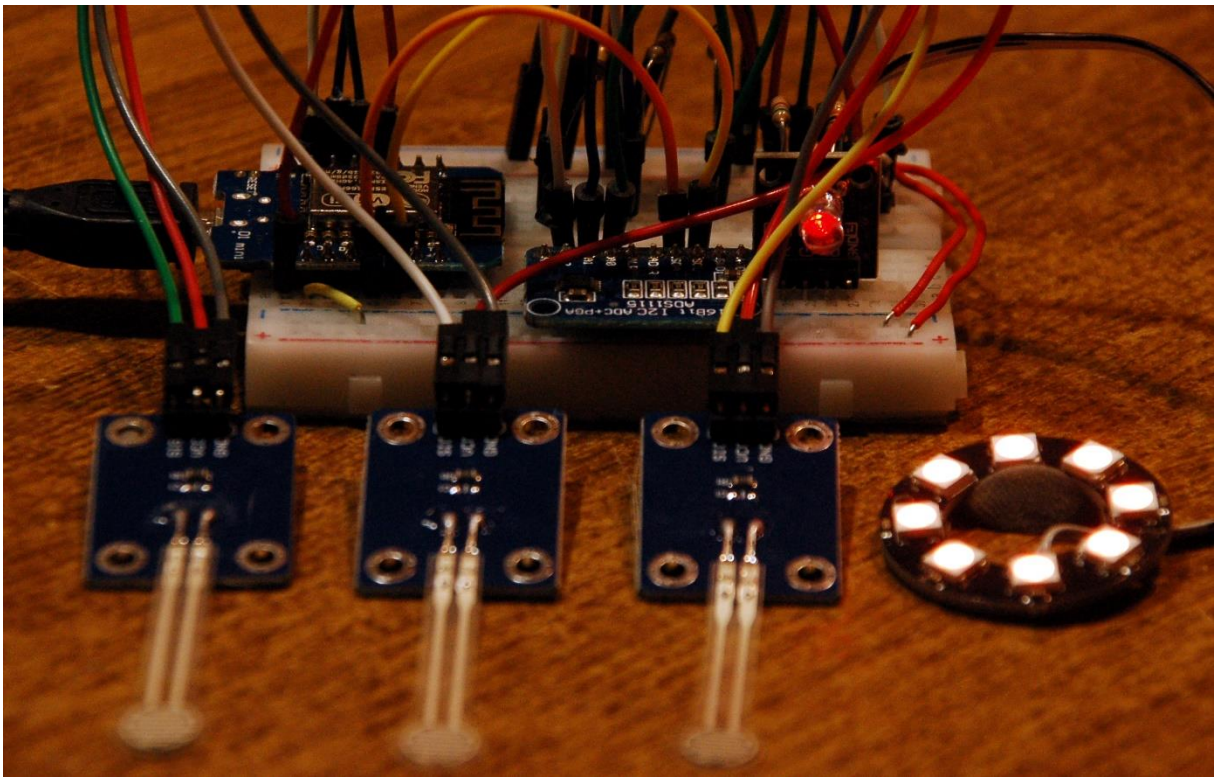

Aufbau ESP8266

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Kann die Lampe, die wir in den vorangegangenen Folgen mit einem ESP32 und Touchpads angesteuert haben, auch mit einem ESP8266 oder gar einem Raspberry Pi Pico laufen? Wenn man für den Touchservice zusätzlichen Aufwand betreibt – ja! Außerdem bedarf es jeweils einer Anpassung des Moduls **touch.py**. Das Betriebsprogramm kann im Wesentlichen vom ESP32 übernommen werden. Aber auch da müssen kleine Änderungen getätigt werden. Woran liegt das?

Der ESP8266 hat zwar einen Analog-Digital-Wandler (ADC) aber eben nur einen mit einem Kanal und wir brauchen drei Eingänge für die drei Touch-Sensoren. Denn Touch-Eingänge hat der Controller nicht. Daher müssen wir auf Drucksensoren ausweichen. Die liefern ein analoges Signal, also eine Spannung bis 3,3 Volt.

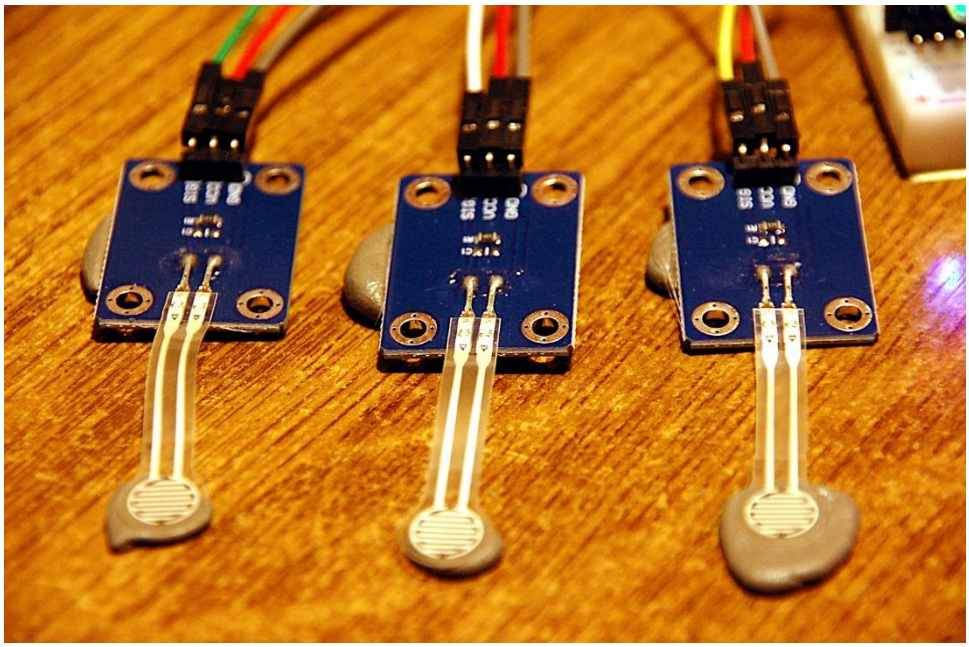

Abbildung 1: Drucksensoren als Touchpadersatz

Am ESP8266 setzen wir einen AD-Wandler vom Typ **ADS1115** ein, mit seinen vier Kanälen kann er leicht unsere Anforderungen erfüllen.

Ein weiteres Problem beim ESP8266 ist das Fehlen des Moduls **asyncio** im Kernel. Ein Parallelbetrieb von Funktionseinheiten wie beim ESP32 ist daher nicht möglich. Außerdem fehlt das Modul NVS, sodass wir zum Speichern der Lieblingsfarbe auf das Dateisystem zurückgreifen müssen.

Beim Raspberry Pi Pico können wir uns den ADS1115-Wandler sparen, denn er besitzt selber drei ADC-Eingänge. Und der MicroPython-Kern des Raspberry Pi Pico spricht auch **asyncio**. Aber auch hier läuft die Speicherung über das Dateisystem.

Nach diesen Vorinformationen geht es jetzt ans Eingemachte. Willkommen zu einer neuen Folge aus der Reihe

MicroPython auf dem (ESP32) ESP8266 und Raspberry Pi Pico

heute

Sensorlampe 3.0

Die Hardware für den ESP8266 und den Raspberry Pi Pico unterscheidet sich, außer im verwendeten Controller, lediglich darin, dass der ESP8266-Aufbau einen externen ADC benötigt. Eine RGB-LED, ein Neopixel-Ring und die drei Drucksensoren sind zusammen mit sechs Widerständen für beide Lösungen nötig.

Die Hardware

ESP8266

1	NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI Wifi Development Board mit CP2102 oder D1 Mini NodeMcu mit ESP8266-12F WLAN Modul kompatibel mit Arduino oder D1 Mini V3 NodeMCU mit ESP8266-12F
1	ADS1115 ADC Modul 16bit 4 Kanäle für Raspberry Pi
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
3	Flexibler Dünnfilm Drucksensor 3,3 / 5V Analog
1	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
1	Li-Po-Akkumulator vom Typ 18650
1	Widerstand 560 Ω
2	Widerstand 10kΩ
3	Widerstand 100kΩ
optional	Logic Analyzer

Für die angegebenen Controller-Boards genügt ein kleines Breadboard. Die Stiftreihen der Platinen liegen mit neun Rastereinheiten eng genug, dass an jeder Seite eine Kontaktreihe des Boards verfügbar bleibt.

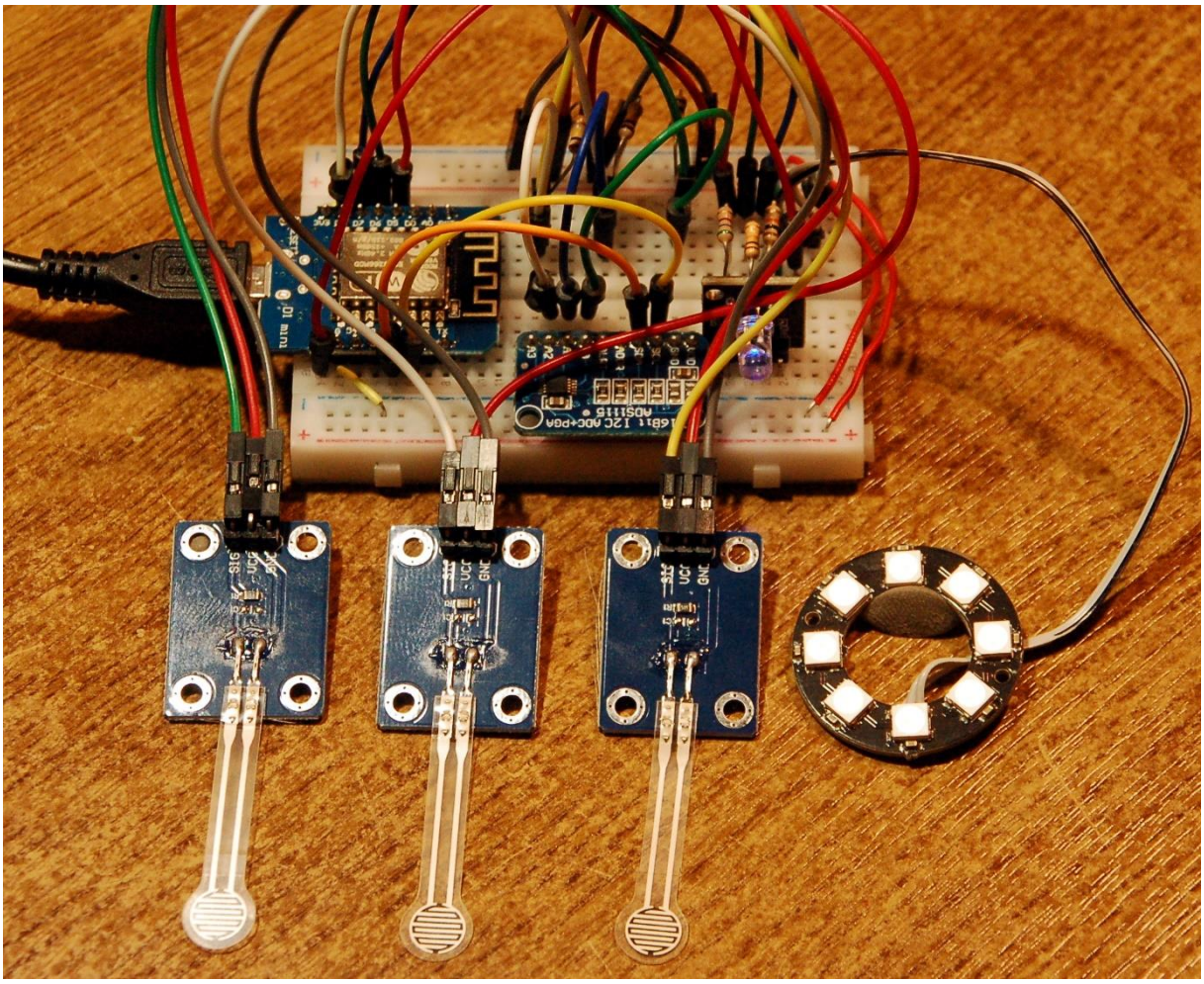

Abbildung 2: Aufbau mit ESP8266

Die RGB-LED verwende ich wieder 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 sechs Zustände geben: rot, grün, blau, hell-dunkel, ein-aus und automatischer Wechsel.

Die Vorwiderstände sind so dimensioniert, dass an der Signal-LED 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 3.

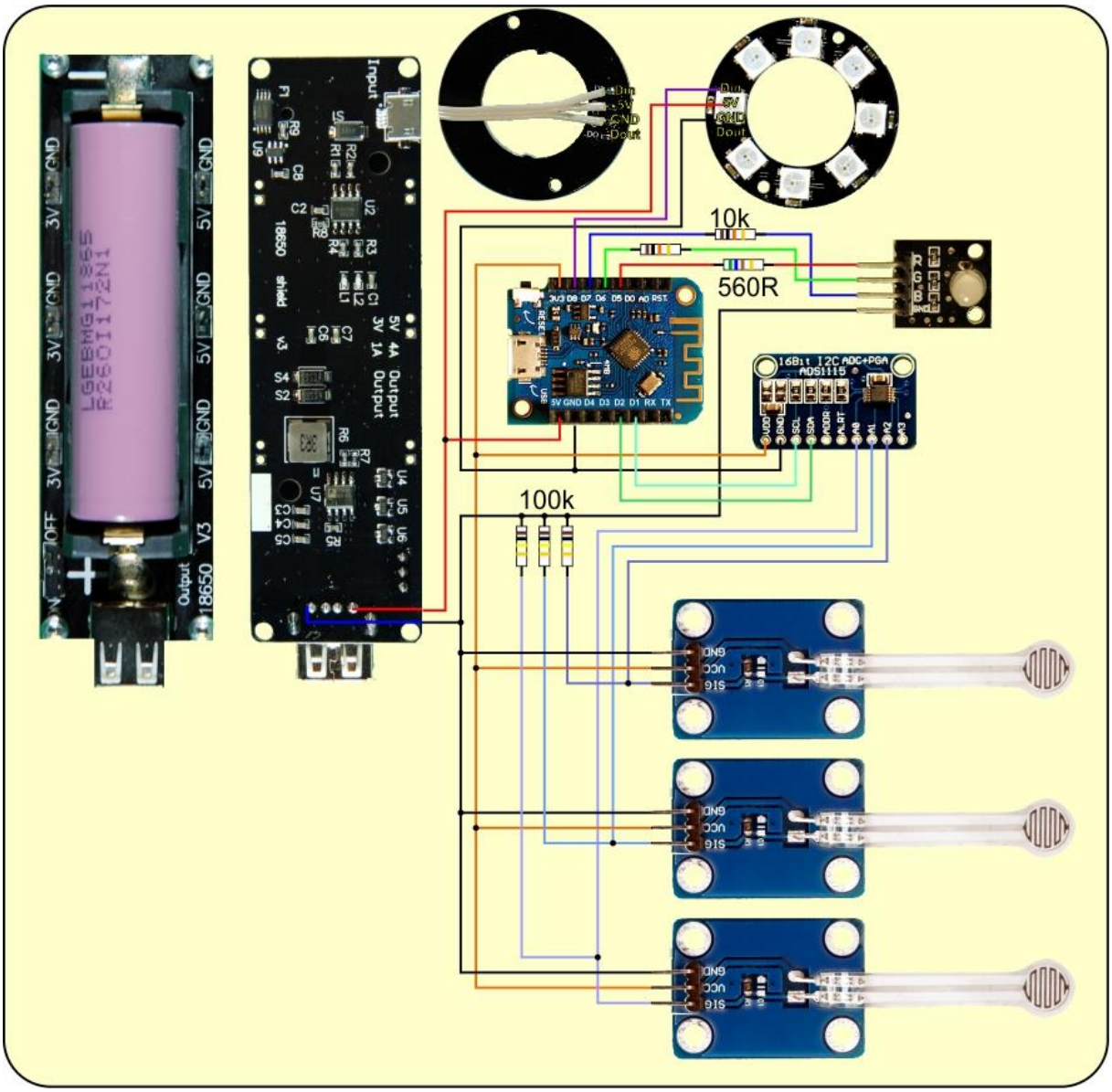

Abbildung 3: Schaltung - ESP8266 mit ADS1115 und Drucksensoren

Raspberry Pi Pico

1	Raspberry Pi Pico RP2040 Mikrocontroller-Board
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	Flexibler Dünnfilm Drucksensor 3,3 / 5V Analog
1	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
1	Widerstand 560 Ω
2	Widerstand 10kΩ
3	Widerstand 100kΩ
optional	Logic Analyzer

Auch beim Raspberry Pi Pico setzen wir auf die Drucksensoren, sparen uns aber den AD-Wandler ADS1115. Die Schaltung passt ebenfalls auf ein kleines Breadboard.

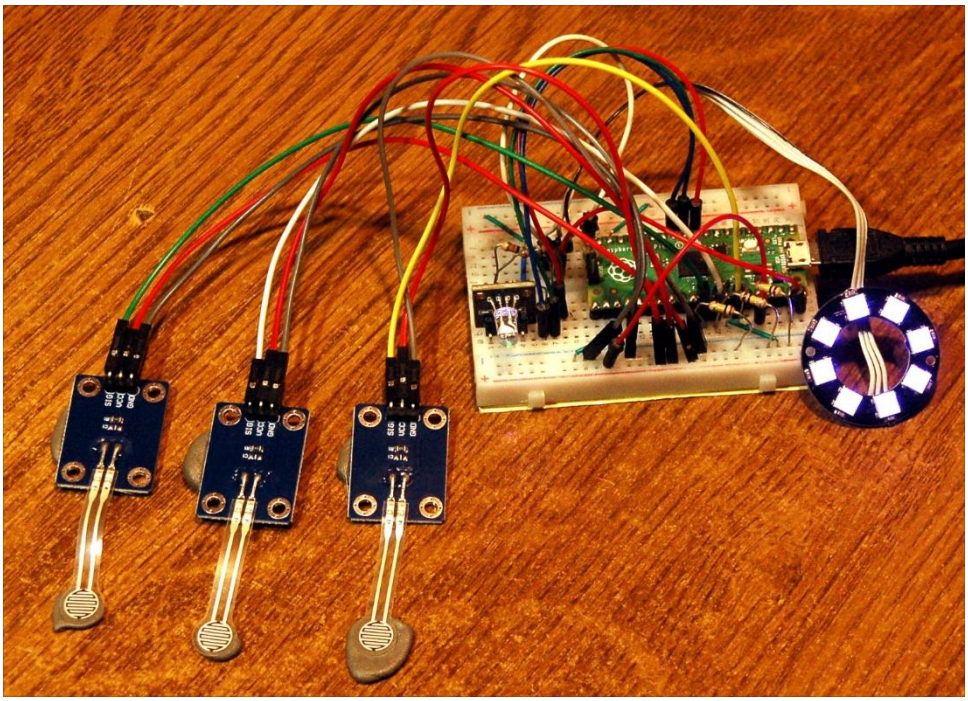

Abbildung 4: Aufbau mit Raspberry Pi Pico

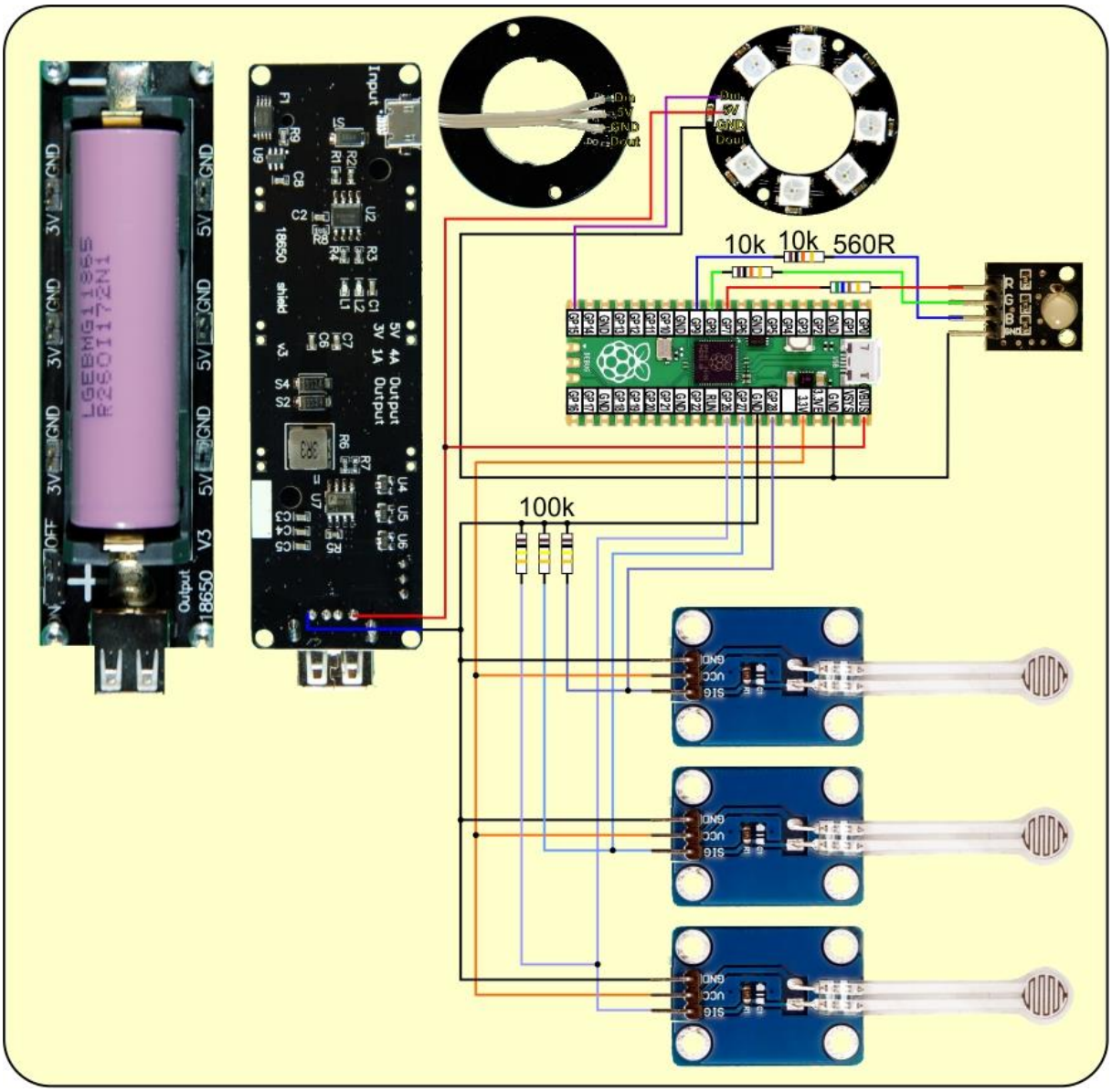

Abbildung 5: Schaltung - Raspberry Pi Pico

Die Platinen der Drucksensoren enthalten neben dem Sensor noch einen Widerstand. Der Sensor selbst stellt auch einen Widerstand dar, dessen Wert sich bei Belastung verringert. Sensor und Festwiderstand bilden eine Serienschaltung, in der der Sensor gegen Vcc geschaltet ist. Die Schaltung bildet also einen Spannungsteiler. Die Spannung am Ausgang steigt im Wert, wenn Druck auf den Sensor ausgeübt wird.

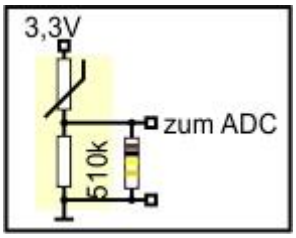

Abbildung 6: Schaltung der Drucksensoren

Bereits bei kleinen Belastungen der Sensorfläche fällt der Widerstand auf Werte im zweistelligen Kiloohm-Bereich. Alleine mit dem 510kΩ-Widerstand nimmt die Ausgangsspannung sprunghaft zu. Will man ein weicheres Verhalten erreichen, dann kann man, wie in den Abbildungen 5 und 6, einen externen Widerstand von 100kΩ parallel zum eingebauten legen. Der Gesamtwiderstand dieser Parallelschaltung beträgt 83kΩ. Damit lassen sich verschieden starke Belastungen besser detektieren.

Die Versuchsschaltung wird über das USB-Kabel mit Energie versorgt. Als Energieversorgung für das Produktionssystem 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 5.

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

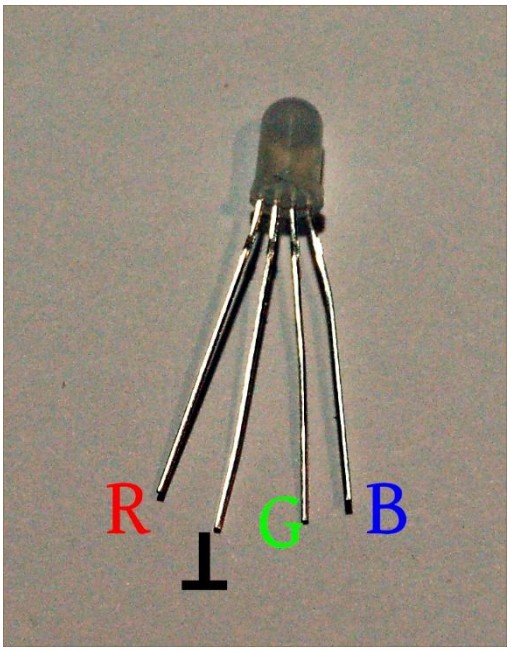

Abbildung 7: RGB-LED mit gemeinsamer Kathode

Die Lampe

Wie die Lampe aufgebaut ist habe ich bereits im ersten Teil ausgeführt. In dieser Folge liegen die Schwerpunkte auf der Anpassung des Programms an die neuen Controller. Die Konstruktion der Lampe an sich ist in der ersten Episode beschrieben. Dort finden Sie auch einen bemaßten Schnittmusterplan.

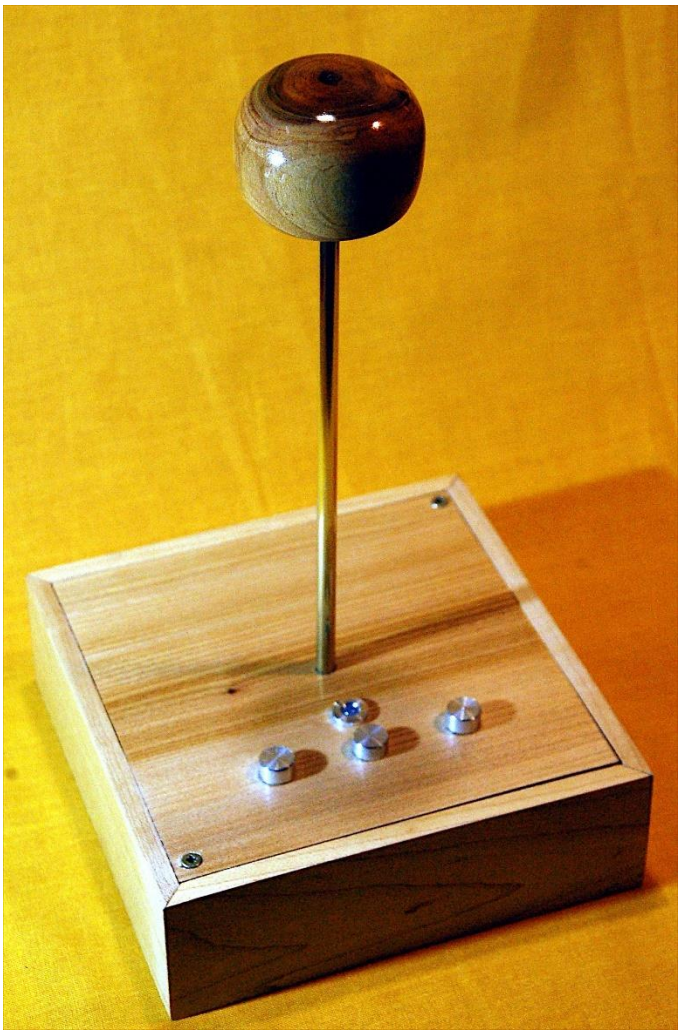

Abbildung 8: Sensor-Lampy 2.0

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

touch8266.py Touchpad-Modul
touchrpp.py Touchpad-Modul
touchlampy8266.py Betriebsprogramm
touchlampyrpp.py Betriebsprogramm Ausbaustufe 1

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. Wie Sie den **Raspberry Pi Pico** einsatzbereit kriegen, finden Sie hier.

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

Das Programm und seine Änderungen

Als Vorlage verwende ich zwei der Vorgängerversionen vom ESP32, touchlampy_2.py und touchlampy_4.py. Natürlich muss auch das Modul touch.py an die Verwendung der Drucksensoren und damit eines AD-Wandlers angepasst werden.

ESP8266

Da der ESP8266 nicht über das Modul **asyncio** verfügt, müssen wir die Abfrage der Sensoren über eine Endlosschleife erledigen. Durch die fehlende Nebenläufigkeit geht uns auch die zufallsgesteuerte Farbsequenz aus touchlampy_4.py verloren, die wir aber durch die programmgesteuerte Version aus touchlampy_2.py ersetzen können.

```
from machine import Pin, SoftI2C
from neopixel import NeoPixel as NP
from time import sleep, sleep_ms
from ads1115 import ADS1115
from touch8266 import TP

i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
ads=ADS1115(i2c)

tp1=TP(ads, 0, 2000)
tp2=TP(ads, 1, 2000)
tp3=TP(ads, 2, 2000)
```

Der Import gibt einige Zeilen ab und bekommt zwei neue dazu. Für die Unterhaltung mit dem ADS1115 brauchen wir die Klasse **SoftI2C**. Eine Instanz davon reichen wir an den Konstruktor der Klasse **ADS1115** weiter. Das Objekt **ads** benötigen die **TP**-Objekte um die Werte von den Drucksensoren zu holen. Von der Klasse **TP** brauchen wir die beiden Methoden **getDuration**() und **touched**(). Nur die letztere muss an die neue Situation angepasst werden. Nach dem Einstellen des Kanals holen wir das Wandlerergebnis ab und vergleichen es mit dem Grenzwert. Eine 1 wird zurückgegeben, wenn **val True** ist, andernfalls wird 0 zurückgegeben.

```
    def touched(self):
        self.adc.setChannel(self.kanal)
        tpin=self.adc.getConvResult()
        val=(tpin >= self.threshold)
        return 1 if val else 0
```

Wir erzeugen das NeoPixel-Objekt, legen den Kanal 0 vor und definieren die Startfarbe. Die Liste **col** brauchen wir bei der Umwandlung der Farbwerte in **color** (0…100) in die Werte, die an den Ring geschickt werden (0…255).

In der Datei **color.txt** werden die Farbwerte und der Helligkeitswert abgelegt. Dafür müssen die Ganzzahlen in Strings umgewandelt werden. Ich verwende gerne die **with**-Anweisung zum Öffnen von dateien, weil ich mir dadurch das Schließen der Datei sparen kann. Das geschieht automatisch beim Verlassen des with-Blocks. Damit jeder Wert in einer eigenen Zeile steht, hänge ich ein New Line = \n =0x0A an.

```
neoPin=Pin(15,Pin.OUT) # D8
neo=NP(neoPin,8)
kanal=0 # sw=0, rt=1, gn=2, bl=3, ws=4, ge=5
color=[16,32,8]
col=[0,0,0]
lumi=100

def storeColor():
    with open("color.txt","w") as f:
        for i in range(3):
            f.write(str(color[i])+"\n")
        f.write(str(lumi)+"\n")
    print("Fardcode gesichert",color)


def loadColor():
    global color, lumi
    with open("color.txt","r") as f:
        for i in range(3):
            color[i]=int(f.readline())
        lumi=int(f.readline())
    rt,gn,bl = color
    print("Fardcode geladen:",[rt,gn,bl])

try:
    loadColor()
except:
    storeColor()
```

Beim zeilenweisen Einlesen der Strings mache ich wieder eine Ganzzahl daraus und lege die Werte in den globalen Variablen **color** und **lumi** ab. Beim Programmstart wird die gespeicherte Farbinfo geladen. Eine Exception wird geworfen, wenn beim allerersten Start noch keine Datei existiert. Dann wird die Datei angelegt und der vorgelegte Farbwert gespeichert.

Die GPIO-Ausgänge für die Signal-LED müssen noch definiert werden, ebenso einige globale Variablen. In **shapes** stehen die Tupel zum Ansteuern der Signal-LED.

```
ledR=Pin(14,Pin.OUT,value=0)
ledG=Pin(12,Pin.OUT,value=0)
ledB=Pin(13,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),)
mode=len(shapes)
channels=("red","green","blue","lumi","auto")
step=1
m=0
program=0
merken=color
merkenL=lumi
gemerkt=False
```

Die Funktion **lum**() berechnet aus dem Farbcode und der Leuchtstärke, die in **lumi** übergeben wird, die Zahlenwerte, die an den Ring gesendet werden sollen. Farbkomponente/100 mal Helligkeit /100 mal 255.

```
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()

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

Im Automatikbetrieb werden a priori Werte im Bereich von 0 bis 255 verwendet. **lumAuto**() sendet die Werte daher direkt.

switchChannel() und **showChannel**() werden aus der Programmversion touchlampy_4.py übernommen, die in Episode 2 beschrieben ist. Allerdings wird daraus eine normale Funktion, weil der ESP8266 ja kein **asyncio** kennt.

```
def switchChannel():
    global kanal, color
    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])
```

Das gleiche Schicksal ereilt die Funktionen **increase**() und **decrease**(). Sie werden in der Hauptschleife zyklisch aufgerufen, wenn die entsprechenden Sensoren angetippt werden. Mit **increase**() werden in Kanal 1 bis 3 die Farbkomponenten im Wert von 0 bis 100 erhöht. Damit stelle ich eine Mischfarbe in der gewünschten Helligkeit ein. Kanal 4 bedient die Gesamthelligkeit. Damit kann man die zuvor eingestellte Farbe in 100 Schritten absenken (**decrease**()) und wieder anheben (**increase**()), ohne den Farbton zu ändern. Abweichungen kann es jedoch bei sehr kleinen Werten geben, weil letztlich ja auf Ganzzahlen gerundet werden muss.

```
def increase():
    global color,merken,merkenL,gemerkt,program,m,lumi
    if tp2.touched():
        n=0
        print("increase",kanal)
        while tp2.touched():
            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)
                print("auto gestartet",m,merken)
                program=1
                auto()
        n+=1
```

Über Kanal 0 (Signal-LED aus) lässt sich der Ring einschalten, nachdem er über Kanal 0 und **decrease**() ausgeschaltet wurde. Beim Ausschalten merken wir uns vorübergehend die zuletzt eingestellten Werte in **merken** und **merkenL** und setzen **gemerkt** auf True. Beim Einschalten mit **increase**() werden die Werte wieder restauriert. Ähnlich verhält sich der Kanal 5, über den die in **auto**() festgelegte Farbsequenz abgespielt und gestoppt wird.

```
def decrease():
    global color,merken,merkenL,m,gemerkt,program,lumi
    print("dec entered")
    if tp3.touched():
        n=0
        print("decrease",kanal)
        while tp3.touched():
            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
                color=merken
                lumi=merkenL
                print("auto gestoppt",merken)
                lum(lumi)
        n+=1
```

Auch die Funktion **auto**() verliert den Status einer Coroutine. In touchlampy_4.py mussten wir uns wegen der Nebenläufigkeit der Tasks nicht extra um die Abfrage der Touchpads kümmern. Hier müssen wir die Abfrage von **tp3** in die Funktion einbauen, weil sonst die automatisch ablaufende Farbsequenz nicht verlassen werden kann.

```
def auto():
    global program, m, color
    print("Automatik gestartet")
    while 1:
        if tp3.touched()==1:
            print("stopped in 1")
            break
        color=[0,0,255]
        m=0
        while program == 1:
            if tp3.touched():
                break
            if 0 < m < 256//step:
                color[0]=(color[0]+step)
                color[2]=(color[2]-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)
            lumAuto()
            m=(m+1) % (1792//step)
```

Im Übrigen muss darauf geachtet werden, dass die Komponentenwerte nicht negativ werden, weil sonst seltsame Effekte auftreten. [-1,0,0] wird dann nämlich zum Beispiel zu hellstem Rot.

>>> -1 & 0xff
255

Die Hauptschleife übernimmt die Sensorabfrage und steuert so den Programmablauf.

```
lum(100)

while 1:
    if tp1.touched():
        switchChannel()
    elif tp2.touched():
        print("inc")
        increase()
    elif tp3.touched():
        print("dec")
        decrease()
```

Raspberry Pi Pico

Weil der Raspberry Pi Pico **asyncio** versteht, können wir das Programm touchlampy_4.py vom ESP32 fast 1:1 übernehmen. Aber nur fast, denn NVS kann der Raspberry Pi Pico nicht. Dafür hat er aber eigene ADC-Eingänge und zwar drei Stück – passt! Die Speicherung und das Laden der Farbkomponenten übernehmen wir vom ESP8266.

Beginnen wir mit der Besprechung der relevanten Objekte des Moduls **touchrpp.py**. Wir holen uns die Klasse **ADC** in Boot.

```
from machine import Pin, ADC
import timeout
```

Dem Konstruktor übergeben wir die Kanalnummer und den Grenzwert, der beim Raspberry Pi Pico zwischen 0 und 65535 liegt. Als Defaultwert habe ich durch Versuchsmessungen 10000 ermittelt. Die Kanalnummer wird in das Instanz-Attribut **kanal** übertragen und bei der Instanziierung des ADC-Objekts als Index in die Liste **ADCs** der ADC-Anschlussnummern verwendet.

```
    def __init__(self, channelNbr,
                 grenze=Grenze):
        self.threshold=0 # Grenzwert
        self.duration=None # Touch-Dauer
        self.grenzwert(grenze)
        if not channelNbr in range(3):
            raise ValueError
            sys.exit()
        self.kanal=channelNbr # Eingang festlegen
        self.adc=ADC(TP.ADCs[channelNbr])
        print("Konstruktor TP",channelNbr)
```

Die Methode **touched**() sieht jetzt so aus.

```
    def touched(self):
        tpin=self.adc.read_u16()
        val=(tpin >= self.threshold)
        return 1 if val else 0
```

Schließlich wird in **touchrpp.py** noch die Funktion **grenzwert**() angepasst.

```
    def grenzwert(self,grenzwert=None):
        if grenzwert is None:
            return self.threshold
        else:
            gw = int(grenzwert)
            gw = (gw if gw > 0 and gw < 65536 else Grenze)
            print("Grenzwert ist jetzt {}.".format(gw))
            self.threshold = gw
            return gw
```

Das Betriebsprogramm der Lampe entspricht mit Ausnahme weniger Zeilen dem Inhalt von touchlampy_4.py. Die Abweichungen sind im Listing fett formatiert.

```
from touchrpp import TP
from machine import Pin
from neopixel import NeoPixel as NP
from time import sleep, sleep_ms
import uasyncio as asyncio
import random
from sys import exit

tp1=TP(0, 10000)
tp2=TP(1, 10000)
tp3=TP(2, 10000)

neoPin=Pin(15,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():
    with open("color.txt","w") as f:
        for i in range(3):
            f.write(str(color[i])+"\n")
        f.write(str(lumi)+"\n")
    print("Fardcode gesichert",color)


def loadColor():
    global color, lumi
    with open("color.txt","r") as f:
        for i in range(3):
            color[i]=int(f.readline())
        lumi=int(f.readline())
    rt,gn,bl = color
    print("Fardcode geladen:",[rt,gn,bl])

try:
    loadColor()
except:
    storeColor()

ledR=Pin(7,Pin.OUT,value=0)
ledG=Pin(8,Pin.OUT,value=0)
ledB=Pin(9,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
merken=color
merkenL=lumi
gemerkt=False

step=2
m=0

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(50)
        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(70)
                if 1 <= kanal <= 3:
                    ptr=kanal-1
                    col=color[ptr]
                    if n < 10:
                        col = col + 1
                    else:
                        col = col + 3
                    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 + 3
                    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(70)
                if 1 <= kanal <= 3:
                    ptr=kanal-1
                    col=color[ptr]
                    if n < 10:
                        col = col - 1
                    else:
                        col = col - 3
                    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 - 3
                    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())
```

Damit Sind Sie in der Lage, das Lämpchen mit einem Controller Ihrer Wahl in MicroPython zu programmieren. Wir haben mit asyncio ein Modul an der Hand, mit dem wir mehrere Prozesse parallel zueinander ablaufen lassen können. Wir wissen jetzt, wie auf den Systemen Daten dauerhaft abgespeichert werden können und wie die fehlende Touchpad-Funktionalität durch Drucksensoren zu ersetzen ist. Diese Features lassen sich natürlich ohne weiteres auch auf eigene Projekte anwenden.

Viel Spaß damit!