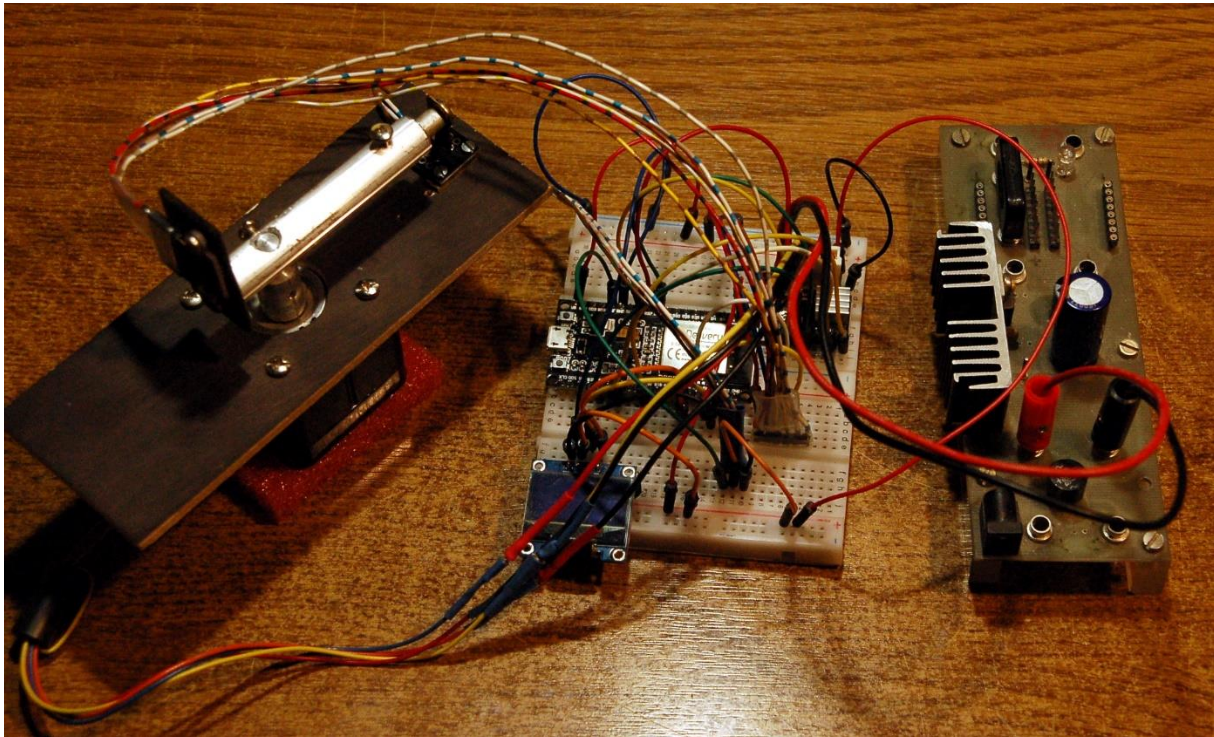

Aufbau des Lichtradars (mit Eigenbau-Netzteil 12V – 5V)

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

In den bisherigen Folgen hatten wir uns den Aufbau und die grundsätzliche Funktion von Schrittmotoren angeschaut und uns mit Treiber-ICs beschäftigt, die uns eine Menge an Programmierarbeit abnehmen. Heute stricken wir, zusammen mit einem Time-Of-Flight-Sensor (TOF), daraus eine Anwendung, die das Profil eines Raumes abtastet, so wie es die Radaranlagen auf Flugplätzen tun. Laut Datenblatt soll der VL53L0X eine Reichweite von bis zu zwei Metern haben. Mein Exemplar funktioniert leider nur im Umkreis bis ca. 1,6 Meter. Dennoch ist das Ergebnis faszinierend. Was ein TOF ist und wie man ihn einsetzen kann, das erfahren Sie in dieser neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Das Licht-Radar

Ein TOF-Sensor misst die Laufzeit von Lichtimpulsen und berechnet daraus die Entfernung zu dem Objekt, welches den Lichtpuls reflektiert. Die Wellenlänge des ausgesandten Lichts liegt im Infrarot-Bereich (IR) und kann vom menschlichen Auge nicht wahrgenommen werden. Aber mit der Kamera eines Handys lässt sich das ausgesandte Licht aufspüren.

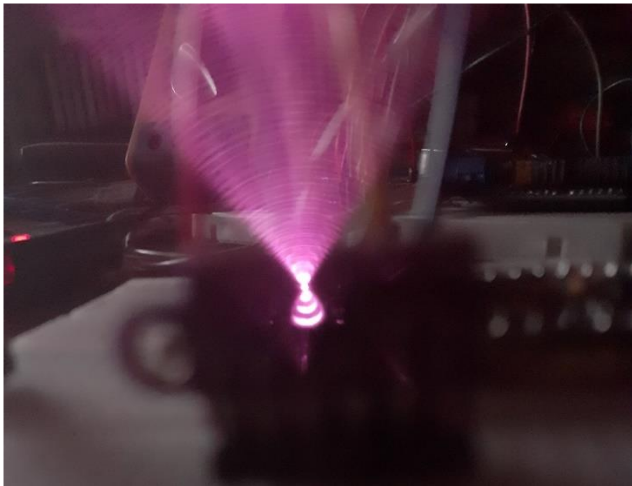

Abbildung 1: Der IR-Lichtkegel des VL53L0X

Auf diese Weise lässt sich übrigens auch die Funktion einer normalen IR-Fernsteuerung von TV-Geräten, Stereoanlagen etc. überprüfen.

Den VL53L0X habe ich schon einige Male zur Entfernungsmessung eingesetzt, zum Beispiel beim Theremin und bei der Messung der Körpergröße von Personen. Dort war der Sensor ruhend montiert. Heute befestigen wir ihn auf der Rotorachse des Schrittmotors, sodass er einen (fast) beliebigen Winkelbereich abtasten kann. Nachdem ein Winkelschritt ausgeführt ist, messen wir die Entfernung zum nächstliegenden Gegenstand. Der ESP32 rechnet diesen Wert herunter, damit er als "Fahrstrahl" auf einem OLED-Display dargestellt werden kann. Eine Gabellichtschranke oder ersatzweise ein Reed-Kontakt dienen als Endpositionssensor. So kann der ESP32 den Rotor, nach dem Booten der Anlage, stets in eine definierte Startposition bringen. Das ist das heutige Programm. Wir starten mit der Hardwareliste sofort durch.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	A4988 Schrittmotor-Treiber-Modul mit Kühlkörper
	oder DRV8825 Schrittmotor-Treiber-Modul mit Kühlkörper
1	Schrittmotor bipolar z.B. Pollin Best.-Nr. 310689 oder 310690
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	VL53L0X Time-of-Flight (ToF) Laser Abstandssensor
1	KY-010 Lichtschranke oder KY-021 Magnet Schalter Mini Magnet Reed Modul Sensor
1	LM2596S DC-DC Netzteil Adapter Step down Modul
20 mm	12mm Ø Alu-Rundstab zur Verlängerung der Rotorachse
17 mm	6mm Ø Alurundstab
65 mm	12mm Ø Alu-Rundstab als Sensortraghebel
30 mm	6mm Ø Alurundstab
1	4mm Sperrholzplatte 55mm x 140mm
1	Pertinax-Platte 20mm x 40mm x 1,5mm
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
1	Ultraflexibles verzinntes Silikonkabel Schaltlitzen Kit
diverse	Jumperkabel
1	Steckernetzteil 12V DC / 2A
Optional	Logic Analyzer

Der Logic Analyzer ist sehr nützlich, wenn es um die Überwachung der Logikleitungen geht, hier speziell zur Lösung von Problemen im Zusammenhang mit dem I2C-Bus.

Für das vorliegende Projekt sind die ESP8266-Module leider nicht geeignet. Ich musste feststellen, dass bereits beim Einbinden des MicroPython-Moduls für den VL53L0X ein Speicherfehler gemeldet wurde. Auch für den ESP32 gab es eine Überraschung, dazu weiter unten mehr.

Die angegebenen Motoren arbeiten bei etwa 10,5V. Wir müssen also ein 12V-Netzteil verwenden und die Stromstärke am Treibermodul A4988 oder DRV8825 mit dem Trimmer, wie in der [letzten Folge](#) beschrieben, einstellen. Alternativ könnte man auch zwei Dioden vom Typ 1N4001 seriell in Durchlassrichtung in +12V-Zuleitung legen. Durch Abzug der Vorwärtsspannung von zweimal 0,7V bleiben für den Motor ca. 10,5V übrig.

Wenn der ESP32 auch aus dieser Versorgung betrieben werden soll, können wir über das Buck-Converter-Modul die 12V auf 5V herunterregeln, **bevor** der Ausgang des Moduls an den ESP32 angeschlossen wird. Der Converter wird nicht benötigt, wenn der ESP32 über die USB-Leitung aus dem PC versorgt wird.

Vorbereiten der Motoreinheit

Abbildung 3 zeigt den Aufbau der Motoreinheit, die wir als Erstes besprechen.

Die Grundplatte wird so vorbereitet, dass sie auf den Motor passt. Für die genannten Motoren gelten die Maße in Abbildung 2. Die Befestigungslöcher bohren wir mit 4mm, um Toleranzen in der Positionierung auszugleichen. Die Platte wird nun mit vier Schrauben M3 x 10 auf den Motorblock des Minebea T15317-01 geschraubt.

Beim Minebea T15429-02 muss das Ritzel mit einer Mini-Trennscheibe angeschnitten und mit einem Flachschraubendreher abgesprengt werden.

Abbildung 2: Bohrungen für die Motoraufnahme

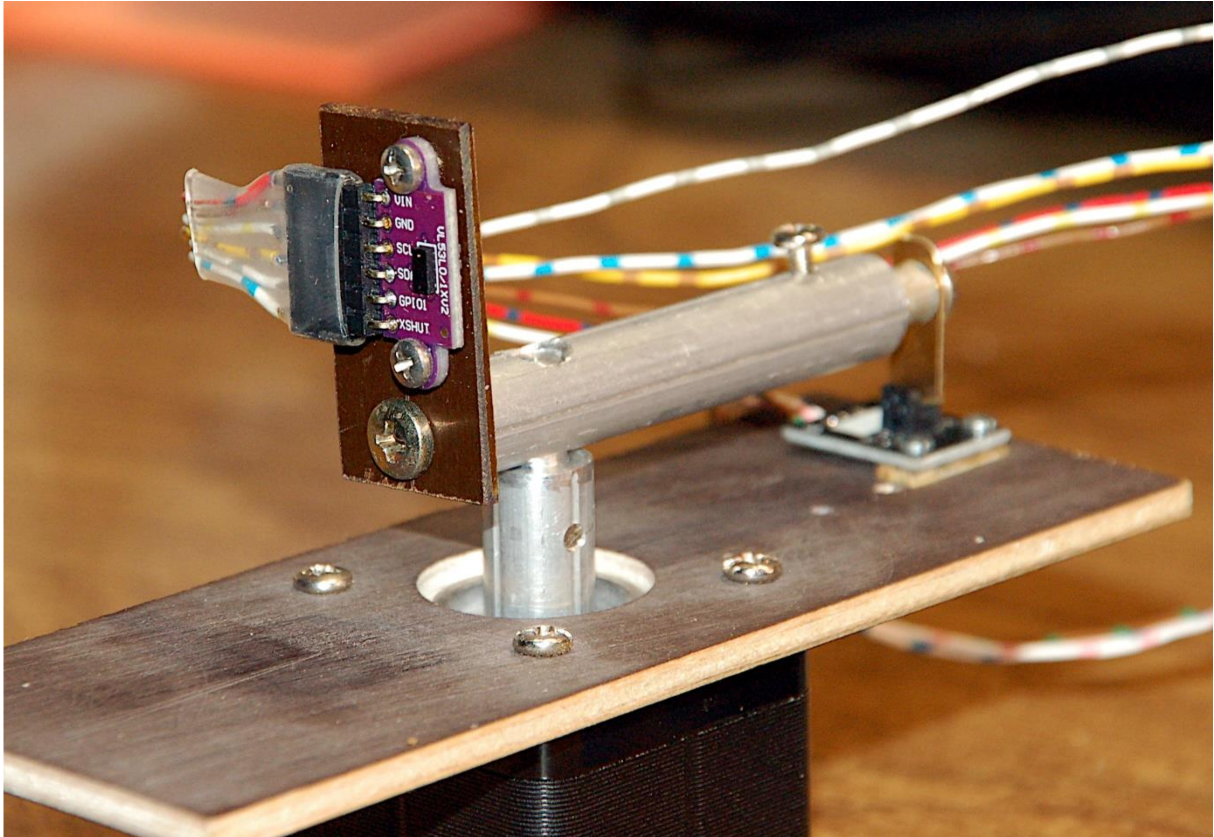

Abbildung 3: VL53L0X auf der Motorwelle montiert

Der 20mm-Rundstab (12mm Ø) wird mit einer zentrischen 4mm-Bohrung versehen. Anschließend wird vom oberen Ende auf eine Länge von 7mm mit einem 6mm-Bohrer aufgeweitet, sodass später der 20mm-Rundstab (6mm Ø) eingeklebt (2-Komponentenkleber) werden kann. Von unten bohren wir auf 6mm Länge auf 5mm Durchmesser auf. Etwa mittig zur verbliebenen 4mm-Bohrung habe ich eine Bohrung mit 2,5mm senkrecht zur Achse angebracht und mit einem Gewinde M3 versehen. Jetzt kann ich die Verlängerung auf die Motorachse setzen und festziehen. Nun kann man den 20mm-Rundstab einkleben und den Kleber aushärten lassen.

Den Rundstab für den Hebel bohre ich 15mm vom Ende quer mit 6mm mittig durch. Das linke Ende versehe ich mit einer 3,3mm-Bohrung, in die ein Gewinde M4 einziehe. Das rechte Ende wird auf eine Länge von 40mm auf 6mm aufgebohrt, und mit einem Querloch von 2,5mm versehen. In dieses ziehe ich ein Gewinde M3 ein.

Den 30mm-Rundstab bohre ich mit 3,3mm axial für ein Gewinde M4 vor. Der Hebel wird auf die verlängerte Achse geklebt (2-Komponentenkleber wie oben). Am kurzen Ende (links) befestige ich die Pertinaxplatte mit dem VL53L0X und am 30mm-Rundstab schraube ich ein kleines Stück Blech fest, das zwischen die Backen der Gabellichtschranke passen muss.

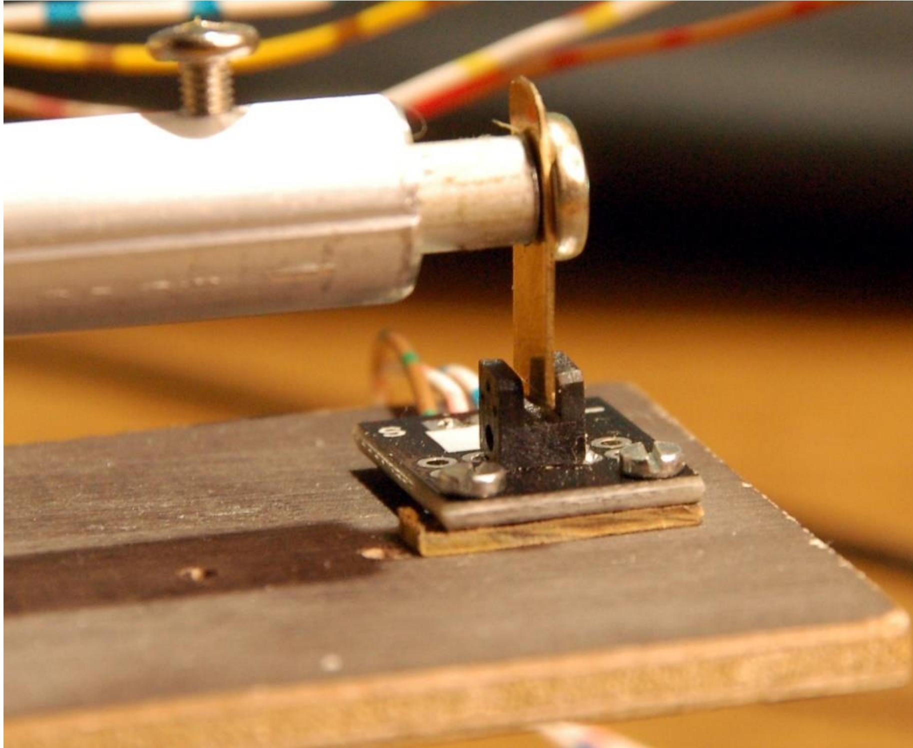

Abbildung 4: Lichtschranke zum Anfahren der Startposition

Damit der Hebel in alle Richtungen ohne Behinderung drehen kann, müssen von der Platine der Gabellichtschranke die Anschlussstifte entfernt werden. Ersatzweise werden Kabelstücke (30cm) direkt angelötet, an deren Ende wieder die Stiftleiste angebracht werden kann. Zum Anpassen des Blechstücks am Hebel montieren wir das Modul mit einem Abstandshalter (ein Streifen Pertinax) auf der Grundplatte.

Auch für das VL53L0X-Modul brauchen wir eine Verbindung, die wir aus sechs Kabelstücken von ca. 30cm Länge herstellen und mit einer Buchsen- und einer Stiftleiste abschließen.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender zum Testen des ESP8266 als UDP-Client und -Server
SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für einen ESP32:

ESP32_GENERIC-20220117-v1.18.bin
Bitte verwenden Sie nur diese Firmware, weil nur damit der VL53L0X korrekt funktioniert – warum auch immer!? Es scheint an der I2C-Klasse zu liegen.

Die MicroPython-Programme zum Projekt:

oled.py Display-API
ssd1306.py OLED-Hardwaretreiber
stepper.py MicroPython-Modul
VL53L0X.py Treibermodul
lightradar.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 Licht-Radar

Im Betriebsprogramm arbeiten wir an zwei Stellen mit Interrupts (IRQ = Interrupt Request = Unterbrechungsanforderung).

Ein Timer-IRQ sorgt für den Betrieb des Motors. In mehr oder weniger gleichen Zeitintervallen wird ein IRQ ausgelöst. Die entsprechende Interrupt-Service-Routine (ISR) verwaltet einen eigenen Zähler, durch dessen Ablauf die Zeitdauer zwischen zwei Step-Impulsen festgesetzt wird.

Beim Programmstart wird die Motorachse in eine rasche Drehung im Gegenuhrzeigersinn versetzt. Damit diese beim Eintritt des Blechs in die Lichtschranke automatisch gestoppt wird, lassen wir durch die ISR eines Pin-Change-IRQs den Timer anhalten und beenden damit (vorübergehend) die Rotation. Ein Pin-Change-IRQ tritt auf, wenn an einem GPIO-Eingang ein Pegelwechsel passiert. Genau dafür sorgt die Lichtschranke oder ersatzweise der Reed-Kontakt.

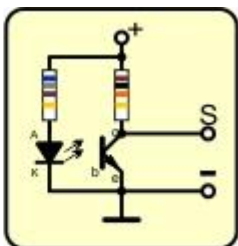

Abbildung 5: Gabellichtschranke - Schaltung

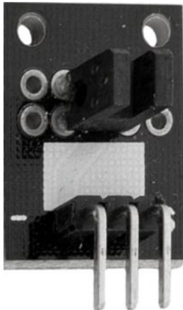

Abbildung 6: Lichtschranken-Modul

In einem der beiden Backen der Lichtschranke sitzt eine IR-LED. Der Fototransistor gegenüber schaltet durch, wenn ihn das Licht der LED trifft. Dadurch geht S auf GND-Pegel. Ein undurchsichtiges Hindernis in der Gabel schattet den Transistor ab wodurch dieser sperrt. Der Widerstand zieht S auf Vcc-Potenzial.

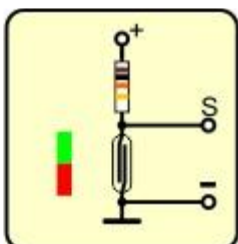

Abbildung 7Reed-Kontakt - Schaltung

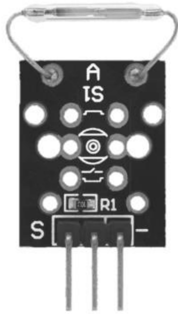

Abbildung 8: Reed-Kontakt mini - Modul

Beim Reed-Kontakt liegen sich in einer Schutzatmosphäre zwei Weicheisen Kontakte nah gegenüber, ohne sich zu berühren, S wird durch den Widerstand auf Vcc gezogen. Nähert man einen kleinen Stabmagneten, dann werden die Kontakte durch magnetische Influenz vorübergehend selbst zu Magneten, die sich gegenseitig anziehen und damit den Kontakt schließen, S geht auf GND-Potenzial. Sollten Sie diese Lösung verwenden, **ein Tipp zur Vorsorge**:

Biegen Sie bitte nicht zu sehr an den Anschlussdrähten des Reedkontakts herum, die sind recht störrisch. Schnell pflegen dann die Durchführungen am Glaskörper auszubrechen und das Bauteil ist kaputt.

Die gesamte Schaltung zeigt Abbildung 9.

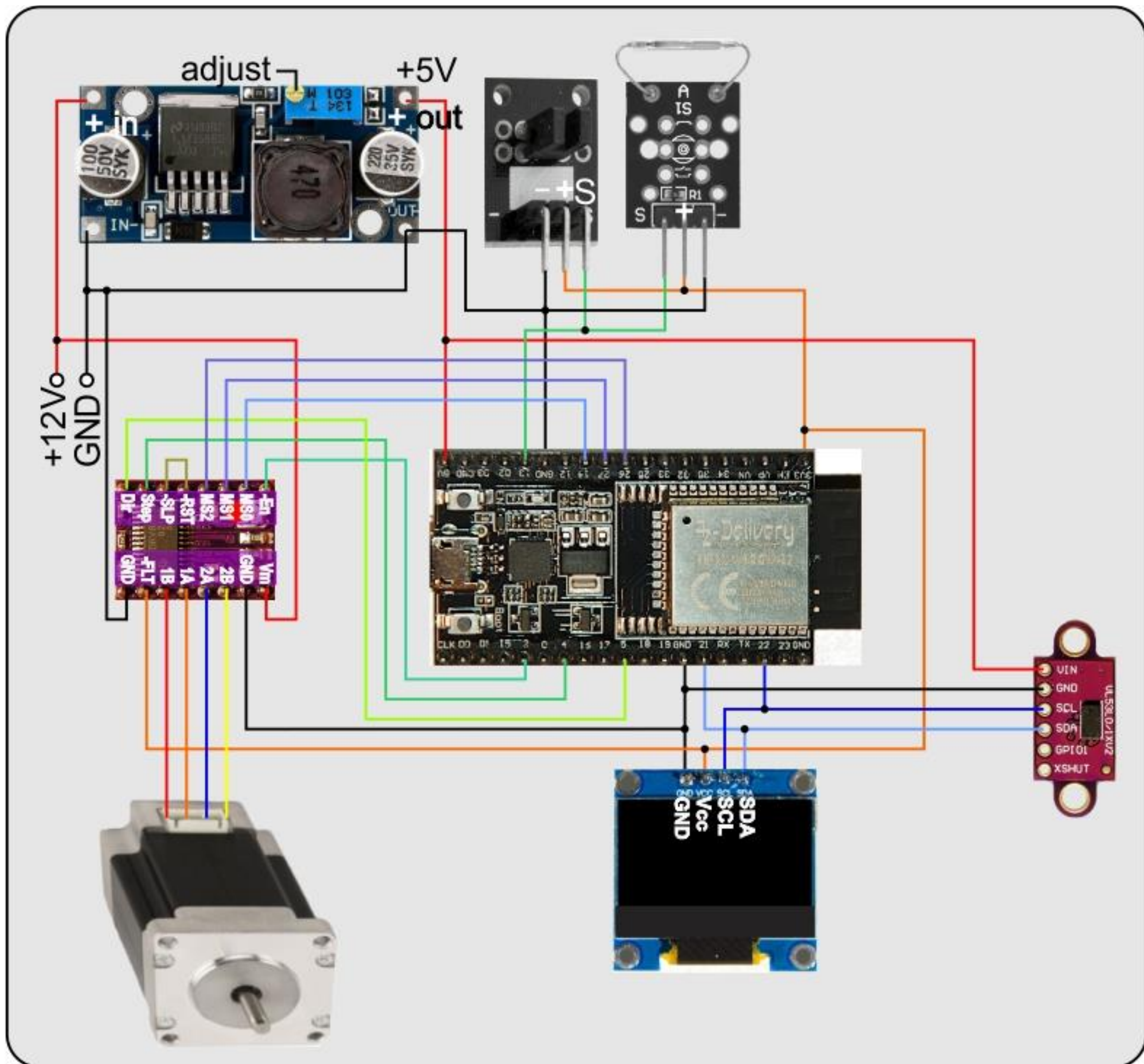

Abbildung 9: Lichtradar - Schaltung

Das Programm

Das Programm **lightradar.py** ist mit seinen 18 Zeilen "nur" der Dirigent in diesem Konzert. Spielen werden die Musiker, sprich Methoden, in den einzelnen Klassen, die wir eingangs einladen müssen, mitzumachen. Ein paar Interpreten, Funktionen, bringt der Chef selber mit. Und natürlich muss er die Instrumente, GPIO-Pins und Datenstrukturen, bestimmen, mit denen gespielt werden soll.

Die GPIO-Pins finden wir im Modul **machine**, ebenso die Klasse **SoftI2C**, welche die Grundlagen für die Unterhaltung des ESP32 mit dem VL53L0X und dem OLED-Display liefert. Für kurze Atempausen sorgt die Funktion **sleep**() aus dem Modul **time**. Die Funktion **exit**() aus dem Modul **sys** erlaubt uns, das Spiel geordnet zu beenden, nachdem aufgeräumt wurde.

```
from machine import Pin,SoftI2C
from time import sleep
from sys import exit
from oled import OLED
import VL53L0X
from stepper import DRV8825
```

Die Klasse **OLED** aus dem Modul **oled** liefert Methoden, welche die Ansteuerung des Displays einfach machen und die Klasse **VL53L0X** fasst die komplexen Vorgänge des Chats mit dem TOF-Sensor zusammen. Das Modul von der Website https://github.com/uceeatz/VL53L0X/blob/master/VL53L0X.py musste für den ESP32 angepasst werden, weil es für wipy-Boards vom Pycom geschrieben wurde. Verwenden Sie daher bitte meine Version. In diesem Zusammenhang erinnere ich daran, den ESP32 unbedingt mit der 18-er Revision der Firmware zu flashen, weil spätere Versionen bereits beim Start des VL53L0X mit Fehlermeldungen abbrechen.

Die Musiker, Verzeihung, Methoden in der Klasse DRV8825 habe ich schon im vorangegangenen Beitrag beschrieben. Sie dienen der Motorsteuerung, die, wie oben schon angedeutet, im Hintergrund durch Timer-IRQ-Steuerung abläuft. Die Klasse DRV8825 erbt von der Klasse A4988, welche die Hauptarbeit leistet. In DRV8825 werden lediglich die Methode **setMode**() und die Modustabelle **ModeTable** überschrieben. Das Programm arbeitet auch ohne Probleme und ohne Änderungen mit dem A4988 zusammen, wenn sie die Importzeile auf folgende Weise anpassen:

```
from stepper import A4988
```

Die nächsten Zeilen verteilen die GPIO-Pins

```
# I2C-Bus
SCL=Pin(22)
SDA=Pin(21)
```

```
# Motorsteuerung
Dir=5
Step=4
Enable=2
M0=14
M1=27
M2=26
referenz=Pin(13)
StepsPerRev=200
```

Ich verwende einen Motor mit 200 Vollschritten pro Umdrehung.

Die Flashtaste am ESP32 nehme ich gerne als Fluchttaste. Während der Entwicklung ermögliche ich dadurch einen sauberen Programmausstieg. Falls sie einen anderen GPIO-Pin benutzen möchten, habe ich vorsorglich den Pin als Eingang mit Pullup deklariert. Bei GPIO0 ist das nicht nötig, da er einen externen Pullup von 10kΩ hat und als Eingang initialisiert wird.

```
# secure exit
taste=Pin(0, Pin.IN, Pin.PULL_UP)
```

Wir erzeugen ein I2C-Bus-Objekt, das wir gleich dem Konstruktor des Display-Objekts übergeben. Wir verwenden ein Display mit 128 x 64 Pixeln.

```
i2c=SoftI2C(SCL,SDA,freq=100000)

d=OLED(i2c,widthw=128,heightw=64)
```

Auch der Konstruktor der VL53L0X-Klasse bekommt das Bus-Objekt zugewiesen. Die Hardware-Adressen der Bausteine sind in den Klassen bereits standardmäßig festgelegt. Sie können über **i2c.scan**() in REPL abgerufen werden. Eingaben formatiere ich immer fett, die Ausgaben kursiv.

>>> i2c.scan()
[41, 60]

Der VL53L0X ist unter 41 = 0x29, das Display unter 60 = 0x3C ansprechbar. Dieser erste Test signalisiert die grundsätzliche Bereitschaft der beiden Geräte am Bus. Instanziieren wir nun das VL53L0X-Objekt. Ich halte mich auch hier an das Beispiel von https://github.com/uceeatz/VL53L0X/blob/master/main.py.

```
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()
distance = tof.read()
print (distance)
```

Mit dem Aufruf von **start**() calibriert der VL53L0X. Genau hier ereignet sich ein Timeout-Fehler, wenn eine andere Firmware als ESP32_GENERIC-20220117-v1.18.bin geflasht ist. Wir lesen den ersten Wert aus, der meist über 8000 liegt, um ihn auch gleich wieder zu vergessen. Das heißt, wir verwenden ihn nicht weiter.

```
m=DRV8825(Dir,Step,Enable,StepsPerRev,M0,M1,M2)
found=False
x0=(d.width-1)//2 # = 63
y0=d.height-1 # = 63
rMax=min(x0,y0)
distMax=2000
propFaktor=rMax/distMax # = 0.0315
```

Dann erzeugen wir eine Motor-Instanz **m**. Wir übergeben die Nummern der Steuer-Pins und die Anzahl Schritte pro Umdrehung. Die Variable **found** wird den Zustand repräsentieren, ob die Startposition des Rotors gefunden, also angefahren wurde, oder nicht. Wir haben bisher noch nicht danach gesucht, also Fehlanzeige, **False**.

Der Mittelpunkt der untersten Zeile in der Anzeige spiegelt die Position des Motors im Raum wider. Unsere Anwendung drückt uns ein polares Koordinatensystem aufs Auge. Der Ursprung ist der eben angesprochene Punkt, der Pol (0, 0). Die unter dem Winkel φ gemessene Entfernung R liefert die Polarkoordinaten des Punktes P_R (R,φ) im Raum. Die Umrechnung in die kartesischen Koordinaten des Displays übernimmt später die Funktion **getPoint**().

Der maximale Radius **rMax** im Display ist das Minimum von halber Breite und Höhe der Anzeige. Die maximal messbare Distanz **distMax** belegen Sie bitte mit dem höchsten zuverlässigen Wert, den Ihr Sensor liefern kann. Der Proportionalitätsfaktor

propFaktor für die Umrechnung von Raum- in Display-Koordinaten ist der Quotient aus **rMax** und **distMax (=R_{max})**.

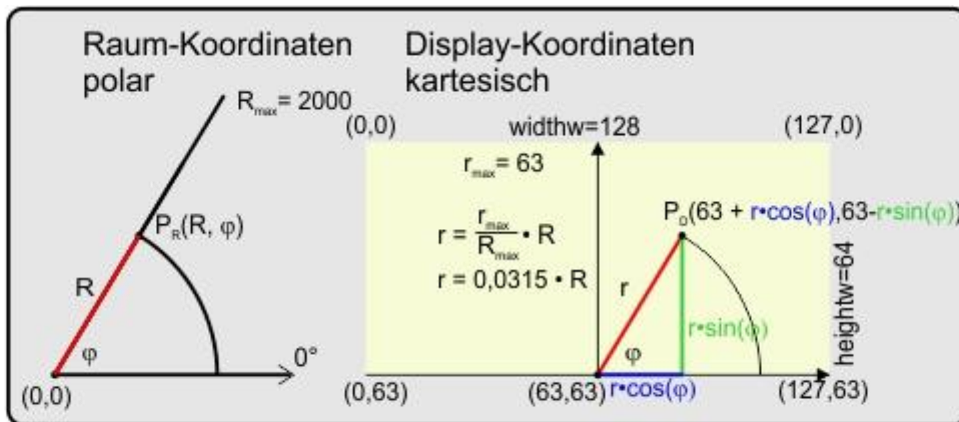

Abbildung 10: Raum- und Display-Koordinaten

Lassen Sie uns ein paar Funktionen deklarieren, die uns helfen, die Hauptschleife zu entlasten und damit klar und lesbar zu halten. Die umfangreichste Routine ist **goToStart**(). Sie schickt den Rotor unter Verwendung der Lichtschranke in die Ausgangsposition. Damit die eventuelle Änderung von **found** den lokalen Kontext der Funktion verlassen kann, wird die Variable als global deklariert. Wir haben sie weiter oben ja bereits mit **False** initialisiert. Wir schalten den Motor ein, setzen die Geschwindigkeit auf 400 Schritte pro Sekunde und stellen den Achtelschritt-Modus ein. Das bewirkt einen Schrittwinkel von 360 / (200 • 8) = 0,225°. Dann starten wir den Timer-IRQ, der letztlich für die automatische Rotation sorgen wird. All diese Aktionen stellt die Klasse A4988 bereit. Durch die Vererbung stehen sie auch der Klasse DRV8825 zur Verfügung, die wir importiert haben.

Falls der Pegel am Referenz-Pin **referenz** jetzt gerade eine logische 1 liefert, befindet sich das Blech am Rotorhebel bereits in der Lichtschranke. Wir drehen den Hebel in Microschritten von 0,225° im Uhrzeigersinn so lange, bis der Referenz-Pin eine 0 liefert, dann ist der Lichtweg frei und die Startposition genau erreicht. Die Methode **relativ**() sagt mit dem übergebenen Argument der ISR des Timer-IRQs, wie viele Schritte zu gehen sind. Wir brauchen uns also um den Ablauf selbst nicht kümmern, **relativ**() ist unser Subunternehmer.

```
# Funktionen deklarieren
def goToStart():
    global found
    m.enable(True)
    m.setSpeed(400)
    m.setMode(8)
    m.start()
    if referenz.value() == 1: # Marke in LS
        while referenz.value() == 1:
            m.relativ(1)
        found=True
    else:
        referenz.irq(handler=pinIRQ,trigger=Pin.IRQ_RISING)
        print("Pin-IRQ started")
        found=False
```

```
        m.relativ(-StepsPerRev *8)
        while not found:
            pass
        while referenz.value() == 1:
            m.relativ(1)
        referenz.irq(handler=None)
        print("Pin-IRQ stoped")
    m.resetPos()
```

Der logische Pegel am Pin **referenz** ist 0, wenn auftreffendes IR-Licht von der LED den Transistor durchschalten lässt. Der Hebel am Rotor zeigt dann in irgendeine unbekannte Richtung und müsste bei einer Volldrehung irgendwann den Fototransistor abschatten. Genau in diesem Moment muss die Drehung gestoppt werden. Um diesen Fall kümmert sich der **else**-Zweig.

Wir machen den referenz-Pin als Erstes interruptfähig und zwar für steigende Flanken. Die ISR wird durch **pinIRQ** referenziert. Diese Funktion werden wir gleich behandeln. Nachdem der IRQ scharfgeschaltet ist, setzen wir **found** auf False und starten eine volle Umdrehung, das sind 200 mal 8 Microschritte, im Gegenuhrzeigersinn. Wir warten, bis die ISR des Pin-Change-IRQ von **referenz** die Variable **found** auf **True** gesetzt hat und tun während dieser Zeit nichts – **pass**.

Nun kann es sein, dass der IRQ erst dann ausgelöst wird, wenn sich die Blechnase bereits ein Stück in der Lichtschranke befindet. Also fahren wir schrittweise wie im if-Zweig wieder aus der Lichtschranke heraus. Der IRQ wird nicht mehr gebraucht, also setzen wir den Handler auf **None**. Die Routine wird verlassen, wenn die Positionszeiger auf 0 gesetzt wurden.

Die ISR des Pin-Change-IRQs muss nicht viel tun. Damit die Variablenbelegung von **found** nach der Wertänderung nach außen sichtbar wird, muss die Variable wieder als global deklariert werden. Der Timer-IRQ wird ausgeschaltet, dadurch stoppt die Rotation. Wir setzen **found** auf **True** und das war's auch schon. Ein genereller Grundsatz für ISRs lautet: Man halte sie so kurz wie möglich.

```
def pinIRQ(pin):
    global found
    m.timerStop()
    found=True
    print("Timer stoped")
```

Um Streuungen bei der Entfernungsmessung zu minimieren, tasten wir nicht nur einmal ab, sondern n-mal. Den Mittelwert als wahrscheinlichsten Wert gibt die Funktion **getDistance**() zurück.

```
def getDistance(n):
    s=0
    for i in range(n):
        s+=tof.read()
    return s/n
```

Die Polarkoordinaten des vermessenen Raumpunkts, oder besser der vermessenen Raumpartie, liefert uns die Funktion **getPoint**() bereits umgerechnet auf die Pixelgröße des Displays. Wir lassen fünfmal abtasten. Die Strahllänge für das Display erhalten wir durch Multiplikation mit dem Proportionalitätsfaktor. Die 0°-Richtung im Display geht vom Pol (63,63) nach links, um genau der Raumorientierung zu entsprechen. Der Sensor schaut in der Startposition ja auch nach links. Daher ist der Peilwinkel φ die Differenz aus 180° und dem Rotorwinkel. Letzteren bekommen wir aus der Schrittposition **position** und dem Schrittwinkel **degPerRev** durch Multiplikation. Die Werte für Radius und Winkel gibt die Funktion über das Tupel **(r,w)** zurück.

```
def getPoint():
    dist=getDistance(5)
    r=propFaktor*dist
    w=180-m.position*m.degPerStep
    return(r,w)
```

Letzte Vorbereitungen laufen. Wir steuern die Startposition an, schalten auf Halbschrittmodus und betreten die Main-Loop.

```
goToStart()
modus=2
m.setMode(modus)
```

Jeden Durchlauf starten wir mit dem Einholen eines Koordinatenpaars für Radius und Winkel als Startschuss für die innere while-Schleife. Der erste Winkel ist 180° und damit größer oder gleich 0°.

```
while 1:
    radius,winkel=getPoint()
    while winkel >=0 :
        gc.collect()
        radius,winkel=getPoint()
#         d.polarPixel(x0,y0,radius,winkel,show=False)
        d.ray(x0,y0,radius,winkel,1)
        m.relativ(1)
        # evtl. Schleife beenden
        if taste.value()==0:
            m.timerStop()
            m.enable(False)
            exit()
    goToStart()
    m.setMode(modus)
    sleep(3)
    d.clearAll()
```

Der Speicher wird aufgeräumt. Dann zeichnen wir einen Strahl vom Ursprung unseres Koordinatensystems zum Messpunkt. Alternativ könnte auch nur dieser Punkt eingetragen werden, je nachdem welche der beiden Zeilen aktiv ist, **polarPixel** oder **ray**. Dann geht es einen Halb-Schritt, also um 360° / (200 • 2) = 0,9°, weiter.

Ein Abbruch wird gewünscht, wenn die Flash-Taste gedrückt ist. In diesem Fall stoppen wir den Timer-IRQ, schalten den Motor aus und verlassen das Programm.

Die innere while-Schleife ist beendet, wenn ein Winkel kleiner 0° erreicht wird. Im Display sehen wir jetzt die Abtastung der Sensorumgebung. Das könnte etwa so aussehen.

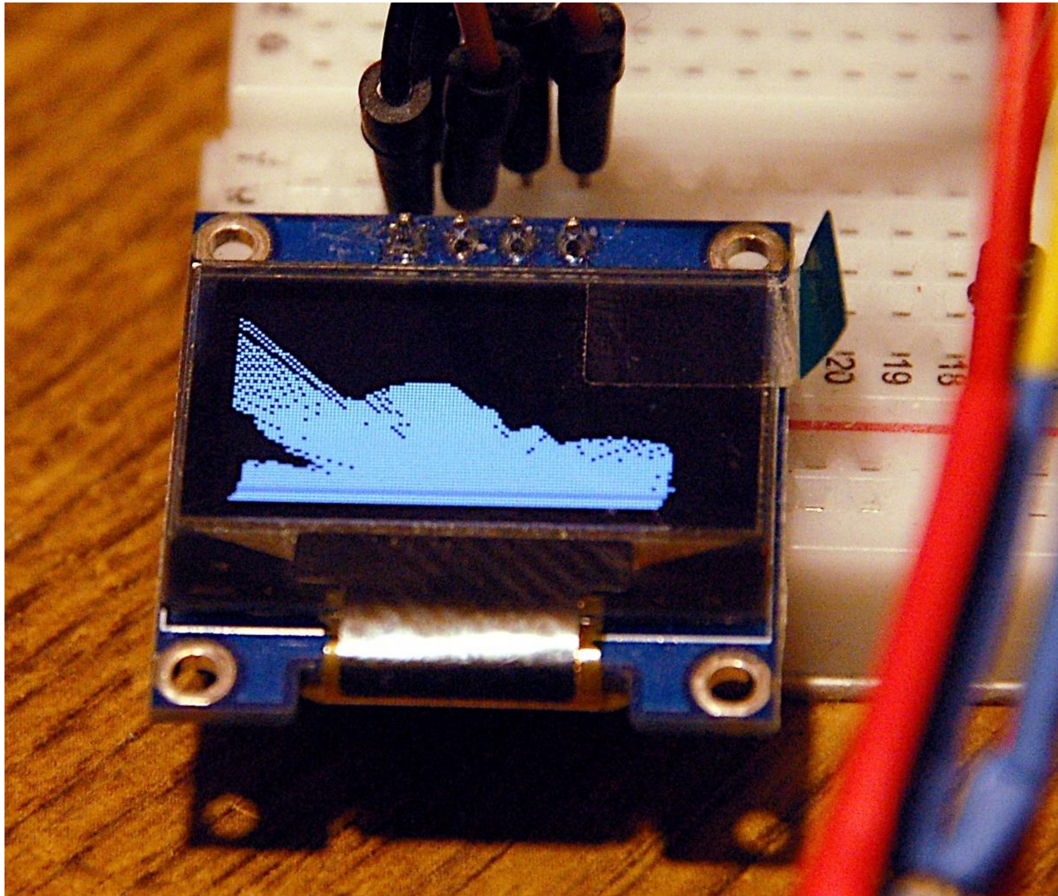

Abbildung 11: Abtastung meines Arbeitstisches um 180 Grad

Die Entfernung Richtung 10 Uhr geht nicht über die Erfassungsgrenze hinaus, aber die dort befindlichen Gegenstände bilden keine glatte Fläche. Somit kommt nicht genügend Licht für eine ordentliche Messung zurück. Der VL53L0X gibt in diesem Fall Werte über 8000 zurück. Das führt zu Strahllängen über den Bildschirmrand hinaus.

In der Main-Loop steht jetzt noch das erneute Aufsuchen der Startposition an. **goToStart**() hinterlässt uns den Modus 8, wir stellen also wieder auf Halbschritt-Modus zurück, machen ein kleines Nickerchen und starten nach dem Löschen des Displays in die nächste Runde.

Seit einiger Zeit sind runde Farb-TFT-Displays mit 240x 240 Pixel im Handel. Das motiviert zum Aufbau eines Radars mit einem Erfassungswinkel von 360°. Mal sehen, vielleicht kommt in einer der nächsten Folgen so ein Teil zum Einsatz.

Bis dann, viel Freude am Basteln und Programmieren – und - bleiben sie dran!