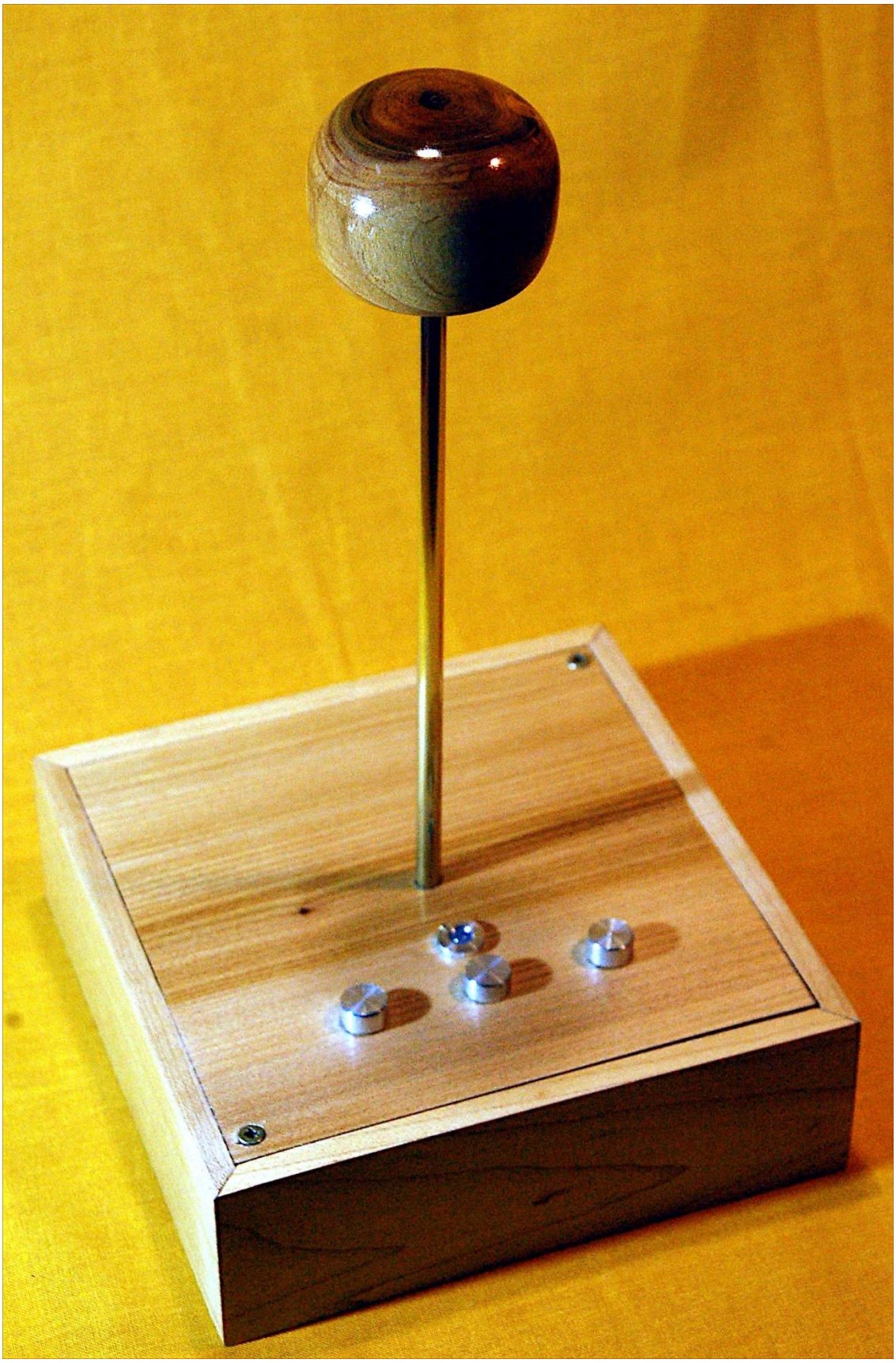

Sensor-Lampy 2.0

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

In dieser Folge benutzen wir dieselbe Hardware wie in der vorangegangenen Episode. Durch die Erweiterung des Programms und eine Umstellung der Philosophie in der Farbverwaltung bekommen wir zwei weitere Steuerkanäle hinzu und machen die Helligkeitssteuerung geschmeidiger. Freuen Sie sich auf eine neue Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Sensorlampe 2.0

Der experimentelle Aufbau auf Breadboards ist derselbe wie im ersten Teil. Dennoch will ich Ihnen die Hardware und die Gründe für die Auswahl der Teile nicht vorenthalten.

Die Hardware

Ein ESP32, eine RGB-LED, ein Neopixel-Ring und, für den Anfang, drei Platinenstücke stellen die Hardware für dieses Projekt. Daneben brauchen wir natürlich die entsprechende mechanische Basis, damit aus der reinen Elektronik ein einsetzbares Gerät wird, dazu kommen wir später.

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	RGB LED Ring 8 bit WS2812 5050 + integrierter Treiber
1	KY-016 FZ0455 3-Farben RGB LED Modul 3 Color oder LED Leuchtdioden Sortiment Kit, 350 Stück, 3mm & 5mm, 5 Farben - 1x Set
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord
1	Widerstand 560 Ω
2	Widerstand 10kΩ
optional	Logic Analyzer

Für das verwendete Controllerboard ESP32 Dev Kit C V4 braucht man für die Entwicklung der Schaltung zwei Breadboards, die über eine Stromschiene seitlich zusammengesteckt werden. Nur so bekommt man genug freie Kontaktstellen für die Jumperkabel.

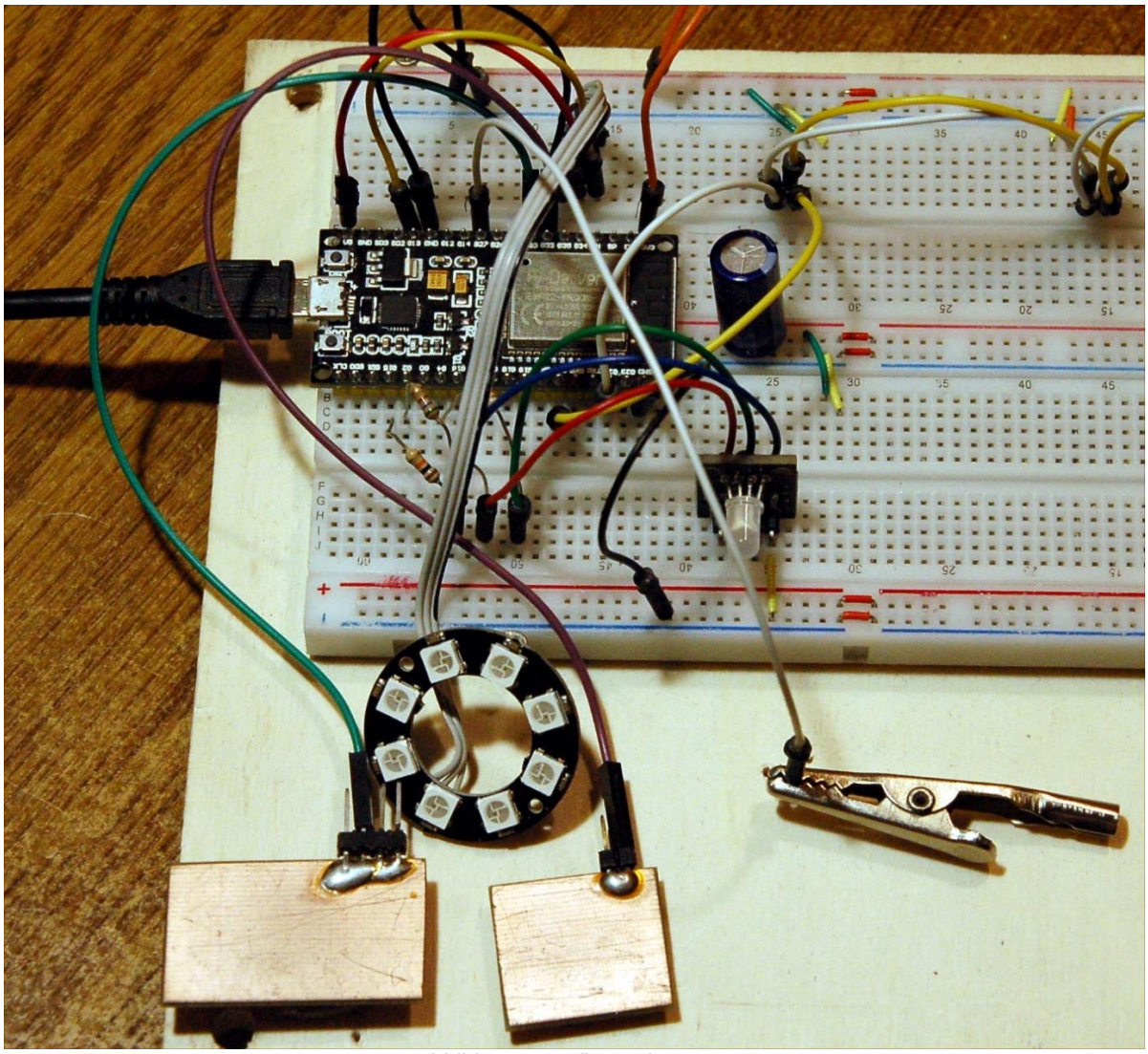

Abbildung 1: Aufbau mit ESP32

Als Touchpads habe ich Reststücke einer Platine und eine Krokse benutzt. Im Produktionssystem werden dafür kleine Metallzylinder dienen.

Die RGB-LED verwende ich zum Anzeigen des Betriebszustands. Die Farben der LEDs am Ring können einzeln rauf- und runtergeregelt werden. Welche Farbe grade dran ist, zeigt die RGB-LED an. Insgesamt wird es fünf Zustände geben: rot, grün, blau, weiß und ein-aus.

Die Vorwiderstände sind so dimensioniert, dass zusammen etwa weißes Licht herauskommt. Die blaue und vor allem die grüne LED sind um vieles heller als die rote. Daher rühren die großen Unterschiede in den Widerstandswerten. Wer eine hellere Anzeige möchte, nimmt kleinere Ohmwerte.

Wie alles zusammengehört, das zeigt das Schaltbild in Abbildung 2.

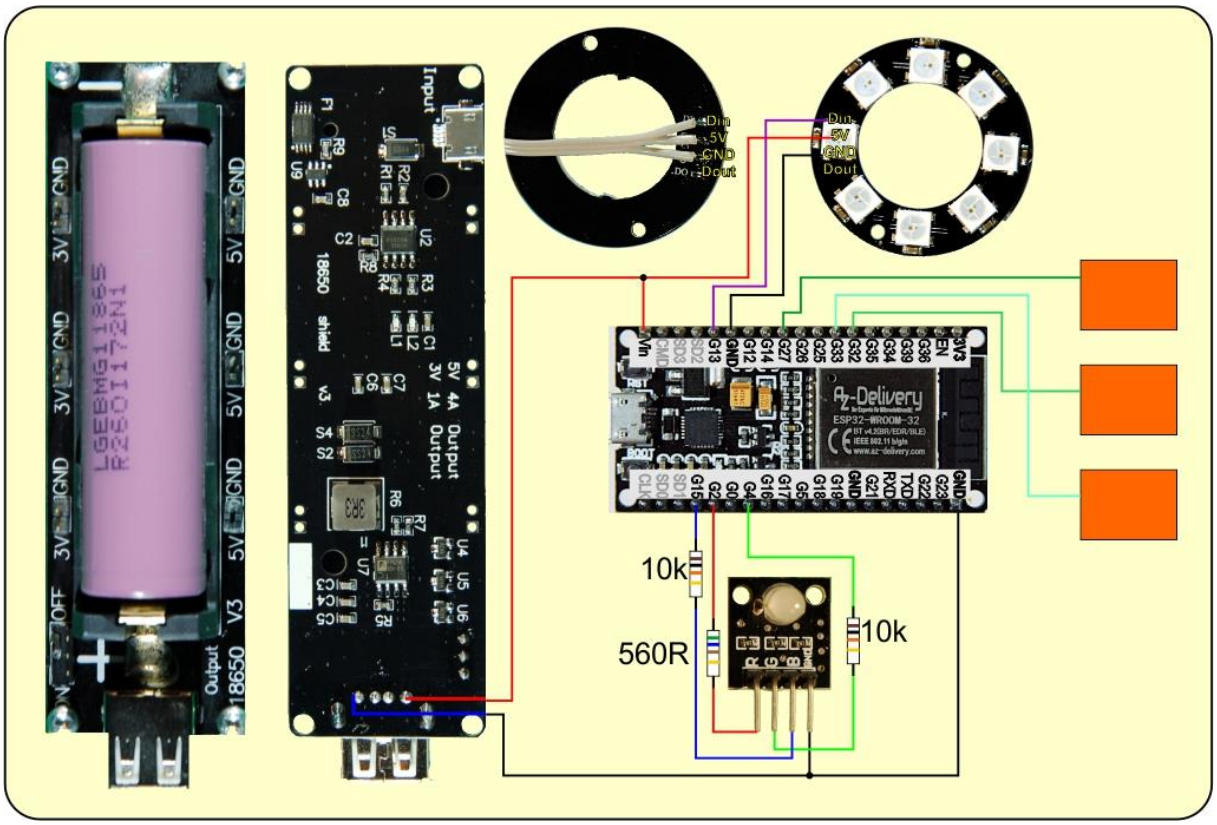

Abbildung 2: Schaltung mit ESP32 und Touchpads

Als Energieversorgung habe ich eine Li-Ion-Zelle vom Typ 18650 gewählt und dazu einen Batteriehalter mit Ladeteil und 5V Ausgang. An die Anschlusspins der USB-A-Buchse habe ich die Versorgungsleitungen für meine Schaltung gelötet. Auf diese Weise komme ich ohne USB-Stecker aus und kann trotzdem den mechanischen Schalter an der Platine nutzen. Er wird im Gehäuse in Richtung Boden zeigen, ebenso wie die Ladebuchse am oberen Ende der Platine in Abbildung 2.

Weil in der Lampe für die Breadboards zu wenig Platz sein wird, habe ich eine Platine entworfen, auf der die wenigen Einzelteile Platzfinden. Sie können das Layout als PDF-Datei herunterladen.

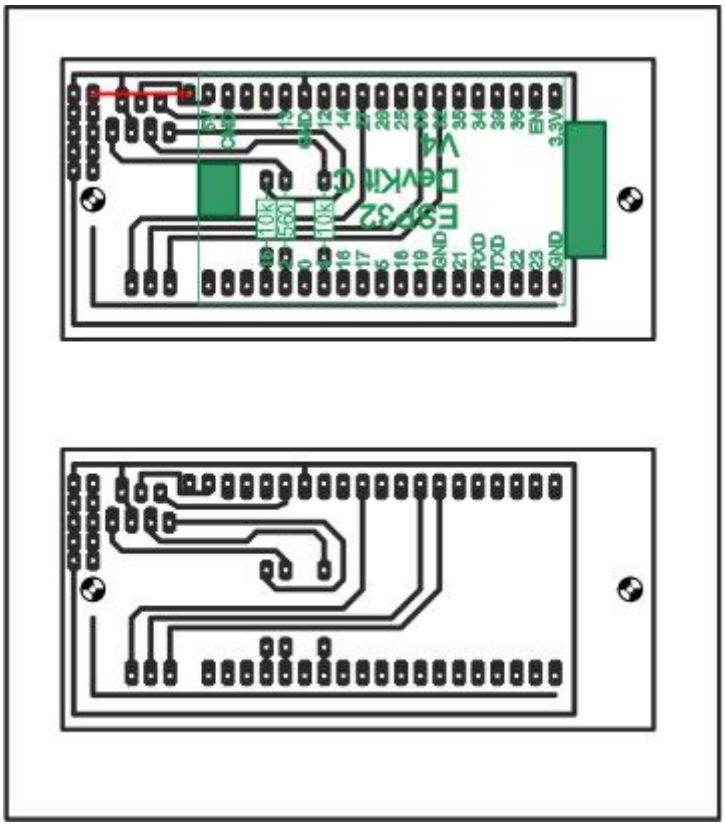

Abbildung 3: Sensorlampe - Layout

Im Versuchsaufbau habe ich ein RGB-LED-Modul verwendet. Für die Lampe selbst fand ich eine der LEDs aus dem Sortiment optimaler, wegen der einfacheren Montage im Deckel.

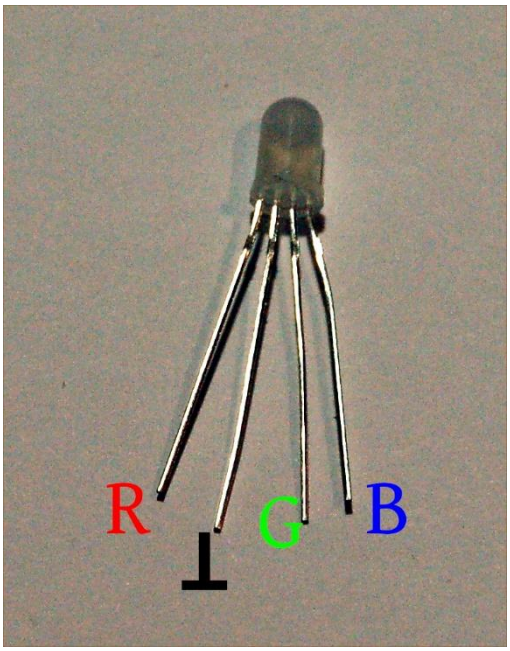

Abbildung 4: RGB-LED mit gemeinsamer Kathode

Die Lampe

Wie die Lampe aufgebaut ist habe ich bereits im ersten Teil ausgeführt. In dieser Folge möchte ich mich auf die Änderungen im Programm konzentrieren und bitte Sie darum, für die Konstruktion der Lampe an sich die vorangegangene Episode zu konsultieren. Dort finden Sie auch einen bemaßten Schnittmusterplan.

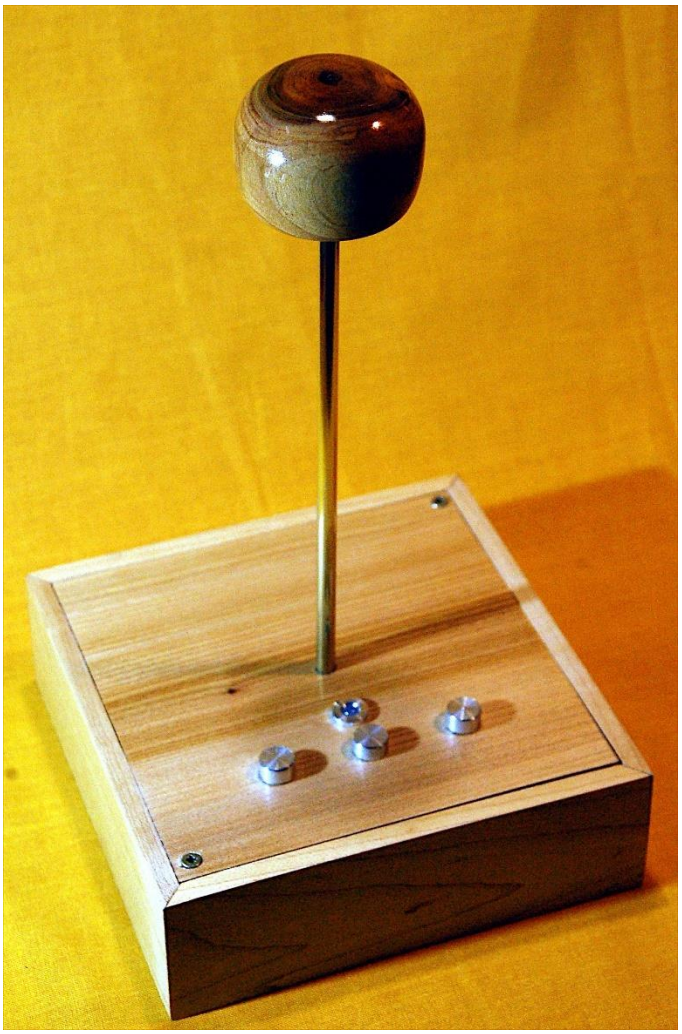

Abbildung 5: Sensor-Lampy 2.0

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Signalverfolgung:

Saleae Logic 2

Verwendete Firmware für einen ESP32:

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

Die MicroPython-Programme zum Projekt:

timeout.py Nichtblockierender Software-Timer
touch.py Touchpad-Modul
touchlampy.py Betriebsprogramm
touchlampy_2.py Betriebsprogramm Ausbaustufe 1
touchlampy_3.py Betriebsprogramm Ausbaustufe 2
touchlampy_4.py Betriebsprogramm Ausbaustufe 3

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.

Der Neopixelring

Auch zu Neopixelring, dessen Funktionsweise und dem Modul NeoPixel habe ich im letzten Beitrag schon das Wesentliche gesagt. Wenn Sie Näheres zur Thematik erfahren möchten, empfehle ich die Lektüre des gleichnamigen Kapitels aus dieser Episode.

Die Programmerweiterungen

Programmgesteuerte Sequenz

Beginnen wir mit dem Aufbau der programmgesteuerten Farbsequenz. An den Vorbereitungen hat sich fast nichts verändert. Die Stellen, an denen gefeilt wurde, habe ich im Listing fett formatiert. Als Erstes habe ich eine sechste Kanalnummer eingeführt und ihr die Signalfarbe gelb zugewiesen.

```
# touchlampy_2.py

from touch import TP
from machine import Pin
from neopixel import NeoPixel as NP
from time import sleep, sleep_ms
import uasyncio as asyncio
from esp32 import NVS

# rise = asyncio.Event()
# fall = asyncio.Event()

nvs=NVS("config")

tp1=TP(33, 350)
tp2=TP(32, 350)
tp3=TP(27, 350)

neoPin=Pin(13,Pin.OUT)
neo=NP(neoPin,8)
kanal=0 # sw=0, rt=1, gn=2, bl=3, ws=4, ge=5
color=[0,0,0]
```

```
def storeColor():
    nvs.set_i32("red",color[0])
    nvs.set_i32("green",color[1])
    nvs.set_i32("blue",color[2])
    nvs.commit()
    print("Farbcode gesichert",color)


def loadColor():
    rt=nvs.get_i32("red")
    gn=nvs.get_i32("green")
    bl=nvs.get_i32("blue")
    print("Farbcode geladen:",[rt,gn,bl])
    return [rt,gn,bl]

try:
    color=loadColor()
except:
    storeColor()

merken=color
gemerkt=False
freeze=color
step=2
m=0
ledR=Pin(2,Pin.OUT,value=0)
ledB=Pin(15,Pin.OUT,value=0)
ledG=Pin(4,Pin.OUT,value=0)
leds=(ledR,ledG,ledB)
#         aus     rot    gruen   blau    weiss    gelb
shapes=((0,0,0),(1,0,0),(0,1,0),(0,0,1),(1,1,1),(1,1,0),)
channels=("red","green","blue","white","yellow")
mode=len(shapes)
program=0
```

Für die programmgesteuerte Farbfolge definieren wir vorab die Variablen **freeze**, **step** und **m** und für die Ansteuerung der Signal-LED kommt das sechste Muster-Tupel (1,1,0) hinzu, rot und grün ergibt gelb. Das Tupel **channels** wird um den Eintrag "yellow" erweitert. Um die Skalierbarkeit der Kanäle zu verbessern, arbeiten wir nicht mehr mit Konstanten als Angabe für die Anzahl an Kanälen, sondern bestimmen deren Anzahl automatisch aus der Länge des Tupels **shapes**. Das Flag **program** dient später zum Ein- und Ausschalten des automatischen Farbwechsels.

Die Funktion **lum**() übernehmen wir ohne Änderung. Sie überträgt die aktuelle Farbe in den Puffer des NeoPixel-Objekts und schickt dessen Inhalt an den Ring.

```
def lum():
    for i in range(8):
        neo[i]=color
    neo.write()
```

Jetzt kommt wieder das Modul **asyncio** ins Spiel. Wir definieren damit drei Funktionen, welche die Touch-Steuerung übernehmen und eine Funktion, die das Abspielen der Farbfolge managt. Mit **tp1** schalten wir die Kanäle durch, speichern die aktuelle Farbe in **NVS.config** und schalten den Ring dunkel. Was jeweils geschehen soll, decodieren wir über die Zeitdauer, mit der das Pad berührt wird.

Eine Funktion, die einen Prozess im asyncio-System darstellen soll, muss mit **async def** eingeleitet werden. Der Prozess wird beendet, sobald die Funktion verlassen wird. Das wollen wir nicht, daher läuft der Prozess als Endlosschleife. Die Objekte **kanal** und **color** erfahren potenzielle Änderungen, die wir andernorts abfragen müssen, deshalb deklarieren wir sie beim Start der Funktion als **global**.

Das normale **sleep_ms**() aus **time** ersetzen wir durch die **asyncio**-Variante **asyncio.sleep_ms**(). Mit der Zeile

await asyncio.sleep_ms(10)

signalisieren wir dem System, dass der Prozess an dieser Stelle unterbrochen werden darf, um andere Prozesse mit Zeitfenstern zu bedienen.

Die grundsätzliche Arbeitsweise der Funktion **switchChannel**() ist bereits im ersten Teil erklärt. Nur beim Ringzähler für das Erhöhen der Steuerkanalnummer wurde die Konstante 5 durch die Variable **mode** ersetzt.

```
async def switchChannel():
    global kanal, color
    while 1:
        await asyncio.sleep_ms(10)
        if tp1.touched():
            tp1.getDuration()
            hold=tp1.dauer()
            print("Dauer",hold)
            if hold < 500:
                kanal = (kanal + 1) % mode
                showChannel(kanal)
            elif 500 <= hold <2000:
                storeColor()
            else:
                kanal=0
                color=[0,0,0]
                showChannel(kanal)
                lum()
```

Auch die Plausibilitätsprüfung auf die Kanalnummer in **showChannel**() erfolgt jetzt über **mode**.

```
def showChannel(num):
    assert num in range(mode)
    for i in range(3):
        leds[i](shapes[num][i])
```

An die Funktion **increase**() wurde die Behandlung von Kanal 5 angehängt. Weil auf die Speicherinhalte von **program**, **m** und **freeze** außerhalb der Funktion oder des Prozesses Zugriff möglich sein muss, werden auch diese Variablen als global definiert.

```
async def increase():
    global color,merken,gemerkt,program,m,freeze
    while 1:
        await asyncio.sleep_ms(10)
        if tp2.touched():
            n=0
            print("increase",kanal)
            while tp2.touched():
                await asyncio.sleep_ms(50)
                if 1 <= kanal <= 3:
                    ptr=kanal-1
                    col=color[ptr]
                    if n < 10:
                        col = col + 1
                    else:
                        col = col + 5
                    col = min(col,255)
                    color[ptr]= col
                    print(channels[ptr], color)
                    lum()
                elif kanal == 4:
                    for i in range(3):
                        col=int(color[i] + 3)
                        color[i]=min(col,255)
                    print("Lumineszenz", color)
                    lum()
                elif kanal == 0:
                    color=merken
                    print(color)
                    lum()
                    gemerkt=False
                elif kanal == 5:
                    if not gemerkt:
                        freeze=color
                        gemerkt=True
                        m=0
                        color=[0,0,0]
                        lum()
                        program=1
                        print("auto gestartet",m,freeze)
                n+=1
```

Kanal 5 startet die programmgesteuerte Farbabfolge und ist durch die gelbe Signal-LED-Farbe erkennbar. Wenn bei aktivem Kanal 5 **increase**() noch nicht aufgerufen wurde (Touch am GPIO32-Pad) oder wenn ein Touch an GPIO27-Pad vorausging, ist **gemerkt** auf **False** und die Sequenz wird durchlaufen. Wir merken uns den Farbcode in **freeze** und setzen **gemerkt** auf True. Dieses Vorgehen erlaubt nur

einen Einzelschuss und schützt davor, dass **freeze** in einem weiteren Durchlauf auf die Farbe "schwarz" = [0,0,0] gesetzt wird. Die Zählervariable **m** wird auf 0 gesetzt, die NeoPixel-LEDs machen wir aus. Damit sind alle Vorbereitungen getroffen. Mit dem Flag **program** signalisieren wir dem Prozess, der die Farbfolge durchführt (Funktion **auto**()), dass er mit der Abarbeitung loslegen kann.
In ähnlicher Weise arbeitet die Funktion **decrease**(), mit deren Aufruf die Automatik angehalten werden kann.

```
async def decrease():
    global color,merken,gemerkt,program,freeze
    while 1:
        await asyncio.sleep_ms(10)
        if tp3.touched():
            n=0
            print("decrease",kanal)
            while tp3.touched():
                await asyncio.sleep_ms(50)
                if 1 <= kanal <= 3:
                    ptr=kanal-1
                    col=color[ptr]
                    if n < 10:
                        col = col - 1
                    else:
                        col = col - 5
                    col = max(col,0)
                    color[ptr]= col
                    print(channels[ptr], color)
                    lum()
                elif kanal == 4:
                    for i in range(3):
                        col=int(color[i] - 3)
                        color[i]=max(col,0)
                    print("Lumineszenz", color)
                    lum()
                elif kanal == 0:
                    if not gemerkt:
                        merken=color
                        print(color)
                        color=[0,0,0]
                        lum()
                        gemerkt=True
                elif kanal == 5:
                    if gemerkt:
                        program=0
                        gemerkt=False
                        await asyncio.sleep_ms(50)
                        color=freeze
                        print("auto gestoppt",freeze)
                        lum()
                n+=1
```

Nur wenn gemerkt den Zustand True aufweist, zuvor also eine Farbe in **freeze** gebunkert wurde, darf die Sequenz durchlaufen. Mit **program** = 0 senden wir das Stoppsignal an die Funktion **auto**() und setzen das Flag gemerkt zurück. Dann müssen wir warten, bis der dort gerade durchgeführte Schleifendurchlauf beendet ist. Durch **await asyncio.sleep_ms(50)** geben wir dem Prozess die Möglichkeit das zu tun. Danach restaurieren wir die zuvor eingestellte Farbe und schicken die Daten zum Ring.

Hier kommt nun die komplett neue Funktion **auto**(), die den Prozess für die Farbfolge festlegt. Auch sie ist asynchron definiert, damit man über die Touchpads jederzeit Änderungswünsche anmelden kann.

Die äußre while-Schleife kapselt den Prozess und sorgt, wie bei den anderen Corroutinen als Endlosschleife dafür, dass der Prozess nicht abbricht. Durch **await asyncio.sleep_ms(10)** bekommen die anderen Prozesse auch Zeitscheiben ab. Wenn jetzt **program** den Wert 1 hat, beginnt ein neuer Schleifendurchlauf. Auch die innere while-Schleife muss den anderen Prozessen die Gelegenheit geben, eine Zeitscheibe zu ergattern, also **await asyncio.sleep_ms(10)**.

Über die Zählervariable **m** decodieren wir, welche Farbsequenz ablaufen soll. Mit **step** können wir angeben, in welcher Schrittweite die Farbänderung erfolgen soll. Zweierpotenzen sind dafür sehr gut geeignete Werte. Die Farbänderung wird durch Addieren oder Subtrahieren der Schrittweite erreicht. Wir starten mit einem anschwellenden Rot und hängen in der nächsten Scheibe über die Mischfarbe gelb ein finales Grün an. Dann nehmen wir das Grün zurück, steigern rot bis 255 und blau bis 127. In der letzten Schicht nehmen wir die blauen LEDs bis 0 zurück.

```
async def auto():
    global program, m, color
    print("Automatik gestartet")
    while 1:
        await asyncio.sleep_ms(10)
        if program==1:
            await asyncio.sleep_ms(10)
            while program == 1:
                await asyncio.sleep_ms(10)
                if 0 < m < 256//step:
                    color[0]=(color[0]+step)
                    print("\nRunde1")
                elif 256//step < m < 512 // step:
                    color[0]=(color[0]-step)
                    color[1]=(color[1]+step)
                    print("\nRunde2")
                elif 512 // step < m < 768 // step:
                    color[0]=(color[0]+ step)
                    color[1]=(color[1]- step)
                    color[2]=(color[2]+ step//2)
                    print("\nRunde3")
                elif 768//step < m < 1024//step:
                    color[0]=(color[0]- step)
                    color[2]=(color[2]+ step//2)
                    print("\nRunde4")
```

```
                elif 1024//step < m < 1280//step:
                    color[1]=(color[1]+ step)
                    color[2]=(color[2]- step//2)
                    print("\nRunde5")
                elif 1280//step < m < 1536//step:
                    color[1]=(color[1]- step)
                    print("\nRunde6")
                elif 1536//step < m < 1792//step:
                    color[2]=(color[2]- step// step)
                    print("\nRunde7")

                print(m, color)
                lum()
                m=(m+1) % (1792//step)
```

In der Testphase habe ich mir den Zähler und die entsprechende Farbkombi ausgeben lassen, aber diese Zeile kann man getrost streichen, wenn das System läuft. Aktuellen Farbcode an den Ring schicken und Ringzähler erhöhen. Dann beginnen wir mit dem nächsten Durchlauf, falls **program** immer noch 1 ist,

Für die Prozesse müssen wir jetzt nur noch eine Eventloop und die Tasks einrichten und die ganze Maschine starten. Das passiert in der Coroutine **main() und im nachfolgenden asyncio.run(main())**.

```
async def main():
    loop = asyncio.get_event_loop()
    loop.create_task(switchChannel())
    loop.create_task(increase())
    loop.create_task(decrease())
    loop.create_task(auto())
    loop.run_forever()


lum()

asyncio.run(main())
```

Die zufällige Farbfolge

Das größte Chaos entsteht durch drei Farben wohl dadurch, dass die RGB-Leds des Rings zu unterschiedlichen Zeiten mit unterschiedlicher Dauer unterschiedlich schnell rauf- und runtergefahren werden. Was wäre dafür besser geeignet, als ein Parallelbetrieb von drei Prozessen, die sich gegenseitig den Zugriff auf die Farbgebung mehr oder weniger zufällig zuspielen. Hoch leben die Coroutinen von **asyncio**.

Es ist nun zweitrangig, ob wir für diesen Ansatz einen neuen Kanal einführen oder einfach nur die Funktion **auto**(), die wir gerade besprochen haben , durch eine neue ersetzen wollen. Damit an dem Programm keine weiteren Änderungen stattfinden

müssen, habe ich den zweiten Weg gewählt. Sicher fällt es Ihnen nicht schwer, auch einen neuen Kanal anzulegen. Den Weg dazu sind wir ja gerade erst gegangen.

Basteln wir uns also eine neue Funktion **auto**(), diesmal aber mit einer Parameterübergabe. Das übergebene Argument soll der Farbindex sein, der Wert 0, 1 oder 2 für rot, grün und blau.

Interessant ist bei dem Ansatz, dass eine Funktionsdefinition für alle drei Prozesse hergenommen werden kann. Es wird also drei Instanzen der Coroutine **auto**() geben, die unabhängig voneinander arbeiten. Aber alle drei Prozesse greifen auf dieselben Objekte **program** und **color** zu. Jeder Prozess wird aber nur seinen Anteil am Farbcode verändern. Jedoch richten sich alle drei nach dem Wert im Flag **program**.

Überlegen wir einmal, welche Parameter die Farbfolge beeinflussen können. Für diese Parameter werden wir dann geeignete Zufallszahlen würfeln lassen. Das kann das Modul **random**, das wir oben bei den Importen mit berücksichtigen müssen.

```
import random
```

Die maximale Leuchtkraft beschreiben wir mit **lumMax**, dann sind da noch die Schrittweite **step**, die Haltedauer **duration** und die Richtung **direction**. Die lokale Variable **col** enthält den Farbcode und mit **steps** legen wir die maximale Anzahl von Durchgängen fest. Diesen Wert erhalten wir, wenn wir die maximale Leuchtkraft durch die Schrittweite dividieren, ganzzahlig versteht sich. Je nach der gewürfelten Richtung muss **col** entweder mit 0 vorbelegt werden – beim Hochzählen oder mit **lumMax** – beim Herunterzählen. Den Durchgangszähler **m** setzen wir auf 0. Farbe zuweisen und zum Ring schicken, dann starten wir die innere Schleife.

```
async def auto(index):
    global program, color
    print("Random gestartet")
    while 1:
        await asyncio.sleep_ms(10)
        if program==1:
            await asyncio.sleep_ms(10)
            lumMax=random.randint(0,160)
            step=random.randint(1,5)
            duration=random.randint(10,100)
            direction=random.getrandbits(1)
            col=0 if direction==1 else lumMax
            steps=lumMax // step
            m=0
            color[index]=col
            lum()
            while program == 1:
                await asyncio.sleep_ms(duration)
                if direction==1:
                    col=col+step
                    col=min(col,lumMax)
                else:
                    col=col-step
                    col=max(col,0)
```

```
                color[index]=col
                lum()
                print(index, m, color)
                m += 1
                if m >= steps:
                    break
```

Die aktuelle Farbintensität wird **duration** Millisekunden gehalten. Wir erhöhen den Farbwert, wenn **direction** 1 ist, sorgen aber dafür, dass **lumMax** nicht überschritten wird. Absteigend darf 0 nicht unterschritten werden.

Den neuen Wert bauen wir in die Liste **color** ein und senden den Code an den Ring. Wenn alles wunschgemäß funktioniert, kann die **print**-Zeile entfernt werden.

Der Durchlaufzähler wird erhöht und wenn die Anzahl zulässiger Schritte erreicht ist, verlassen wir die innere Schleife. Sofern **program** immer noch 1 ist, würfeln wir erneut und starten mit den neuen Einstellungen.

Die Coroutine **auto**() wird nun mit dem Farbindex 0, 1 oder 2 jeweils einem Task zugewiesen.

```
async def main():
    loop = asyncio.get_event_loop()
    loop.create_task(switchChannel())
    loop.create_task(increase())
    loop.create_task(decrease())
    loop.create_task(auto(0))
    loop.create_task(auto(1))
    loop.create_task(auto(2))
    loop.run_forever()


lum()
```

Eine geschmeidigere Farbverwaltung

Jetzt steht noch eine bessere Lösung für das Anheben und Absenken eines Farbmischtons aus. Das Problem besteht darin, dass bisher beim Anheben der Helligkeit irgendwann der Farbcode (255,255,255) erreicht wird und dadurch die ursprüngliche Einstellung des Farbtons verloren geht.

In diesem neuen Ansatz stelle ich die gewünschte Lichtfarbe ein. Die erreichte Leuchtstärke werte ich als 100%. Wird nun der Kanal 4 ausgewählt, dann kann mit den Plus- Minus-Pads die Helligkeit in hundert Stufen abgesenkt werden. Das gelingt durch die Einführung eines Faktors für die Helligkeit, der bei der Berechnung der Komponentenwerte berücksichtigt wird.

Natürlich bedarf es dafür einer erweiterten Funktion **lum**() und einiger Anpassungen an verschiedenen Stellen. Die Passagen sind im Gesamtlisting des Programms **touchlampy_4.py fett** formatiert.

```
# touchlampy_4.py

from touch import TP
from machine import Pin
from neopixel import NeoPixel as NP
from time import sleep, sleep_ms
import uasyncio as asyncio
from esp32 import NVS
import random

nvs=NVS("config")

tp1=TP(33, 350)
tp2=TP(32, 350)
tp3=TP(27, 350)

neoPin=Pin(13,Pin.OUT)
neo=NP(neoPin,8)
kanal=0 # sw=0, rt=1, gn=2, bl=3, ws=4, ge=5
color=[0,0,0] # rt, gn, bl
lumi=100
col=[0,0,0]

def storeColor():
    nvs.set_i32("red",color[0])
    nvs.set_i32("green",color[1])
    nvs.set_i32("blue",color[2])
    nvs.set_i32("lumi",lumi)
    nvs.commit()
    print("Farbcode gesichert",color,lumi)


def loadColor():
    global color, lumi
    rt=nvs.get_i32("red")
    gn=nvs.get_i32("green")
    bl=nvs.get_i32("blue")
    lumi=nvs.get_i32("lumi")
    print("Farbcode geladen:",[rt,gn,bl],lumi)
    color=[rt,gn,bl]

try:
    loadColor()
except:
    storeColor()

merken=color
merkenL=lumi
gemerkt=False

step=2
m=0
```

```
ledR=Pin(2,Pin.OUT,value=0)
ledB=Pin(15,Pin.OUT,value=0)
ledG=Pin(4,Pin.OUT,value=0)
leds=(ledR,ledG,ledB)
# Signal: aus     rot    gruen   blau    weiss    gelb
shapes=((0,0,0),(1,0,0),(0,1,0),(0,0,1),(1,1,1),(1,1,0),)
channels=("red","green","blue","lumi","auto")
mode=len(shapes)
program=0

def aus():
    global color
    color=[0,0,0]
    lum()

def lum(lumi):
    global col
    for i in range(3):
        col[i]=int(color[i]/100*255*lumi/100)
    for i in range(8):
        neo[i]=col
    neo.write()

async def switchChannel():
    global kanal, color
    while 1:
        await asyncio.sleep_ms(10)
        if tp1.touched():
            tp1.getDuration()
            hold=tp1.dauer()
            print("Dauer",hold)
            if hold < 500:
                kanal = (kanal + 1) % mode
                showChannel(kanal)
            elif 500 <= hold <2000:
                storeColor()
            else:
                kanal=0
                color=[0,0,0]
                showChannel(kanal)
                lum()

def showChannel(num):
    assert num in range(mode)
    for i in range(3):
        leds[i](shapes[num][i])

async def increase():
    global color,merken,merkenL,gemerkt,program,m,lumi
    while 1:
        await asyncio.sleep_ms(10)
        if tp2.touched():
```

```
            n=0
            print("increase",kanal)
            while tp2.touched():
                await asyncio.sleep_ms(50)
                if 1 <= kanal <= 3:
                    ptr=kanal-1
                    col=color[ptr]
                    if n < 10:
                        col = col + 1
                    else:
                        col = col + 5
                    col = min(col,100)
                    color[ptr]= col
                    print(channels[ptr], color)
                    lum(100)
                elif kanal == 4:
                    if n < 10:
                        lumi = lumi + 1
                    else:
                        lumi = lumi + 5
                    lumi = min(lumi,100)
                    print("Lumineszenz", color, lumi)
                    lum(lumi)
                elif kanal == 0:
                    if gemerkt:
                        color=merken
                        lumi=merkenL
                        print(color)
                        lum(lumi)
                        gemerkt=False
                elif kanal == 5:
                    if not gemerkt:
                        merken=color
                        merkenL=lumi
                        gemerkt=True
                        m=0
                        color=[0,0,0]
                        lum(0)
                        program=1
                        print("auto gestartet",m,freeze)
                n+=1

async def decrease():
    global color,merken,merkenL,m,gemerkt,program,lumi
    while 1:
        await asyncio.sleep_ms(10)
        if tp3.touched():
            n=0
            print("decrease",kanal)
            while tp3.touched():
                await asyncio.sleep_ms(50)
                if 1 <= kanal <= 3:
```

```
                    ptr=kanal-1
                    col=color[ptr]
                    if n < 10:
                        col = col - 1
                    else:
                        col = col - 5
                    col = max(col,0)
                    color[ptr]= col
                    print(channels[ptr], color)
                    lum(100)
                elif kanal == 4:
                    if n < 10:
                        lumi = lumi - 1
                    else:
                        lumi = lumi - 5
                    lumi = max(lumi,0)
                    print("Lumineszenz", color, lumi)
                    lum(lumi)
                elif kanal == 0:
                    if not gemerkt:
                        merken=color
                        merkenL=lumi
                        print(color)
                        color=[0,0,0]
                        lum(0)
                        gemerkt=True
                elif kanal == 5:
                    if gemerkt:
                        program=0
                        gemerkt=False
                        await asyncio.sleep_ms(50)
                        color=merken
                        lumi=merkenL
                        print("auto gestoppt",merken)
                        lum(lumi)
                n+=1

async def auto(index):
    global program, color
    print("Random gestartet")
    while 1:
        await asyncio.sleep_ms(10)
        if program==1:
            await asyncio.sleep_ms(10)
            lumMax=random.randint(0,100)
            step=random.randint(1,3)
            duration=random.randint(50,300)
            direction=random.getrandbits(1)
            col=0 if direction==1 else lumMax
            steps=lumMax // step
            m=0
            color[index]=col
```

```
            lum(100)
            while program == 1:
                await asyncio.sleep_ms(duration)
                if direction==1:
                    col=col+step
                    col=min(col,lumMax)
                else:
                    col=col-step
                    col=max(col,2)
                color[index]=col
                print(index, m, color)
                lum(100)
                m += 1
                if m >= steps:
                    break

async def main():
    loop = asyncio.get_event_loop()
    loop.create_task(switchChannel())
    loop.create_task(increase())
    loop.create_task(decrease())
    loop.create_task(auto(0))
    loop.create_task(auto(1))
    loop.create_task(auto(2))
    loop.run_forever()


lum(lumi)

asyncio.run(main())
```

In dieser und der letzten Episode konnten wir die Features des ESP32 und MicroPython gut nutzen. Im Vordergrund standen vor allem die nichtflüchtige Speicherung im NVS-Bereich, die Touchpads und das parallele Ausführen von Aufgaben mit asyncio. Der kleine Bruder, der ESP8266, kann diese angenehmen Eigenschaften leider nicht von sich aus bieten. Dennoch kann man ihn, mit mehr externem Equipment, dazu bewegen, mit nahezu demselben Programm dieselben Ergebnisse zu liefern. Wie das beim ESP8266 und auch beim Raspberry Pi Pico funktioniert, das erfahren Sie in der nächsten Blogfolge.

Bis dann, bleiben Sie dran!