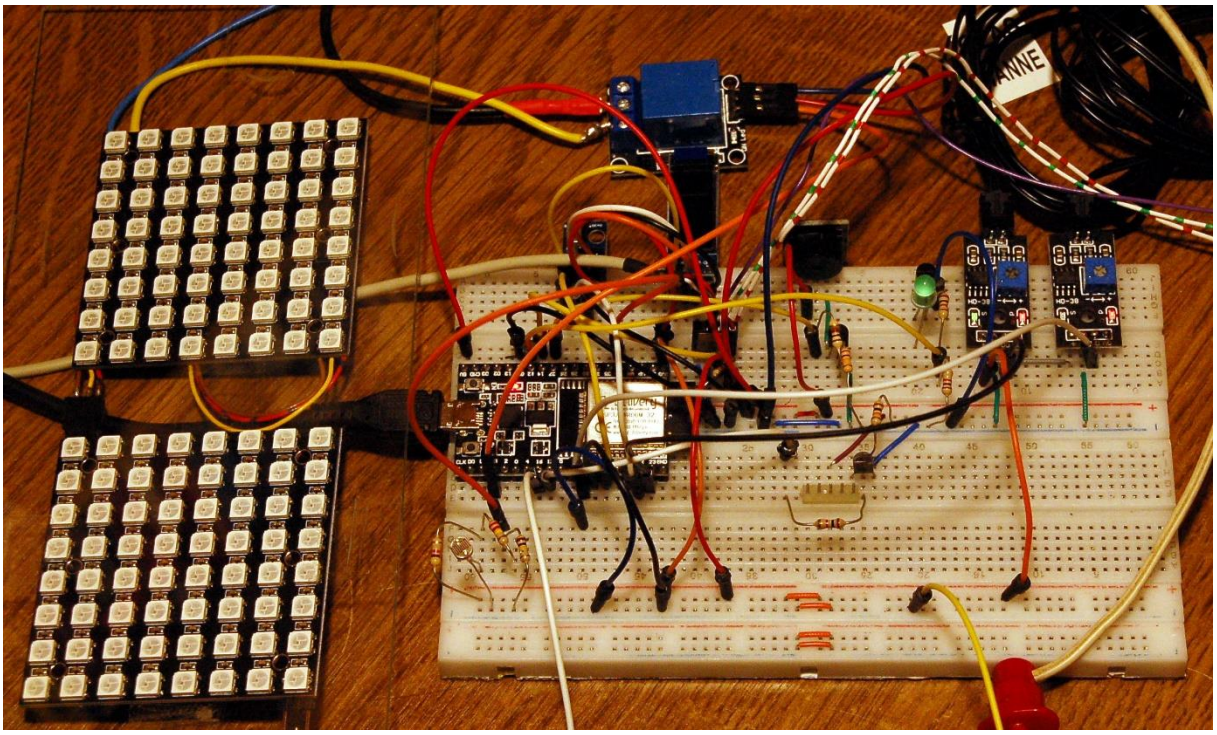
Gesamter - Aufbau

Diese Folge ist auch als PDF-Dokument erhältlich.

Pflanzen brauchen Licht, um zu gedeihen. Genau darum werden wir uns heute kümmern. Zwar werden die Tage schon deutlich länger, wenn wir uns mit der Anzucht von Sämlingen beschäftigen, aber es kann nicht schaden, eine zusätzliche Beleuchtung mit der richtigen Lichtqualität zur Verfügung zu haben. Gärtnereien lassen ihre Pflanzen ja sogar nachts mit besonderen Lampen bescheinen. Ein solche Lichtquelle der etwas anderen Art werden wir heute herstellen und programmieren. Natürlich gibt es auch wieder den einen oder anderen MicroPython-Programmiertrick. Damit willkommen zu einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Teil 2 – Es werde Licht

Den Pflanzen geht es wie uns, Licht heitert die Stimmung auf und regt zu Aktivitäten an. Deswegen wird meine Saatbox eine Lichtberieselungseinrichtung erhalten.

Während das menschliche Auge im grünen Spektralbereich die größte Empfindlichkeit aufweist, bevorzugen Pflanzen Licht im roten und blauen Spektralband.

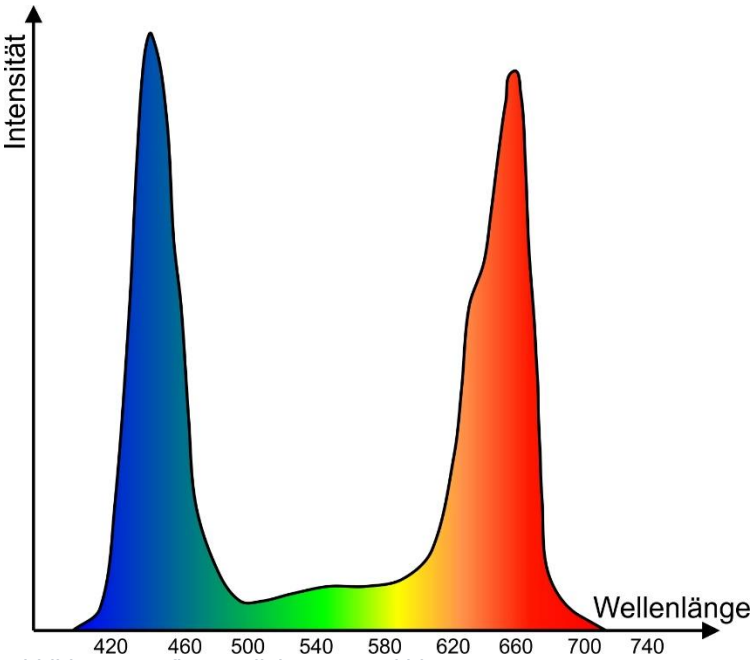

Abbildung 1: Pflanzen lieben rot und blau

Nun, wenn dem so ist, sollen auch genau diese Anteile zur Beleuchtung genutzt werden. Das geht hervorragend durch den Einsatz von Neopixel-LEDs. Während herkömmliche Pflanzenlampen den Grünanteil herausfiltern, erzeugen wir den erst gar nicht, das spart ein Drittel Energie. Ich habe zwei 8x8-Panele ausgewählt, die sollten eine ausreichende Helligkeit erbringen. Die Ansteuerung mit einem ESP32 ist denkbar einfach. Die Teile haben nur einen kleinen Nachteil, auch wenn keine LED leuchtet, fließt ein Ruhestrom von 120mA. Das allein ist der rund 10-fache Wert dessen was der ESP32 zieht. Um die Schluckspechte zu bändigen habe ich deshalb ein Relais in die Versorgungsleitung der Neopixel-Panele gelegt. Damit sind wir auch schon bei der Hardwareliste angekommen.

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	1-Relais 5V KY-019 Modul High-Level-Trigger oder 3-er
1	2-Relais Modul 5V mit Optokoppler Low-Level-Trigger

1	U 64 LED Matrixpanel
1	Fotowiderstand Photo Resistor
1	Widerstand 47k
1	Widerstand 220k
1	Widerstand 100k
1	Widerstand 10k
1	Widerstand 1k
1	Transistor BC548 o. ä.
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
licht.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

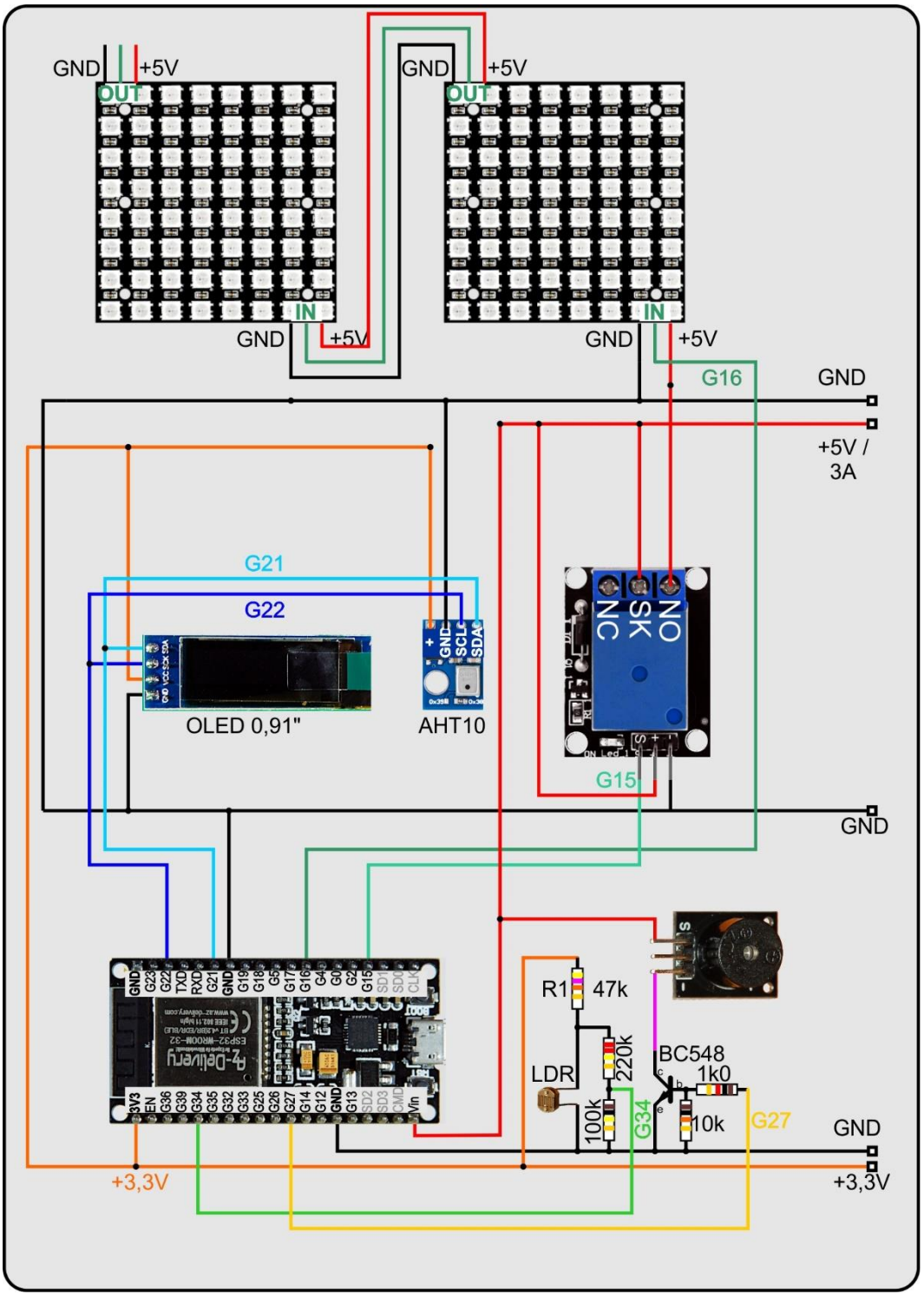

Abbildung 2: Licht und Sound - Schaltung:

Zu OLED-Display und AHT10 aus der vorangegangenen Folge kommen ein LDR (Light Dependent Resistor), fünf normale Kohleschichtwiderstände, ein NPN-Transistor BC548 o. ä., ein aktiver Piezo-Buzzer, ein Relais und die beiden LED-Panele dazu. Den Buzzer brauche ich im Zusammenhang mit den Wasserstandssensoren, die ich im nächsten Beitrag behandeln werde. Wie funktioniert das nun alles?

Tages-Helligkeit erfassen

Wenn bei Sonnenschein die Tageshelligkeit ausreicht, brauchen wir natürlich keine Extrabeleuchtung. Also muss der ESP32 wissen, wann er die LEDs anmachen soll und wann die natürliche Beleuchtung genügt. Dafür gibt es Widerstände, deren Widerstandswert sich abhängig vom Lichteinfall ändert, LDRs. Unser LDR hat bei völliger Dunkelheit einen Wert von 2MΩ, am Fenster bei trübem Wetter mit Bewölkung 440Ω und direkt an der Zimmerbeleuchtung 80Ω.

Weil der ESP32 den Widerstand nicht direkt messen kann, müssen wir halt selbst den Widerstand in eine Spannung umwandeln. Das geht mit einem sogenannten Spannungsteiler. Dazu werden zwei Widerstände in Reihe geschaltet und natürlich vom gleichen Strom I durchflossen. Für beide Widerstände gilt die Widerstandsformel. Daraus kann man ableiten, dass sich die Spannungen, die an den Widerständen abfallen, im selben Verhältnis stehen wie die Widerstandwerte selbst.

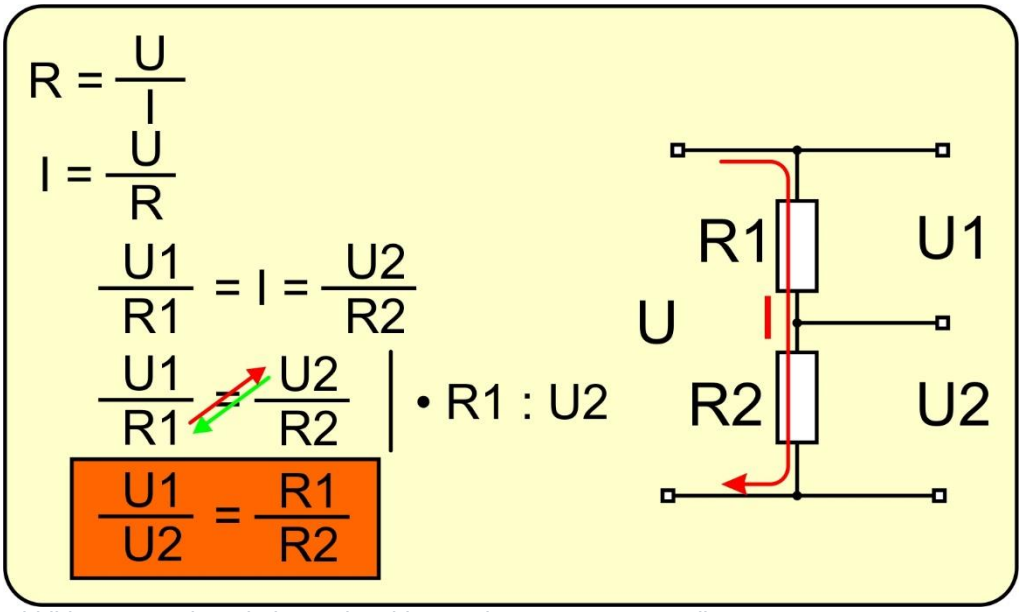

Abbildung 3: Serienschaltung als Widerstands-Spannungs-Wandler

Mit einem festen Wert für R1 und dem LDR als R2 erhalten wir als U2 niedrige Werte, wenn es draußen hell ist und Werte bis U2 = 3,3V bei Dunkelheit.

Damit haben wir aber auch schon das nächste Problem am Hals. Der Analogeingang GPIO34 verträgt nur 1.1 V. Also brauchen wir einen zweiten Spannungsteiler, um die maximal 3,3V auf höchstens 1V herabzusetzen. Damit der zweite Spannungsteiler den eigentlichen Messwert nicht (nennenswert) beeinflusst, müssen die Teilwiderstände zusammen größer sein als der Bereich, in dem der LDR arbeitet. Der soll bei einsetzender Dämmerung eine Spannung von ca. 1,5V liefern. Das tut er, wenn sein Wert ca. 45kΩ beträgt und wir als R1 einen Widerstand von 47kΩ nehmen. Für den nachfolgenden Spannungsteiler wähle ich daher Werte im 10-fachen Bereich.

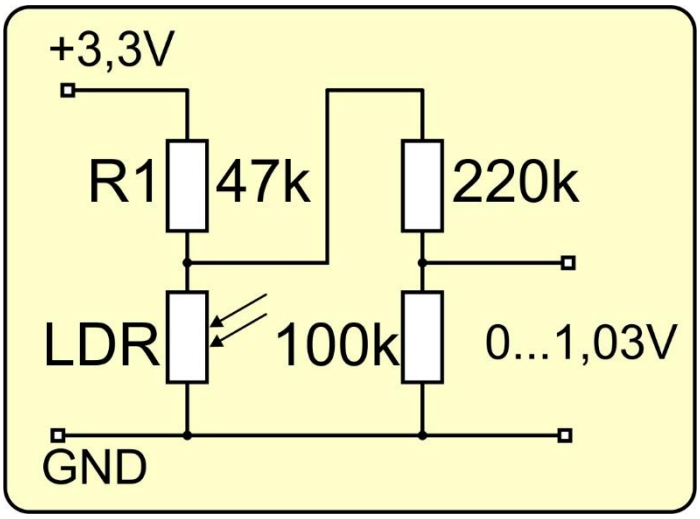

Abbildung 4: Verringerung der Spannung für den Analogeingang

Das war der sensorische Bereich zum Thema Licht, kommen wir zu den Aktoren Relais und LED-Panel.

Der Lichtschalter

Weil der ESP32 an seinen Ausgängen maximal 12mA liefern kann, die Panele aber bis zu 3,0 A ziehen können, brauchen wir entweder einen fetten Transistor, der solche Stromstärken vertragen kann als Schalter oder einen elektromagnetisch betätigten Schalter, ein Relais. Eine Spule mit Eisenkern bildet einen Elektromagneten, der einen Schaltkontakt bewegen kann. Der Schaltkontakt S oder auch Anker liegt an einem weiteren Kontakt, der im Ruhezustand geschlossen ist, normaly closed (NC). Dem gegenüber befindet sich ein Kontakt, der Im Ruhezustand gegen S offen ist, normaly open (NO). Fließt Strom durch die Spule, dann wird S gegen NO geschlossen und S gegen NC geöffnet.

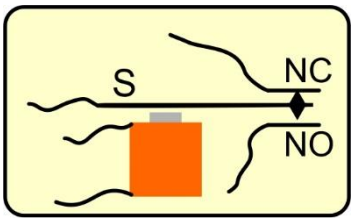

Abbildung 5: Funktion eines Relais

Selbst der Erregerstrom des Elektromagneten von ca. 30mA ist noch zu hoch für einen GPIO-Ausgang. Deshalb wohnt auf dem Relaismodul ein Schalttransistor, der mit wenigen Milliampere angesteuert werden kann und dann den Strom durch die Spule fließen lässt. Der Panelstrom kann dann über die Kontakte S und NO fließen.

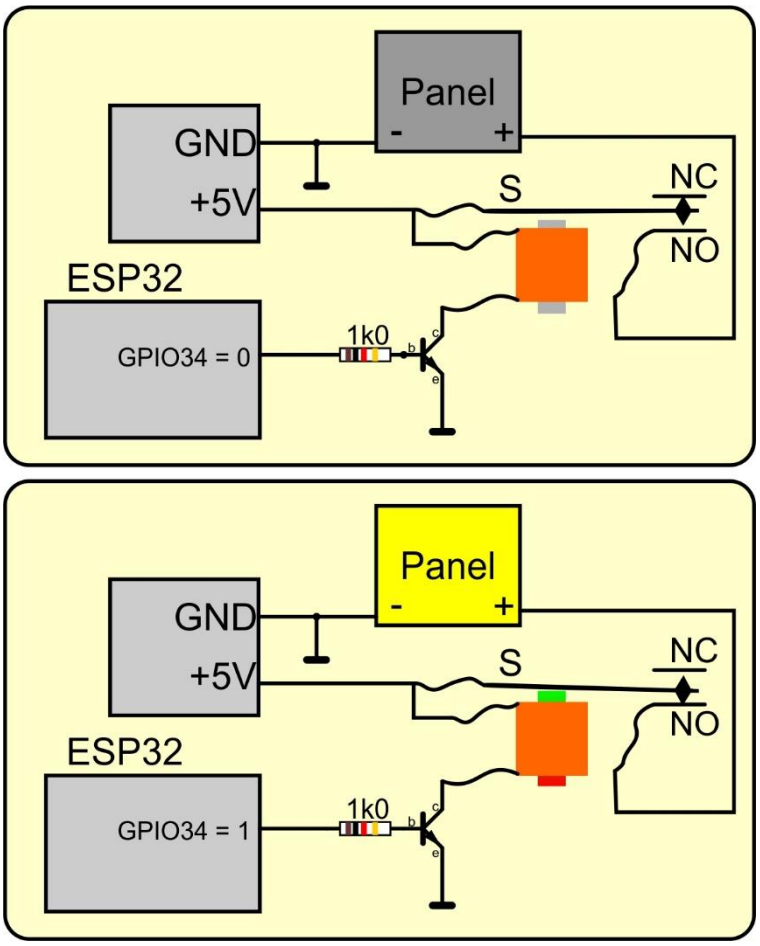

Abbildung 6: Relais in Aktion:

Die Panele

Die LED-Panele sind mit WS2812B-Neopixel-LEDs bestückt, wobei der Ausgang einer LED mit dem Eingang der nächsten verbunden ist. Jedes Pixel enthält eine rote, grüne und blaue LED und einen Controller. Dieser empfängt die Daten über die Leitung DIN, schnappt sich die ersten drei Bytes, verdaut sie, und gibt die nachfolgenden Dreiergruppen verstärkt an DO aus. So ein Dreierpacket an Daten enthält die Farbinformationen für grün, rot und blau. Weil mit einem Byte 256 Zustände codiert werden können, kann jede Einzel-LED 256 Helligkeitsstufen der jeweiligen Farbe wiedergeben und damit kann jede Neopixel-Einheit in 256 x 256 x 256 = 16,77 Millionen Farbtönen erstrahlen.

In MicroPython werden Neopixel-Module über einen GPIO-Pin mit den Tools aus der Klasse **NeoPixel** gesteuert, die bereits im Kernel enthalten ist. Nach dem Import dieser Klasse aus dem Modul **neopixel** wird ein Neopixel-Objekt instanziiert. Als Argumente muss man den GPIO-Pin und die Anzahl der LEDs an den Konstruktor übergeben.

```
>>> from neopixel import NeoPixel
>>> np = NeoPixel(Pin(16, 4)
>>> np.buf
bytearray(b'\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')
```

Beim Instanziieren wird ein **bytearray** als Datenpuffer erzeugt, der dreimal so viele Elemente enthält wie LEDs auf dem Panel sind, hier in dem Beispiel also 12. Ich schreibe jetzt für die erste LED das Tupel (64,65,66) in diesen Puffer.

>>> np[0]=(65,66,67)
>>> np.buf
bytearray(b'BAC\x00\x00\x00\x00\x00\x00\x00\x00\x00')

Aus der Folge rot, grün, blau im Tupel wird die Bytefolge für grün, rot, blau im Puffer. So wie die Bytes im Puffer stehen, werden sie auch an die kaskadierten LEDs des Panels mittels der Methode **np.write**() übertragen.

Der Ruhepegel auf der Neopixel-Leitung ist LOW. Die einzelnen Bits werden als Puls-Pause-Kombination gesendet. Die Periodenlänge sollte 1,25 µs betragen, daraus folgt, dass die Übertragungsfrequenz 800kHz ist. Die Pulslänge codiert den Bitwert. Die Pulslänge und die Periodendauer haben eine Toleranz von +/- 150ns.

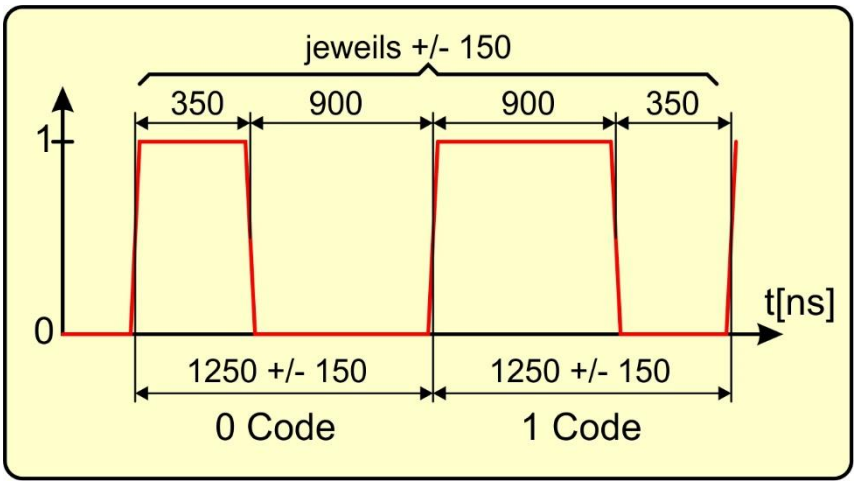

Abbildung 7: Impulsschema der WS2812B-LEDs

Die Attribute eines Neopixel-Objekts kann man mit Hilfe des Object Inspectors auflisten. Den öffnet man über das Menü View.

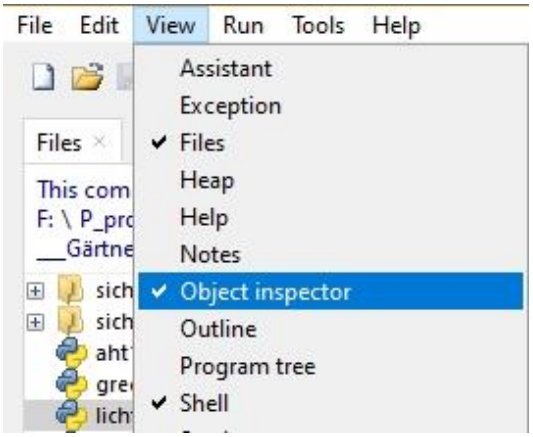

Abbildung 8: Object Inspektor aufrufen

Im Terminal wird einfach das Neopixel-Objekt aufgerufen.

>>> np

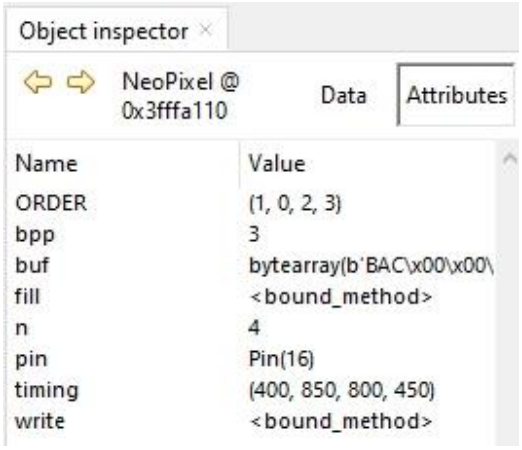

Abbildung 9: Neopixelobjekt im Objekt Inspector

Order gibt die Reihenfolge an, in der die Farbcodes aus dem Tupel (65,66,67) gesendet werden: 66, 65, 67. **bpp** gibt an, dass wir 3 Bytes pro Pixel verwenden. Im Tuple **timing** stehen die Puls- Pause-Zeiten. Sie weichen vom Datenblatt ab, sind aber innerhalb der Toleranzgrenzen. Auch MicroPython arbeitet nicht genau mit diesen Werten, wie ich weiter unten zeigen werde. Falls die Übertragung nicht sauber funktioniert, kann man die Werte des Vierer-Tupels selbst anpassen.

>>> np.timing
(400, 850, 800, 450)
>>> np.timing=(350,900,900,350)
>>> np.timing
(350, 900, 900, 350)

So, wie die Pixel auf dem Panel aneinandergereiht sind, kann man auch mehrere Panele kaskadieren, indem man den Ausgang OUT des einen mit dem Eingang IN des nächsten Panels verbindet. Meine beiden Panele habe ich mit Klebestreifen auf einer 2mm-Glasscheibe fixiert. Links oben führe ich die Spannung vom Netzteil zu, rechts unten kommt die Leitung von GPIO34 des ESP32. **Während der Entwicklungsphase muss unbedingt der GND-Anschluss des Netzteils mit dem GND-Potenzial des Aufbaus verbunden werden, um Spannungsunterschiede zwischen diesen Teilen auszugleichen.**

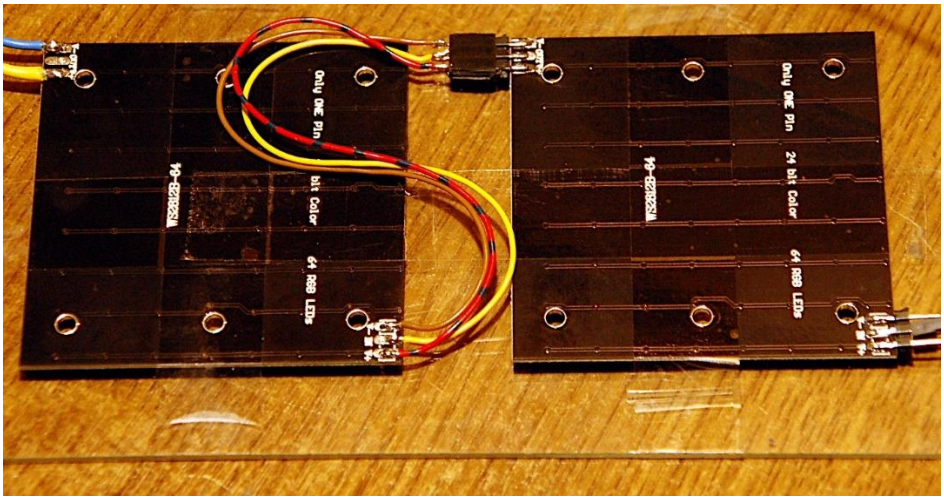

Abbildung 10: Neopixelpanele von hinten

Ein Burst (Impulsfolge) für die 128 Pixel dauert ca. 4ms. Ich habe das mit dem Logic Analyzer und dem kostenlosen Programm Logic 2 aufgezeichnet.

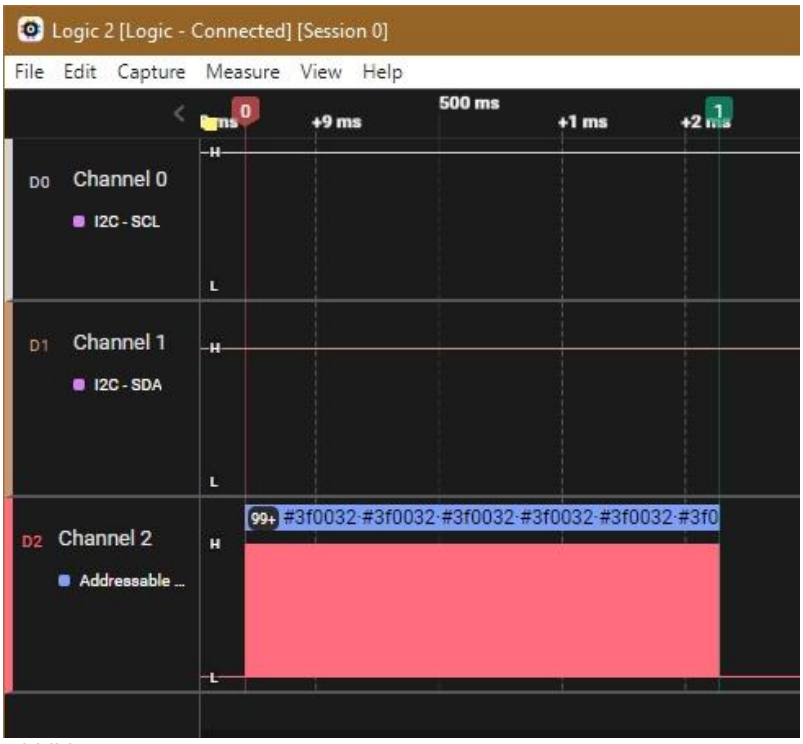

Abbildung 11

Ich zoome jetzt mal in den Anfang rein.

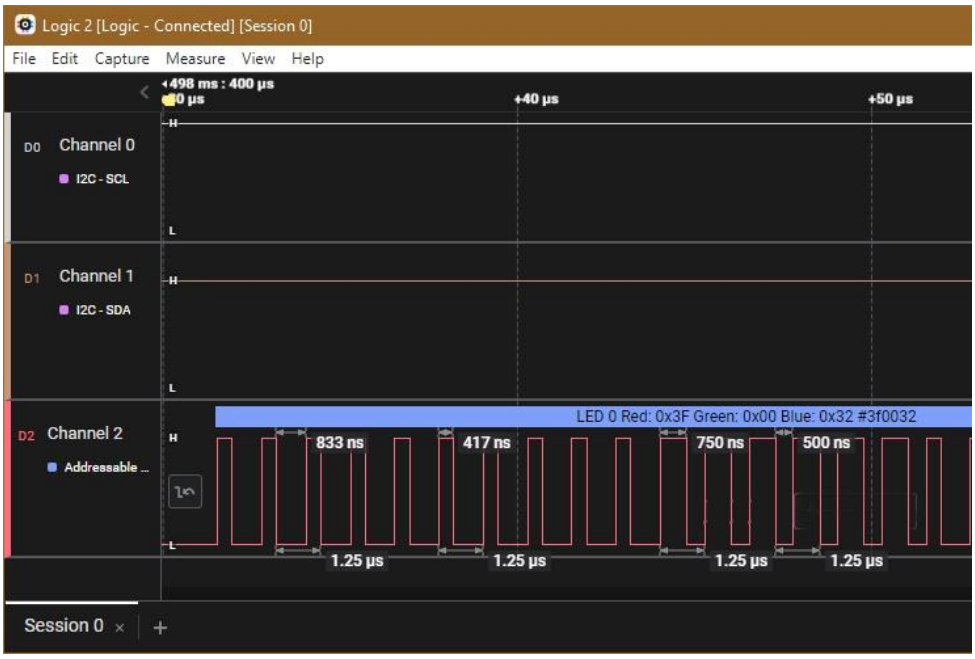

Abbildung 12: Tatsächliche Pulsfolge aufgenommen mit Logic 2

Wie Sie sehen können, weichen die Werte von den Angaben im Objekt Inspector zum Teil erheblich ab, sind aber immer noch im Toleranzbereich des Datenblatts.

Das Beleuchtungsprogramm

Wie die allermeisten MicroPython-Programme beginnt auch das Beleuchtungsprogramm **licht**.py mit dem Importgeschäft.

```
# licht.py
from machine import SoftI2C, Pin, ADC
from time import sleep, ticks_ms, sleep_ms
from oled import OLED
import sys
from neopixel import NeoPixel
```

Die Variable **debug** dient, im Fall von Fehlern oder mystischem Programmverhalten dazu, Textausgaben an neuralgischen Punkten auszulösen, wenn sie auf **True** gesetzt ist.

```
debug=False
```

Die I2C-Schnittstelle brauche ich für das OLED-Display. Der Konstruktor bekommt das I2C-Objekt und die Höhe des Displays in Pixeln.

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

Das ADC-Objekt wird mir die Spannung am LDR einlesen. Wir bekommen Spannungen bis maximal 1 Volt bei völliger Dunkelheit und Werte um 0V bei normalem Tageslicht.

```
ldr=ADC(Pin(34))
ldr.atten(ADC.ATTN_0DB)
ldr.width(ADC.WIDTH_10BIT)
maxCnt=1023
```

Das erste Neopixel-Panel liegt an GPIO16. Zusammen haben beide Panele 128 Pixel. Das Timing wird auf die Angaben des Datenblatts des WS2812B gesetzt und die Variablen **an** und **aus** definieren die Farbcodes für Licht an und aus. Weil der Ruhestrom der Panele recht hoch ist, brauche ich einen General-Lichtschalter, das Relais. **GPIO15** schaltet bei 1 ein und bei 0 aus.

```
neo=16
neoCnt=128
neoPin=Pin(neo,Pin.OUT)
np = NeoPixel(neoPin, neoCnt)
np.timing=(350,900,900,350)
aus=(0,0,0)
an =(63,0,50)
neoRel=Pin(15,Pin.OUT,value=0)
```

Um an einem definierten Punkt das Programm verlassen zu können, nutze ich die Flashtaste des ESP32 als Abbruchtaste. Mit Strg + C würde der Abbruch an einer zufälligen Position erfolgen. Dann weiß man nie, in welchem Zustand die Schaltung gerade ist.

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

Ein paar Funktionen werden deklariert, allen voran **licht**(). Die Funktion nimmt optional ein 3er-Tupel mit RGB-Werten. Als Default wird das Tupel **an** vorgelegt. Ist der Rot-Wert ungleich 0, aktiviere ich das Relais und fülle den Puffer. Das könnte über eine for-Schleife geschehen, aber **fill**() macht das mit einer einzigen Anweisung und zwar mit dem übergebenen Tupel. Alle Pixel leuchten dann in derselben Farbe. Nach einer kurzen Wartezeit, lasse ich den Puffer an die Panele senden. Das ist der Burst aus Abbildung 11.

Ist der Rot-Wert aber 0, wird einfach das Relais und damit die Versorgungsleitung der Panele ausgeschaltet.

```
def licht(val=an):
    r,g,b=val
    if r != 0:
        neoRel.value(1)
        np.fill(val)
        sleep(0.3)
        np.write()
    else:
        neoRel.value(0)
```

getADC() nimmt ein ADC-Objekt und optional einen Wert, der die Anzahl an einzelnen Konvertierungen angibt. Durch das übergebene ADC-Objekt kann die Funktion flexibel auch für die anderen analogen Eingänge genutzt werden. Die Summenvariable **sum** wird auf 0 gesetzt. Die for-Schleife arbeitet mit der temporären Variable _, weil der Laufindex im Schleifenkörper nicht gebraucht wird. Jeder neu eingelesene Wert wird zur bisherigen Summe addiert. Als Ergebnis wird der ganzzahlige Anteil des arithmetischen Mittelwerts zurückgegeben.

```
def getADC(adc,n=50):
    sum=0
    for _ in range(n):
        sum = sum + adc.read()
    avg = int(sum/n)
    return avg
```

Die Funktion **beleuchtung**() braucht das Ergebnis von **getADC**() und ruft daher die Funktion mit dem ADC-Objekt **ldr** als Argument auf. Der Rückgabewert in **h** wird im Terminal von Thonny ausgegeben, wenn **debug** auf **True** gesetzt ist. Der Wert, mit dem h verglichen wird, entscheidet darüber, bei welchem Schummerlicht von draußen die Beleuchtung in der Saatbox eingeschaltet wird. Höhere Werte von h

bedeuten zunehmende Dunkelheit. Den Grenzwert ermitteln Sie am besten empirisch. Zum Ein- und Ausschalten wird **licht**() mit den Variablen **an** und **aus** aufgerufen.

```
def beleuchtung():
    h=getADC(ldr)
    if debug: print("LDR: ",h)
    if h > 140:
        licht(an)
    else:
        licht(aus)
    return h
```

Nach der Ausgabe der Überschrift betreten wir die Main-Loop. Die muss nicht viel tun, Aufruf der Funktion **beleuchtung**(), Ausgabe des Werts im Display, mal schnell eine Sekunde warten und prüfen, ob die Flashtaste gedrückt ist. Wenn ja, Display löschen und Abbruchmeldung ausgeben. Dann vor allem Licht aus und tschüs.

Für den Test reicht es, wenn Sie mit den Werten in **an** = (63,0,25) fahren. Der Strom durch die Panele liegt dann bei 900mA. In Produktionsbetrieb stellen Sie die Helligkeit dann nach Ihren Bedürfnissen ein. Wenn Sie volle Kanne fahren, wird die Stromstärke so um die 3,0 A betragen.

In der nächsten Folge kümmern wir uns um die Wasserversorgung der Pflanzschale. Feuchtefühler werden dabei die Pegelstände in der Wanne und im Vorratsgefäß überwachen, und eine Pumpe besorgt das Nachfüllen der Wanne. Wenn kein Wasser mehr da ist, warnt der Piezo-Piepser, den ich auf ungewöhnliche Weise ansteuere. Lassen Sie sich überraschen.

Bis dann!