WS2812B Nachtlicht mit Raspberry Pi Pico

Nach nunmehr 5 Jahren hat unsere geschenkte Nachtlichtschildkröte den Geist aufgegeben und meiner Kinder waren sehr traurig. Gerade zum Abend haben meine Kinder es immer genossen, wenn die Schildkröte mit einstellbarer Helligkeit das Schlafzimmer erhellt hat. Als Maker und mittlerweile geschickter Konstrukteur mit dem 3D-Drucker, wollte ich mir die Chance nicht entgehen lassen, ein eigenes Nachtlicht zu bauen. Die Idee dahinter war, dass das Nachtlicht mittels transparenten Filaments die Farbe ins Schlafzimmer projizieren kann und ggf. sogar einfach erweitert werden kann. Damit MicroPython oder Pyhton allgemein mal wieder ein bisschen aufgefrischt wird, soll alles in der vorgesehen Programmiersprache erstellt werden. Gleichzeitig möchte ich Ihnen zeigen, wie Sie einfach ein Entprellen von Tastern selbst programmieren können, ohne auf Funktionen von MicroPython zurückgreifen zu müssen.

Benötigte Hard- und Software

Die Hardware für diesen Versuchsaufbau ist relativ überschaubar, siehe Tabelle 1.

Pos	Anzahl	Bauteil	Link
1	1	Raspberry Pi Pico	https://www.az-delivery.de
2	1	Streifenrasterplatine	amazon.de
3	1	NeoPixel Ring mit 32 WS2812	amazon.de
4	1	Hex Abstandshaltern	amazon.de
5	1	560 Stück Breadboard Jumper Wires	amazon.de
6	1	Mehrere M2-Sechskantschrauben	amazon.de
7	1	Drucktaster	amazon.de

Tabelle 1: Hardwareteile für das WS2812 Nachtlicht

Gerade bei den Amazon-Links handelt es sich um Beispiele, jedoch sollten Sie bei den Neopixel Ring darauf achten, dass dieser einen maximalen Außendurchmesser von 104mm aufweist! Andernfalls wird mein zur Verfügung gestelltes Gehäuse nicht passen. Gleichzeitig empfehle ich bei den Hex Abstandshaltern nicht die Version aus Nylon, sondern aus Messing zu nehmen, dadurch ersparen Sie sich Frust beim Reindrehen in den Gehäuseboden.

Zusätzlich brauchen Sie alles zu löten (Lötstation, Lötzinn, Lupe, etc.) und Werkzeug, um die Streifenrasterplatine zu bearbeiten (aussägen und Kupferkontaktfläche trennen).

Möchten Sie mein Modell für das Nachtlicht verwenden, so brauchen Sie einen 3D-Drucker und Filament. Am besten hat sich bei mir schwarzes Filament für den Boden und transparentes für alle anderen Teile erwiesen.

Damit Sie später das Programm von mir verwenden und modifizieren können, nutzen Sie bitte das Programm Thonny, welches es für alle gängigen Betriebssysteme (und den Raspberry Pi) gibt. Unser Autor Andreas Wolter hat dazu einen guten Einstiegsblog geschrieben, der die Grundeinrichtung erklärt.

Die Grundidee für das Nachtlicht und die Komponenten

Wie am Anfang von meinem Blogbeitrag erwähnt, brauchte ich, für meine Kinder, eine schnelle Lösung für ein neues Nachtlicht. Das alte hat mit einem Motor und LEDs Meereswellen an die Decke projiziert. Ich wollte aber eher mehr Farbe und das Zimmer in weichen sanften Tönen mit nicht zu hektischen Farbänderungen eintauchen. Damit das

klappt, brauchte ich einen festen Untergrund, die Streifenrasterplatine für den Pico, auf dem auch gleichzeitig die Stecker für weitere Hardware gelötet werden soll. Die Außenwände mussten transparent sein und Platz für den WS2812 Neopixel-Ring bieten. Dieser wird an der Oberseite vom Gehäuse eingesetzt. Der Clou an der Sache ist, dass der Deckel austauschbar sein soll. Denn ich werde die Platine schon so vorbereiten, dass weitere Sensoren verwendet werden könne. Dies kann zum Beispiel ein PIR-Bewegungsmelder sein.
Mit diesen Ideen im Kopf und etwas Fusion 360, habe ich mein Gehäuse erstellt, siehe Abbildung 1.

Abbildung 1: Das fertig zusammengebaute Gehäuse in Fusion 360

Damit das so funktioniert, musste der Boden schon so erstellt sein, dass der Pico samt Streifenrasterplatine montiert werden kann. In meinem Fall muss die Platine rund sein, da der Boden genau diese Form aufweist, siehe Abbildung 2.

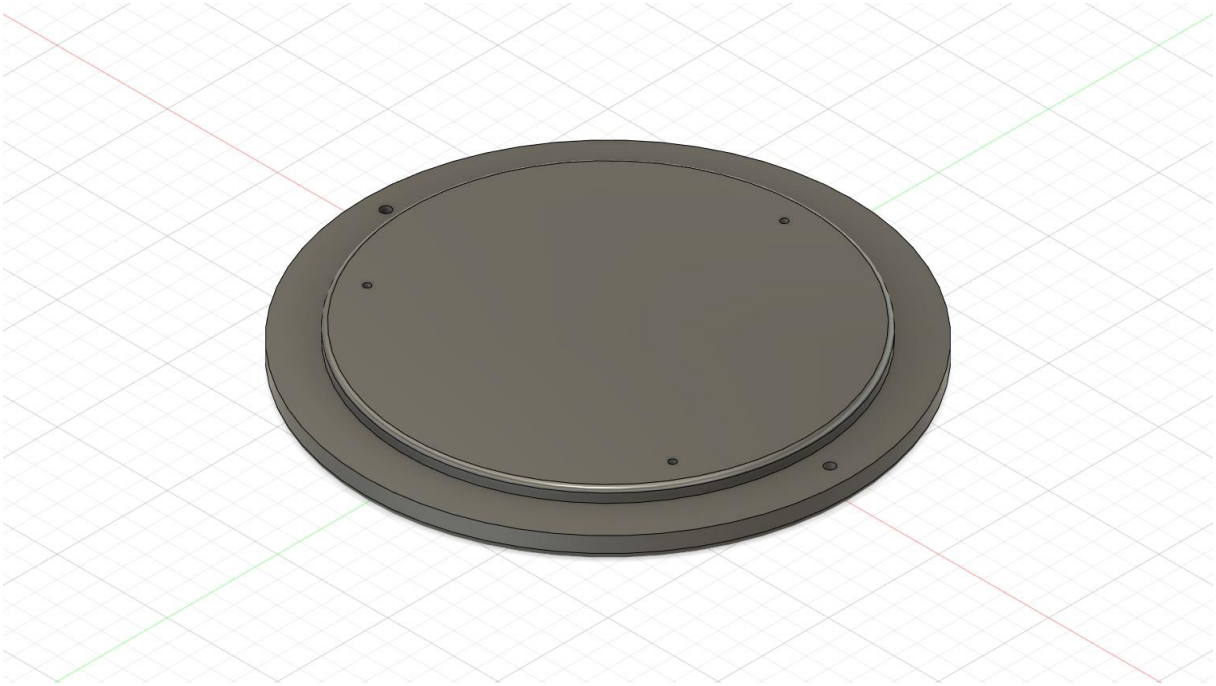

Abbildung 2: Der Boden der Nachtleuchte

Damit der Pico auch später seiner Aufgabe gerecht wird, muss zunächst die Verkabelung angefertigt werden, siehe Abbildung 3. An der Stelle habe ich einen weiteren Pin für den Näherungssensor noch nicht eingeplant, während meiner Lötarbeiten ist dieser aber noch hinzugekommen. Dieser ist daher in der Grundverdrahtung nicht zu sehen.

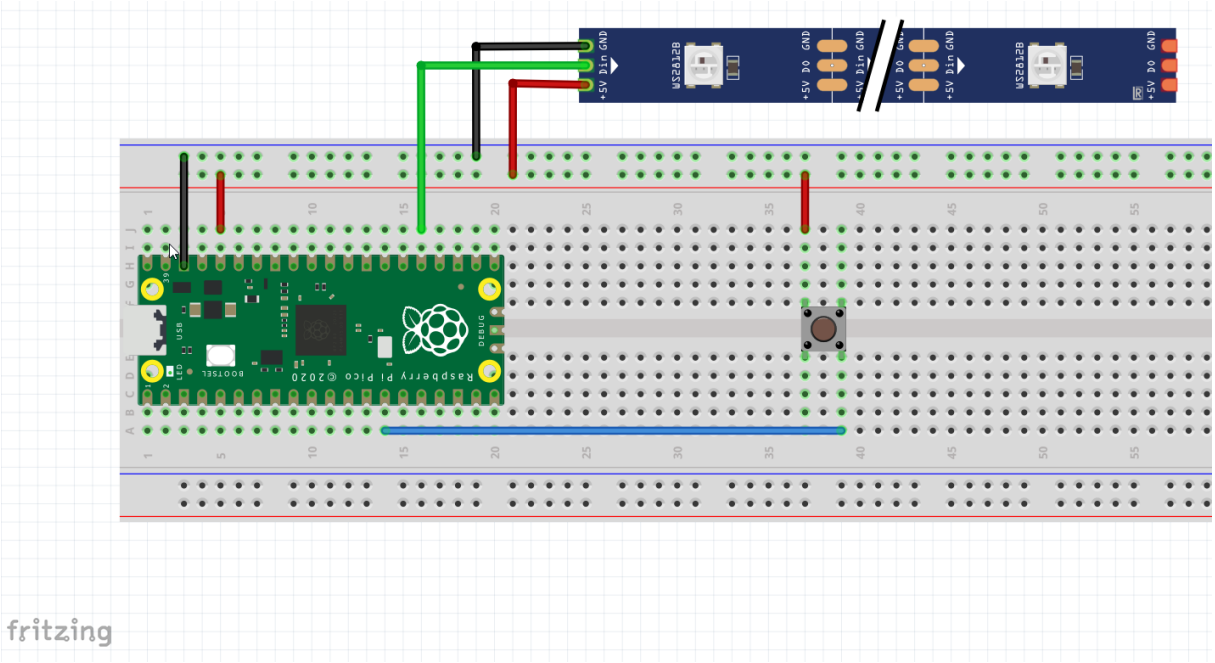

Abbildung 3: Das Grundschema der Verdrahtung

Trotz der überschaubaren Verkabelung ist die Platine am Ende doch recht umfangreicher gelötet worden als gedacht, siehe Abbildung 4. Nicht zuletzt musste ich darauf achten, dass die Bohrlöcher an der richtigen Stelle sitzen. Eine entsprechende Zeichnung, um die Bohrungen korrekt zu setzen, findet sich in meinem Repository.

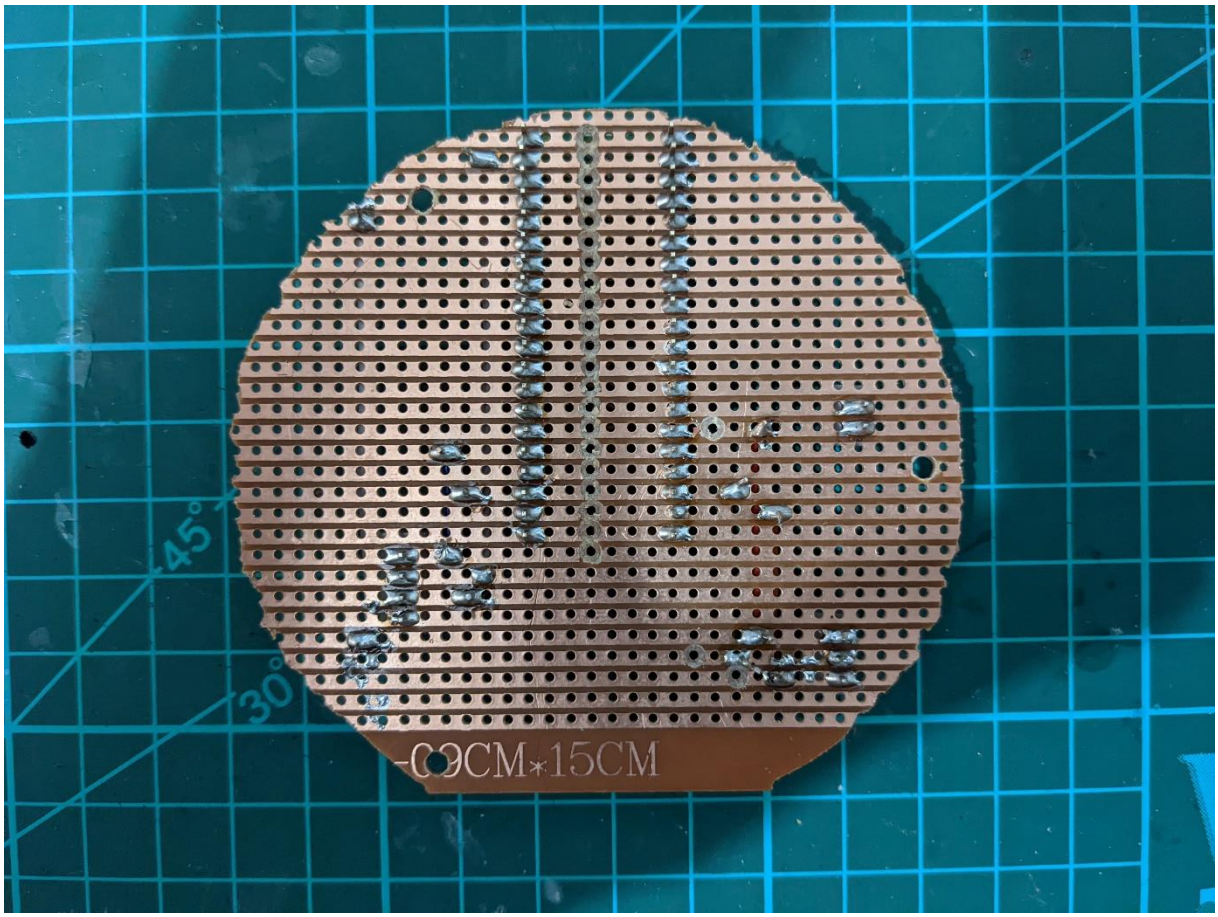

Abbildung 4: Raspberry Pi Pico mit allen Adaptern verlötet

Wie in Abbildung 5 zu sehen, habe ich mich dagegen entschieden, die Verbindungen zu Taster und später vllt. sogar einem Bewegungsmelder über direkt angelötete Punkte vorzunehmen.

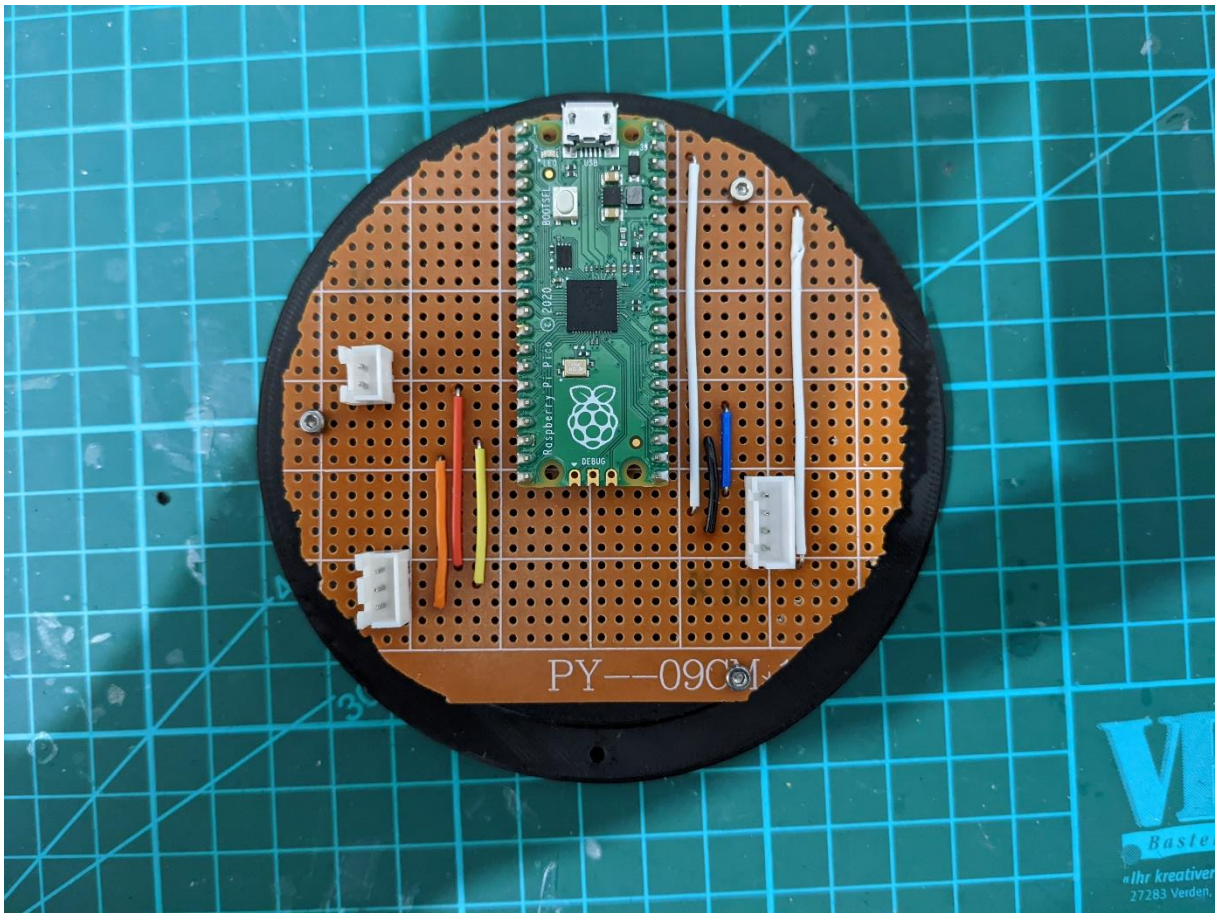

Abbildung 5: Platine mit allen Steckern auf Grundplatte montiert

Stattdessen habe ich JST-XH-Verbinder genutzt, womit ich schnell und einfach die Anschlüsse tauschen kann. Ich muss gleich klarstellen, dass hier auch die Stecker dann selbst gecrimpt werden müssen, aber mit ein bisschen Übung gelingt auch das.

Mit dem Lampenschirm und dem Deckel musst ich mir noch etwas einfallen lassen. Zum einen wollte ich den WS2812B-Ring nicht verschrauben, dieser sollte aber so im Lampenschirm fixiert sein, dass er eben nicht frei im Inneren herumfallen kann. Letztlich habe ich eine einfache Nut in den Lampenschirm konstruiert, an welcher der Ring mit etwas Druck fixiert wird. Der Druck selbst ist nicht kompliziert, siehe Abbildung 6, jedoch durch die Höhe des Lampenschirms hat der Druck aller Teile knapp 20 Stunden gedauert.

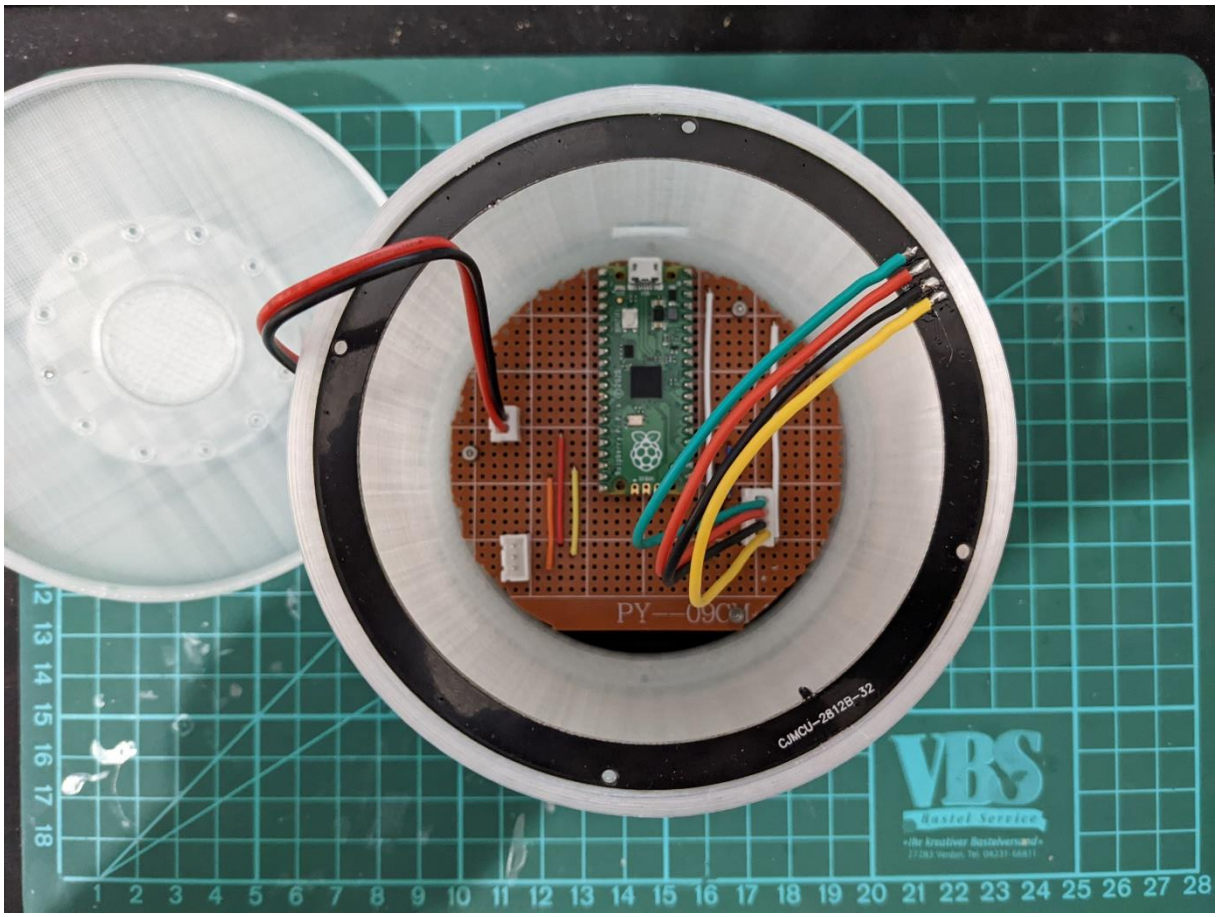

Abbildung 6: Das Innenleben und alle Teile vom Nachtlicht

An der Seite vom Deckel und dem Lampenschirm sind Löcher, um mit kleinen M2-Schrauben alles zu fixieren. Damit ist im Grunde die Hardware für das Nachtlicht fertig und es steht nur noch die Programmierung aus.

Die Software

Ursprünglich war geplant gewesen, mit der Arduino IDE dem Raspberry Pi Pico zu programmieren. Jedoch, gerade weil die Verwendung von einem Arduino IDE zusammen mit einem Raspberry Pi Pico bei mir in der Vergangenheit immer wieder zu Problemen geführt hat, habe ich mich für MicroPython entschieden. Ein Vorteil an der Stelle ist, dass die Programme nicht immer wieder kompiliert werden müssen und ich schneller ein Ergebnis sehe.
Damit der Raspberry Pi Pico auch den WS2812B-Ring ansteuern kann, habe ich lange nach einer passenden und funktionieren Lib gesucht. Leider habe ich mir den Entwickler nicht gemerkt, jedoch konnte ich nach etwas suchen eine relativ schlanke Lib mit dem Namen neopixel.py bei github finden, siehe Code 1.

```
import array, time
from machine import Pin
import rp2

# PIO state machine for RGB. Pulls 24 bits (rgb -> 3 * 8bit) automatically
@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT,
autopull=True, pull_thresh=24)
def ws2812():
    T1 = 2
```

```
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop().side(0)                       [T2 - 1]
    wrap()

# PIO state machine for RGBW. Pulls 32 bits (rgbw -> 4 * 8bit) automatically
@rp2.asm_pio(sideset_init=rp2.PIO.OUT_LOW, out_shiftdir=rp2.PIO.SHIFT_LEFT,
autopull=True, pull_thresh=32)
def sk6812():
    T1 = 2
    T2 = 5
    T3 = 3
    wrap_target()
    label("bitloop")
    out(x, 1)               .side(0)    [T3 - 1]
    jmp(not_x, "do_zero")   .side(1)    [T1 - 1]
    jmp("bitloop")          .side(1)    [T2 - 1]
    label("do_zero")
    nop()                   .side(0)    [T2 - 1]
    wrap()


# Delay here is the reset time. You need a pause to reset the LED strip back to the initial
LED
# however, if you have quite a bit of processing to do before the next time you update the
strip
# you could put in delay=0 (or a lower delay)
#
# Class supports different order of individual colors (GRB, RGB, WRGB, GWRB ...). In order
to achieve
# this, we need to flip the indexes: in 'RGBW', 'R' is on index 0, but we need to shift it left by 3
* 8bits,
# so in it's inverse, 'WBGR', it has exactly right index. Since micropython doesn't have [::-1]
and recursive rev()
# isn't too efficient we simply do that by XORing (operator ^) each index with 3 (0b11) to
make this flip.
# When dealing with just 'RGB' (3 letter string), this means same but reduced by 1 after
XOR!.
# Example: in 'GRBW' we want final form of 0bGGRRBBWW, meaning G with index 0 needs
to be shifted 3 * 8bit ->
# 'G' on index 0: 0b00 ^ 0b11 -> 0b11 (3), just as we wanted.
# Same hold for every other index (and - 1 at the end for 3 letter strings).

class Neopixel:
    def __init__(self, num_leds, state_machine, pin, mode="RGB", delay=0.0001):
        self.pixels = array.array("I", [0 for _ in range(num_leds)])
        self.mode = set(mode)   # set for better performance
        if 'W' in self.mode:
            # RGBW uses different PIO state machine configuration
```

```
            self.sm = rp2.StateMachine(state_machine, sk6812, freq=8000000,
sideset_base=Pin(pin))
            # dictionary of values required to shift bit into position (check class desc.)
            self.shift = {'R': (mode.index('R') ^ 3) * 8, 'G': (mode.index('G') ^ 3) * 8,
                    'B': (mode.index('B') ^ 3) * 8, 'W': (mode.index('W') ^ 3) * 8}
        else:
            self.sm = rp2.StateMachine(state_machine, ws2812, freq=8000000,
sideset_base=Pin(pin))
            self.shift = {'R': ((mode.index('R') ^ 3) - 1) * 8, 'G': ((mode.index('G') ^ 3) - 1) * 8,
                    'B': ((mode.index('B') ^ 3) - 1) * 8, 'W': 0}
        self.sm.active(1)
        self.num_leds = num_leds
        self.delay = delay
        self.brightnessvalue = 255

    # Set the overal value to adjust brightness when updating leds
    def brightness(self, brightness=None):
        if brightness == None:
            return self.brightnessvalue
        else:
            if brightness < 1:
                brightness = 1
        if brightness > 255:
            brightness = 255
        self.brightnessvalue = brightness

    # Create a gradient with two RGB colors between "pixel1" and "pixel2" (inclusive)
    # Function accepts two (r, g, b) / (r, g, b, w) tuples
    def set_pixel_line_gradient(self, pixel1, pixel2, left_rgb_w, right_rgb_w, how_bright =
None):
        if pixel2 - pixel1 == 0:
            return
        right_pixel = max(pixel1, pixel2)
        left_pixel = min(pixel1, pixel2)

        for i in range(right_pixel - left_pixel + 1):
            fraction = i / (right_pixel - left_pixel)
            red = round((right_rgb_w[0] - left_rgb_w[0]) * fraction + left_rgb_w[0])
            green = round((right_rgb_w[1] - left_rgb_w[1]) * fraction + left_rgb_w[1])
            blue = round((right_rgb_w[2] - left_rgb_w[2]) * fraction + left_rgb_w[2])
            # if it's (r, g, b, w)
            if len(left_rgb_w) == 4 and 'W' in self.mode:
                white = round((right_rgb_w[3] - left_rgb_w[3]) * fraction + left_rgb_w[3])
                self.set_pixel(left_pixel + i, (red, green, blue, white), how_bright)
            else:
                self.set_pixel(left_pixel + i, (red, green, blue), how_bright)

    # Set an array of pixels starting from "pixel1" to "pixel2" (inclusive) to the desired color.
    # Function accepts (r, g, b) / (r, g, b, w) tuple
    def set_pixel_line(self, pixel1, pixel2, rgb_w, how_bright = None):
        for i in range(pixel1, pixel2 + 1):
            self.set_pixel(i, rgb_w, how_bright)

    # Set red, green and blue value of pixel on position <pixel_num>
    # Function accepts (r, g, b) / (r, g, b, w) tuple
    def set_pixel(self, pixel_num, rgb_w, how_bright = None):
```

```
        if how_bright == None:
            how_bright = self.brightness()
        pos = self.shift

        red = round(rgb_w[0] * (how_bright / 255))
        green = round(rgb_w[1] * (how_bright / 255))
        blue = round(rgb_w[2] * (how_bright / 255))
        white = 0
        # if it's (r, g, b, w)
        if len(rgb_w) == 4 and 'W' in self.mode:
            white = round(rgb_w[3] * (how_bright / 255))

        self.pixels[pixel_num] = white << pos['W'] | blue << pos['B'] | red << pos['R'] | green <<
pos['G']

    # Converts HSV color to rgb tuple and returns it
    # Function accepts integer values for <hue>, <saturation> and <value>
    # The logic is almost the same as in Adafruit NeoPixel library:
    # https://github.com/adafruit/Adafruit_NeoPixel so all the credits for that
    # go directly to them (license:
https://github.com/adafruit/Adafruit_NeoPixel/blob/master/COPYING)
    def colorHSV(self, hue, sat, val):
        if hue >= 65536:
            hue %= 65536

        hue = (hue * 1530 + 32768) // 65536
        if hue < 510:
            b = 0
            if hue < 255:
                r = 255
                g = hue
            else:
                r = 510 - hue
                g = 255
        elif hue < 1020:
            r = 0
            if hue < 765:
                g = 255
                b = hue - 510
            else:
                g = 1020 - hue
                b = 255
        elif hue < 1530:
            g = 0
            if hue < 1275:
                r = hue - 1020
                b = 255
            else:
                r = 255
                b = 1530 - hue
        else:
            r = 255
            g = 0
            b = 0

        v1 = 1 + val
```

```
        s1 = 1 + sat
        s2 = 255 - sat

        r = ((((r * s1) >> 8) + s2) * v1) >> 8
        g = ((((g * s1) >> 8) + s2) * v1) >> 8
        b = ((((b * s1) >> 8) + s2) * v1) >> 8

        return r, g, b


    # Rotate <num_of_pixels> pixels to the left
    def rotate_left(self, num_of_pixels):
        if num_of_pixels == None:
            num_of_pixels = 1
        self.pixels = self.pixels[num_of_pixels:] + self.pixels[:num_of_pixels]

    # Rotate <num_of_pixels> pixels to the right
    def rotate_right(self, num_of_pixels):
        if num_of_pixels == None:
            num_of_pixels = 1
        num_of_pixels = -1 * num_of_pixels
        self.pixels = self.pixels[num_of_pixels:] + self.pixels[:num_of_pixels]

    # Update pixels
    def show(self):
        # If mode is RGB, we cut 8 bits of, otherwise we keep all 32
        cut = 8
        if 'W' in self.mode:
            cut = 0
        for i in range(self.num_leds):
            self.sm.put(self.pixels[i], cut)
        time.sleep(self.delay)

    # Set all pixels to given rgb values
    # Function accepts (r, g, b) / (r, g, b, w)
    def fill(self, rgb_w, how_bright = None):
        for i in range(self.num_leds):
            self.set_pixel(i, rgb_w, how_bright)
        time.sleep(self.delay)
```

Code 1: Die Bibliothek neopixel.py

Vom Prinzip steuert die Bibliothek später die einzelnen LEDs, ähnlich wie bei dem Arduino, nur eben im MicroPython. Laden sie die Datei auf den Pico. Hierzu am besten einfach eine neue Datei anlegen und den Inhalt aus Code 1 einfügen und auf dem Pico mit dem Namen neopixel.py speichern. Ist diese Datei auf den Pico geladen, kommen wir zu der main.py. Diese muss:

- Den aktuellen Taster entprellen und den Status abfragen und speichern
- Den WS2812B-Ring steuern, egal mit wie vielen Elementen
- Den Farbverlauf vorberechnen, hier mittels HSV, siehe Wikipedia
- Die Timer überwachen und entsprechende Aktionen am Ende ausführen

Die erste Frage, die Sie sich nun wahrscheinlich stellen werden, ist, warum ich eingangs erwähnt habe, dass ich eine eigene Entprell-Funktion geschrieben habe. Ja es gibt in MicroPython so etwas, jedoch hatte ich das Problem, dass diese Funktion mir nicht das Ergebnis geliefert hat, was ich aus meiner Arduino-Programmierung sonst immer genutzt

habe. Daher wollte ich, auch um wieder etwas meine Programmierkenntnisse aufzufrischen, wie Funktionen in MicroPython geschrieben werden, etwas Eigenes kreieren. Herausgekommen ist die Funktion aus Code 2

```
def btn_pressed():
    global btnLastState, tmLastUpdate
    btnState = False
    if btnLastState != btn.value():
        tmLastUpdate = ticks_ms()
    if (ticks_ms() - tmLastUpdate) > 100:
        btnState = btn.value()

    btnLastState = btn.value()
    return btnState
```

Code 2: Die Funktion zum entprellen eines Tasters

Vom Prinzip sind es neun Zeilen, die recht schnell erklärt sind. Bei jedem Durchlauf wird meine Funktion **btn_pressed()** aufgerufen und der zuletzt gespeicherte Wert überprüft. Hat sich dieser geändert, so wird ein Variable mit der aktuellen Zeit gesetzt und in den nächsten Durchläufen geprüft ob irgendwann die Grenze von 100ms erreicht ist. Gleichzeitig wird am Ende der Funktion der aktuelle Stand gespeichert, damit diese Timer-Funktion nur bei jeder Änderung durchgeführt wird. Sind 100ms abgelaufen, wird der entprellt Wert zurückgeliefert. Somit ist das Problem des Prellens recht einfach gelöst. Der komplette Code, siehe Code 3, ist nicht komplex, sondern mit gerade einmal 75 Zeilen (Leerzeichen und Kommentarzeilen mit eingerechnet) recht schlank.

```
from neopixel import Neopixel
from time import sleep, ticks_ms
from machine import Pin,Timer

## Setup
NUM_LEDS = 32
LED_PIN = 19
BTN_PIN = 10
SENSOR_PIN = 14 #For later feature
pixels = Neopixel(NUM_LEDS, 0, LED_PIN, "GRB")
pixels.brightness(30)
btn = Pin(BTN_PIN, Pin.IN, Pin.PULL_DOWN)

## Constants
DEFTIMEBTN = 15*60*1000 #Define for lights on
TIMECOLORCHANGE = 50 #Timer colorchange
TIMECOLOROFF = 0 #Timer
red = pixels.colorHSV(0, 255, 255)
green = pixels.colorHSV(21845, 255, 255)
blue = pixels.colorHSV(43691, 255, 255)

## Variables
lastValue = ticks_ms()
lastActiveSet = ticks_ms()
bActiveLight = False
bButtonPressed = False
btnLastState = False
btnControl = False
tmLastUpdate = 0
hue = 0
```

```
#Thats a simple function to debounce btn
def btn_pressed():
    global btnLastState, tmLastUpdate
    btnState = False
    if btnLastState != btn.value():
        tmLastUpdate = ticks_ms()
    if (ticks_ms() - tmLastUpdate) > 100:
        btnState = btn.value()

    btnLastState = btn.value()
    return btnState

## Loop
while True:
  bButtonPressed = btn_pressed() #Check button pressed or not
  #Function-loop if button pressed
  if bButtonPressed and not btnControl:
      if not bActiveLight:
         TIMECOLOROFF =  DEFTIMEBTN
         bActiveLight = True
         lastActiveSet = ticks_ms()
      else:
        TIMECOLOROFF =  0
        bActiveLight = False
      btnControl = True
  elif not bButtonPressed and btnControl:
      btnControl = False
  #Function-loop to recolor LEDs
  if (ticks_ms() - lastValue) > TIMECOLORCHANGE and bActiveLight:
      hue += 50
      if(hue > 65535):
          hue = 0
      color = pixels.colorHSV(hue, 255, 255)
      pixels.fill(color)
      pixels.show()
      lastValue = ticks_ms()
  elif not bActiveLight:
      hue = 0
      color = pixels.colorHSV(hue, 0, 0)
      pixels.fill(color)
      pixels.show()
  #Timer to set lights off
  if (ticks_ms() - lastActiveSet) > TIMECOLOROFF:
    bActiveLight = False
```

Code 3: Der Nachtlichcode nur für Taster

Das bringt den Vorteil, dass Sie (sofern das Projekt gefällt) recht schnell modifiziert werden kann. Auch ist das Konzept, wie der WS2812B seine Farbe verändert recht schnell einsehbar. Letztlich wird alle 50msder Wert für den Farbbereich erhöht. Die Zeit ist über die Variable **TIMECOLORCHANGE** einstellbar. Überschreitet der Code die Magische Grenze von 65535, so wird der Farbbereichswert wieder auf Null gesetzt.

Die Zeichnung, die Modelle und das Programm

Da ich alles mit Autodesk Fusion 360 entworfen habe, kann ich leider die Urmodelle nicht so weitergeben. Jedoch können Slicer wie Cura mit dem 3mf-Format umgehen, welches ich mittlerweile sehr gerne zum Exportieren verwende. Auch die maßstabsgetreue Zeichnung zum ausschneiden der Streifenrasterplatine habe ich auf DIN A4 in meinem GitHub Repository bereitgestellt.
Wie bei meinen letzten Beiträgen zum Pico gewünscht, finden Sie nicht nur das Programm, sondern auch die verwendete Bibliothek bei GitHub.

Zusammenfassung

Wie Sie sehen, können Sie mit etwas löten, einem 3D-Drucker und ein paar Zeilen MicroPython Kindern eine kleine Freude machen. Als ich das fertige Projekt meiner Frau und Kindern gezeigt habe, waren die Augen groß und das Kinderzimmer leuchtet nun nicht mehr „nur“ blau, sondern in „allen Regenbodenfarben“.
Gleichzeitig habe ich das Projekt und den Code so gehalten, dass Sie ohne Probleme Helligkeit, Farbwechsel und Anzahl der LEDs verändern können. Möglich ist auch weitere Hardware mit dem Raspberry Pico zu verlöten, in meinem Fall ein Bewegungssensor, der dann die Lampe zum Leuchten bringt. Lassen Sie Ihrer Fantasie freien Lauf, die Basis für spannende Projekte ist gemacht.

Weitere Projekte für Az-Delivery von mir, finden Sie unter https://github.com/M3taKn1ght/Blog-Repo.