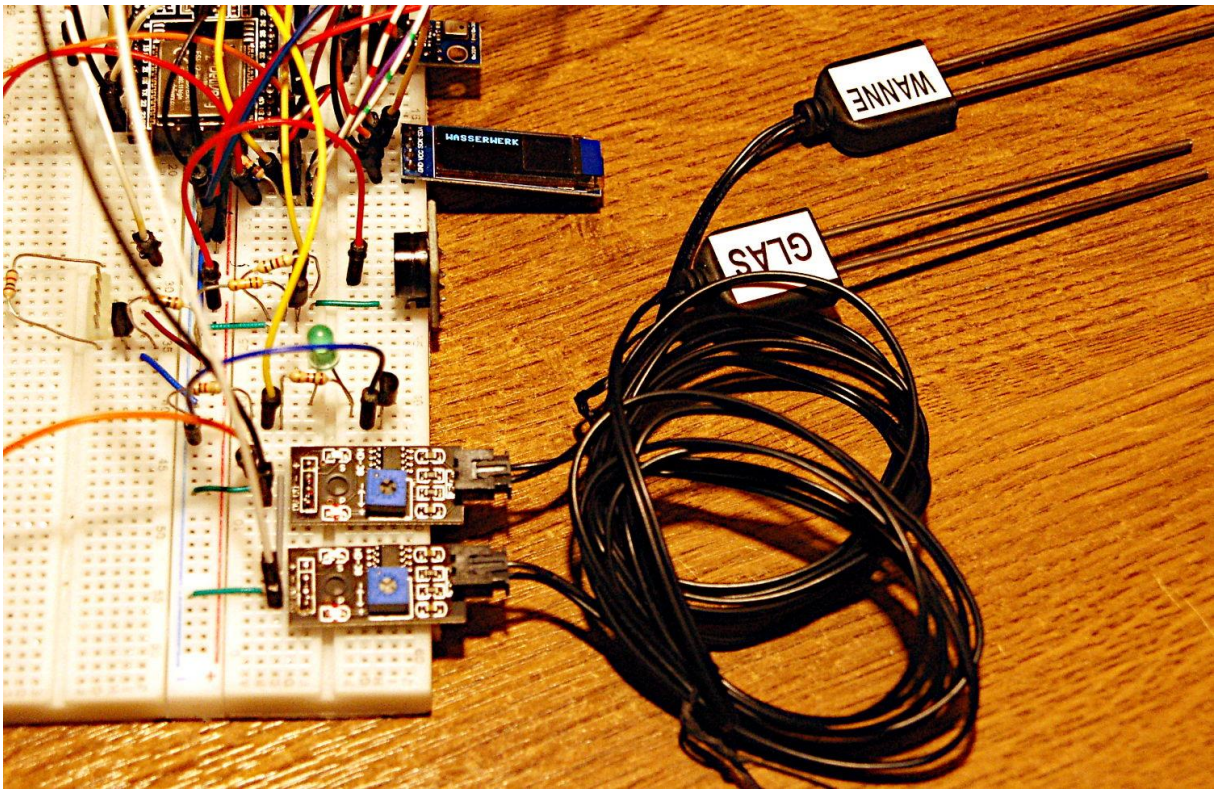

Wasserstands-Sensoren

Diese Folge ist auch als PDF-Dokument erhältlich.

Mit der Beleuchtung unserer Pflanzschale allein werden die Gewächse sicher nicht glücklich. Die Pflänzchen brauchen Nährstoffe und die sind üblicherweise im Wasser gelöst. Um die Wasserzufuhr geht es im heutigen Beitrag. Neben der Programmierung gibt es ein auch wenig Physik. Welche Rolle die spielt und wie die Planung hardwaremäßig und programmtechnisch umgesetzt werden kann, das erfahren sie in dieser Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Teil 3 – Wasser marsch

Die Ansaatschale ist so konzipiert, dass die Pflanztöpfchen mit ihren Bohrungen am Boden ein paar mm in das darunter befindliche Wasser eintauchen. Das ist einfacher, als an jedes Gefäß eine eigene Wasserleitung zu verlegen. Zum einen gäbe das ein heilloses Schlauchwirrwarr und andererseits müsste ein Druckausgleich zwischen den einzelnen Leitungen hergestellt werden, sonst ersaufen die einen Pflänzchen, während die anderen verdorren, weil sie nix abkriegen.

Natürlich muss ein Wasservorratsgefäß da sein, damit, mittels Pumpe, die Wanne nachgefüllt werden kann. In diesem Zusammenhang gibt es aber einen versteckten Haken und das ist die Leitung vom Vorratsgefäß zur Wanne.

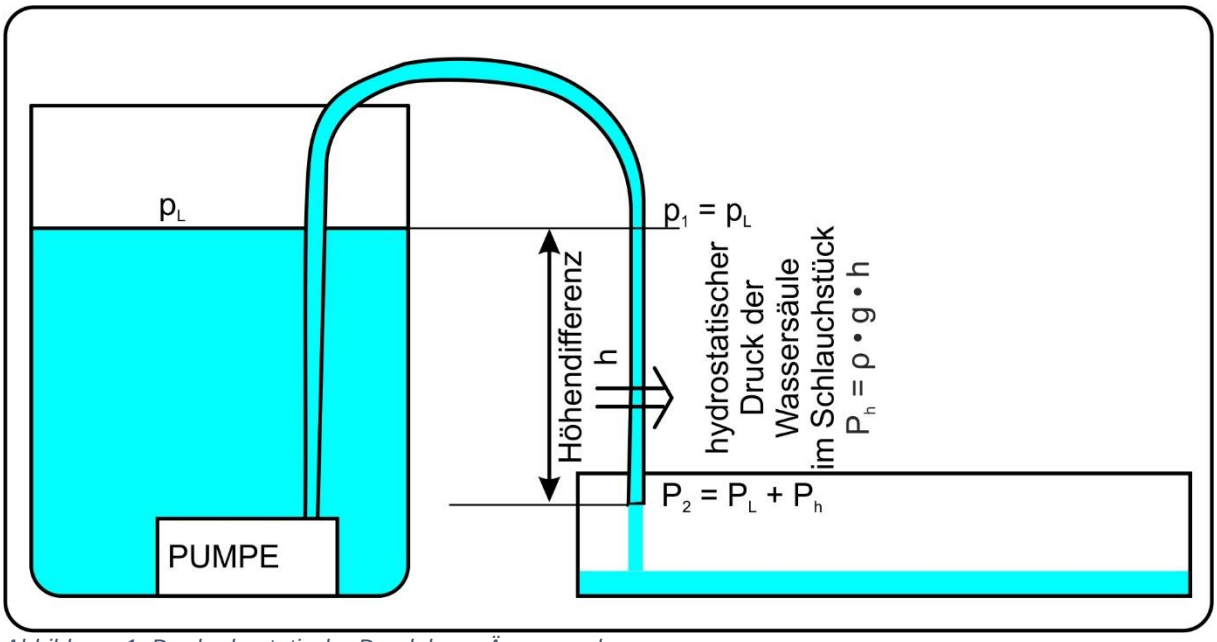

Abbildung 1: Der hydrostatische Druck kann Ärger machen

Die Pumpe arbeitet wie die Laugenpumpe einer Waschmaschine mit einem Flügelrad in einer Kammer. Das Wasser wird durch eine Bohrung an der Stirnseite angesaugt und durch einen Rohrstutzen in den Schlauch befördert. Steht der Motor still, ist diese Kammer in beiden Richtungen durchlässig, es gibt keine Ventile.

Hat die Pumpe den Schlauch gefüllt, dann baut sich im rechten Teil durch den Höhenunterschied ein hydrostatischer Druck auf, der das Wasser weiterlaufen lässt, auch wenn die Pumpe steht. Der Schlauch wirkt als Saugheber, solange das Wasser im Glas höher steht als in der Wanne.

Auf die offenen Flächen im Glas und am Schlauchende wirkt von außen der normale Luftdruck. Druck breitet sich nach allen Seiten gleichmäßig aus. Das führt dazu, dass der Druck p_1 im Schlauch an der oberen Marke auch gleich dem Luftdruck ist. Am Schlauchende kommt aber von innen der hydrostatische Druck p_h der Wassersäule dazu. Die Druckkraft, senkrecht zur Querschnittsfläche des Schlauchs befördert das Wasser nach außen.

$P_h = \rho \cdot g \cdot h$

ρ ist die Dichte der Flüssigkeit, g die Erdbeschleunigung und h die Höhendifferenz

Dieser Effekt kann dazu führen, dass im schlimmsten Fall die Wanne überläuft. Die Pflanztöpfchen gehen auf jeden Fall baden. Damit das nicht passiert, muss man die Wanne einfach so hoch aufstellen, dass der Wasserspiegel darin stets höher liegt als derjenige im Glas. Dann fließt das Wasser aus dem Schlauch nach dem Abstellen des Motors, nach derselben physikalischen Begründung, wieder in das Glas zurück.

Ein ähnlicher Effekt stellt sich aber ein, wenn der Schlauch in der Wanne bis auf den Boden reicht. Ist die Pumpe eine Zeit lang aktiv gewesen, dann hat sich in der Wanne Wasser angesammelt, in das der Schlauch eintaucht. Liegt der Wasserspiegel in der Wanne höher als im Glas, wird Wasser in dieses zurückfließen. Der Effekt kann gewünscht sein, wenn es gilt, die Pflanzgefäße nur kurz zu tauchen. Das kann sinnvoll sein, weil dann die Pflanzen nicht ständig nasse Füße haben.

Allerdings müsste dann eine Zeitsteuerung die Pumpe regelmäßig in Aktion setzen, weil sich der Bewässerungsvorgang dann nicht mehr selbst steuern kann, wie es im vorliegenden Programm der Fall ist.

Hardware

Als Controller habe ich einen ESP32 gewählt, weil der mit genügend frei wählbaren GPIO-Anschlüssen aufwarten kann, davon brauchen wir beim Vollausbau 10 Stück aufwärts.

Die ESP32-Modelle in der Teileliste sind alle brauchbar. Lediglich beim ESP32-Lolin-Board muss für den I2C-Anschluss SDA statt GPIO21 der GPIO25-Pin genommen werden.

1	ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Miniwasserpumpe 5V / 450mA
2	Widerstand 10k
2	Widerstand 1k
1	NPN-Transistor BC548 o. ä.
1	NPN-Transistor BC517 o. ä. I_C >= 1A
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
diverse	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard
1	Netzteil 5V / 3A
optional	Logic Analyzer

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
wasser.py Das Programm zur Lichtsteuerung

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 18.06.2022) 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.

Schaltung

Abbildung 2 zeigt die Schaltung des Projekts. Ausgegraut ist der bereits besprochene Teil aus der ersten und zweiten Blogfolge.

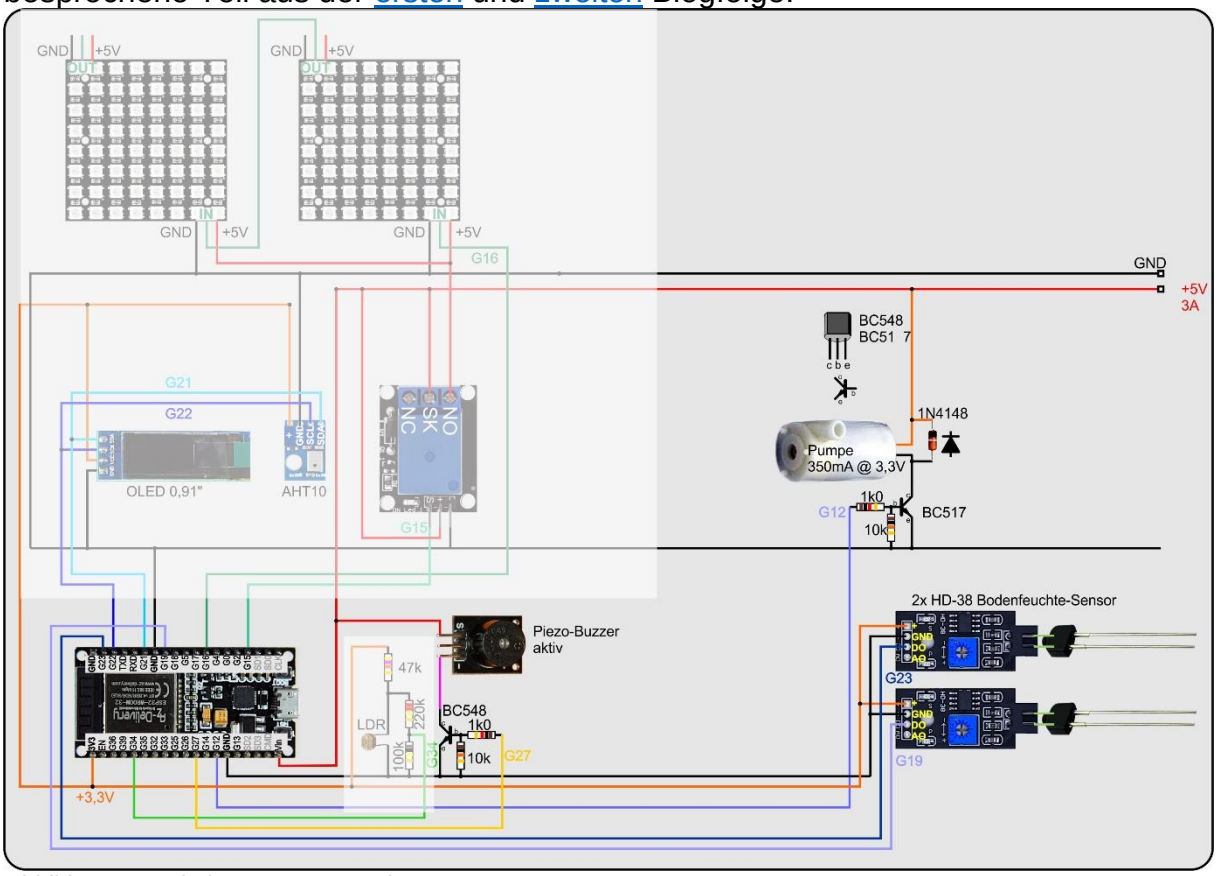

Abbildung 2: Schaltung-Wasserwerk

Wie die LED-Panele wird auch die Pumpe an der externen Spannungsquelle betrieben, denn mit einer Stromaufnahme von knapp 0,5A macht die Versorgung vom USB-Bus schlapp und die Pumpe quäkt nur müde vor sich hin. Damit das Wasser einen Höhenunterschied von 10 bis 15cm schafft, braucht man 5 Volt und eine Stromquelle, die auch bei Belastung diese Spannung hält. Wichtig ist die Freilaufdiode am Pumpenmotor. Sie verhindert, dass Spannungsspitzen an der Spule den Transistor killen. Achten Sie aber bitte beim Einbau auf die Richtung des Bauteils.

Beim Booten des ESP32 sind alle GPIOs als Eingänge geschaltet. Damit der Pegel an der Basis des BC517 eindeutig bestimmt ist, habe ich einen 10kΩ-Widerstand von Basis auf GND gelegt damit der Transistor sicher sperrt. Ein HIGH an GPIO12 hebt den Pegel auf 2,2 Volt, wodurch der Transistor sicher durchschaltet. Die Stromzuleitungen zur Pumpe sollten möglichst kurz gehalten werden und nicht zu dünn sein. Jumperkabel sind für den Produktionsbetrieb ungeeignet, weil an dem relativ hohen Leitungswiderstand einfach zu viel Spannung abfällt. Was dadurch an Energie an der Zuleitung in Wärme umgewandelt wird, fehlt für die mechanische Arbeit der Pumpe.

Den Wasserstand im Glas und in der Wanne überprüfen wir mit zwei Bodenfeuchtesensoren.

Abbildung 3: Feuchte_Fühler

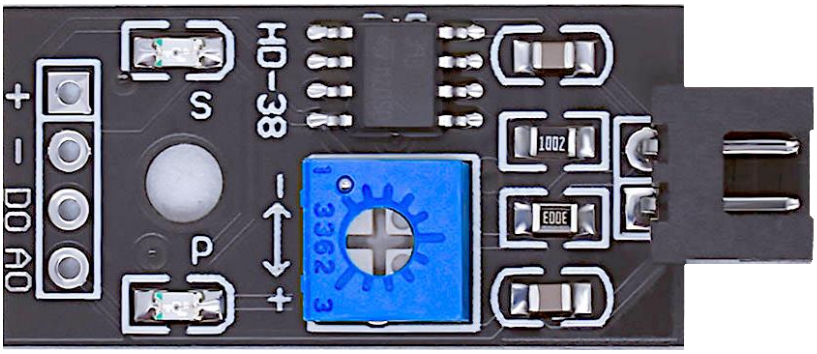

Abbildung 4: Bodenfeuchte-Modul

Die Potitrimmer auf den beiden Boards sind so eingestellt, dass bereits der kleinste Kontakt der Fühlerstäbe mit der Wasseroberfläche die grüne LED (S) auf dem BOB (Break Out Board) zum Aufleuchten bringt. Der digitale Ausgang DO geht dann auf LOW (0V). Sinkt der Wasserpegel, werden die Fühlerstäbe freigegeben und DO wird HIGH (3,3V). Die Fühler müssen nun so im Behälter platziert werden, dass DO der Einheit in der Wanne auf HIGH geht, wenn der Wasserstand knapp unter dem Boden der Pflanztöpfe liegt. DO der Einheit im Glas muss auf HIGH gehen, wenn der sinkende Pegel im Glas fünf Millimeter über der Ansaugöffnung der Pumpe liegt. **Die Pumpe darf nicht trockenlaufen!**

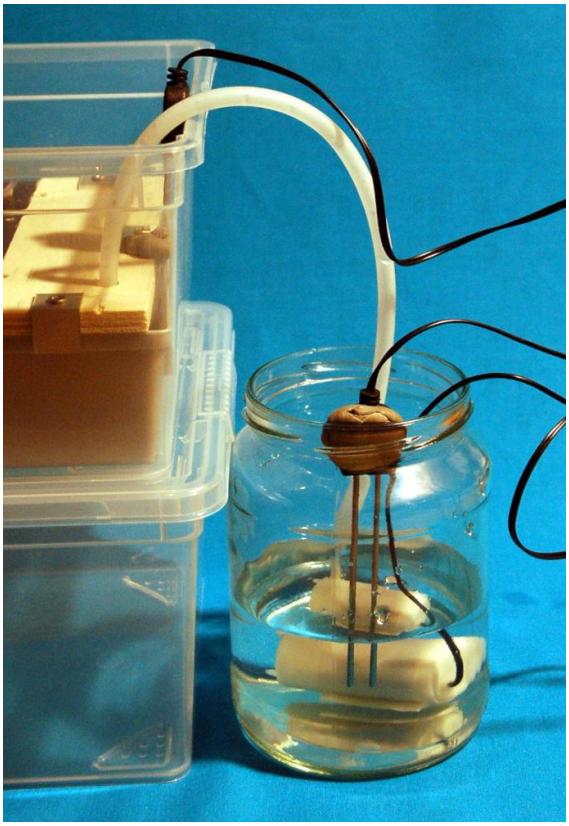

Abbildung 5: Wasserversorgung

Kritisch wird es für die Pflanzen, wenn kein Wasser mehr im Glas ist. In diesem Fall kommt die Alarmtröte zum Einsatz. Ich habe einen aktiven Piezo-Buzzer gewählt, weil ich mich dann nicht um die Erzeugung der Tonfrequenz kümmern muss. Dafür lasse ich den Buzzer intervallweise ertönen. Mehr dazu bei der Programmbesprechung. Als Treibertransistor reicht hier ein BC548, den ich in derselben Weise ansteuere wie den BC517 an der Pumpe.

Das Programm

Das Importgeschäft ist mit vier Zeilen erledigt.

```
# wasser.py
from machine import SoftI2C, Pin, PWM
from time import sleep,ticks_ms
from oled import OLED
import sys
```

SoftI2C brauche ich für das Display, **Pin** für die **SoftI2c** und die Transistoransteuerung und mit **PWM** setze ich die Impulssteuerung des Buzzers um. Pausen werden mit **sleep** erzeugt und **ticks_ms** brauche ich für einen Timer. Die Klasse **OLED** bietet eine API für das OLED-Display und **sys** liefert die Funktion **exit**(), mit der ich das Programm unter definierten Bedingungen beenden kann. Dazu gehört, dass beim Programmausstieg die Pumpe sicher ausgeschaltet wird. Würde ich mit Strg+C beenden, dann passiert das an einer zufälligen Programmstelle, an der vielleicht gerade die Pumpe an ist und "Land unter" hervorrufen könnte, bis ich sie händisch stoppen kann.

```
debug=False
```

Mit Variable **debug** = **True** habe ich die Möglichkeit während der Entwicklung an bestimmten Stellen eine Zustandsmeldung im Terminalbereich ausgeben zu lassen.

```
i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
d=OLED(i2c,heightw=32)
```

Das I2C-Objekt **i2c** übergebe ich an den Konstruktor der OLED-Klasse zusammen mit der Displayhöhe **heightw** in Pixeln. Die Breite von 128 Pixeln entspricht dem Defaultwert des optionalen Parameters widthw und muss nicht angegeben werden.

```
wanne=Pin(23, Pin.IN)
glas =Pin(19, Pin.IN)
```

Die Feuchte-Module liefern an DO digitale Werte. Eine Spannung von 3,3V oder logisch 1 bedeutet, dass der Fühler nicht in Wasser eintaucht. Ein Pegel von 0V oder logisch 0 sagt uns, dass die Fühlerspitzen Wasserkontakt haben. GPIO23 und GPIO19 tasten die Pegel an den Modulen als Eingänge ab.

```
pump=Pin(12,Pin.OUT, value=0)
```

An GPIO12 hängt die Steuerleitung für den Pumpen-Transistor. Wir deklarieren den Pin als Ausgang im ausgeschalteten Zustand, value=0.

```
taste=Pin(0,Pin.IN,Pin.PULL_UP)
```

Die Verwendung des Pin-Objekts **taste** habe ich oben bereits beschrieben.

```
pwmPin=Pin(27,Pin.OUT,value=0)
pwm=PWM(pwmPin,5)
pwm.freq(5)
pwm.duty(0)
```

Den Pin 27 schalte ich als Ausgang mit Anfangszustand 0. Ich verwende den Pin als PWM-Ausgang. Die PWM-Frequenz setze ich initial auf 5 Hz und schalte den Ausgangspegel auf 0, indem ich den Duty Cycle auf 0 setze, das entspricht prozentual 0%. Die Werte für das Tastverhältnis (auch Tastgrad), die an die Funktion **duty**() übergeben werden, müssen zwischen 0 und 1023 liegen, sind also keine Prozentwerte. 1023 liefert eine konstante Spannung von 3,3 Volt am GPIO-Pin. Bei 50% oder 512 beträgt die Pulsdauer die Hälfte der Periodendauer.

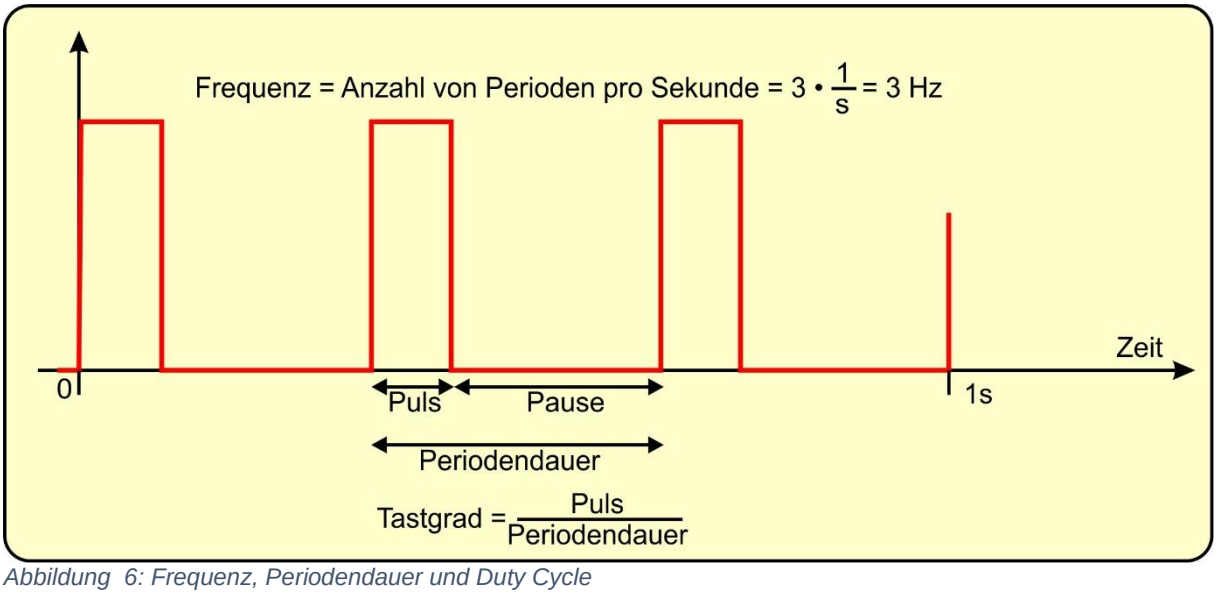

Abbildung 6: Frequenz, Periodendauer und Duty Cycle

```
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare
```

Normalerweise sind Objekte, die innerhalb einer Funktion deklariert werden, lokal und außerhalb der Funktion nicht sichtbar. Sie wandern ins Nirwana, wenn die Funktion verlassen wird. Das kann man umgehen, wenn die Funktion Objekte mit **return** zurückgibt. Im Normalfall werden das Werte sein, die innerhalb der Funktion berechnet oder zum Beispiel über GPIO-Pins erhoben wurden.

TimeOut() gibt nun keinen Zahlenwert, sondern die Funktion **compare**() zurück, genauer gesagt eine Referenz darauf. Beim Aufruf von **TimeOut**() wird eine Zeitdauer in Millisekunden an den Parameter **t** übergeben. In **start** wird dann der

aktuelle Stand des Millisekunden-Zählers gespeichert. **compare**() subtrahiert diesen Wert vom laufend aktualisierten Stand und vergleicht den Differenzwert mit der Zeitdauer in **t** und gibt **True** zurück, wenn der Differenzwert den Wert in **t** überschreitet. Dann ist der Timer abgelaufen.

TimeOut() gibt also eine Referenz auf die Funktion **compare**() zurück, die wir uns in einer Variable merken können.

>>> periode=TimeOut(5000)

Durch diesen Trick bleibt der Zugriff auf **t, start** und die Funktion **compare**() auch nach dem Verlassen der Funktion **TimeOut**() erhalten. **periode** ist ein Alias für den Namen **compare**. In **periode** liegt die Referenz auf die Funktion **compare**() und die rufen wir jetzt auf.

>>> periode()

Wir bekommen den momentanen Status des Zeitablaufs als Rückgabewert von **compare**() geliefert.

Wenn die letzte Eingabe innerhalb 5 Sekunden nach der ersten Anweisung erfolgt, wird es **False** sein, danach dann **True**.

Mit der Closure **TimeOut**() haben wir somit einen Software-Timer realisiert, der den Programmablauf, nicht blockiert, so wie es **sleep**(5) tut. Denn zwischen zwei Aufrufen von **periode**() kann der ESP32 beliebige andere Dinge erledigen. Closures gehören schon zu den Geheimwaffen von MicroPython. Damit lassen sich zum Beispiel auch Punktezähler bei Spielen, Spielzustandsmerker und viele weitere Dinge elegant programmieren.

```
def pumpe(val=None):
    assert val in (0,1,None)
    if val is not None:
        pump.value(val)
    else:
        return state[pump.value()]
```

Die Funktion **pumpe**() stellt erst einmal fest, ob für **val** der richtige Wert übergeben wurde. Ohne Wertübergabe beim Aufruf, hat **val** den Defaultwert **None**. Das führt im **else**-Zweig dazu, dass der Klartext "an" oder "aus" aus der Liste **state** zurückgegeben wird. Ist **val** nicht **None**, dann wird der übergebene Wert dazu benutzt, die Pumpe ein- (1) oder auszuschalten (0).

```
def sound(dauer,freq=8):
    period=TimeOut(dauer*1000)
    pwm.freq(freq)
    pwm.duty(512)
    while not period():
        pass
    pwm.duty(0)
```

die Funktion **sound**() nimmt im Positionsparameter **dauer** eine Zeitspanne in Sekunden und in **freq** eine Frequenz in Hertz. Damit wird der PWM-Ausgang mit einem Duty Cycle von 50% geschaltet. Der Defaultwert für **freq** ist 8, das heißt, dass der Buzzer 8-mal pro Sekunde zirpt. Die Sekunden in **dauer** rechnen wir in Millisekunden um und stellen damit den Timer **period**. Solange **period**() (aka **compare**()) noch nicht abgelaufen ist, erhalten wir **False**. Mit **pass** erfüllen wir die Bedingung, dass ein Schleifenkörper nicht leer sein darf. Diese Anweisung tut gar nichts, sie ist einfach nur da. Liefert **period**() ein **True**, dann beendet das die Schleife und wir müssen nur noch den Tweeter ausschalten, bevor wir die Funktion verlassen.

```
def wasserstand():
    pegelWanne=wanne.value() # 1 = leer; 0 = voll
    pegelGlas=glas.value()
    if debug: print("Wanne",pegelWanne)
    if pegelWanne == 0:
        if debug: print("Wanne: Pegel OK")
        pumpe(0)
        return 1 # Wanne OK
    else:
        # Wanne zu wenig
        if debug: print("Glas",pegelGlas)
        if pegelGlas==0:
            if debug: print("Glas Pegel OK, pumpen")
            pumpe(1) # Wasser vom Glas zur Wanne
            return 2
        else:
            if debug: print("Glas nachfüllen")
            pumpe(0) # Pumpe aus
            sound(2,5) # kein Wasser im Pool
            return 0
```

Die Funktion **wasserstand**() muss drei Situationen abbilden:

- Die Wanne ist ausreichend gefüllt, OK
- In der Wanne ist zu wenig Wasser; im Vorratsglas ist genug drin, die Wanne kann nachgefüllt werden
- Auch das Glas ist leer; Alarm auslösen

Wir lesen erst einmal die Wasserstände ein. Falls **debug** den Wert **True** hat, wird der Pegelzustand im Terminalbereich ausgegeben. Meldet der Sensor eine 0, dann ist alles OK und es muss keine Aktion folgen außer es ist vorher die Pumpe an

gewesen, dann muss sie ausgeschaltet werden, weil der Wasserstand jetzt jedenfalls stimmt. Wir geben 1 an das aufrufende Programm zurück.

Ist **pegelWanne** aber 1, dann sollten wir Wasser nachfüllen. Mal nachschauen, ob im Glas noch Wasser ist. Dann hat **pegelGlas** den Wert 0, wir werfen die Pumpe an und geben eine 2 zurück.

Andernfalls ist der Wasservorrat zu Ende, und wir müssen schleunigst die Pumpe ausschalten, denn sie soll ja nicht trockenlaufen. Jetzt ist es wichtig, dass der Fühler im Glas nicht zu tief sitzt, damit die Pumpe aus ist, bevor der Wasserstand die Ansaugöffnung erreicht.

Wir lassen den Buzzer zwei Sekunden lang 5-mal pro Sekunde piepen und geben 0 zurück. Im aufrufenden Programm würde das als **False** interpretiert, während jeder andere Wert als **True** gilt.
Die Hauptschleife muss jetzt nur noch die Funktion **wasserstand**() aufrufen, die erledigt alles Notwendige. Wir lassen den Zustand im Terminal ausgeben, fragen die Flashtaste ab und schicken den ESP32 für eine Sekunde ins Bett.

Jetzt fehlt uns nur noch die Temperaturregelung. Ich werde ein Thermoelement als Wärmequelle zusammen mit zwei Kühlkörpern als Wärmetauscher einsetzen. Das ist im Prinzip nichts anderes als eine Wärmepumpe. Natürlich muss die Temperatur der Warmseite überwacht werden, denn sie darf nicht höher als 138°C sein. Daneben messen wir mit einem wasserdicht verpackten DS18B20 die Temperatur im Substrat.

Bleiben Sie dran!

Bis dann