **led**.. Indem ich für den Ausgang **bussy** das Alias **led** einführe, kann ich dieselbe LED ohne Änderungen am Text des Bausteins sowie am Programmtext einmal als bussy und ein andermal als led verwenden.

```
bussy=Pin(2,Pin.OUT,value=0)
led=bussy
```

Hat die Initialisierung der Karte geklappt, dann muss der Speicher als Ordner in den Verzeichnisbaum des ESP32 eingehängt, gemountet, werden. Wie bei der Instanziierung des Karten-Objekts, sichern wir den Vorgang mit **try – except** ab.

```
try:
    os.mount(sd, '/sd')
    print("SD-Card is mounted on /sd")
    d.writeAt("SD IS MOUNTED",0,1)
except OSError as e:
    print(e)
    print("SD-Card previously mounted")
```

Die Karte ist jetzt über das Verzeichnis /sd ansprechbar. Wir müssen nur den Verzeichnisbaum des ESP32 auffrischen, dann können wir das Verzeichnis in Thonny öffnen.

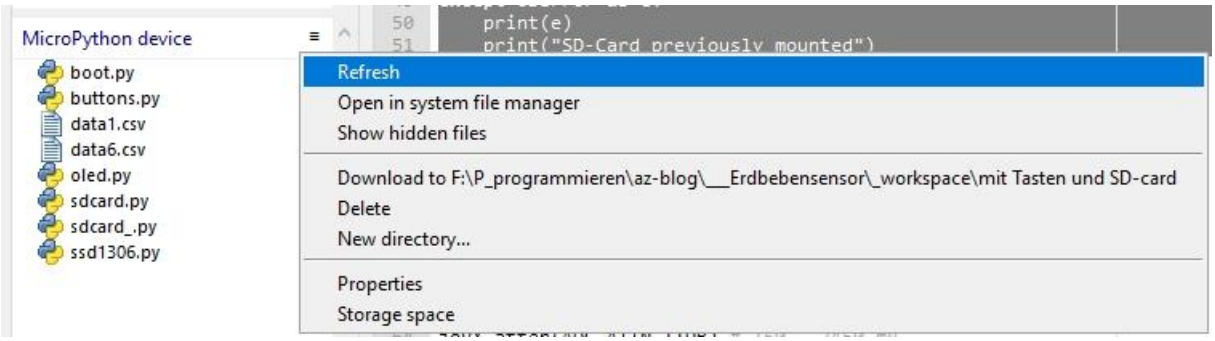

Abbildung 5: Das Verzeichnis sd in die Anzeige holen

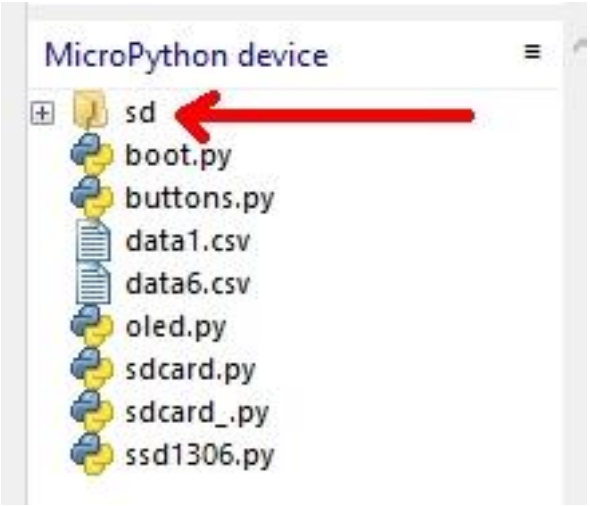

Abbildung 6: Die Karte ist gemountet

Mit Doppelklick auf **sd** sehen wir die enthaltenen Dateien, oder besser, wir sehen sie nicht, weil wir ja in **sd** noch nichts abgelegt haben.

Ein Großteil der nächsten 50 Programmzeilen wurden 1:1 aus dem Listing der Vorgängerversion dieser Ausgabe übernommen. Dort sind sie auch dokumentiert. Die neuen Bereiche sind fett formatiert. Hier geht es um die Deklaration und Initialisierung der analogen Anschlüsse für den Joystick. Dabei hat sich herausgestellt, dass eine Änderung der Auflösung bei den Analogeingängen nur bei Pin 36 möglich ist. Die anderen Analoganschlüsse arbeiten, auch wenn zum Beispiel WIDTH_9BIT, wie im Listing, eingestellt wird, trotzdem mit 12 Bit. Wenn der Knüppel auf dem rechten Anschlag liegt, erhalten wir statt 511 4095:

>>> joyX.read()
4095

```
sleep(3)
nvs=NVS("Quake")

adcPinNumber=36
adc=ADC(Pin(adcPinNumber))
joyXnumber=39
joyYnumber=34
joyX=ADC(Pin(joyXnumber))
joyY=ADC(Pin(joyYnumber))
```

```
joyX.width(ADC.WIDTH_9BIT)
joyX.atten(ADC.ATTN_11DB) # 150 - 2450 mV

joyY.width(ADC.WIDTH_9BIT)
joyY.atten(ADC.ATTN_11DB) # 150 - 2450 mV

adc.atten(ADC.ATTN_11DB) # 150 - 2450 mV
adc.width(ADC.WIDTH_12BIT)
# 0...4095; LSB = 3149mV/4095cnt=0,769mV/cnt
# @313mV/g  (313mV/g)/0,769mV/cnt = 408cnt/g
# LSBg = 1/408cnt/g = 2,45mg/cnt
lsbC=3149/4095 # mV / cnt
lsbG=lsbC/313  # g / mV

s0=0
su=4095
so=0
n=1000
m=[]
for i in range(n):
    s=adc.read()
    m.append(s)
    s0+=s
    su=min(su,s)
    so=max(so,s)
s0=int(s0/n)
nvs.set_i32("s0",s0)
nvs.commit()
dsu=int(s0-su+1)
dso=int(so-s0+1)
dsc=max(dsu,dso)*2
ds = dsc*lsbC*lsbG
print ("s0= {}; su={}; so= {}; dsu={}; dso=
{}".format(s0,su,so,dsu,dso))
print ("Rauschen: {} cnts= {:0.2f} g".format(dsc,ds))

sx,sy=0,0 # Nullstellung des Joysticks bestimmen und merken
for i in range(50):
    sx+=joyX.read()
    sy+=joyY.read()
nvs.set_i32("joyx",int(sx/50))
nvs.set_i32("joyy",int(sy/50))
nvs.commit()
```

Die Werte, die sich aus der Ruhestellung des Joysticks ergeben, legen wir, wie den s0-Wert, auch im nichflüchtigen Speicher ab.

getAdc() und **getS0**() wurden ebenfalls unverändert übernommen.

```
def getAdc(n):
    s=0
    for i in range(n):
        s+=adc.read()
    return int(s/n)
```

```
def getS0():
    return nvs.get_i32("s0")
```

Ganz neu ist die Joystickabfrage. Der Joystick enthält zwei Potentiometer, deren Schleifer durch Neigen des Knüppels verstellt wird. Die Enden der Potis liegen an +3,3V und GND, sodass wir Spannungen aus diesem Bereich an den Kontakten x und y des Joystick-Shields erhalten. Den x-Anschluss habe ich an GPIO39 geführt, y liegt an GPIO34.

Die Funktion für die Joystickabfrage arbeitet, ähnlich wie die Abfrage des Accelerometers, zunächst mit Mittelwertbildung. Allerdings wird nicht der Absolutwert der ADC-Counts weitergegeben, sondern der Relativwert zur Mittelstellung, reduziert mit dem Teiler 100. Es ergeben sich somit Werte zwischen -20 und +20.

```
def joystick(n):
    sx,sy=0,0
    for i in range(n):
        sx+=joyX.read()
        sy+=joyY.read()
    x,y=int(sx/n),int(sy/n)
    x=int((x-joyX0)/100)
    y=int((y-joyY0)/100)
    return x,y
```

Geändert hat sich die Routine zum Einlesen von Dateien, aber nur an der **Stelle**, wo es um den Aufbau des Dateinamens geht. Der muss natürlich jetzt das Prefix ***/sd/*** erhalten.

```
def readData(n):
    global s
    s=[]
    name="/sd/data"+str(n)+".csv"
    try:
        with open(name,"r") as f:
            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".format(n),0,5)
        sleep(3)
        d.clearAll()
```

Einen größeren Umbau hat die Funktion **grafik**() erhalten. Lief vorher der Graph einer Aufzeichnung einfach durch, so wird der Vewport jetzt mittels Joystick quasi über dem Datensatz eines Events hin- und hergeschoben. Betrachtet wird stets die Liste s, in welche der Inhalt einer Datei vorher eingelesen werden muss.

```
def grafik(v):
    d.clearAll()
    ml=dheight//2+1
    laenge=len(v)
    print(laenge)
    so=max(v)
    su=min(v)
    s0=getS0()
    ds=max(so-s0,s0-su)
    yFaktor=(dheight/(ds*2))
    i=0
    while 1:
        x,y=joystick(5)
        i=min(i+x,laenge - 64)
        i=max(1,i)
        d.clearAll(False)
        d.hline(0,ml,127,1)
        d.writeAt(str(i),16-len(str(i)),0,False)
        x1=0
        y1=ml-int((s[i]-s0)*yFaktor)
        bis=63 if i+64 < laenge else laenge - i
        for j in range(i,i+bis):
            x2=(j-i+1)*2
            y2=ml-int((v[j]-s0)*yFaktor)
            d.line(x1,y1,x2,y2,1)
            x1=x2
            y1=y2
        d.show()
        if getTouch(stop): break
```

Bis zur while-Schleife, die die äußere for-Schleife der Vorgängerversion ersetzt, blieben die Zeilen unverändert.

Wir starten mit dem Einlesen der Joystickposition durch fünf Einzelmessungen. Der Index **i** in die Liste **s** darf höchstens bis zur 64. Position vor dem Listenende laufen. Dass die Summe aus der gegenwärtigen Position und der Joystickstellung x diese Marke nicht überschreiten kann, garantiert die Funktion **min**(). **max**() stellt sicher, dass die 1 nicht unterschritten wird, wenn ein negativer x-Wert den Index verringert.

Wir löschen die Anzeige verdeckt (False) und zeichnen die Mittenlinie. Die Funktion **hline**() erbt **OLED** über **SSD1306_I2C** von der Klasse **framebuf.FrameBuffer**. Die Funktionen von FrameBuffer schreiben fast alle nur in den Puffer. Die Funktion **show**(), die durch die Vererbung in den Namensraum von OLED geholt wurde, sendet die Pufferdaten dann erst ans Display. Auch den Index i schreiben wir rechtsbündig und verdeckt in die rechte obere Ecke des Displays.

Der ADC-Wert an der Position i wird in eine y-Koordinate umgerechnet, Differenz zum Ruhewert S0 mal y-Skalenfaktor. Den ganzzahligen Anteil davon subtrahieren wir vom y-Wert der Mittenlinie. Subtraktion deshalb, weil die y-Achse des Displays von oben nach unten zeigt. Positive Werte kommen damit in die obere Hälfte, negative in die untere.

Damit der Index beim Zeichnen nicht über das Ende der Liste s hinausrennt, muss der Summand **bis**, mit dem wir den Bereich der x-Werte festlegen, ebenfalls begrenzt werden. Wir berechnen die Koordinaten des nächsten Pixels, ziehen eine Linie von der ersten zur zweiten Position und übernehmen die Koordinaten des zweiten Punkts in den ersten.

Nach dem Verlassen der for-Schleife ist der Puffer gefüllt, wir schieben die Daten mit **show**() ins Display.

Durch die Deklaration der Funktion **save2sd**() wird die Sicherung der Daten aus der Hauptschleife ausgelagert. Das erhöht die Lesbarkeit der Hauptschleifenstruktur und verringert die Durchlaufzeit. In **v** übergeben wir die Liste **s** (oder einen Teil davon) und in **n** die Nummer der zu bildenden Datei.

```
def save2sd(v,n):
    aw=len(v)
    name="/sd/data"+str(n)+".csv"
    print("***** Speichere {} Werte auf SDCard *****".\
          format(aw))
    d.writeAt("SAVING data{}".format(n),0,4)
    with open(name,"w") as f:
        for i in range(aw):
            f.write("{};{}\n".format(i,v[i]))
    print("============= FERTIG ================")
    d.writeAt("DONE!",0,5)
    sleep(3)
    d.clearFT(0,4)
```

Die Länge der Liste wird bestimmt und der Dateiname zusammengebaut. Im Terminal und Display werden wir über den Stand der Dinge informiert. Mit **with** öffnen wir die Datei zum Schreiben. Die **for**-Schleife schickt die Nummer-Wert-Paare auf die SD-Karte. Vor dem Verlassen der Funktion löschen wir den Bereich der Mitteilungen im Display, Zeile 4, Spalte 0 bis Displayende Zeile 5, Spalte 15.

Die sechs Tasten erlauben zwar bereits die Auslösung verschiedener Aktionen, aber mit einer Ausdehnung auf weitere Features würden wir sicher an Grenzen stoßen. Abhilfe schafft ein Menü. Dadurch können wir den Tasten mehrere Funktionen zuweisen, programmtechnisch mit einem OOP-Begriff gesprochen, wir können sie überladen.

```
def Menu():
    while 1:
        d.clearAll(False)
        d.writeAt("MENU",0,0,False)
        d.writeAt("D Dateien",0,1,False)
        d.writeAt("C EXIT",0,5)
        t=waitForAnyKey((keyD,keyC),5000)
        if t==0:
            if waitForRelease(keyD,2):
                files()
        elif t==1:
            d.clearAll()
            return
```

In der while-Schleife geben wir Überschrift, Tastenbezeichnung und Bedeutung aus. Die Funktion **waitForAnyKey**() aus dem Modul **buttons** wartet 5 Sekunden lang auf die Betätigung einer der Tasten, deren Instanzen als Elemente des Tupels an den Positionsparameter **tasten** übergeben wurden. Wird 5000ms lang keine Taste gedrückt, kommt der Wert **None** zurück, sonst die Positionsnummer des Tasten-Objekts im Tupel.

Wir prüfen auf diese Nummern. Weil im Fall der Taste D die Funktion **files**() aufgerufen wird und dort dieselbe Taste eine weitere Aktion auslöst, warten wir kurz darauf, dass die Taste losgelassen wird. Kommt **True** zurück, wird **files**() aufgerufen, sonst passiert gar nix.

Mit C steigen wir aus dem Menü aus. Das Menü kann leicht erweitert werden, indem neue Menüpunkte an die Reihe der Displayausgaben angehängt werden und die if – elif – Struktur durch weitere Vergleiche ergänzt wird.

Die Funktion **files**() behandelt Aktionen rund um das Thema Dateien. Nach dem Löschen des Displays holen wir die Liste der in **/sd** vorhandenen Dateien, bestimmen die Anzahl und sortieren die Liste. Der Laufindex i wird mit 0 initiiert.

In der **while**-Schleife geben wir den ersten Dateinamen aus und fragen den Joystick ab. In y-Richtung bewegt, blättern wir uns durch die Dateinamenliste nach oben aufsteigend nach unten absteigend. Mit **min**() und **max**() erzwingen wir nur gültige Indexwerte.

Die Taste D schaufelt den Inhalt der aktuell angezeigten Datei in die Liste s, indem die Funktion **readData**() aufgerufen wird.

Mit der Taste F löschen wir die Anzeige und kehren zum aufrufenden Programm zurück.

Die Taste E dient zum Löschen der aktuell angezeigten Datei. **remove**() erledigt das auf der Speicherkarte, die Anweisung **del** entfernt den Namen aus der Liste **dateien**. Wir merken uns schon mal den Namen der Datei. Die Anzahl der Namen und die Anzeige müssen nach dem Löschen angepasst werden.

Zur Anzeige der Grafik kommen wir mit der Taste B.

```
def files():
    d.clearAll()
    dateien=os.listdir("/sd")[1:]
    n=len(dateien)
    dateien.sort()
    i=0
    while 1:
        d.writeAt("{}    ".format(dateien[i]),0,0)
        x,y=joystick(5)
        if y>0:
            i=min(i+1,n-1)
        elif y<0:
            i=max(i-1,0)
        if getTouch(keyD): # Datei laden
            num=dateien[i][4:dateien[i].index(".")]
            d.writeAt("load data{}    ".format(num),0,4)
            readData(num)
            d.writeAt("got data{}     ".format(num),0,5)
            print("eingelesen:",dateien[i])
            sleep(3)
            d.clearFT(0,4)
        if getTouch(stop): # F files verlassen
            d.clearAll()
            return
        if getTouch(confirm): # E Datei loeschen
            num=dateien[i][4:dateien[i].index(".")]
            os.remove("/sd/"+dateien[i])
            del dateien[i]
            n=len(dateien)
            d.writeAt("data{} killed ".format(num),0,5)
            sleep(3)
            d.clearFT(0,5)
        if getTouch(keyB):
            grafik(s)
            d.clearAll(False)
        sleep(0.2)
```

```
joyX0,joyY0=nvs.get_i32("joyx"), nvs.get_i32("joyy")
```

Wir holen die Nullstellung des Joysticks aus dem nichtflüchtigen Speicher. Es folgen Informationen zum Basiswert und dem Grundrauschen auf dem Display. Mit Taste A wird das Menü aufgerufen. Messwertliste leeren und Dateinummerierung auf 0.

```
d.clearAll()
d.writeAt("s0 ={}".format(s0),0,0)
d.writeAt("dsc={}".format(dsc),0,1)
d.writeAt("A >> Menu".format(dsc),0,5)

s=[]
n=0
```

In der Hauptschleife kümmern wir uns sogleich um einen aktuellen ADC-Wert. Die Differenz mit **S0** liefert uns die Höhe des eventuellen Trigger-Pegels. Der absolute Wert davon ist die Abweichung **ds**, die wir mit dem Grundrauschen **dsc** vergleichen.

Übersteigt **ds** das Grundrauschen, liegt eine Bewegung des Untergrunds vor. Die LED geht an, und eine leere Messwertliste wird vorgelegt. Wir stellen den Timer für die Messdauer. In der **while**-Schleife wird die aktuelle Messung an die Liste angehängt, ein neuer Wert wird geholt, kurze Pause von einer Millisekunde. Ist die **Stoptaste F** nicht gedrückt, geht es in die nächste Runde, sonst wird der Schleifendurchlauf mit **break** beendet. Das Ende der Aufzeichnung ist auch dann erreicht, wenn der Timer **over**() abgelaufen ist. Die LED geht aus, und wir erfahren den Wert des Triggers. Der untere Bereich der Anzeige wird geputzt, die Liste s gespeichert und die Dateinummer erhöht – fertig.

Mit der Taste A kommen wir ins Menü, die Flash-Taste des ESP32 beendet den Programmlauf nachdem die SD-Karte ausgehängt wurde

```
while 1:
    s1=getAdc(3)
    trigger=s1-s0
    ds=abs(trigger)
    if ds > 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,ds))
        d.writeAt("{}. Trig:{}".format(n,trigger),0,2)
        d.clearFT(0,4)
        save2sd(s,n)
        n+=1
    if getTouch(keyA):
        Menu()
    if getTouch(ende): # Flash
        os.umount("/sd")
        d.clearAll()
        d.writeAt("PROGRAM",0,0)
        d.writeAt("CANCELED",0,1)
        sys.exit()
```

Um die Speicherkarte aus dem Slot zu entfernen, sollten Sie das Programm mit der Flashtaste beenden, damit sie automatisch aus dem Dateisystem ausgehängt wird, **umount**() übernimmt das. Alternativ wären zwei weitere Menüpunkte eine interessante Möglichkeit, Taste A – Karte aushängen (umount), Taste E – Karte einhängen (mount). Denken Sie aber auch daran, versehentliches Auslösen der

Aktionen abzusichern, Stichwort **waitForRelease**(), wenn Sie den Vorschlag in die Tat umsetzen.

Immerhin haben Sie mit der SD-Karte eine passable Möglichkeit, die Daten auf den PC zu holen, um sie mit EXCEL auswerten zu können. Wie das geht habe ich ja schon im vorangegangenen Beitrag beschrieben. In der nächsten Folge wird sich ein RTC-Modul (DS3231) dazugesellen, und wir werden die Dateien per WLAN an beliebige Empfänger übertragen. Damit im Zusammenhang steht eines von zwei Problemen, deren Lösung ich Ihnen dann auch verraten werde.

Bleiben Sie dran, bis dann!