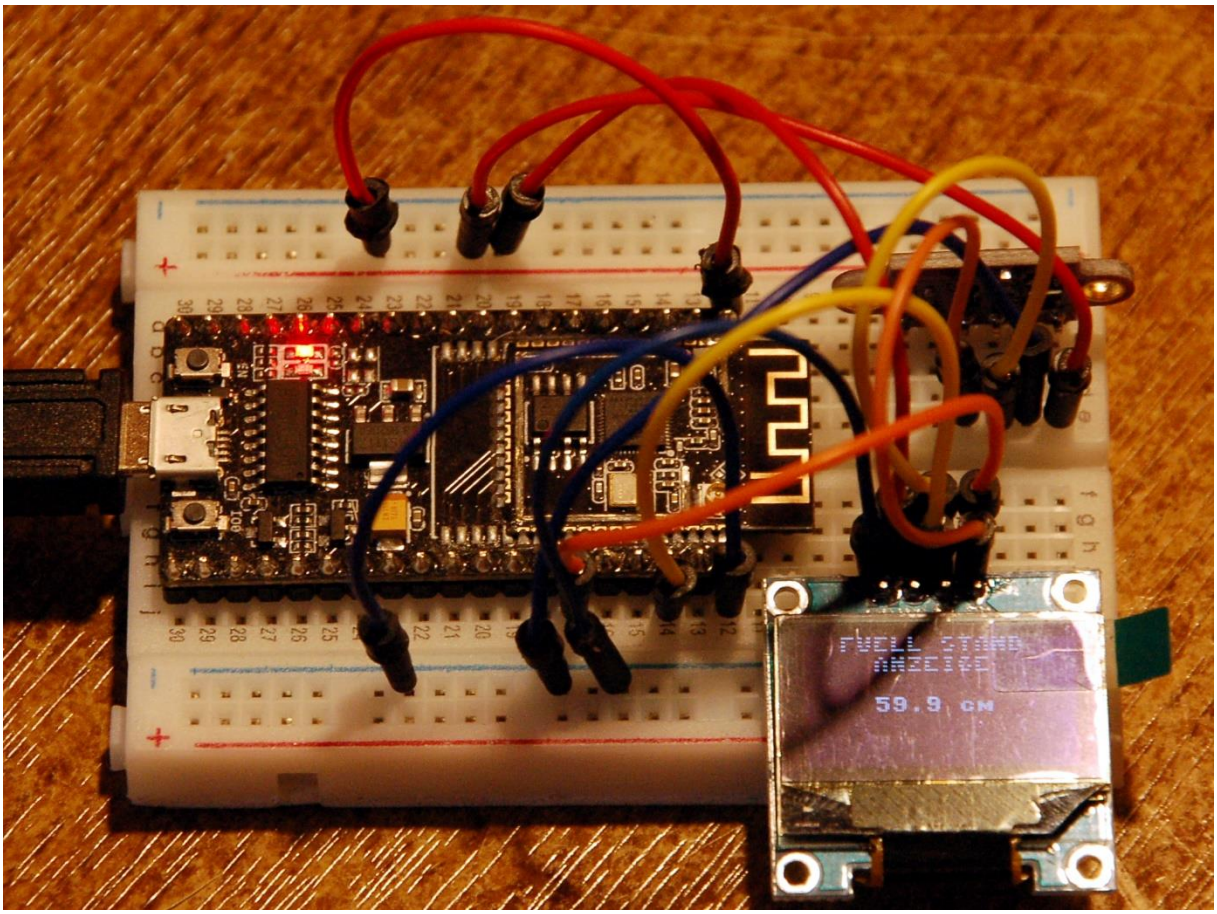

Füllstand Erfassung mit VL53L0X

Beim Theremin hatten wir 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. Willkommen bei einer neuen Folge der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Der ESP32 als Wasserstandsmelder (Teil 1)

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, der das IR-Filter herausoperiert wurde.

Als Reflektor können aber auch Flüssigkeitsoberflächen dienen, zum Beispiel die Wasseroberfläche in einem Regenwasserfass. Um genau diese Anwendung geht es in diesem Beitrag.

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	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins

Der Aufbau gestaltet sich sehr einfach, Sensor und OLED-Display werden beide am gemeinsamen I2C-Bus angeschlossen. Den Rest der Arbeit übernimmt ein ESP32. Wegen des sehr umfangreichen VL53L0X-Treibermoduls scheidet der Einsatz eines ESP8266 aus. Und weil später eine Funkverbindung dazukommen soll, ist auch ein Raspberry Pi pico nicht geeignet.

Warum ich mich für den ToF-Sensor und nicht für einen Ultraschallsensor entschieden habe, das erfahren Sie weiter unten.

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
fuellstand_einfach.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.

Das Treibermodul des VR53L01 für den ESP32 ertüchtigen

Das Datenblatt für den VR53L01 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'))
```

wird 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 VR53L01 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 lauter nette Entfernungswerte in mm ausgegeben.

```
import sys,os
import time
from machine import Pin,PWM
from machine import I2C
import VL53L0X

i2c = I2C(0,scl=Pin(21),sda=Pin(22))

# Create a VL53L0X object
tof = VL53L0X.VL53L0X(i2c)
sound=PWM(Pin(18), freq=20, duty=512)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[0], 18)

tof.set_Vcsel_pulse_period(tof.vcsel_period_type[1], 14)
```

```
tof.start()

while True:
# Start ranging
    tof.read()
    print(tof.read())
```

Bereit für das Wasserfass

Im ersten Teil der Beiträge zum Thema Wasserfass soll es nur um die Anwendung des Sensors und die Darstellung der Messergebnisse auf dem Display gehen.

Eine Alternative zum VL53L0X wäre ein Ultraschallmodul von der in Abbildung 1 gezeigten Art gewesen.

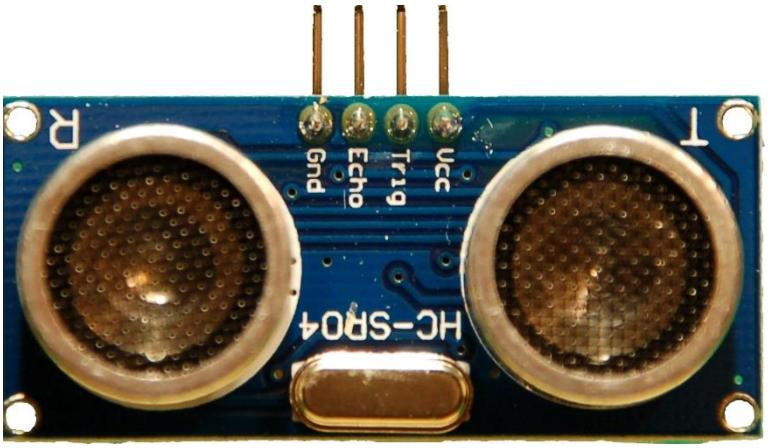

Abbildung 1: ULTRASCHALLMODUL

Warum ein Ultraschallsensor nicht in die engere Wahl kam liegt an der unzuverlässigen Messwerterfassung dieses Sensors. Vor allem in dem engen Fass gab es wohl immer wieder unvorhersehbare Reflexionen, die zu stark abweichenden Messergebnissen führten. Ein Testprogramm für einen Amica 8266 12F könnte etwa so aussehen:

```
# uschall.py
# 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
from time import sleep_us, ticks_us, sleep

#GPIO Pins zuweisen
trig = Pin(14, Pin.OUT)
echo = Pin(12, Pin.IN)


def distance():
    trig.value(1)
    sleep_us(10)
```

```
    trig.value(0)
    start,stop=0,0

    # Start mit steigender Flanke an echo
    while echo.value() == 0:
        start = ticks_us()

    # Stop mit fallender Flanke an echo
    while echo.value() == 1:
        stop = ticks_us()

    # Schalllaufzeit gesamt in Sekunden
    laufzeit = (stop - start)/1000000
    # Schallgeschwindigkeit (34300 cm/s)
    # v = s/t => s = v*t
    # hin und zurueck => s=s/2
    abstand = (laufzeit * 34300) / 2
    return abstand


while True:
    print ("Entfernung: {:.2f} cm".format(distance()))
    sleep(1)
```

Das Modul misst ähnlich wie der VL53L0X eine Laufzeit, hier die Laufzeit eines Ultraschallpulses bis zum Hindernis und zurück. Die relative Genauigkeit liegt im Zimmer bei einer Größenordnung von 1cm. Aber der Wert ist nur reproduzierbar, wenn der Reflektor eine große, ebene Fläche ist und keine weiteren Hindernisse im Schallkegel liegen.

Entfernung: **48.93** cm
Entfernung: 48.95 cm
Entfernung: 48.93 cm
Entfernung: 48.93 cm
Entfernung: 49.19 cm
Entfernung: 48.93 cm
Entfernung: 49.55 cm
Entfernung: 48.93 cm
Entfernung: **50.61** cm

Andernfalls kann die Messwerttabelle im Zimmer so aussehen. Im Außenbereich gab es noch größere Abweichungen.

Entfernung: **53.47** cm
Entfernung: 53.04 cm
Entfernung: 52.84 cm
Entfernung: **21.83** cm
Entfernung: **51.40** cm
Entfernung: 52.34 cm
Entfernung: 52.24 cm

Das war mir zu unzuverlässig und deswegen habe ich mich für den ToF-Sensor entschieden.

Hier kommen Schaltplan und Aufbau.

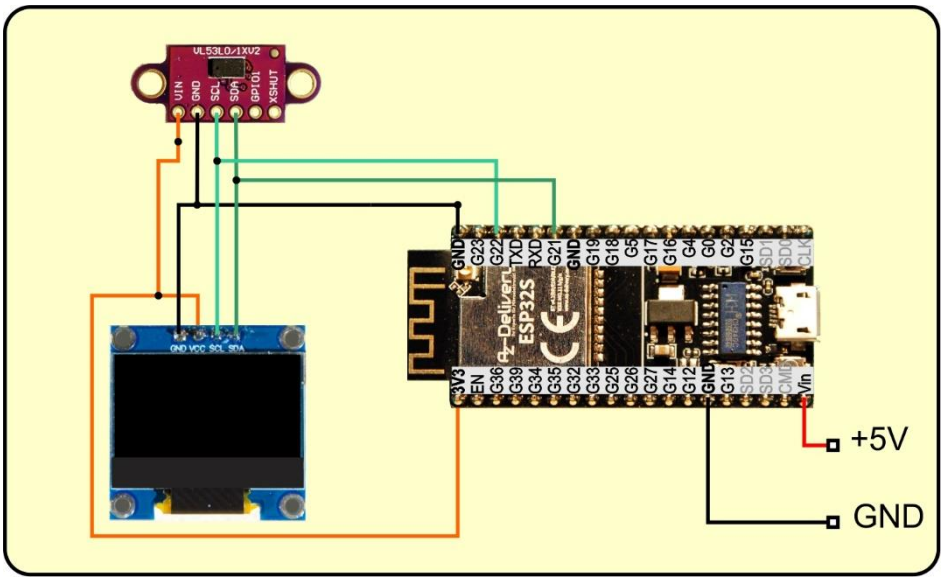

Abbildung 2: Füllstandsmessung - Schaltung

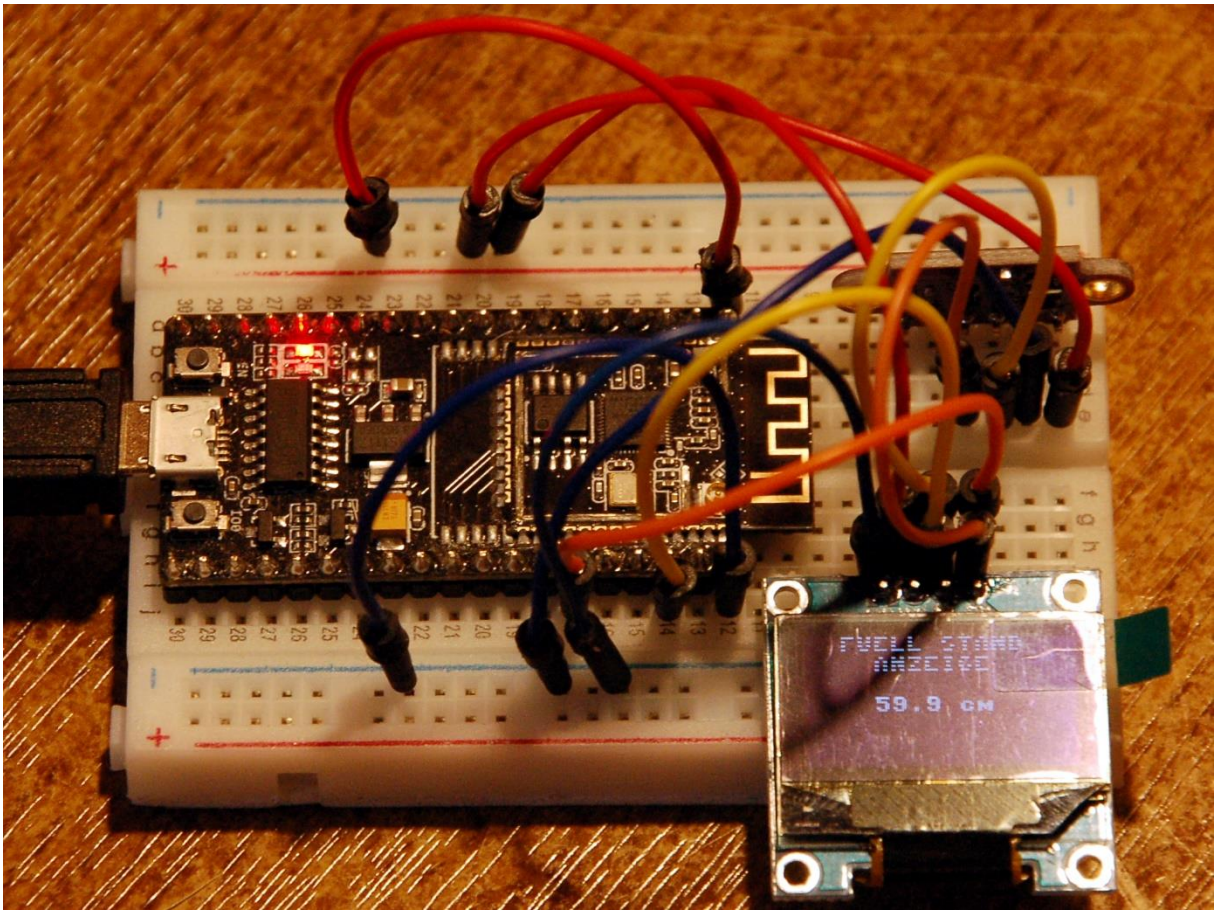

Abbildung 3: Füllstand Erfassung mit VL53L0X

Das Programm

Betrachten wir das Programm, das wie stets mit dem Import von Modulen beginnt. Falls unbeabsichtigte Neustarts nerven, kann es auch hilfreich sein, **webrepl**, die Funk-Kommandozeile, auszuschalten. Dazu gebe ich

>>> **import webrepl_setup**

im Terminalbereich von Thonny ein, quittiere den Prompt mit "**d**" und lasse den Controller neu durchstarten. Bei ESP8266-Modulen, kann es auch nötig sein, das AP-Interface auszuschalten.

```
# fuellstand.py

# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!

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
```

Da ein ESP8266 wegen des umfangreichen Moduls VL53L0X nicht in Frage kommt, vergebe ich feste Pin-Nummern für die I2C-Leitungen. Mit dem I2C-Objekt i2c wird auch gleich die OLED-Instanz erzeugt.

```
SCL=Pin(22)
SDA=Pin(21)

i2c=SoftI2C(SCL,SDA)
d=OLED(i2c)
```

Die Programmabbruchtaste wird deklariert, dann erzeuge ich das Time of Flight-Objekt **tof** und starte es.

```
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")
```

Natürlich kann ich meine Schaltung nicht am Boden des Wasserfasses verankern. Ich bringe sie also ein Stück oberhalb des Fassrandes möglichst mittig über der Öffnung an. Der Wasserstand wird dann durch Differenzbildung bestimmt. Dazu messe ich die Entfernung wischen Fassboden und Sensor einfach mit dem Klappmeterstab und setze den Wert für **hoeheUeberGrund** im Programm ein. Der Wert für **distance** wird deklariert und mit 9999 vorbelegt, was andeutet, dass noch keine Messung stattgefunden hat.

```
distance=9999
hoeheUeberGrund=120
```

Die erste Messung wird gestartet und verworfen. Dann bereite ich das Display für die laufende Anzeige vor.

```
tof.read()
d.clearAll()
d.writeAt("FUELL-STAND",2,0,False)
d.writeAt("  ANZEIGE",2,1,False)
d.writeAt("{:.1f} cm    ".format(distance),4,3)
sleep(2)
```

In der Hauptschleife lasse ich den Speicher bereinigen, setze die Arbeitsvariable **distance** auf 0 und addiere dann n=10 Messwerte darin auf. Weil die Ergebnisse vom VL53L0x in mm geliefert werden addiere ich 5mm zum Ergebnis und dividiere anschließend durch n und 10, um den gerundeten Mittelwert in der Einheit cm zu erhalten. Der Füllstand ergibt sich jetzt als Differenz aus der Höhe bis zum Fassboden und der gemessenen Distanz bis zur Wasseroberfläche. Das Ergebnis wird im Display angezeigt und nach einer Sekunde Pause startet die nächste Messung, es sei denn die Flashtaste am ESP32-Modul wurde gedrückt, um das Programm abzubrechen. In diesem Fall wird die entsprechende Meldung ausgegeben und wir landen am REPL-Prompt im Terminalbereich.

```
while True:
    gc.collect()
    distance=0.0
    n=10
    for i in range(n):
        distance=distance+tof.read()
    distance=(distance+5)/(10*n)
    fuellstand=hoeheUeberGrund - distance
    d.writeAt("{:.1f} cm    ".format(fuellstand),4,3)
    sleep(1)
    # Schleife beenden
    if taste.value()==0:
        d.clearAll()
        d.writeAt("FUELL-STAND",4,0,False)
        d.writeAt("  ANZEIGE",4,1,False)
        d.writeAt("TERMINATED",3,2)
        sys.exit()
```

Damit das Programm autonom starten kann, fehlt noch ein letzter Schritt. Der Programmtext muss im Editorfenster in eine neue Datei kopiert werden. Ich kopiere also den gesamten Text in die Zwischenablage, lege via File – New eine neue Datei an und kopiere den Text dort hinein. Dann speichere ich diese Datei unter dem Namen **main.py** auf dem **ESP32** ab. Nach einem Reset startet das Programm automatisch auch dann, wenn das Modul nicht am PC angeschlossen ist.

Hier folgt nun noch das gesamte Listing des Programms, das Sie natürlich auch herunterladen können, ebenso wie diesen Blogbeitrag als PDF-Dokument.

```
# fuellstand.py

# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!

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)

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
hoeheUeberGrund=120

tof.read()
d.clearAll()
d.writeAt("FUELL-STAND",2,0,False)
d.writeAt("  ANZEIGE",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=distance+tof.read()
    distance=(distance+5)/(10*n)
    fuellstand=hoeheUeberGrund - distance
    d.writeAt("{:.1f} cm    ".format(fuellstand),4,3)
    sleep(1)
    # Schleife beenden
    if taste.value()==0:
        d.clearAll()
        d.writeAt("FUELL-STAND",4,0,False)
        d.writeAt("  ANZEIGE",4,1,False)
        d.writeAt("TERMINATED",3,2)
        sys.exit()
```

Ausblick:
Die Erkenntnisse aus diesem Projekt werden als nächstes um eine WLAN-Funktionalität erweitert. Der Füllstand ist dann an einer Station im Haus oder über das Handy abrufbar.

Als weitere Einsatzmöglichkeit des VL53L0X bietet sich ein Nahfeldauslöser für die Nikon 50 und ähnliche Modelle an. Wenn ein Objekt in ein definierbares Entfernungsfenster vor dem Sensor eintritt, wird über IR-Licht eine Serie von Fotos ausgelöst. Das erweitert die bereits vorgestellten Projekte zu diesem Thema (IR-Fototimer und IR-Remote-Auslöser).