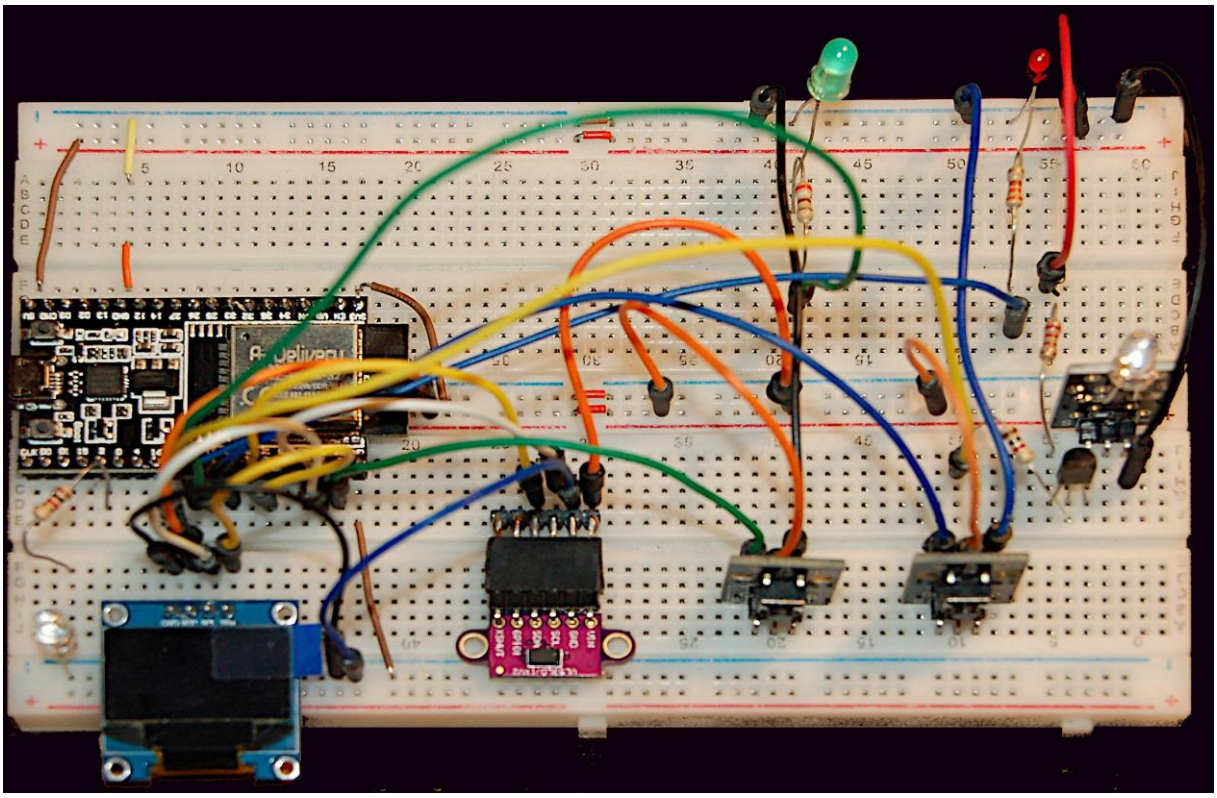

Der Nikon IR-Auslöser hat noch einen Nachschlag erhalten. Mit Hilfe eines VL53L0X wird es möglich, Ereignisse in einem einstellbaren Entfernungsfenster als Auslöser für die Kamera zu erfassen. Natürlich ist Anwendung nicht auf die Kamera beschränkt, sondern lässt sich, mit geeigneten Aktoren und Treibern auch für andere Steuerungen verwenden. Der Sensor wurde bereits in der Blogfolge **Füllstandsmesser** und zuvor im Post **Theremin** besprochen. Der VL53L0X ist ein sogenannter Time of Flight-Sensor (ToF). Er sendet Laserimpulse im IR-Bereich aus und misst die "Flugzeit" bis zum Wiedereintreffen nach der Reflexion an einem Hindernis. Die Genauigkeit wird vom Hersteller mit 1mm angegeben, was auf Laborumgebungen zutreffen mag. Im wirklichen Leben und der Reflexion an nicht genormten Flächen, betreffend Farbe und Struktur, bringt man es auf ca. 1cm, was in diesem Fall völlig ausreicht. Interesse geweckt? Dann willkommen zu einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Foto-Auslöser mit Entfernungsmessung

Wie man eine IR-Fernsteuerung auslesen kann, habe ich im ersten Teil ausführlich beschrieben, ebenso das Nachbilden der IR-Impulse durch Software. Das Programm in diesem Post bedient sich wieder der Methode **ausloesen**(), die um die Befehle zum Ein- und Ausschalten einer LED (blau) am Pin GPIO2 ergänzt wurde. Diese LED zeigt somit die Auslösung einer Aufnahme durch einen kurzen Lichtblitz an.

Damit sind wir auch schon bei der benötigten Hardware angekommen. Der Einsatz des VL53L0X erfordert einen ESP32 als Prozessor, ein ESP8266 ist mit dem Umfang des Treibermoduls für den ToF-Sensor hoffnungslos überfordert.

Hardware

2	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
2	KY-004 Taster Modul
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
2	Widerstand 330 Ω
1	Widerstand 220 Ω
2	Widerstand 1kΩ
1	PNP-Transistor BC558 ähnlich
1	LED rot
1	LED grün
1	LED blau
diverse	Jumperkabel
2	MB-102 Breadboard Steckbrett mit 830 Kontakten
1	Logic Analyzer optional

Zur Einstellung des Entfernungsbereichs dienen die beiden Taster. Dabei ist ein OLED-Display hilfreich, denn das System soll ja unabhängig vom PC arbeiten können. Die rote und die grüne LED signalisieren, zusammen mit dem Display die Aktionsbereitschaft und Funktionalität der Tasten. Der PNP-Transistor wird über einen Widerstand von 1kΩ vom Pin GPIO17 angesteuert. Er schaltet durch, wenn GPIO17 auf LOW geht und die Basis auf Kollektor-Potenzial legt. Dadurch wird die IR-LED aktiviert, die über einen Begrenzungswiderstand an GND liegt. Durch die Verwendung des Transistors kann die IR-LED an die 5V-Betriebsspannung gelegt werden. Damit wird eine höhere Energiezufuhr und somit eine größere Reichweite ermöglicht. Die genauen Zusammenhänge zeigt der Schaltplan.

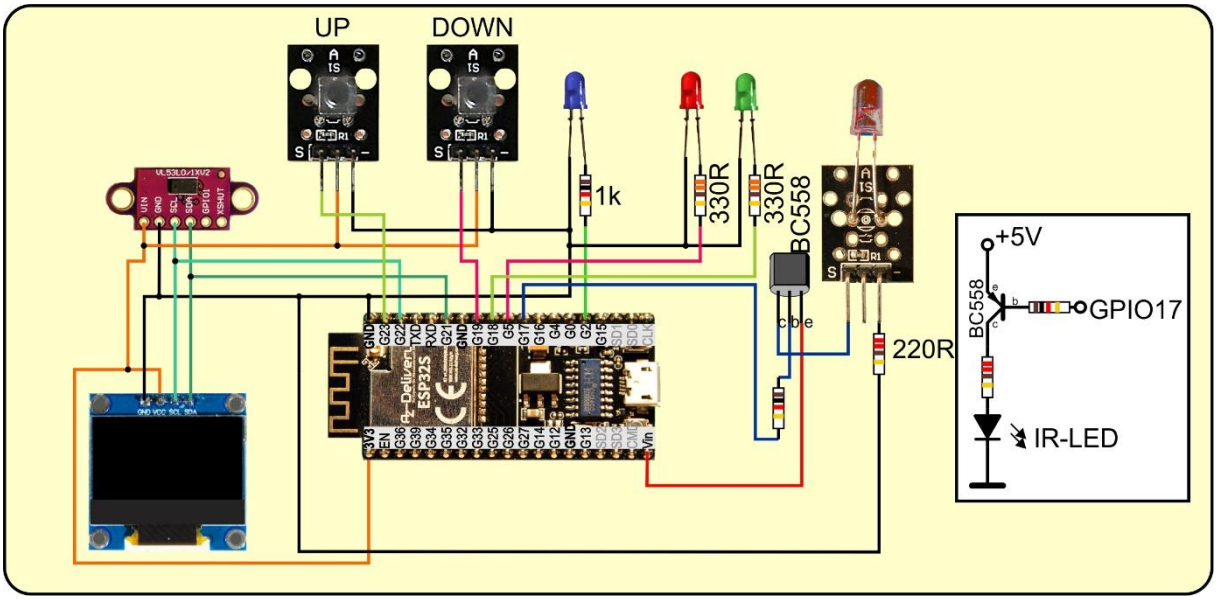

Abbildung 1: Distanz-Sensor Schaltung

Und hier der Aufbau der Testschaltung.

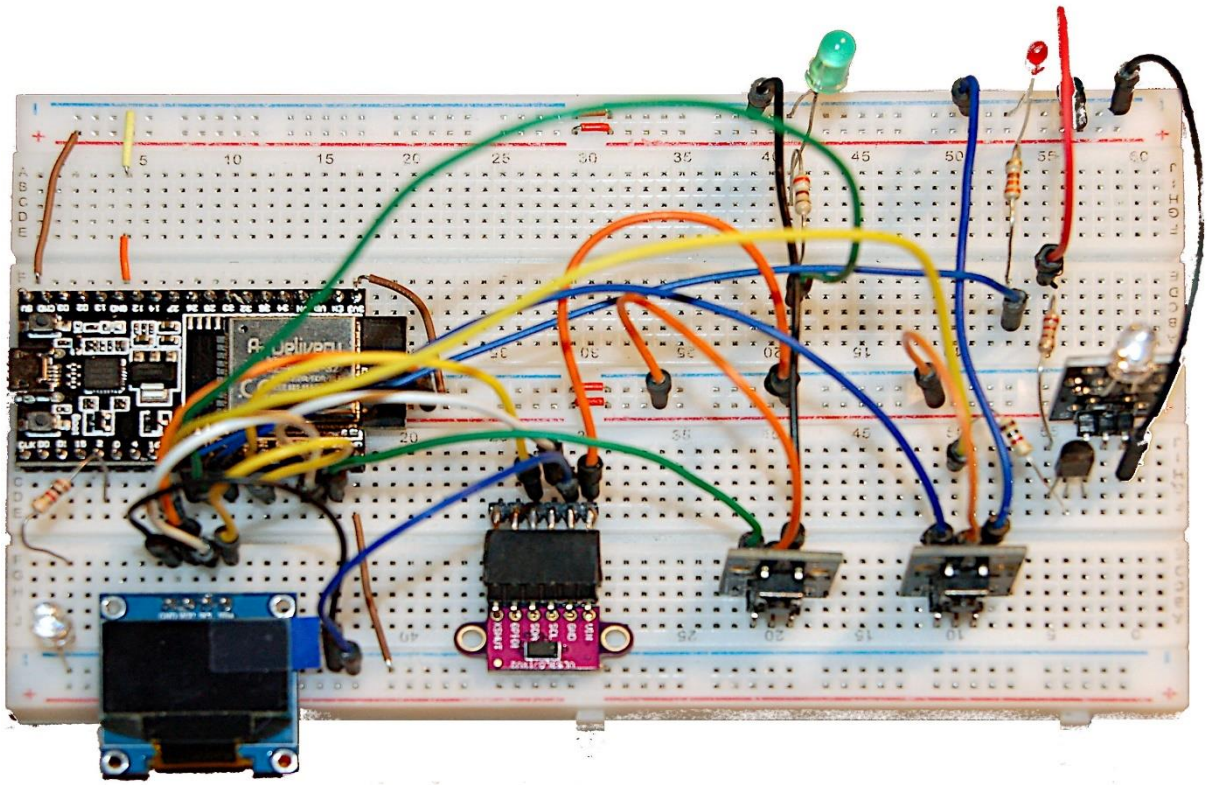

Abbildung 2: Distanz-Sensor Aufbau.jpg

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
SALEAE – Logic-Analyzer-Software für Windows 8, 10, 11

Verwendete Firmware für den ESP32:

MicropythonFirmware

ESP32 mit 4MB Version **1.18** Stand 25.03.2022

Die Versionsnummer ist entscheidend für die Umsetzung des Projekts.
Die 1.18 ist die einzige Version, die sowohl das Modul **bitstream** im Modul **machines** bereitstellt und gleichzeitig den Treiber für den VL53L0X unterstützt.

Durch langwierige Tests mit verschiedenen Controllern und den Firmware-Versionen von 1.15 bis 1.19 habe ich zum Teil gravierende Unterschiede, die Treiberunterstützung betreffend, festgestellt. So beherrscht die Version 1.18 zum Beispiel **bitstream** aber leider nur ein fehlerhaftes PWM, das in 1.19 wieder korrigiert wurde. Dagegen liefert die 1.19 mit dem VL53L0X nur stets denselben schrottigen Messwert, der eigentlich vom Wert her keiner sein kann, weil der Sensor maximal

2000 mm erfassen kann und eben keine 12000 plus! Die MicroPython-Entwickler-Crew flickt also immer wieder einmal Löcher in der Firmware, um andernorts neue aufzureißen. Das führt dann mitunter zu Äußerungen wie: "Verdammt, das hat doch letzten Monat noch prima funktioniert. Wieso funzt das jetzt nich mehr?" Die Antwort findet man dann (vielleicht?) darin, dass entweder im Aufbau der Controller gewechselt wurde oder die Firmware oder beides. Aber Kontinuität ist anders!

Die MicroPython-Programme zum Projekt:

oled.py OLED-Frontend
ssd1306.py OLED-Treibermodul
ir_ausloeser.py Testprogramm für die Auslösesequenz
nikon_timer_nahfeld.py Betriebssoftware
buttons.py Treiber für die Tasten
VL53L0X.py Treiber für den ToF-Sensor

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.

Statische Methoden

Vor etlichen Monaten hatte ich für eine Anwendung ein Modul zur Erfassung von Tasten geschrieben. Darin gab es einige Methoden, die für alle Tasten gebraucht wurden. Diese Methoden wurden als Instanzmethoden implementiert, sodass jedes Tastenelement die gleichen Sequenzen enthielt. Bei vielen Tasten verschwendet das Speicherplatz und deshalb habe ich für dieses Projekt das Modul überarbeitet und aus den Instanzmethoden statische Methoden gemacht. Die werden nun nur einmal im Speicher abgelegt und von allen Tasten benutzt. Wie hängt das zusammen? Ich erläutere es anhand der statischen Methode **buttons.getTouch**() und der Instanz-Methode **buttons.Buttons.ledOn**() aus der Klasse **Buttons**.

```
def getTouch(taste):
    v=taste.tpin.value()
    if taste.invert:
        v=(False if v else True)
    return (True if v==1 else False)
```

Die Methode **getTouch**() ist im Modul **buttons**.py aber außerhalb der Klasse **Buttons** deklariert. Damit daraus eine statische Methode wird, benutze ich innerhalb der Klassendefinition die Decorator-Funktion **staticmethod**(). Dadurch wird **getTouch**() innerhalb der Klasse Buttons bekanntgemacht und kann nun auch durch Objekte der Klasse Buttons referenziert werden.

getTouch=staticmethod(getTouch)

Entscheidend ist, dass es bei statischen Methoden keinen automatischen Bezug auf ein Objekt der Klasse gibt. Bei Instanzmethoden wird dieser Bezug durch das Prefix **self** bei Objekten oder durch den Parameter **self** bei Methoden hergestellt. Während Instanzmethoden innerhalb der Klassendefinition auch mit einem vorangestellten **self** referenziert werden, entfällt das self bei den statischen Methoden.

Jedem Tastenobjekt, das über den Konstruktor der Klasse **Buttons** erzeugt wird, ist unter anderen Attributen auch eine LED über ein Pin-Objekt zugeordnet. Zum Ein- und Ausschalten benutze ich die Instanz-Methoden **ledOn**() und **ledOff**(), die das Vorhandensein einer LED und der Aktivitätsstatus abfragen, bevor sie bedient werden.

Damit **getTouch(taste)** den Zustand einer Taste abfragen kann, muss ich deren Instanz über den Parameter **taste** als Argument übergeben. Anders verhält es sich bei der Instanz-Methode **ledOn**(). Weil die Methode an ein Tastenobjekt gebunden ist, weiß sie auch, welche LED geschaltet werden muss.

Während **getTouch**() nur ein einziges Mal im Speicher vorliegt, gibt es ebenso viele Sequenzen von **ledOn**() wie es Tasteninstanzen gibt.

Wenn ich **buttons**.py im Editorfenster starte, wird es als Hauptprogramm ausgeführt und deshalb wird nicht nur die statische Methode **getTouch**() definiert sondern am Ende werden auch die Tastenobjekte t und s instanziiert.

```
if __name__ == "__main__":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=100000)
    from oled import OLED
    d=OLED(i2c)

t=Buttons(23,invert=True,name="up",ledPin=18,active=1,d=d,x=0,
y=1)

s=Buttons(19,invert=True,name="down",ledPin=5,active=1,d=d,x=0
,y=1)
```

Nun untersuchen wir Folgendes:

>>> **t.getTouch**
<*function getTouch at 0x3ffe5240*>

>>> **s.getTouch**
<*function getTouch at 0x3ffe5240*>

Wir erkennen, dass in beiden Fällen die Funktionsdefinition an derselben Speicherstelle zu finden ist.

>>> **t.ledOn**
<*bound_method*>

Abbildung 3: Objekt-Inspektor t

>>> **s.ledOn**
<*bound_method*>

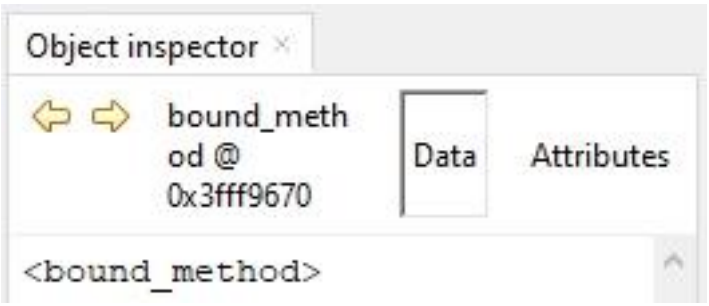

Abbildung 4: Objekt-Inspektor s

Im Objektinspektor, den ich über das Menü **View – Object inspector** aktiviere, kann ich mich davon überzeugen, dass **ledOn**() für t und s an verschiedenen Speicherpositionen liegt.

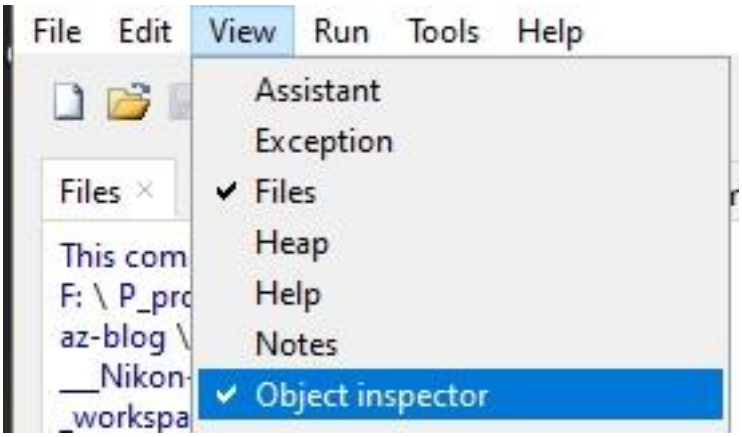

Abbildung 5: View-Objekt inspector

In ähnlicher Weise sind auch die Funktionen **anyKey(), getValue(), TimeOut(), counter(), waitForAnyKey(), waitForTouch()** und **waitForRealease()** als statische Methoden definiert. Die API des Moduls **buttons** ist im Programmtext **buttons**.py kurz beschrieben.

Das Programm

Das Hauptprogramm an sich ist mit knapp 20 Zeilen nicht spektakulär. In der Mainloop wird im Dauerlauf fünfmal die Entfernung gemessen und daraus ein Mittelwert in der Dimension cm berechnet. Weil die Werte vom VL53L0X-Modul in mm geliefert werden ergibt sich so der Teiler 50.

Liegt dieser Messwert im Entfernungsfenster zwischen dem unteren Wert **bottom** und dem oberen Wert **top**, dann werden in Abständen von 2 Sekunden insgesamt drei Fotos ausgelöst. Danach folgt eine Pause von 4 Sekunden, damit die Bilder auch sicher abgespeichert werden können. Das Display informiert über die jeweilige Phase, **RUNNIG** oder **PAUSE**.

Das Ende der Schleife bildet die Abfrage der Flash-Taste am ESP32-Modul. Das Programm wird abgebrochen, wenn die Taste gedrückt wurde.

Den Rest schauen wir uns von oben nach unten an. Wir starten mit dem Import der nötigen Module, Klassen und Methoden.

```
# nikon_timer_nahfeld.py
#
import sys
from time import sleep_ms,ticks_ms, sleep_us
from machine import SoftI2C, Pin, bitstream
from oled import OLED
from ssd1306 import SSD1306_I2C
import VL53L0X
import buttons

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

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

taste=Pin(0,Pin.IN)
auf=buttons.Buttons(23,invert=True,name="up",ledPin=18,active=
1,d=d,x=0,y=1)
ab=buttons.Buttons(19,invert=True,name="down",ledPin=5,active=
1,d=d,x=0,y=1)
keys=(auf,ab)

timeout=buttons.Buttons.TimeOut

out=Pin(17,Pin.OUT,value=0)  # IR-LED
control=Pin(2,Pin.OUT,value=0) # Kontroll-LED

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)
```

Der I2C-Bus bekommt seine Pins zugewiesen und mit dem i2c-Objekt wird das OLED-Objekt d instanziiert. Dieses benötigt das Modul **ssd1306**, welches zusammen

mit **buttons**, **oled** und **VL53L0X** in den Flash des ESP32 hochgeladen werden muss.

Als Abbruchtaste definiere ich die Flashtaste an GPIO0, ferner die Button-Instanzen **auf** und **ab**. Die Tastenmodule KY-004 sind LOW-aktiv, weshalb ich den Parameter **invert** auf True setze. Die Methode **getTouch**() liefert mir dadurch eine 1, wenn die Taste gedrückt wird. Die Tasten erhalten Namen und eine LED zugewiesen. Außerdem übergebe ich das Displayobjekt d und die Positionen, an denen Ausgaben zur Tastensteuerung erscheinen sollen. Die Tasteninstanzen fasse ich im Tuple **keys** zusammen, damit ich via der Methoden **anyKey**() und **waitForAnyKey**() beide Tasten in einem Abwasch behandeln kann.

Damit die Closure TimeOut kürzer aufgerufen werden kann, definiere ich für die statische Methode das Alias **timeout**.

Die IR-LED bekommt den GPIO17 als Ausgang und die Kontroll-LED hänge ich an GPIO2. Am ESP32S kann man sich diese LED sparen, weil bereits eine HIGH-aktive blaue LED auf dem Board vorhanden ist. Der ESP32 Dev Kit V4 besitzt keine Onboard-LED.

Die Deklaration des ToF-Sensors schließt die Vorbereitungen ab.

Eine Herausforderung stellte die Konfiguration des "heißen" Bereichs für die Entfernung dar. Unter- und Obergrenze sollten nur durch zwei Tasten einstellbar und rekonfigurierbar sein. Weil der Bereich immerhin über 200 einstellbare Stufen verfügt, sollten ein Einzel- und ein Mehrschrittmodus implementiert werden.

Die Funktion **config**() erledigt das zusammen mit dem Modul **buttons**.py und dem Display ganz gut. Eine weitere Zutat bringt der nicht blockierende Softwaretimer **TimeOut**() ein, der dafür sorgt, dass nach einer bestimmten Zeit ohne Tastenbetätigung der zuletzt eingestellte Wert automatisch übernommen wird. Nach jeder Tastenbetätigung wird je nach Nummer der Taste, 0 oder 1, der Grenzwert hinauf oder herunter gezählt und danach der Ablauftimer neu gestellt. Die Zeile

```
taste=buttons.waitForAnyKey(keys,300)
```

wartet auf eine Tastenbetätigung, fügt eine kurze Verzögerung ein und gibt die Tastennummer zurück. Wird eine Taste dauerhaft gedrückt, dann schaltet die while-Schleife nach dem 5. Durchlauf auf Turbobetrieb. Von 1 wird die Schrittweite dann auf 5 erhöht. Mit jedem Durchlauf wird der Zählerstand angezeigt. Wenn keine Taste gedrückt ist, verringert sich die Schrittweite wieder auf 1.

Nachdem **bottom** und **top** eingestellt sind, werden die beiden Werte als Klartext in der Datei **config.txt** abgelegt. Die **with**-Anweisung sorgt dafür, dass die Datei nach dem Durchlaufen des Blocks automatisch geschlossen wird.

```
def config(delay=300):
    global bottom, top
    d.clearAll()
    d.writeAt("NIKON-TIMER",2,0,False)
    d.writeAt("UNTERGRENZE:",1,1,False)
```

```
    d.writeAt("{}".format(bottom),4,2,False)
    d.writeAt("UP/DOWN/None",3,3)
    ende = timeout(5000)
    sleep_ms(500)
    while not ende():
        taste=buttons.waitForAnyKey(keys,300)
        if taste is not None:
            n+=1
            if n<5:
                bottom=(bottom+1 if taste==0 else bottom-1)
            else:
                bottom=(bottom+5 if taste==0 else bottom-5)
            d.writeAt("{}   ".format(bottom),4,2)
            sleep_ms(delay)
            ende=timeout(5000)
        else:
            n=0
    d.clearAll()
    d.writeAt("NIKON-TIMER",2,0,False)
    d.writeAt("OBERGRENZE:",1,1,False)
    d.writeAt("{}   ".format(top),4,2)
    d.writeAt("UP/DOWN/None",3,3)
    ende = timeout(5000)
    while not ende():
        taste=buttons.waitForAnyKey(keys,300)
        if taste is not None:
            n+=1
            if n<5:
                top=(top+1 if taste==0 else top-1)
            else:
                top=(top+5 if taste==0 else top-5)
            d.writeAt("{}   ".format(top),4,2)
            sleep_ms(delay)
            ende=timeout(5000)
        else:
            n=0
    with open("config.txt","w") as f:
        f.write("{}\n".format(bottom))
        f.write("{}\n".format(top))
```

Die Funktion **auslösen**() bildet die Impulsfolge der offiziellen Nikon-IR-Steuerung nach. Sie wurde im ersten Teil bereits detailliert beschrieben. Dort ist auch erklärt, wie man mit Hilfe eines Logic Analizers und einer IR-Empfangsdiode ein serielles IR-Signal abtasten kann.

```
def ausloesen():
    control.value(1)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*9)+b'\xf0')
    out(1)
    sleep_us(27800)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
```

```
    out(1)
    sleep_us(1580)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(3480)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(63100)
    control.value(0)
```

Bei jedem Programmstart wird geprüft, ob die Datei **config.txt** existiert. Ist das der Fall, dann werden die Werte für **bottom** und **top** eingelesen. Sonst werden Standardwerte dafür gesetzt und abgespeichert. Einen Fehler beim Öffnen oder Lesen der Datei fange ich mit **try – except** ab.

```
# Lesen oder setzen der Konfiguration
try:
    with open("config.txt","r") as f:
        bottom=int(f.readline())
        top=int(f.readline())
except:
    bottom=20 # cm
    top=60    # cm
    with open("config.txt","w") as f:
        f.write("{}\n".format(bottom))
        f.write("{}\n".format(top))
```

Das Display wird gelöscht und informiert dann über die nächsten Schritte. Der Parameter für die Wartezeit (delay) der Methode **waitForAnyKey**() ist auf 5 Sekunden voreingestellt und muss deshalb nicht angegeben werden.

```
d.clearAll()
d.writeAt("NIKON-TIMER",2,0,False)
d.writeAt("UP-Button",3,2,False)
d.writeAt("to run config",1,3)
d.writeAt("DOWN-Button",2,4,False)
d.writeAt("to start",4,5)
# 5 Sekunden auf Taste warten
if buttons.waitForAnyKey(keys) == 0:
    config(300)
```

Dann starte ich den VL53L0X-Prozess. Das Display meldet **RUNNING** und gibt die Grenzwerte aus. Danach beginnt die Hauptschleife der Anwendung, die wir ja bereits besprochen haben.

Zum Schluss noch das gesamte Programm, das Sie herunterladen können im Zusammenhang.

```
# nikon_timer_nahfeld.py
```

```
#
import sys
from time import sleep_ms,ticks_ms, sleep_us
from machine import SoftI2C, Pin, bitstream
from oled import OLED
from ssd1306 import SSD1306_I2C
import VL53L0X
import buttons

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

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

taste=Pin(0,Pin.IN)
auf=buttons.Buttons(23,invert=True,name="up",ledPin=18,active=
1,d=d,x=0,y=1)
ab=buttons.Buttons(19,invert=True,name="down",ledPin=5,active=
1,d=d,x=0,y=1)
keys=(auf,ab)
timeout=buttons.Buttons.TimeOut

out=Pin(17,Pin.OUT,value=0)  # IR-LED
control=Pin(2,Pin.OUT,value=0) # Kontroll-LED

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)

def config(delay=300):
    global bottom, top
    d.clearAll()
    d.writeAt("NIKON-TIMER",2,0,False)
    d.writeAt("UNTERGRENZE:",1,1,False)
    d.writeAt("{}".format(bottom),4,2,False)
    d.writeAt("UP/DOWN/None",3,3)
    ende = timeout(5000)
    sleep_ms(500)
    while not ende():
        taste=buttons.waitForAnyKey(keys,300)
        if taste is not None:
            n+=1
            if n<5:
                bottom=(bottom+1 if taste==0 else bottom-1)
            else:
                bottom=(bottom+5 if taste==0 else bottom-5)
            d.writeAt("{}   ".format(bottom),4,2)
            sleep_ms(delay)
            ende=timeout(5000)
        else:
            n=0
```

```
    d.clearAll()
    d.writeAt("NIKON-TIMER",2,0,False)
    d.writeAt("OBERGRENZE:",1,1,False)
    d.writeAt("{}   ".format(top),4,2)
    d.writeAt("UP/DOWN/None",3,3)
    ende = timeout(5000)
    while not ende():
        taste=buttons.waitForAnyKey(keys,300)
        if taste is not None:
            n+=1
            if n<5:
                top=(top+1 if taste==0 else top-1)
            else:
                top=(top+5 if taste==0 else top-5)
            d.writeAt("{}   ".format(top),4,2)
            sleep_ms(delay)
            ende=timeout(5000)
        else:
            n=0
    with open("config.txt","w") as f:
        f.write("{}\n".format(bottom))
        f.write("{}\n".format(top))


def ausloesen():
    control.value(1)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*9)+b'\xf0')
    out(1)
    sleep_us(27800)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(1580)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(3480)
    bitstream(out,0,(0,0,13160,13160),(b'\xff'*1)+b'\xfe')
    out(1)
    sleep_us(63100)
    control.value(0)

# Lesen oder setzen der Konfiguration
try:
    with open("config.txt","r") as f:
        bottom=int(f.readline())
        top=int(f.readline())
except:
    bottom=20 # cm
    top=60    # cm
    with open("config.txt","w") as f:
        f.write("{}\n".format(bottom))
        f.write("{}\n".format(top))
```

```
d.clearAll()
d.writeAt("NIKON-TIMER",2,0,False)
d.writeAt("UP-Button",3,2,False)
d.writeAt("to run config",1,3)
d.writeAt("DOWN-Button",2,4,False)
d.writeAt("to start",4,5)
# 5 Sekunden auf Taste warten
if buttons.waitForAnyKey(keys) == 0:
    config(300)

# VL53L0X starten
tof.start()
print("started")
d.clearAll()
d.writeAt("NIKON-TIMER",2,0,False)
d.writeAt("RUNNING",4,2,False)
d.writeAt("UG = {} cm".format(bottom),2,3,False)
d.writeAt("OG = {} cm".format(top),2,4)

while True:
    distance=0
    for i in range(5):
        distance=distance+tof.read()
    distance=(distance+5)//50
    if (bottom <= distance <= top):
        for i in range(3):
            ausloesen()
            print("Foto {} bei {} cm".format(i,distance))
            sleep_ms(2000)
        d.writeAt(" PAUSE ",4,2)
        sleep_ms(4000)
        d.writeAt("RUNNING",4,2)
    sleep_ms(50)
    if taste.value()==0:
        d.clearAll()
        d.writeAt("NIKON-TIMER",2,0,False)
        d.writeAt("TERMINATED",3,2)
        sys.exit()
```

Natürlich gibt es auch wieder den Blogbeitrag als PDF zum Download.