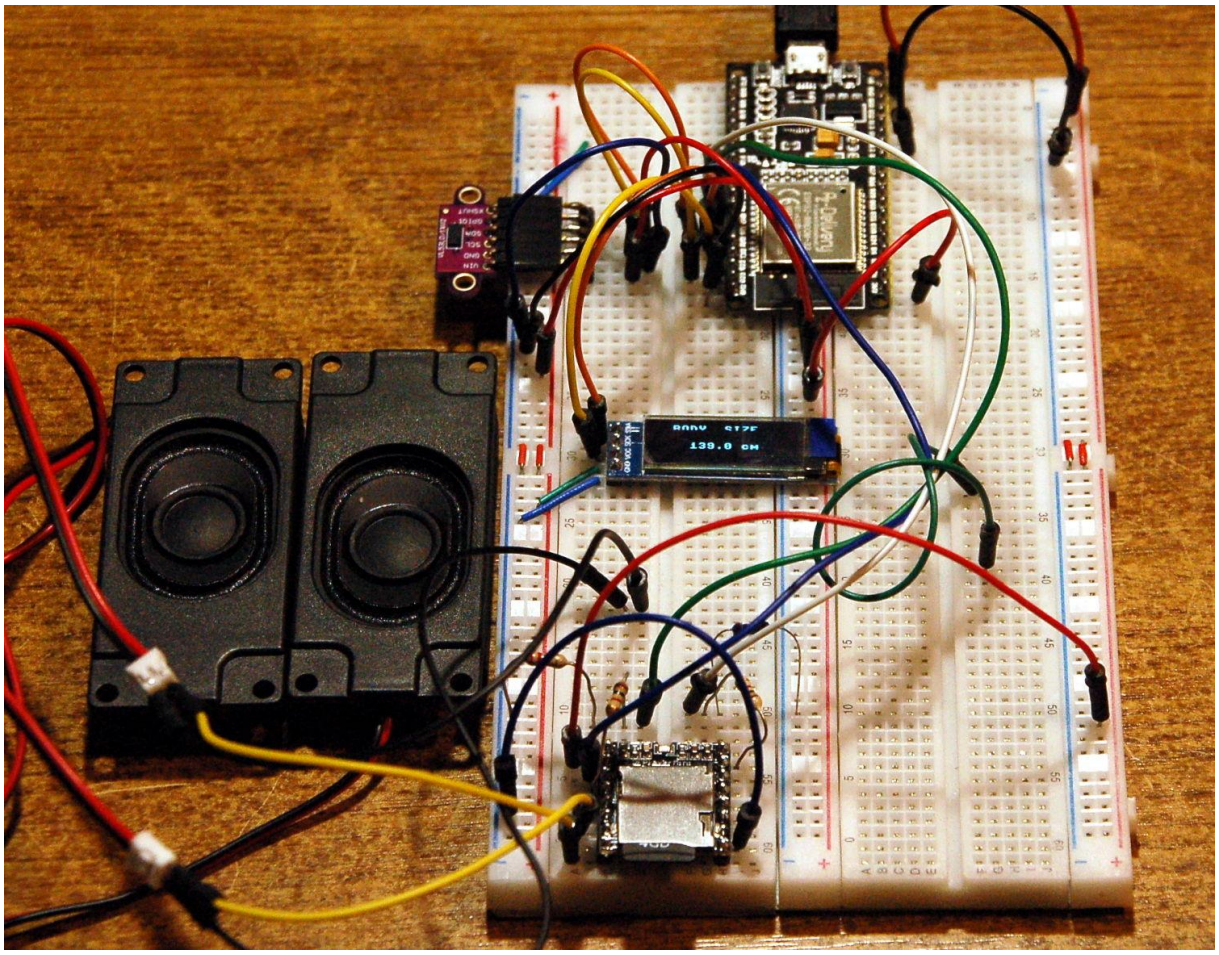


Body-Size-Meter - Aufbau

Diesen Beitrag gibt es auch als PDF-Dokument.

Beim Theremin hatte ich den Time of Flight-Sensor VL53L0X zum Verstellen der Tonhöhe eines Synthesizers benutzt. Entfernungswerte wurden in die Frequenzen eines PWM-Signals umgesetzt. In diesem Projekt wollen wir die relativ genaue Entfernungsmessung per se nutzen. Abgesehen von dem sehr umfangreichen Treiber-Modul des VL53L0X, das wir aber nicht im Detail beleuchten wollen, ist es ein recht einfaches Projekt mit Ausbaupotenzial.

Außerdem möchte ich einmal eine ganz andere Art von Ausgabemedium verwenden. Zu dem OLED-Display, dem VL53L0X und dem ESP32 wird sich noch eine vierte Baugruppe gesellen, mit der eine Sprachausgabe möglich wird. Daneben werde ich ein wenig aus der Werkstatt plaudern. Es gab bei der Entwicklung nämlich ein vertracktes Problem, nicht hausgemacht, sondern vom MicroPython-Entwickler-Team aufs Auge gedrückt.

Willkommen bei einer neuen Folge der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Wie groß bin ich?

ToF ist das Akronym für Time of Flight, und das Modul **VL53L0X Time-of-Flight (ToF) Laser Abstandssensor** macht nichts anderes als die Flugzeit eines von ihm ausgesandten IR-Laserimpulses bis zur Reflexion an einem Hindernis und zurück zu messen.

Das gelingt bei den meisten festen Oberflächen, wenn sie im IR-Bereich genügend Licht reflektieren. Wieviel das ist, können wir mit bloßem Auge leider nicht wahrnehmen. Bestenfalls kann das eine IR-Kamera oder eine umgebaute Web-Cam, aus der das IR-Filter herausoperiert wurde.

Als Reflektor können aber auch ganz einfach Leute dienen, zum Beispiel beim Passieren einer Tür. Um genau diese Anwendung geht es in diesem Beitrag. Der Aufbau soll die Größe von Personen feststellen und über einen Lautsprecher an einem Mini-MP3-Player das Ergebnis mitteilen.

Hardware

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	VL53L0X Time-of-Flight (ToF) Laser Abstandssensor
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Mini MP3 Player DFPlayer Master Module
1	2 Stück 3 Watt 8 Ohm Mini-Lautsprecher
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
2	Widerstand 1kΩ
2	Widerstand 2,2kΩ

Der Aufbau gestaltet sich sehr einfach, Sensor und OLED-Display werden beide am gemeinsamen I2C-Bus angeschlossen. Den Chefposten übernimmt ein ESP32. Für dessen Einsatz bedarf es zweier Breadboards, die mit einer Stromschiene in der Mitte verbunden werden, damit man mit den Abständen der Pinreihen hinkommt und am Rand auch noch Kabel gesteckt werden können.

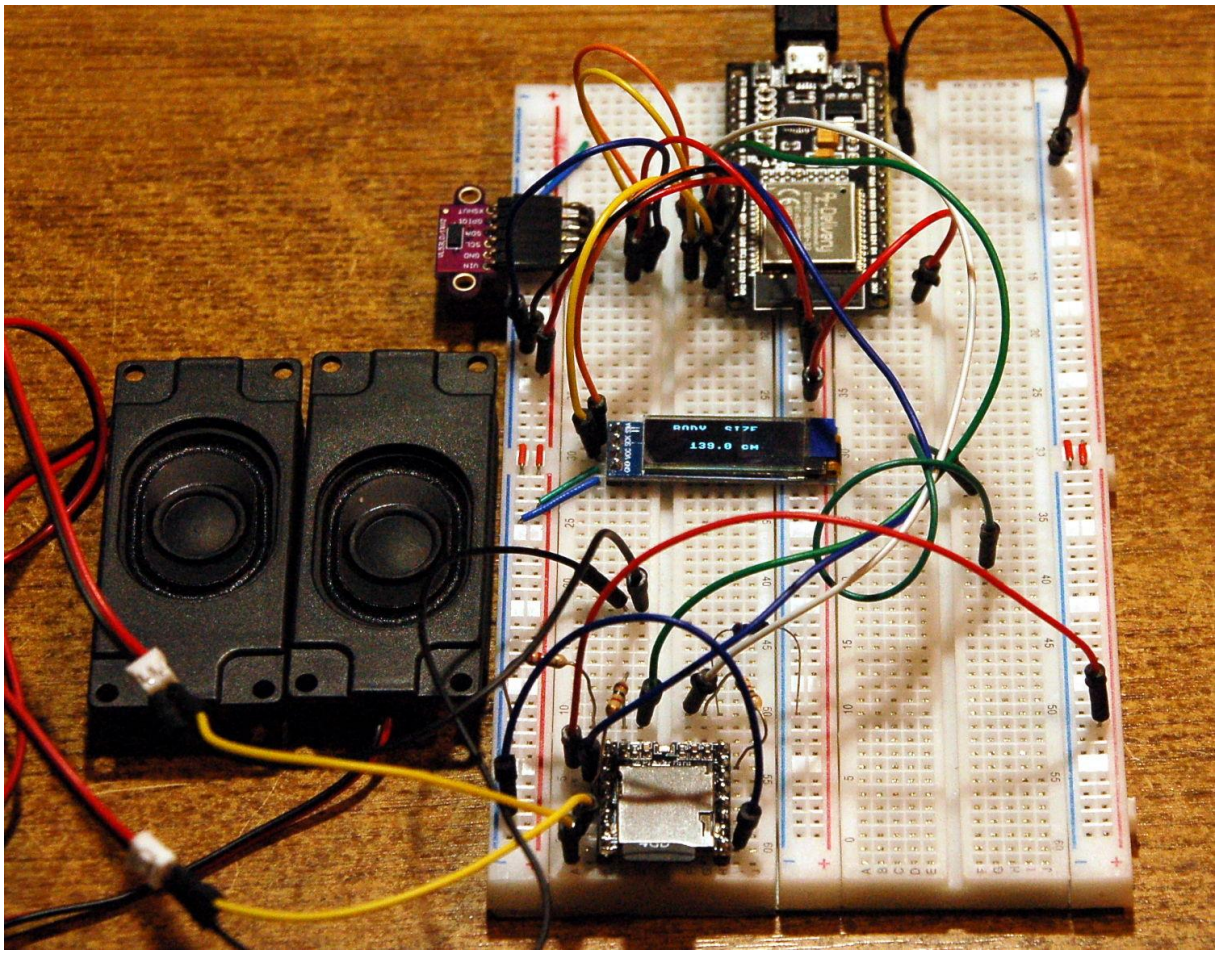


Abbildung 1: Body-Size-Meter - Aufbau

Wegen des sehr umfangreichen VL53L0X-Treibermoduls scheidet der Einsatz eines ESP8266 aus. Aber nicht nur aus diesem Grund ist der kleine Bruder des ESP32 ungeeignet, er kann nämlich auch keine zweite UART-Schnittstelle bieten, und die brauchen wir zur Ansteuerung des DF-Players.

UART ist das Akronym für **Universal Asynchronous Receiver / Transmitter**. Über so ein Interface sprechen wir auch mit unserem Controller vom Terminal aus in Thonny. Der ESP32 /ESP8266 ist über das USB-Kabel mit dem PC verbunden und ein USB-RS232-TTL-Konverter gibt den Datenverkehr an die Schnittstelle UART0 des Controllers weiter. Deshalb können wir UART0 auch nicht benutzen, um Befehle an den Mini-MP3-Player zu senden und Daten von ihm zu empfangen. Der ESP8266 hat zwar eine zweite UART-Schnittstelle, aber das ist eigentlich nur eine halbe, weil nur eine TXD-Leitung (Sendeleitung) und keine RXD-Leitung (Empfangsleitung) zur Verfügung steht.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

MicropythonFirmware
Bitte unbedingt die hier angegebene Version flashen, weil sonst die VL53L0X - Steuerung nicht korrekt funktioniert.
v1.18 (2022-01-17) .bin

Die MicroPython-Programme zum Projekt:

VL53L0X.py modifiziertes Treibermodul für den ToF-Sensor auf ESP32(S) adaptiert
ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für OLED-Displays
distance.py Testprogramm für den VL53L0X
bodysize.py Betriebssoftware

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

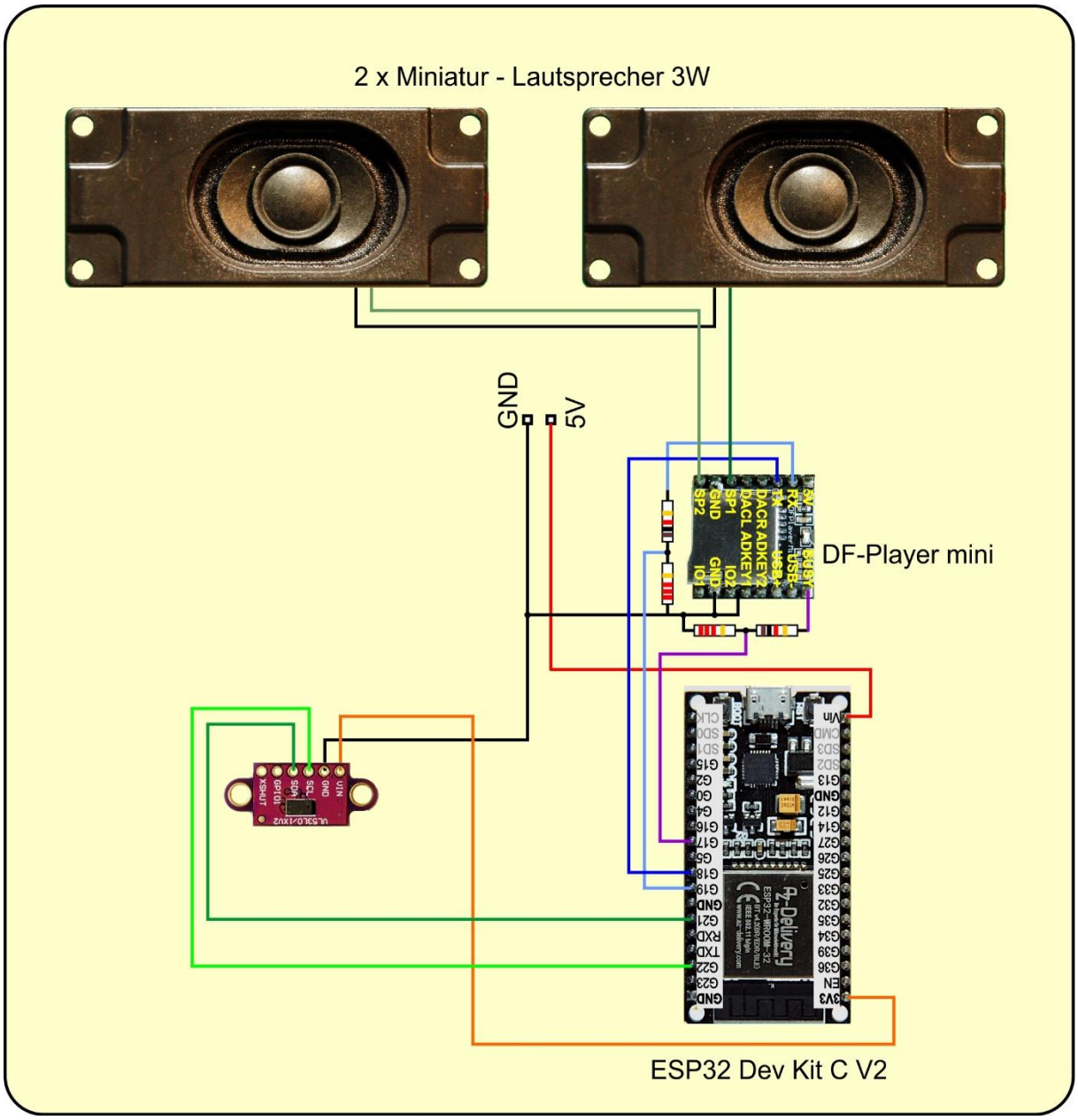


Abbildung 2: Body-Size-Meter - Schaltung

Weil der DF-Player mini mit 5 Volt versorgt werden muss, und daher auch auf den Leitungen TX und Bussy 5V-Pegel anliegen, der ESP32 aber nur 3,3V an seinen GPIOs verträgt, brauchen wir dort Spannungsteiler, welche die Spannung auf ein verträgliches Maß reduzieren. Ich habe 1kΩ und 2,2kΩ gewählt. Sie können auch andere Werte nehmen, die etwa im Verhältnis 1:2 stehen.

Wenn die Lautsprecher mit der schwarzen Leitung an GND gelegt werden, fließt im 5V-Kreis in Strom von ca. 450mA! Das liegt daran, dass die Anschlüsse SP1 und SP2 5V-Pegel führen. Schließen Sie daher die beiden Speaker nur an SP1 und SP2 an, nicht an GND. Auch wenn Sie nur einen Lautsprecher verwenden, gehören dessen Leitungen an SP1 und SP2. Andernfalls müsste man die Lautsprecher über einen Elko anschließen.

Das Treibermodul des VL53L01 für den ESP32 ertüchtigen

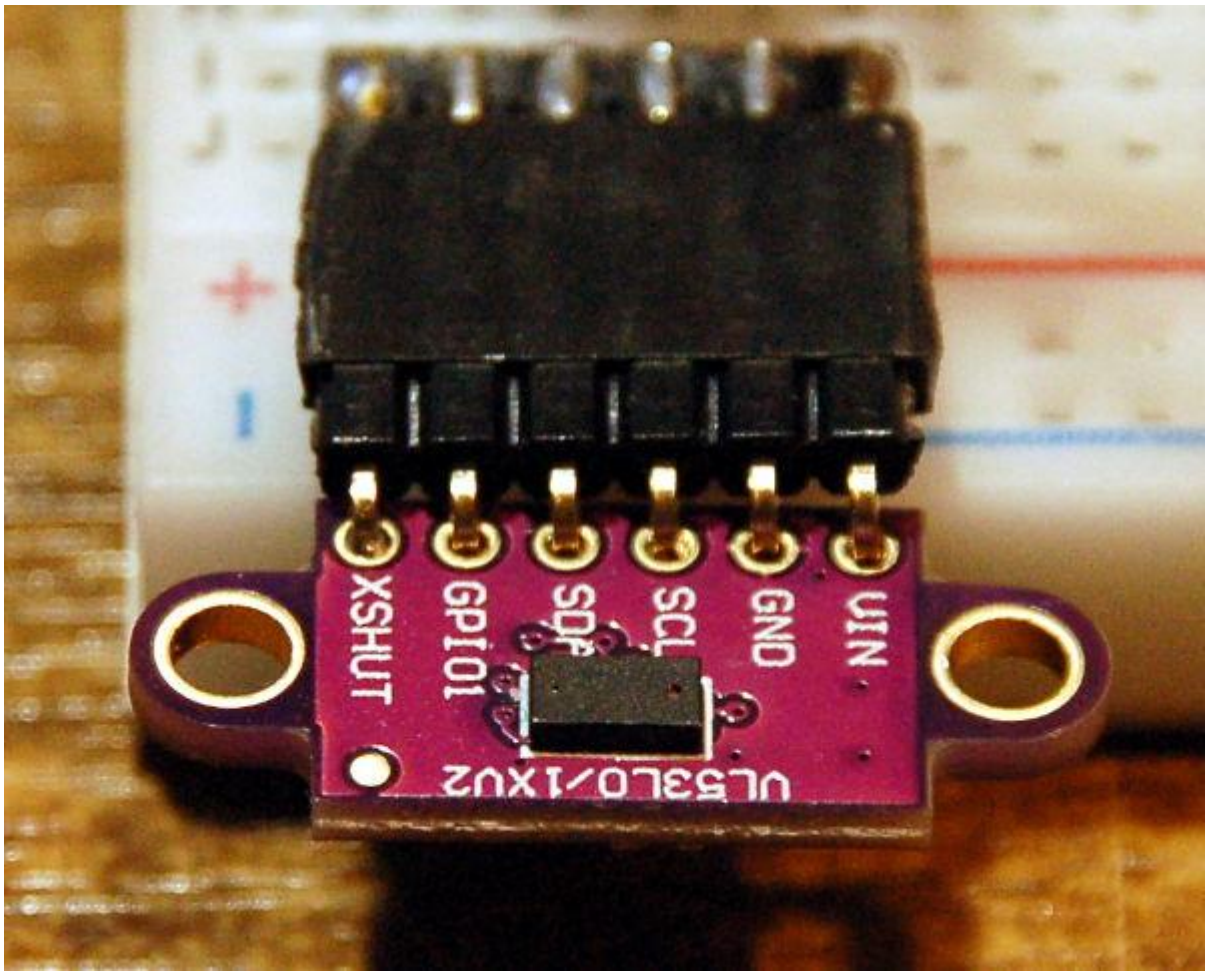


Abbildung 3: VL53L0X - Laser-Entfernungsmesser

Das Datenblatt für den VL53L01 liefert nicht, wie üblich, eine Register-Map und eine Beschreibung für die zu setzenden Werte. Stattdessen wird eine API beschrieben, die als Interface zum Baustein dient. Ellenlange Bezeichner in einer bunten Vielfalt prasseln auf den Leser herunter. Das hat meinen Eifer, ein eigenes Modul zu schreiben, sehr rasch auf null heruntergeschraubt. Glücklicherweise bin ich, nach ein bisschen Suchen, auf Github fündig geworden. Dort gibt es ein Paket, das für einen wipy geschrieben wurde, mit einer Readme-Datei, einem Beispiel zur Anwendung und mit dem sehr umfangreichen Modul VL53L0X.py (648 Zeilen). Der erste Start der Beispieldatei **main**.py brachte eine Ernüchterung, obwohl ich das Modul zum ESP32S hochgeladen und die Pins für den ESP32 umgeschrieben hatte.

Das Original

```
from machine import I2C
i2c = I2C(0)
i2c = I2C(0, I2C.MASTER)
i2c = I2C(0, pins=('P10','P9'))
```

wurde zu

```
from machine import SoftI2C
i2c = I2C(0,scl=Pin(21),sda=Pin(22))
```

Der MicroPython-Interpreter meckerte ein fehlendes Chrono-Objekt an, im Modul VL53L0X.VL53L0X in Zeile 639. Nachdem das Modul für einen anderen Port geschrieben ist, **wipy devices made by Pycom**, konnte es gut sein, dass noch mehr derartige Fehler auftauchen würden. Andere Firmware, andere Bezeichner für GPIO-Pins, andere Einbindung von Hardware-Modulen des ESP32…

Nachdem aber außer den I2C-Pins keine weitere Verbindung zwischen ESP32 und VL53L01 bei dem Beispiel gebraucht wird, stufte ich die Wahrscheinlichkeit, weiterer Problemstellen als gering ein. Ich nahm mir also die Methode **perform_single_ref_calibration**() im Modul VL53L0X.VL53L0X im Editor vor. Der Name Chrono deutete auf ein Zeit-Objekt hin.

Und tatsächlich, in der Firmware des WiPy ist die Benutzung der Hardware-Timer anders gelöst wie beim nativen ESP32. Es geht aber im Prinzip nur um die Initialisierung und Überprüfung eines Timeouts.

```
from machine import Timer
...
...
    def perform_single_ref_calibration(self, vhv_init_byte):
        chrono = Timer.Chrono()
        self._register(SYSRANGE_START, 0x01|vhv_init_byte)
        chrono.start()
        while self._register((RESULT_INTERRUPT_STATUS & 0x07)
= 0):
            time_elapsed = chrono.read_ms()
            if time_elapsed > _IO_TIMEOUT:
                return False
        self._register(SYSTEM_INTERRUPT_CLEAR, 0x01)
        self._register(SYSRANGE_START, 0x00)
        return True
```

Dafür habe ich aber meine eigene Softwarelösung. Eine Funktion, Verzeihung, eine Methode, wir bewegen uns ja in der Definition einer Klasse, also eine Methode **TimeOut**(), die eine Zeitdauer in Millisekunden nimmt und die Referenz auf eine Funktion in ihrem Inneren zurückgibt. So ein Konstrukt nennt man Closure. Damit habe ich nun einfach die Zeitsteuerung ersetzt, kürzer aber genauso effektiv.

```
    def TimeOut(self,t):
        start=time.ticks_ms()
        def compare():
            return int(time.ticks_ms()-start) >= t
        return compare
```

```
    def perform_single_ref_calibration(self, vhv_init_byte):
        self._register(SYSRANGE_START, 0x01|vhv_init_byte)
        chrono = self.TimeOut(_IO_TIMEOUT)
        while self._register((RESULT_INTERRUPT_STATUS & 0x07)
== 0):
            if chrono() :
                return False
        self._register(SYSTEM_INTERRUPT_CLEAR, 0x01)
        self._register(SYSRANGE_START, 0x00)
        return True
```

Außerdem machte sich Erleichterung breit, als sich beim erneuten Start des Demoprogramms keine weitere Fehlermeldung mehr zeigte. Im Gegenteil, im Terminal wurden nach dem Start meines Testprogramms **distance**.py lauter nette Entfernungswerte in cm ausgegeben.

```
# distance.py

import sys
from time import sleep_ms,ticks_ms, sleep_us, sleep
from machine import SoftI2C, Pin
from oled import OLED
from ssd1306 import SSD1306_I2C
import VL53L0X

# ************* Objekte und Variablen erzeugen **************
SCL=Pin(22)
SDA=Pin(21)

i2c=SoftI2C(SCL,SDA)
d=OLED(i2c, heightw=32)

taste=Pin(0,Pin.IN)

tof = VL53L0X.VL53L0X(i2c)
tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 18)
tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 14)
tof.start()
print("started")

distance=9999

tof.read()
d.clearAll()
d.writeAt("BODY  SIZE",2,0,False)
d.writeAt("  METER   ",2,1,False)
d.writeAt("{:.1f} cm    ".format(distance),4,3)
sleep(2)

while True:
    gc.collect()
```

```
    distance=0.0
    n=10
    for i in range(n):
        distance += tof.read()
    distance /= (n*10) #  cm
    d.writeAt("{:.1f} cm    ".format(distance),4,2)
    print (distance)
    sleep(1)
    # Schleife beenden
    if taste.value()==0:
        d.clearAll()
        d.writeAt("BODY  SIZE",4,0,False)
        d.writeAt("  METER   ",4,1,False)
        d.writeAt("TERMINATED",3,2)
        sys.exit()
```

Bis hierher lief alles recht gut. Aber als ich dann auf ein größeres Breadboard umgestiegen bin, um die weiteren BOBs (Break-Out-Boards) unterbringen zu können, wäre ich bald verzweifelt. Auf dem zweiten Board saß bereits ein ESP32 drauf, also sparte ich mir das Umstecken. Aber alles, was grade noch perfekt gelaufen war, funktionierte nun nicht mehr, der VL53L0X brachte nur noch völlig unbrauchbare Werte. Nach diversen ergebnislosen Versuchen, ich tauschte den VL53L0X, setzte einen weiteren ESP32 ein, nahm statt eines ESP32 Dev Kit V2 ein V4, kontrollierte und verglich mehrmals die beiden Aufbauten, prüfte die Kontakte, stets das Gleiche, keine ordentliche Funktion. Der I2C-Bus funktionierte wohl normal, der VL53L0X meldete sich zum Dienst. Das war's dann aber auch schon. Als ich doch noch den ESP32 vom ersten Board nahm, kam die Erleuchtung. Siehe da, der funktionierte richtig. Und dann bemerkte ich den wesentlichen Unterschied. Der ESP32 auf dem ersten Versuchsboard hatte einen MicroPython-Kernel 1.18 und alle anderen hatten das letzte Release 1.19. Nachdem ich auch den zweiten ESP32 mit der Version 1.18 geflasht hatte, schnurrte der auch wie ein zufriedenes Kätzchen.

Ich hatte so etwas Ähnliches schon einmal in Bezug auf die Verwendung von PWM. Ich verstehe nicht, warum den MicroPython-Entwicklern solche Bugs passieren. Da funktioniert ein Feature perfekt und beim Updaten mit der Folgeversion kriegste graue Haare, weil nix mehr geht. Dabei sollten in der neuen Version Fehler beseitigt sein und neue eingebaut.

Sicher, es kann schon auch einmal vorkommen, dass ein Bauteil defekt ist, gar so ein komplexes wie ein Controller. Wenn beim Experimentieren oder beim Nachbau einer Schaltung etwas nicht so funktioniert wie es sollte, dann suchen Sie nicht nur nach Fehlern im Programm oder in der Schaltung, sondern denken Sie auch daran, dass das Betriebssystem eines ESP, der Kernel, buggy sein kann. Deshalb habe ich oben schon darauf hingewiesen, dass Sie unbedingt die 1.18 verwenden müssen, damit dieses Projekt funktioniert.

Der DF-Player mini

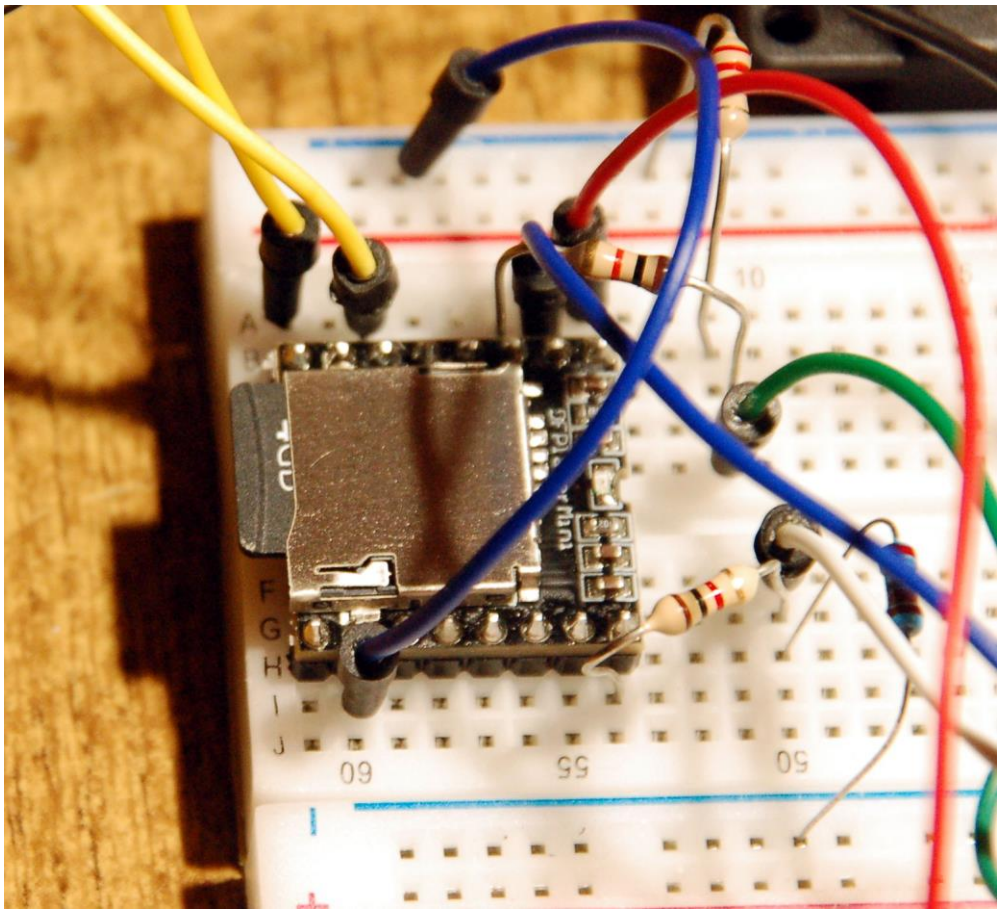


Abbildung 4: DF-Player mini mit 4 GB SD-Karte

Für den MP3-Player wird über eine RS232-Leitung angesprochen. Ich hatte früher schon einmal ein MicroPython-Modul dafür geschrieben. Weil das auf einem ESP8266 lief, hatte ich mir damals die Mühe gespart, Routinen für den Datenempfang aufzunehmen, der Kleine hat ja, wie ich schon bemerkt habe, dafür keine Leitung frei, die wird boardintern gebraucht als SD-D1 zum externen Flash-Chip. Für den ESP32 habe ich das Modul deshalb erweitert. Man kann jetzt zum Beispiel auch die Version, die Anzahl von Dateien auf der SD-Karte oder den Lautstärkepegel abfragen.

Die Kommunikation mit dem DF-Player geschieht stets in der gleichen Weise. Es werden Blöcke von 10 Bytes gesendet und empfangen. Der Aufbau eines solchen Blocks sieht wie folgt aus. Die Methode **sendCommand**() kümmert sich um die korrekte Übermittlung.

Startbyte 0x7E
Versionsnummer 0xFF
Länge der Payload 0x06 (von Versionsnummer bis Parameter 2 inkl.)
Commandbyte
Feedback 0x00 (nein) oder 0x01 (ja)
Parameter 1
Parameter 2
Checksum high
Checksum low
Endebyte 0xEF

Die Befehlscodes kann man im Datenblatt finden. Dieses habe ich benutzt, um die wesentlichsten Codes in Methoden des Moduls umzusetzen. Wir brauchen für das aktuelle Projekt nur ein paar davon. Werfen Sie aber ruhig mal einen Blick in die Datei **pfplayer**.py.

Damit eine Sprachausgabe möglich wird, brauchen wir die Ziffern und Zahlen als einzelne, kurze Sounddateien. Die Texte dafür habe ich mit dem Handy in einem Stück aufgenommen und via Bluetooth auf den PC übertragen, wo ich dann die Datei mit Audacity in die Sprachclips aufgeteilt habe. Audacity ist ein Freewaretool zum Bearbeiten von Audiodateien.

Um eigene Sprachbausteine herzustellen, Nehmen Sie folgende Texte auf. Die Zahlen in der zweiten Spalte sind die Nummern, die später als Dateinamen der Clips beim Speichern auf der SD-Karte gebraucht werden.

ein	000
eins	001
zwei	002
…	
zehn	010
elf	011
zwölf	012
dreizehn	013
…	
neunzehn	019
zwanzig	020
dreißig	030
…	
neunzig	090
einhundert	100
hundert	101
und	102

Die Datei vom Handy hat wahrscheinlich die Endung **m4a**. Damit kann Audacity leider nichts anfangen. Wir müssen eine Typumwandlung nach **mp3** vornehmen. Das geht prima mit dem Online-Tool Convertio. Folgen Sie dem Link und ziehen Sie dann Ihre m4a-Datei auf das rote Feld.

Abbildung 5: convertio - Startfenster

Im nächsten Fenster klicken Sie auf **Konvertieren**. Die Datei wird zum Server hochgeladen und …

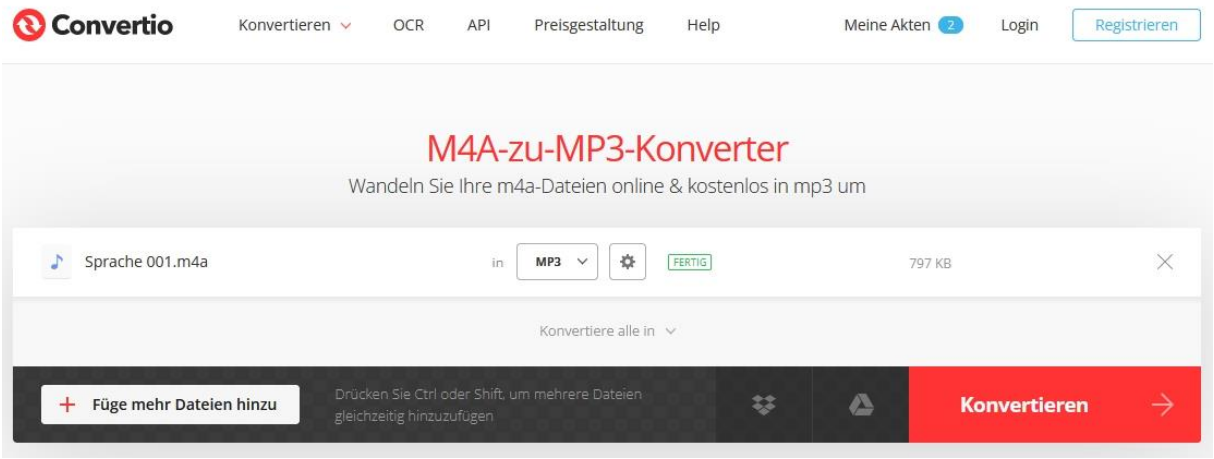


Abbildung 6: convertio - Konvertieren

nach ein paar Sekunden können Sie Ihre mp3-Datei herunterladen.

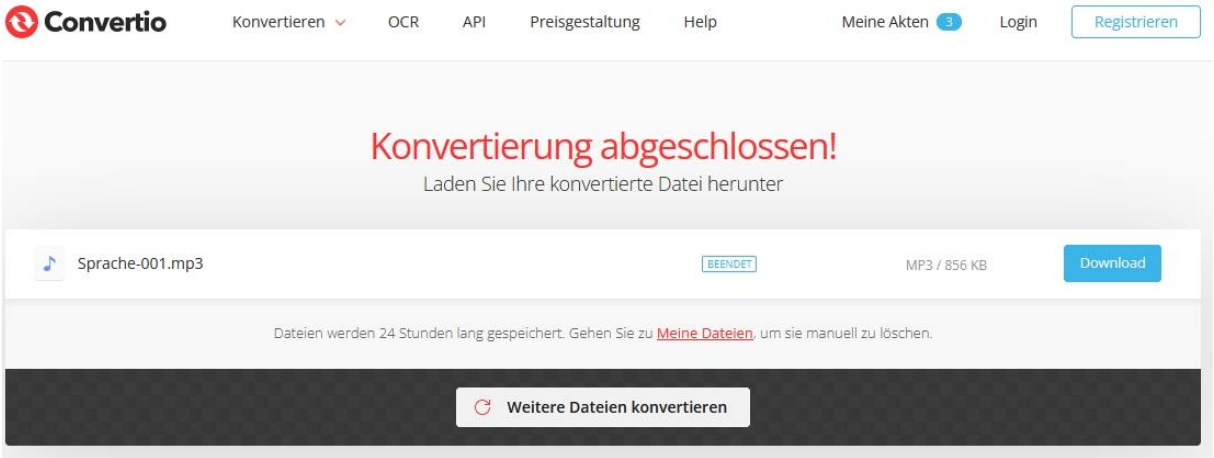


Abbildung 7: convertio - Download

Starten Sie jetzt Audacity und ziehen Sie die Datei ins Bearbeitungsfenster.

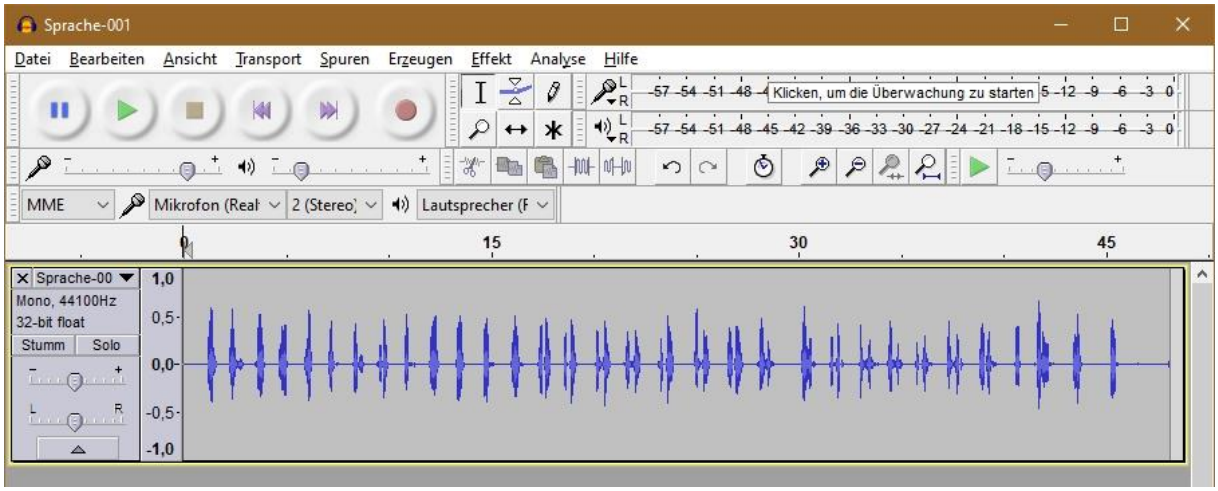
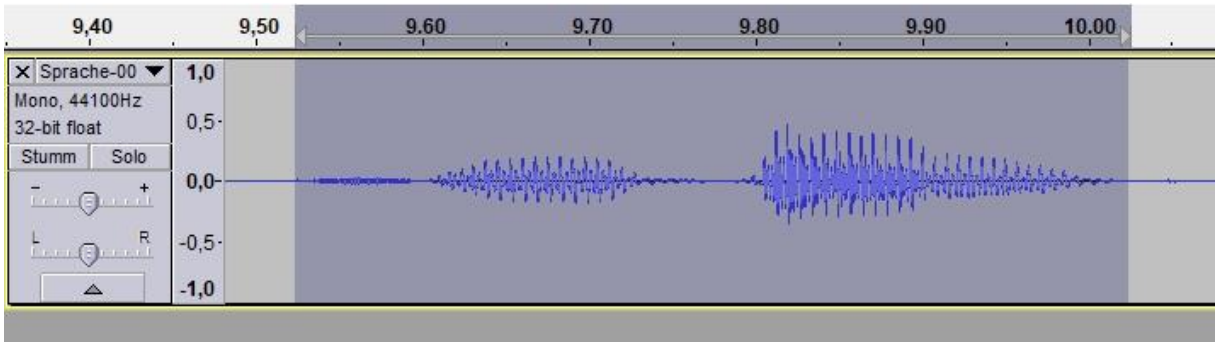


Abbildung 8:Audacity mit Sound-Datei

Mit der Strg-Taste und dem Mausrad zoomen Sie in die Spur hinein und heraus. Markieren Sie nun einen Clip durch Ziehen mit der Maus.



Mit Strg + Shift + A starten Sie den Speichervorgang der Auswahl. Geben Sie als Dateinamen die Nummern aus der obigen Liste ein. Sie können auch eine Textergänzung hinten anfügen, etwa 007-sieben. Klicken Sie auf Speichern. Danach dreimal Enter, weiter geht's zum nächsten Clip.

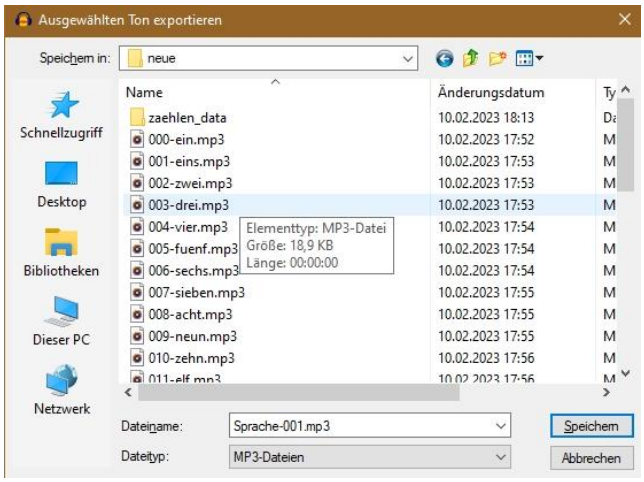


Abbildung 9: Audacity - Clip speichern

Kopieren Sie nun die Clips auf eine mini SD-Karte in den dort angelegten Ordner 00.

Wie das Programm bodysize.py arbeitet

Vier externe Module müssen auf den ESP32 hochgeladen werden: VL53L0X.py, dfplayer.py, oled.py und ssd1306.py. Dann starten wir mit den Importen.

```
# bodysize.py

import VL53L0X
from dfplayer import DFPlayer
from machine import SoftI2C,Pin
from oled import OLED
from time import sleep
import sys

tof = VL53L0X.VL53L0X(i2c)
```

Wir instanziieren ein I2C-Objekt, das wir gleich an den Konstruktor der OLED-Klasse und der Klasse VL53L0X weiterreichen.

Das VL53L0X-Objekt wird initialisiert und gestartet. Der erste Wert wird eingelesen und eingestampft, das heißt wir machen damit nichts. In **height** legen wir die Höhe in Zentimeter ab, in der später der Sensor angebracht wird, zum Beispiel oben am Türrahmen.

```
tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 18)
tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 14)
tof.start()
size=tof.read()
print("ToF started")
height=200
```

Dann ist der DF-Player dran. Die Pin-Nummern der seriellen Leitungen RXD und TXD sowie der Bussy-Leitung werden deklariert und an den Konstruktor weitergegeben. Wir warten zwei Sekunden, bis der Player initialisiert hat. Die Verbindung klappt, wenn der Player uns die Anzahl der Titel auf der Karte mit 31 meldet.

```
RX_Pin = 18
TX_Pin = 19
bussyPin = 17
df=DFPlayer(bussyPinNbr=bussyPin, txd=TX_Pin, rxd=RX_Pin,
volume=80)
sleep(2)
numOfTitles=df.getNumberOfFiles()
print("Anzahl Titel:",numOfTitles)
```

Ein Pin-Objekt für die Flash-Taste am ESP32 wird deklariert, dann definieren wir zwei Funktionen, die die eigentliche Arbeit erledigen, sprechen und die Körpergröße ermitteln.

```
def say(n):
    assert n <= 199, "Der Wert muss zwischen 1 und 199 liegen"
    h=n // 100
    rest=n % 100
    z=rest // 10
    e=n % 10
    print(h,z,e)
    s=[]
```

Die auszusprechende Zahl muss zwischen 1 und 199 liegen. Das sichern wir mit **assert** ab. Wurde ein falscher Wert in **n** übergeben, wird eine AssertationError-Exception geworfen.

Dann spalten wir die Zahl in die Hunderter, den Zehnerrest, die Zehnerziffer und die Einerziffer auf. Eine leere Liste **s** wird angelegt und gleich mit den Nummern der abzuspielenden Soundclips gefüllt.

```
    if h == 1:
        s.append(100)
```

Auf die Anweisung **df.play**(0,100) spielt der DF-Player die Datei 100-einhundert.mp3 im Ordner 00 ab. Das tut er, wenn die Hunderterziffer 1 ist.

```
    if 10 <= rest < 20:
        s.append(rest)
```

Eine Sonderbehandlung brauchen die Zahlen zwischen 10 inklusive und 20 exklusive. Der Zehnerrest ist Namensbestandteil der entsprechenden Dateien und wird daher an die Soundliste s angehängt.

Weitere Sonderfälle sind die ganzen Zehner, zehn, zwanzig und so weiter. Hier ist der Zehnerrest größer oder gleich 20 und die Einerziffer ist 0. Alle weiteren Zehnerreste folgen der Regel Einerziffer "und" Zehnerziffer wobei im Falle einer 1 als Einerziffer zum Beispiel nicht "eins" "und" "dreißig", sondern "ein" " und" "dreißig" herauskommen muss.

```
    elif rest >= 20:
        if e == 0:
            s.append(10*z)
        else:
            e= e if e != 1 else 0
            s.append(e)
            s.append(102)
            s.append(10*z)
```

Der else-Zweig behandelt die Fälle 101 bis 109 und 1 bis 9. Zur Kontrolle lassen wir uns die Liste s im Terminal ausgeben. Diese Anweisung kann später wegfallen.

```
    else:
        s.append(e)
    print(s)
```

Die for-Schleife liest uns jetzt die einzelnen Bestandteile der Liste **s** vor. Wenn der Player nicht gerade beim Abspielen einer Datei ist, fängt er damit an, den Track **t** im Ordner 00 darzubieten. Dann wartet der ESP32, bis der Player damit fertig ist. Das ist der Fall, wenn die Bussy-Leitung auf 0 geht. Die Methode **isPlaying**() überprüft das.

```
    for t in s:
        if not df.isPlaying():
            df.play(0,t)
            while df.isPlaying():
                pass
    return s
```

Die Funktion **getSize**() bestimmt die Körpergröße der Person, die unter dem Sensor durchgeht. Die Montagehöhe **height** desselben muss bekannt sein. Der Sensor erfasst den Abstand **distance** zwischen ihm und dem Kopf der Person Die Körpergröße **size** ergibt sich dann aus der Differenz aus height und distance. Für genauere Ergebnisse führen wir mehrere Einzelmessungen durch deren Anzahl wir im Parameter n übergeben. Der VL53L0X liefert seine Messwerte in Millimetern. Weil wir Zentimeter bevorzugen, teilen wir bei der Berechnung des Mittelwerts zusätzlich durch 10. Die Werte gehen in die Anzeige. Der auf eine Ganzzahl aufgerundete Wert der Körpergröße wird zurückgegeben.

Vor dem Eintritt in die Hauptschleife stellen wir die Lautstärke des Players auf 80%.

```
df.volume(80)
while 1:
    size=getSize(10)
    if size >= 50:
        try:
            say(size)
        except AssertationError as e:
            d.clearAll()
            d.writeAt(" ERROR",4,0,False)
            d.writeAt(" RANGE",4,1,False)
            d.writeAt("OVERRUN",4,2)
    if taste.value()==0:
        d.clearAll()
        d.writeAt("BODY SIZE",2,0,False)
        d.writeAt(" PROGRAM ",2,1,False)
        d.writeAt("TERMINATED",2,2)
        sys.exit()
```

Wir fordern die Körpergröße mit zehn Einzelmessungen an. Damit der ESP32 nicht jedes Mal Laut gibt, wenn die Miezekatze den Sensor passiert, soll der Player nur

plappern, wenn das Objekt größer als 50 cm ist. Sollten von **getSize**() fehlerhafte Werte über 199 geliefert werden, fangen wir die Exception mit try ab und geben eine Fehlermeldung im Display aus. Sonst wird der Wert abgespielt.

Um während der Entwicklung geordnet aus dem Programm auszusteigen, kann man die Flash-Taste am ESP32 drücken. Die Module sind dann noch aktiv. Sie können also händisch noch einige Experimente durchführen. Entdecken Sie zum Beispiel die Methoden des DF-Player-Moduls oder nehmen Sie einen Begrüßungstext zusätzlich zu den Zahlen und Ziffern auf, den Sie beim Programmstart vorlesen lassen. Ein besonderer Gag ist es, Sensor- und Ansageteil zu trennen, dann hängen am Türstock keine Lautsprecher rum. Ein ESP32 erledigt den Messjob ein zweiter die Ansage. Die Übertragung des Messwerts kann per WLAN über UDP erfolgen. Im weiteren Ausbau könnte eine Messwertanzeige mit einem großen Display aus 8x8_LED-Anzeigen entstehen. Denkbar wäre auch ein integrierter Passantenzähler…

Viel Spaß beim Basteln und Programmieren!