Fertige Lampe

Diesen Beitrag gibt es auch als PDF-Dokument.

Direkteingaben im Terminal von Thonny (REPL) sind im Text **fett** formatiert und am Prompt **>>>** zu erkennen. Antworten vom ESP32/ESP8266 sind ***kursiv*** gesetzt.

Das Projekt ist wegen der wenigen Baugruppen und der einfachen Programmierung für Einsteiger gut geeignet. Allerdings wird ein wenig handwerkliches Geschick gefordert. Wer einen 3D-Drucker besitzt, kann das Lampengehäuse auch damit bauen.

Kürzlich war ich bei einem Bekannten zu Gast. Ganz stolz präsentierte er mir seine dimmbare Tischleuchte. Und weil seit einiger Zeit Neopixelringe mit 8 LEDs zu haben sind, hat's bei mir sofort klick gemacht – so ein Ding bau ich mir auch, nur nicht mit ausschließlicher hell-dunkel-Funktion, sondern mit Farbmischer und Dimm-Funktion. Ein kräftiger Li-Akku mit 2400 mAh sorgt dann auch für eine höhere Betriebsdauer, denn das Muster meines Bekannten war schon nach knapp eineinhalb Stunden aus. In diesem ersten Teil der Bau und Programmieranleitung wird es um die Herstellung der Hardware und die Programmierung des Controllers gehen. Willkommen also zu einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Eine Tischlampe mit Neopixelring und ESP8266.

Als Bausteine für das Projekt brauchen wir neben einem Controller, das kann ein ESP8266 oder ESP32 sein, nur noch einen Neopixelring. Weil nur eine Ausgangsleitung benötigt wird, genügt sogar schon ein ESP8266-01(S). Als Bedienpult für die Lampe habe ich eine Handy-App gebastelt, deren Beschreibung im zweiten Teil folgt.

Im Zusammenhang mit dem ESP8266 gibt es eine wichtige Anmerkung. Wenn man einen Controller aus dieser Familie verwenden möchte, also zum Beispiel ESP8266-01, ESP8266 D1 mini oder Amica, gibt es eine Einschränkung in der Funktionalität. Irgendeine Stelle im MicroPython-Kern des Controllers sorgt ständig für Neustarts im Abstand von ca. 30 Sekunden, wenn das Accesspoint-Interface aktiv ist. Dieser Umstand verhindert daher den autonomen Betrieb der Lampe, wenn kein WLAN-Router zur Verfügung steht, über den die Verbindung mit dem Handy geführt werden kann. Für diesen Fall ist ein ESP32 nötig, der problemlos einen Accesspoint abgeben kann. Damit klappt die Verbindung dann auch beim kuschligen Tagesausklang mitten in der Pampa.

Hardware

Die Lamp-Ware

Den Gehäuserahmen für die Lampe habe ich aus 15mm-Birken-Sperrholz-Platten gesägt und auf Gehrung verleimt, nachdem ich eine Stufe ausgeklinkt hatte. Dort hinein passt genau eine Platte aus Pappelsperrholz von 6mm Dicke. Mittig habe ich einen Quader aus Buchenholz untergeklebt, der eine mittige Bohrung von 6mm und eine seitliche Bohrung von 3,2mm aufweist. Erstere nimmt das 6mm dicke Messingrohr als Halterung für den Lampenkopf auf, in letztere habe ich ein 4mm Gewinde gedreht. Mit einer 4mm-Rändelschraube kann man so das Rohr fixieren.

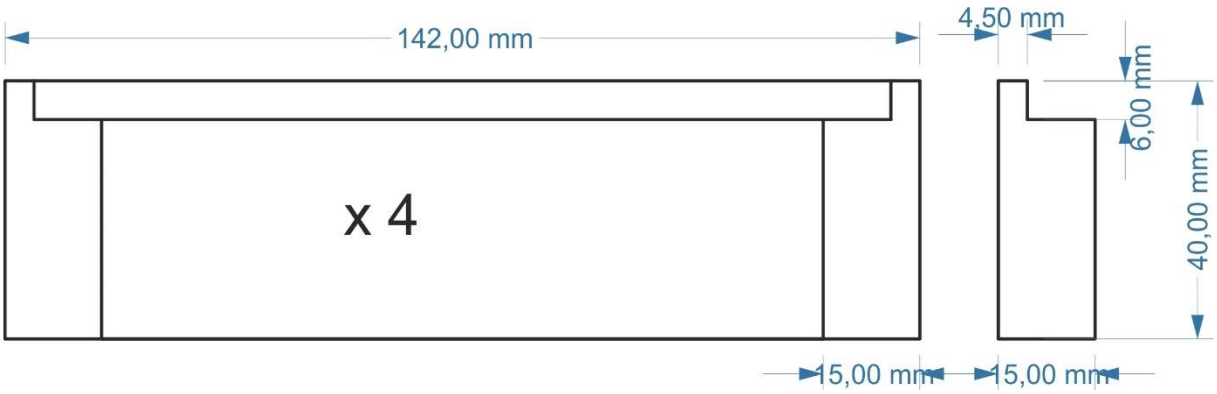

Abbildung 3: Tischlampe_Rahmen

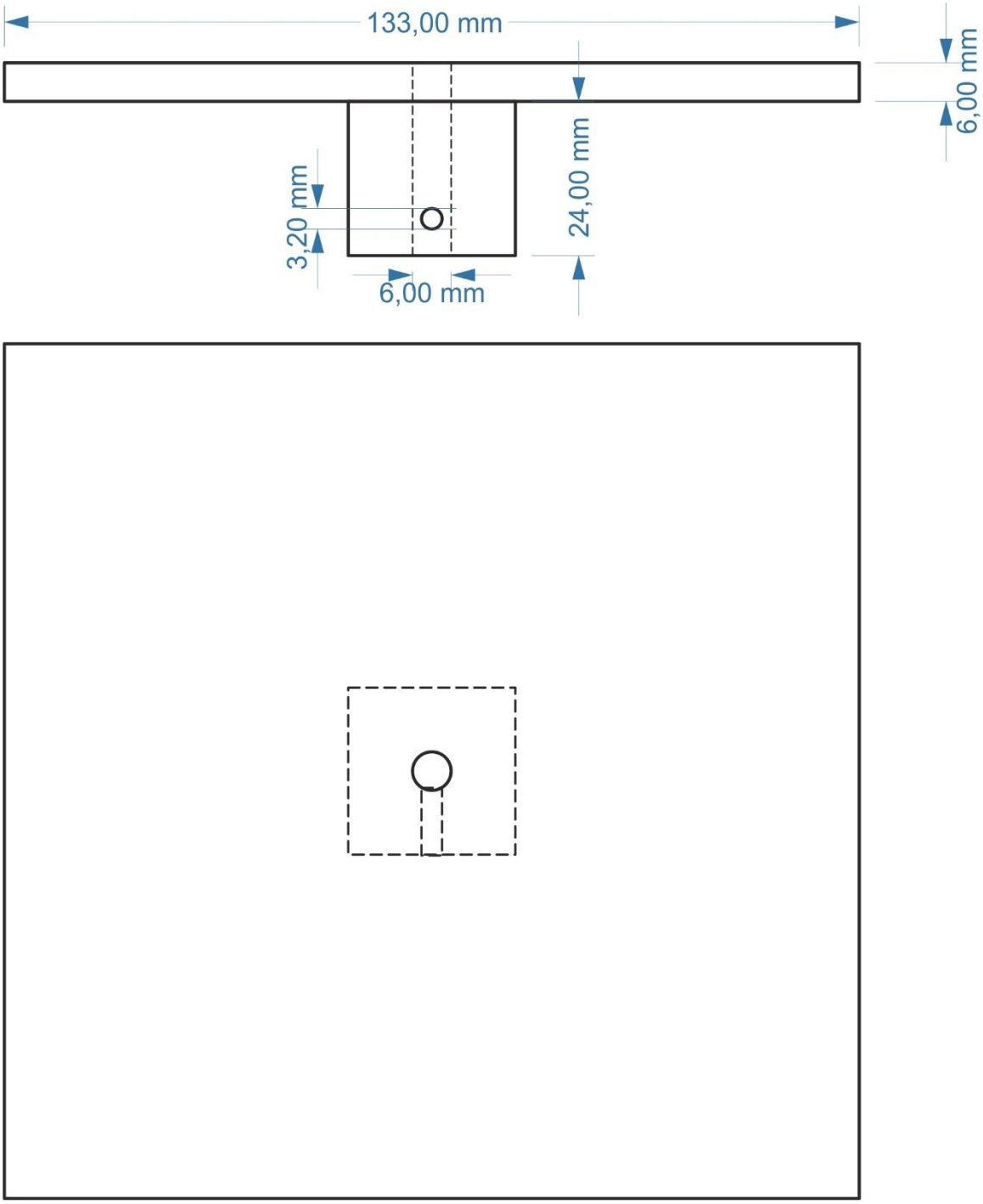

Abbildung 4: Tischlampe_Abdeckplatte mit Rohrhalterung

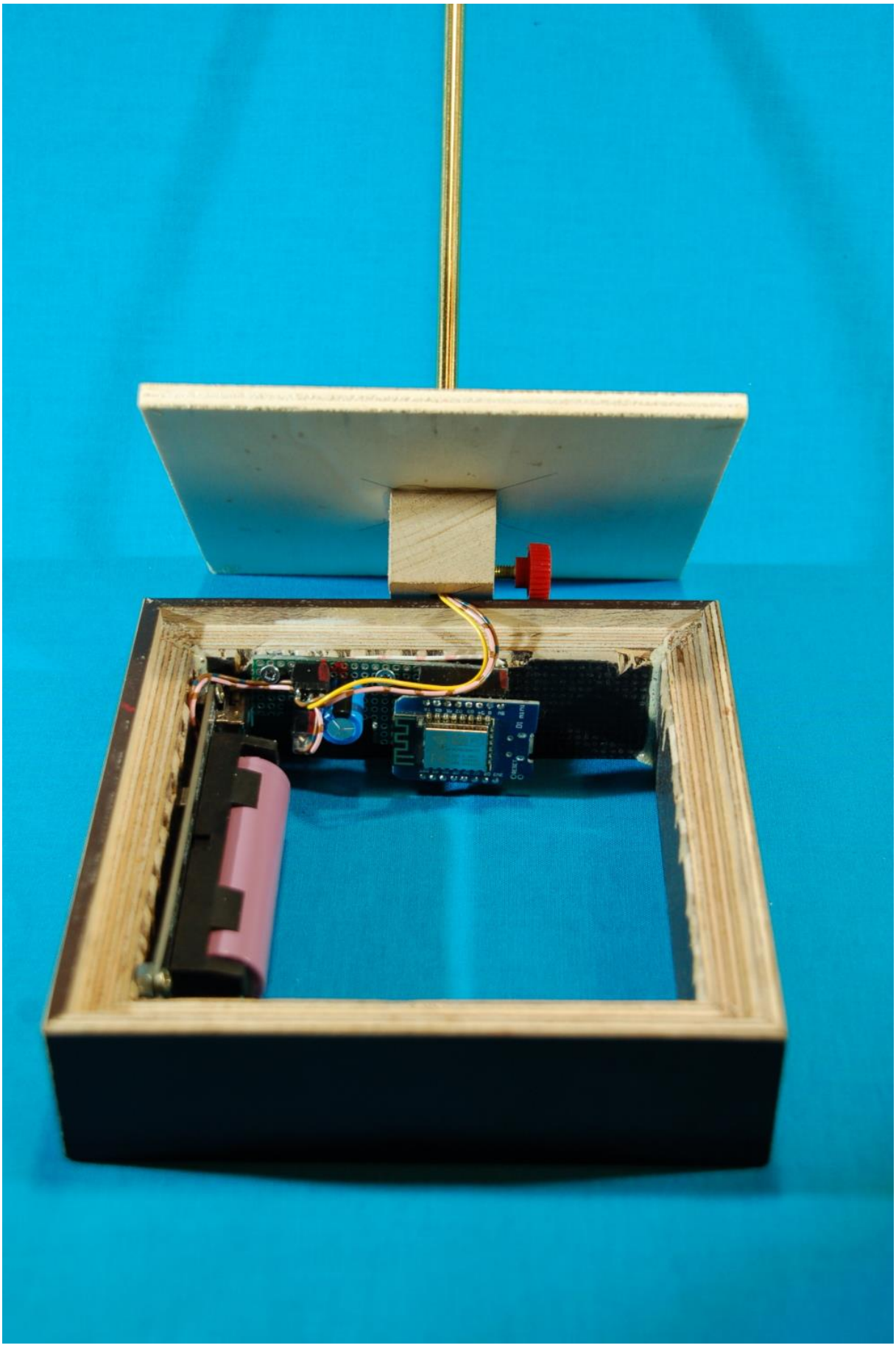

Abbildung 5: Batterie und Controller

Das 6mm-Messingrohr erhält am oberen Ende eine seitliche Bohrung für das Herausführen der drei Leitungen zum LED-Ring. Die Öffnung sitzt am unteren Ende eines Messingvollzylinders, der ein Stück weit auf 5mm Durchmesser abgedreht ist und eine axiale Sackloch-Bohrung von 2,5mm aufweist. Hier wird ein M3 Gewinde eingeschnitten. Der Vollzylinder wird ins Rohr gesteckt und mit ihm verlötet.

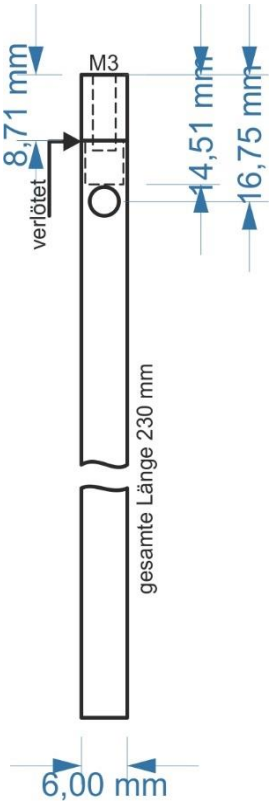

Abbildung 6: Tischlampe_Messingrohr mit Zapfen für Kopfhalterung

Den Lampenkopf habe ich aus einer 40mm-Bambusplatte hergestellt. Der LED-Ring wird auf ein rechteckiges Stück Plexiglas geschraubt, das seinerseits im Lampenkopf verschraubt ist.

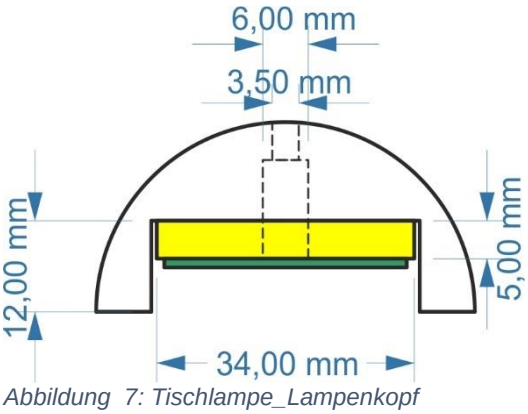

Abbildung 7: Tischlampe_Lampenkopf

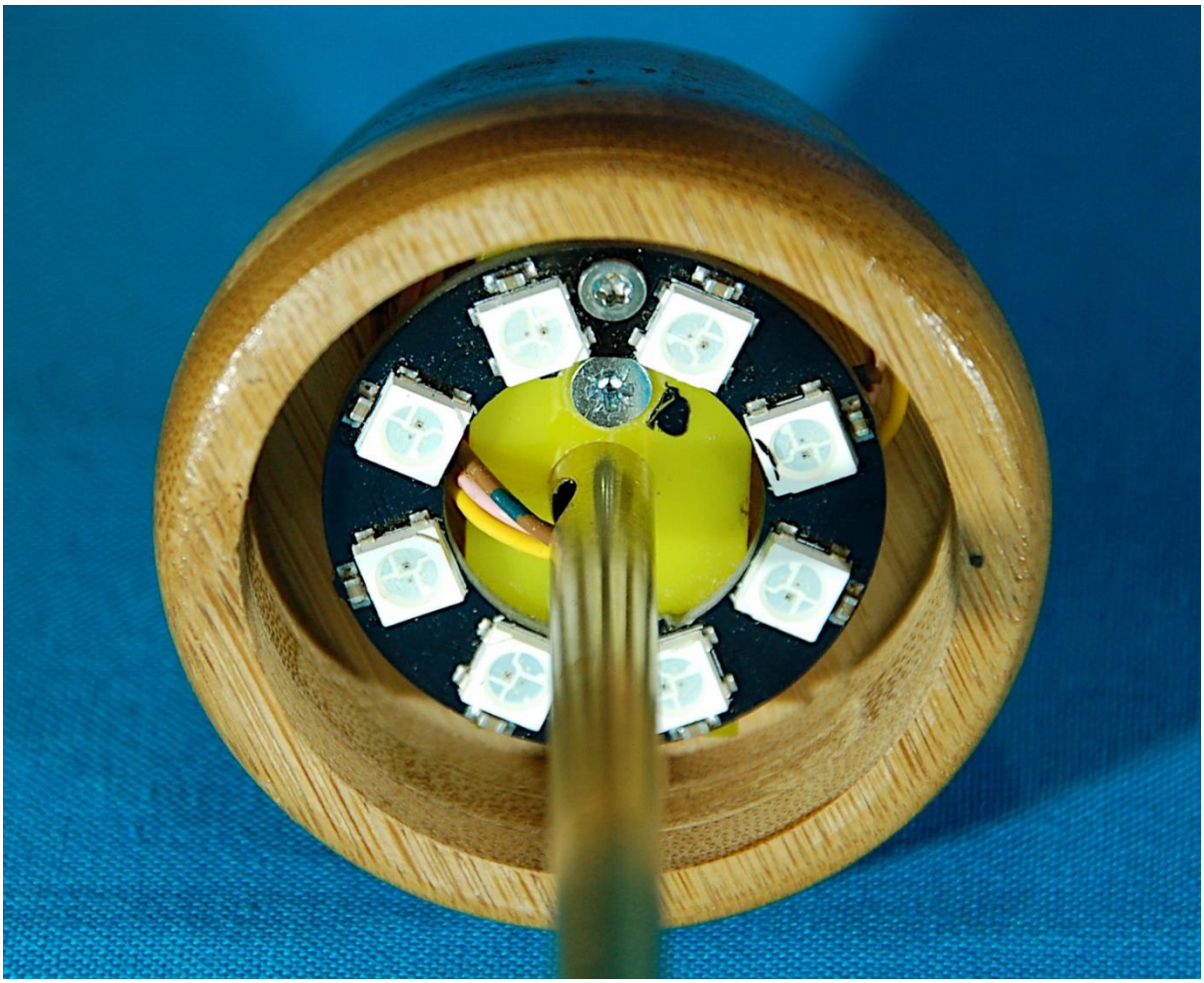

Abbildung 8: Lampenkopf mit 8-er Neopixelring

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender zum Testen des ESP8266 als UDP-Client und -Server
SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für einen ESP32:

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

Verwendete Firmware für einen ESP8266:

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

Die MicroPython-Programme zum Projekt:

timeout.py API für den Betrieb von Tasten
tischlampe.py Betriebsprogramm

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 05.02.2022) 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.

Die Schaltung

Wie in Abbildung 1 zu sehen ist, beziehe ich die Betriebsspannung für die Schaltung an der USB-Buchse des Batterie-Boards nach dem Schalter. Damit spare ich einen separaten Schalter ein, was den Aufbau vereinfacht.

Den Probeaufbau während der Entwicklung habe ich auf einem ESP8266-01S umgesetzt. Weil das marginale kleine Board keinen Spannungsregler an Bord hat, musste die Betriebsspannung von 3,3 Volt durch einen externen Spannungsregler aus einer 5-Volt-Quelle oder aus dem Batterieboard bereitgestellt werden. Ferner braucht man einen externen USB-TTL-Umsetzer, wie den FT232-AZ USB zu TTL Serial Adapter und zwei Taster.

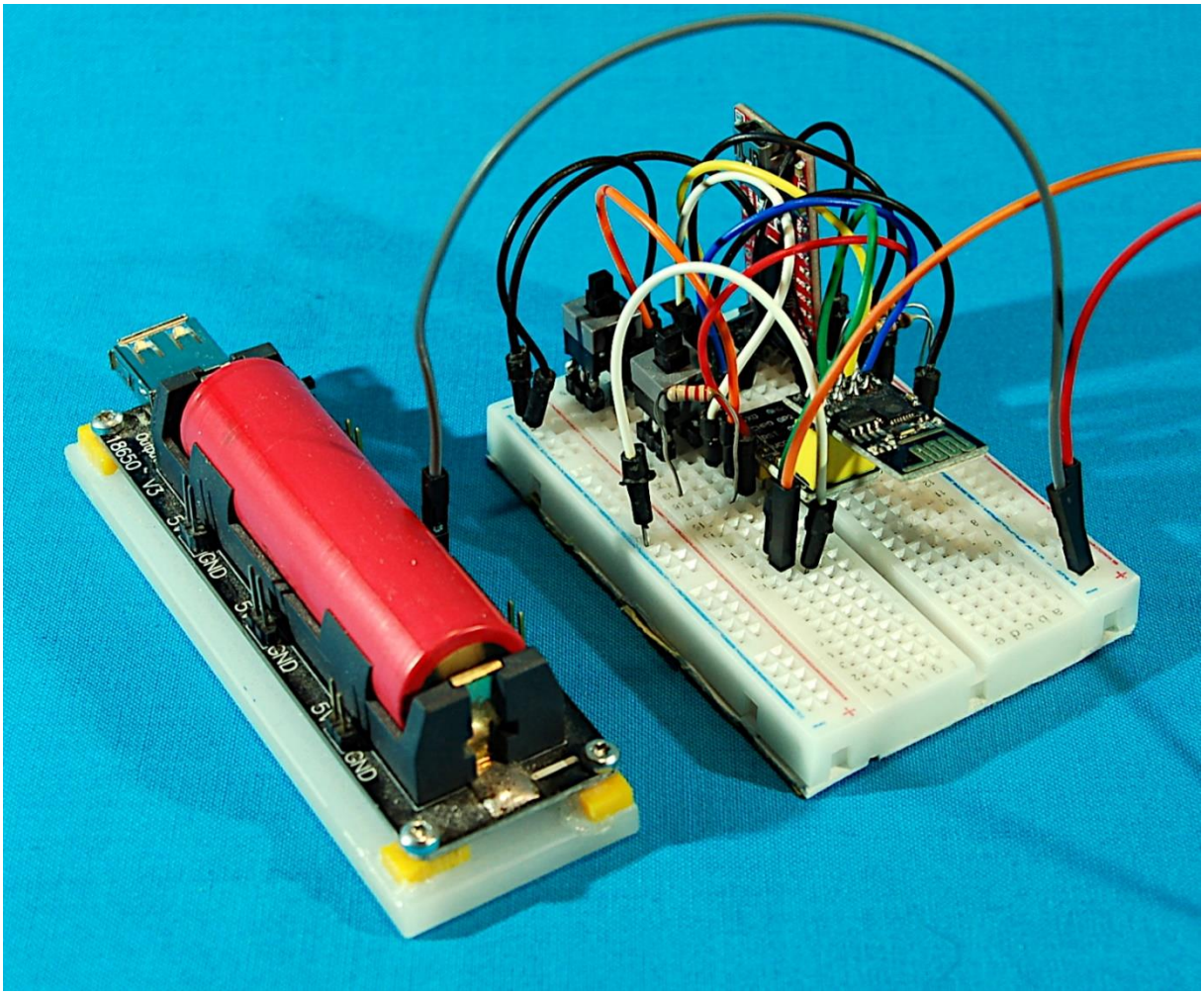

Abbildung 9: Aufbau mit ESP8266-01

Um den Aufwand zu minimalisieren, habe ich im fertigen Gerät dann doch lieber einen **ESP8266 D1 mini** als Controller verwendet. Die Lampe wird in der Nähe des Hauses betrieben, somit kann ich auf ein WLAN zugreifen und im Controller das Station-Interface verwenden. Dadurch ist der ordnungsgemäße Betrieb gewährleistet.

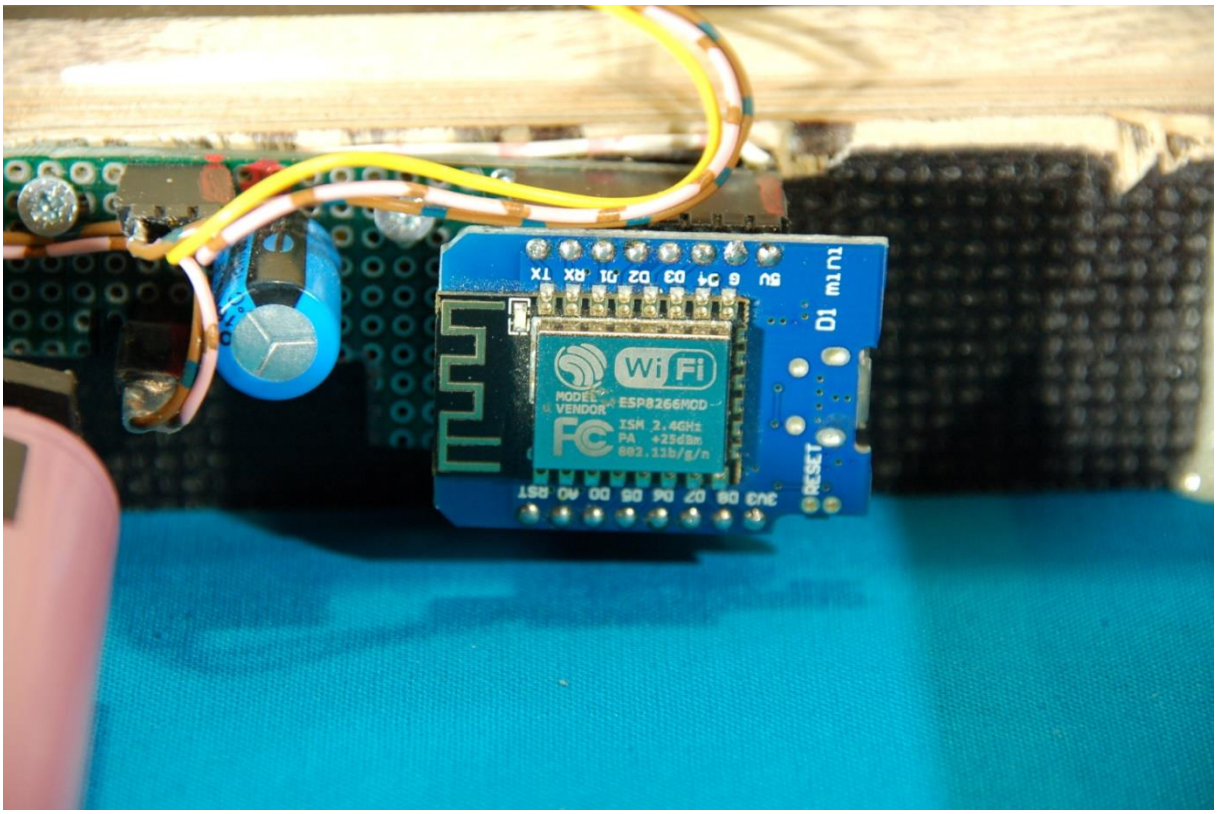

Abbildung 10: Aufbau auf Lochrasterplatine

Der Aufbau erfolgte auf einem Stück Lochrasterplatine. Für das D1-Board habe ich zwei 8-er Buchsenleisten verlötet, desgleichen für die Anschlüsse vom Batterieboard (zweier) und dem LED-Ring (dreier). Wer möchte, kann sich eine Platine auch selbst ätzen. Das Layout ist in dem PDF-Dokument cotrollerboard.pdf zu finden.

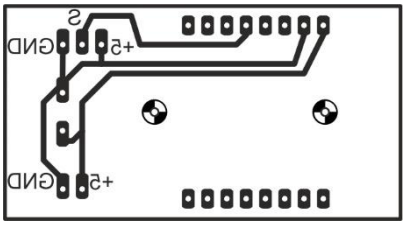
Abbildung 11: tischlampe_pcb

Der Neopixelring

Neopixel-LEDs vom Typ WS2812 enthalten drei einzelne Pixel, die rotes, grünes oder blaues Licht abgeben. Angesprochen werden sie von einem Controller, der seine Anweisungen über eine Art Bussystem erhält, das mit 800kHz getaktet wird.

Beim I2C-Bus oder beim SPI-Bus erreichen die Signale vom Controller, zum Beispiel einem ESP32, alle Slaves am Bus in gleicher Weise, alle sehen alles. Bei den WS2812-Bausteinen ist das anders. Jeder Baustein hat einen Dateneingang und einen Datenausgang. Mehrere Bausteine können kaskadiert werden, indem man den Dateneingang jedes weiteren Bausteins an den Datenausgang des Vorgängers anschließt. Der erste Baustein in der Kette bekommt vom ESP32 eine Pulskette von

drei Bytes für R, G und B zugespielt, die er selbst futtert, also vom gesamten Pulszug entfernt. Alle nachfolgenden Impulse werden durchgewunken und am Datenausgang abgegeben. Jeder Baustein in der Kette holt sich auf dieselbe Weise seinen Anteil und gibt den Rest weiter. So wird es möglich, dass jeder Baustein in der Kette individuell angesteuert werden kann. Die Intensität jeder Farbe kann in 256 Stufen variiert werden, je nach dem Wert des empfangenen Bytes für die einzelnen Farben. Wir geben den Farbcode in Form eines Tupels für jeden WS2812 als Element einer Liste an. Zuerst werden die Klassen **Pin** und **NeoPixel** importiert. Ich erzeuge ein Pin-Objekt als Ausgang und instanziiere damit ein Neopixel-Objekt mit acht Bausteinen.
Die NeoPixel-Instanz **neo** enthält ein Bytearray **buf** und die Methode **write**(), mit welcher der Inhalt des Bytearrays zum Neopixelring übertragen wird.

```
>>> from machine import Pin
>>> from neopixel import NeoPixel
>>> np=Pin(4,Pin.OUT) # D2
>>> neo=NeoPixel(np,8)
>>> neo[0]=(0xe0,0x07,0x3c)
>>> neo[1]=(0xf0,0xf0,0xf0)
>>> neo.write()
>>> neo.buf
```

bytearray(b'\x07\xe0<\xf0\xf0\xf0\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')

Mit **neo**[0] spreche ich die ersten drei Elemente des Arrays an und übergebe die Werte für rot, grün und blau, 0xe0, 0x07 und 0x3c. Intern macht das die private Funktion __setitem__(). In den Buffer werden die Werte in veränderter Reihenfolge eingetragen. Wie wir gleich am Kurvenzug sehen werden, den ich mit Hilfe des Logic Analyzers aufgezeichnet habe. Die Werte werden so gesendet, wie sie im Puffer stehen. Einer 0 entspricht ein schmaler Impuls von ca. 250ns Breite gefolgt von einer Pause von ca. 1000ns. Die 1 wird durch einen Puls von 750ns und einer Pause von 500ns gesendet.

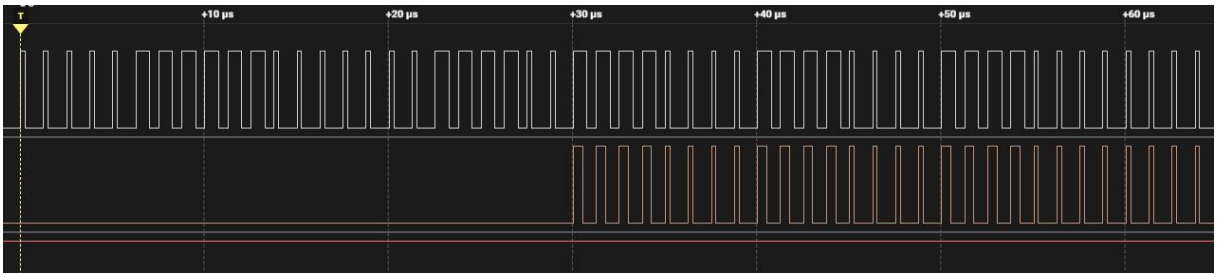

Abbildung 12: Pulsfolge für RGB = 0xE0, 0x07, 0x3C

Am Ausgang der ersten WS2812 fehlen die drei Bytes 0xe0, 0x07 und 0x3c, die dieser Baustein gefressen hat. Stattdessen kommt der Code 0xff, 0xff, 0xff heraus, der Code für den zweiten Baustein. Der Eingang für Kanal 1 des Logic Analyzers war für diese Messung am Eingang der ersten LED, Kanal 2 am Ausgang derselben angeschlossen.

Jetzt wissen wir wie das Känguru beim WS2812 läuft und können uns dem Programm für die Lampe zuwenden.

Das Programm Tischlampe.py

Das Programm hat zwei Aufgaben. Es muss den Neopixelring ansteuern. Das geschieht im Prinzip so wie wir es oben über REPL bereits gezeigt haben. Dabei werden alle acht LEDs mit dem gleichen, nicht mit demselben, Code gefüttert. Ahnen Sie, warum nicht mit demselben? Eine Ausnahme gibt es, das ist die LED mit der Hausnummer 0. Die hat als zweiten Job den Auftrag, zu blinken, wenn Statusmeldungen weitergegeben werden sollen. Wir haben ja kein Display, also lassen wir blinken.

Ja und dann muss das Programm noch Funkaufträge entgegennehmen. Das macht es über eines der beiden WLAN-Interfaces. Ich habe mich für das Station-Interface entschieden, weil ich die Lampe nur im und ums Haus betreiben möchte. Der Code fürs Accesspoint-Interface ist zusätzlich im Programmtext enthalten. Will man in diesem Modus arbeiten, dann ist ein ESP32 statt dem ESP8266 die bessere Wahl. Den Grund dafür habe ich eingangs ja bereits dargestellt.

Also legen wir los.

```
# tischlampe.py

#LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8 RX TX
#ESP8266 Pins 16  5  4  0  2 14 12 13 15  3  1
#Achtung      hi sc sd hi hi          lo hi hi

# Nach dem Flashen der Firmware auf dem ESP8266:
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
# Dann: >>> import network
# >>> nic=network.WLAN(network.AP_IF)
# >>> nic.active(False)
#
```

Ein paar Kommentarzeilen geben Hinweise auf die Eigenheiten des ESP8266. Das fängt mit den GPIO-Pin-Bezeichnungen an. Sie gelten in LUA. Für MicroPython brauchen wir die Nummern aus der ESP8266-Zeile. Achtung weist auf besondere Belegungen der Pins hin. Damit der Controller überhaupt startet, müssen die Pins 0, 1, 2, 3, und 16 high sein, 15 muss auf GND-Potenzial liegen.

Wenn ein ESP8266neu geflasht wurde, versucht er beim Booten Kontakt mit einem WLAN-Router aufzunehmen, was wohl an Web-REPL liegt, dem Funk -Terminal das über einen Browser bedient wird. Das führt dann auch zu ständigen Neustarts, wenn die Verbindung nicht zustande kommt. Diese Eigenwilligkeit kann man dem Controller abgewöhnen, wenn man **webrepl_setup** importiert und am Prompt ein **d** für disable eingibt. Danach sollte ein Neustart mit Reset ausgelöst werden. Das Prozedere muss aber nur einmal nach dem Flashen der MicroPython-Firmware erfolgen. Zur Sicherheit kann man auch noch das Accesspoint-Interface ausschalten. Dieses paranoide Verhalten zeigt nur die ESP8266-Familie, ESP32er verhalten sich normal. Weshalb die ESP8266-er sich so verhalten wissen wohl nur die Götterväter des MicroPython-Consortiums.

Auch interessant ist, dass die ESP8266er das Station-Interface beim Start automatisch wieder mit dem WLAN-Router verbinden, mit dem sie zuletzt Kontakt hatten. Die Daten werden wohl in einem nicht flüchtigen Speicherbereich gebunkert. Daher geht beispielsweise folgendes, ohne dass man sich nach einem Reset oder Kaltstart beim WLAN-Router anmelden muss.

>>> import network
>>> nic = network.WLAN(network.STA_IF)
>>> nic.active(True)
>>> nic.isconnected()
True

Als Erstes erledigen wir jetzt das Importgeschäft.

```
from machine import Pin
from neopixel import NeoPixel
import network, socket
from timeout import *
from time import sleep, sleep_ms
import gc,sys
```

Für In-Out-Aktionen brauchen wir die Klasse **Pin** und natürlich **NeoPixel**. Importiert man Dinge in dieser Weise, dann werden die Namen von Konstanten, Funktionen und Methoden etc. transparent in den globalen Namensraum eingefügt. Um GPIO-Ein- oder Ausgänge zu erzeugen genügt

>>> taste=Pin(0,Pin.IN) # D3
>>> np=Pin(4,Pin.OUT) # D2

Das verhält sich bei **import network, socket** anders. Um zum Beispiel ein WLAN-Verbindungs-Objekt zu erzeugen muss es jetzt heißen

>>> nic = network.WLAN(network.AP_IF)

Eine weitere Importmethode ist die mit dem "*****". Sie arbeitet ähnlich wie die ersten beiden Importe mit **from**, bindet aber alle Bezeichner des Moduls in den globalen Namensraum ein und zwar auch Bezeichner, die im Modul **timeout** selbst importiert wurden. Das sind zum Beispiel **ticks_ms** und **time** aus dem Modul **time**.

>>> from timeout import *
>>> time
<*function*>

>>> TimeOutUs
<*function TimeOutUs at 0x3ffef480*>

Die Funktion **sleep_ms**() aus dem Modul **time** kann nach dem Import der Funktion direkt so aufgerufen werden.

>>> sleep_ms(1000)

Die Funktion **exit**() aus dem Modul **sys** braucht aber das Prefix **sys**.

>>> sys.exit()

```
taste=Pin(0,Pin.IN, Pin.PULL_UP) # D3
np=Pin(4,Pin.OUT) # D2
neo=NeoPixel(np,8)
state=" "
# state="ap"
```

Die Pin-Objekte stellen den Eingang für eine Taste und einen Ausgang für die Steuerleitung zum Neopixelring dar. Mit der GPIO-Instanz **np** erzeugen wir ein Neopixel-Objekt **neo**. Es stellt unter anderem die Funktion **write**() und das Bytearray **buf** mit den 8 x 3 = 24 Farb-Bytes zur Verfügung.

>>> neo
<NeoPixel object at 3ffef380>
>>> dir(neo)
*['__class__', '__getitem__', '__init__', '__len__', '__module__', '__qualname__', '__**setitem**__', **'write'**, '__dict__', 'fill', 'pin', **'buf'**, 'ORDER', 'n', 'bpp', 'timing']*

Die private Methode __**setitem**__() übernimmt ein Code-Tupel und die Adresse des anzusteuernden Ringelements und stellt die RGB-Werte in das Array **buf** ein. die Listenreferenz **neo[2]** übergibt die Werte an __**setitem**__(2,(59,64,128))

>>> neo[2]=(59,64,128)
>>> neo[2]
(59, 64, 128)

>>> neo.buf
*bytearray(b'\x00\x00\x00\x00\x00\x00@**;\x80**\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00\x00')*

>>> neo[2]=(59,64,128)
und
>>> __setitem__(2,(59,64,128))
Sind also synonym zu gebrauchen.

Mit **state=" "** wird eine Statusvariable erzeugt, die das Verbindungsinterface festlegt und entsprechend die SSID, Passwort und IP Vergabe steuert. Mit **state="ap"** wird das Accesspoint-Interface gestartet, sonst das Station-Interface. In beiden Fällen werden dieselbe IP und dieselbe Portnummer verwendet.

Mit **color = [0,0,0]** deklariere ich eine Variable, die im Programm die aktuellen Farbwerte aufnehmen wird.

```
if state=="ap":
    mySSID = 'lampy'
    myPass = 'everybody'
else:
    mySSID = 'EMPIRE_OF_ANTS'
    myPass = 'nightingale'
myIP="10.0.1.96"
myPort=9091
```

Die Variable **connectstatus** liefert über das Dictionary die Übersetzung der Statusmeldungen beim Verbindungsaufbau in Klartext. Statt der Statusnummer wird der zugewiesene String ausgegeben.

```
#********************Variablen deklarieren
**********************
# Die Dictionarystruktur (dict) erlaubt spaeter die
Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN"
    }
```

Auch die Funktion **hexMac**() ist als Übersetzer tätig. Die Funktion **nic.config**('mac') liefert ein meist kryptisches **bytes**-Objekt, welches **hexMac**() in einen normal lesbaren String aus Hexadezimalziffern übersetzt.

>>> nic = network.WLAN(network.STA_IF)
>>> nic.config('mac')
b'\xa6\xcf\x12\xdf_h'
>>> hexMac(nic.config('mac'))
'a6-cf-12-df-5f-68'

Die Funktion **lum**() schreibt die Farbwerte aus dem Tupel **color** in die Bytes des Farbpuffers **neo.buf** und überträgt dessen Inhalt schließlich an den Ring.

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

Die Funktionen **red**(), **green**() und **blue**() nehmen einen Helligkeitswert von 0 bis 255 und tragen ihn in das Farb-Tupel **color** an der entsprechenden Stelle ein, rot an Position 0, grün an Position 1 und blau an Position 2. Dann wird **lum**() aufgerufen, damit die Änderung sichtbar wird.

```
def red(val):
    global color
    color[0]=(val)
    lum()

def green(val):
    global color
    color[1]=(val)
    lum()

def blue(val):
    global color
    color[2]=(val)
    lum()
```

Der Name von **aus**() ist Programm, denn durch den Aufruf gehen alle LEDs aus.

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

Mit **blink**() kann ich Systemmeldungen sichtbar machen. Die Funktion nimmt die Leuchtdauer, die Pause danach, die Anzahl von Blitzen und die Farbe im Tupel **col**.

```
def blink(dauer,pause,count,col):
    merken=neo[0]
    for _ in range(count):
        neo[0]=col
        neo.write()
        sleep_ms(dauer)
        neo[0]=(0,0,0)
        neo.write()
        sleep_ms(pause)
    neo[0]=merken
    neo.write()
```

Die aktuelle Farbe wird in der lokalen Variablen **merken** gesichert. Die for-Schleife wird **count**-mal durchlaufen. Dann setze ich die LED 0 auf den Farbwert von **col**. Die LED leuchtet für **dauer**-Millisekunden und verfinstert sich für **pause**-Millisekunden. Danach wird die vorherige Farbe der LED restauriert.

Jetzt geht es an das Einrichten des Funk-Interfaces. Mit **state = "ap"** wird ein Accesspoint-Interface erzeugt.

```
if state=="ap":
    # ***************** AP einrichten **********************
    # Constructoraufruf erzeugt WiFi-Objekt nic
    nic = network.WLAN(network.AP_IF)
    nic.active(True)  # Objekt nic einschalten
    #
    MAC = nic.config('mac')  # binaere MAC-Adresse abrufen und
    myMac=hexMac(MAC)        # in Hexziffernfolge umgewandeln
    print("AP MAC: \t"+myMac+"\n") # ausgeben
    #
    nic.ifconfig((myIP,"255.255.255.0",myIP,myIP))
    print(nic.ifconfig())
    nic.config(authmode=0)
    print("Authentication mode:",nic.config("authmode"))
    nic.config(essid=mySSID, password=myPass)
    while not nic.active():
      print(".",end="")
      sleep(1)
    print("NIC active:",nic.active())
```

Wir erzeugen ein Interface-Objekt, das wir auch gleich aktivieren. Dann rufen wir die MAC des Objekts ab, decodieren sie und lassen uns den String ausgeben. Wir lassen den Controller logisch als Server arbeiten und vergeben daher eine feste IP und Portnummer, das erledigt .**ifconfig**(). Das Interface ist offen, erfordert daher keine Authentifizierung. Trotzdem muss der Methode .**config**() ein Passwort übergeben werden, weil sonst ein Fehler gemeldet wird. Wir warten bis das Interface bereit ist und setzen jede Sekunde einen Punkt im Ausgabefenster von REPL. Abschließend rufen wir den Bereitschaftszustand ab. Beachten Sie bitte, dass diese Variante für den ESP8266 unter MicroPython nicht funktioniert, sehr wohl aber bei einem ESP32.

```
else:
    nic=network.WLAN(network.AP_IF)
    nic.active(False)
    nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt
    nic.active(True)                    # nic einschalten
    MAC = nic.config('mac')# binaere MAC-Adresse abrufen und
    myMac=hexMac(MAC)      # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    sleep(1)
    nic.ifconfig((myIP,"255.255.255.0",myIP,myIP))
    if not nic.isconnected():
      nic.connect(mySSID, myPass)
      print("Status: ", nic.isconnected())
      while nic.status() != network.STAT_GOT_IP:
        print(".",end='')
        blink(100,900,1,(0,64,0))
    print("\nStatus: ",connectStatus[nic.status()])
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    print()
```

Alternativ können wir beim ESP32 ein Station-Interface einrichten, beim ESP8266 ist das die einzige Möglichkeit. Zuerst deaktivieren wir definitiv das Accesspoint-Interface, um danach sofort das Station-Interface zu aktivieren. Nach der Abfrage der MAC-Adresse braucht der Controller eine kurze Verschnaufpause.

Wir übergeben .ifconfig() IP-Adresse, Netzwerkmaske, Gateway- und DNS-Adresse. Die letzteren beiden müssen angegeben werden, können aber auf die IP des Controllers gesetzt werden, wenn kein Zugriff ins WWW geplant ist. Sonst stehen hier die IP-Adressen des WLAN-Routers, der die entsprechenden Dienste zur Verfügung stellt. Steht noch keine Verbindung, dann melden wir uns beim WLAN-Router mit seiner SSID und dem dort hinterlegten Passwort an. Während wir auf die Verbindung warten lassen wir die Status-LED (LED0) im Sekundentakt grün blinken. Während der Entwicklungsphase werden die gesetzten Daten im REPL-Terminal ausgegeben.

In jedem Fall brauchen wir jetzt ein Socket-Objekt. Das ist das Tor für den Datentransfer, so wie ein Datei-Handle den Transfer in eine oder aus einer Datei steuert.

```
# UDP-Server einrichten
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("listening on port",myPort)
s.settimeout(0.1)
```

Die Abwicklung läuft über IP-V4-Adressen (**socket.AF_INET**) und das UDP-Protokoll (**socket.SOCK_DGRAM**). Mit der Option **socket.SO_REUSEADDR** ermöglichen wir die erneute Verwendung derselben IP und Portnummer nach einem Neustart des Programms ohne vorherigen Reset.

Der Aufruf von .**bind**() bindet die Portadresse in **myPort** an die IP-Adresse, die wir oben dem Interface zugewiesen haben.

Schließlich setzen wir einen Timeout von 0,1 Sekunden fest. Das führt dazu, dass die Abfrage der Empfangsschleife des Sockets die Hauptschleife nicht blockiert. Wurden keine Zeichen empfangen, wirft der Prozess eine **OSError-Exception**, die wir abfangen und damit den ungestörten Ablauf der Hauptschleife gewährleisten. Würden wir keinen Timeout definieren, dann würde die Empfangsschleife erst verlassen, wenn Zeichen angekommen sind. Bis dahin wäre die Hauptschleife blockiert.

Die Main Loop ist eine Endlos-Schleife, weil 1 stets als True interpretiert wird. Als Erstes sammeln wir den Datenmüll und versuchen dann, Zeichen aus dem Empfangspuffer zu holen. Wenn es nichts zu holen gibt, sichert der **except**-Zweig, in welchem **pass** einfach nichts tut den weiteren Durchlauf der Hauptschleife. Mindestens eine Anweisung muss hier stehen, sonst bekommen wir einen Syntaxfehler.

```
# Serverschleife
while 1:
    gc.collect()
    try:
        # Nachricht empfangen
        rec,adr=s.recvfrom(150)
        rec=rec.decode().strip("\r\n")
        print(rec,adr)
        # Nachricht "r,g,b" parsen und
        # Aktionen ausloesen
        r,g,b=rec.split(",")
        color=[int(r),int(g),int(b)]
        lum()
        print(color)
        # Ergebnisse encodiert oder als String senden
        # es kann an mehrere Adressen gesendet werden
#         reply="M;started\n".encode()
#         s.sendto(reply,adr)
        rec=""
    except:
        pass # timeout uebergehen
```

Ich habe hier ein **sys.exit**() unmittelbar vor der Hauptschleife platziert und das Programm gestartet. Dadurch sind nach dem Programmstopp die Netzwerkverbindung und der Socket etabliert, und ich kann jetzt darauf von REPL aus zugreifen. Mit dem Tool **packetsender.exe** kann ich dann vom PC aus UDP-Pakete an den Controller versenden und prüfen, was der damit anstellt.

Zum Test sehr gut geeignet ist die Freeware packetsender. Man kann hier Steuerbefehle von Hand eingeben, via UDP an den Server auf dem ESP8266 schicken und so die Reaktion des Programms testen. später übernimmt das Smartphone das Kommando, wenn alles perfekt funktioniert. Installieren Sie also jetzt die Software und stellen Sie ganz unten den gewünschten UDP-Port Ihres Rechners ein, hier UDP:9091. Die Daten des ESP8266 werden in der vierten Zeile eingetragen, IP, Port und Protokoll. Schicken Sie nun den Farbcode "32,32,8" an den ESP8266 ab.

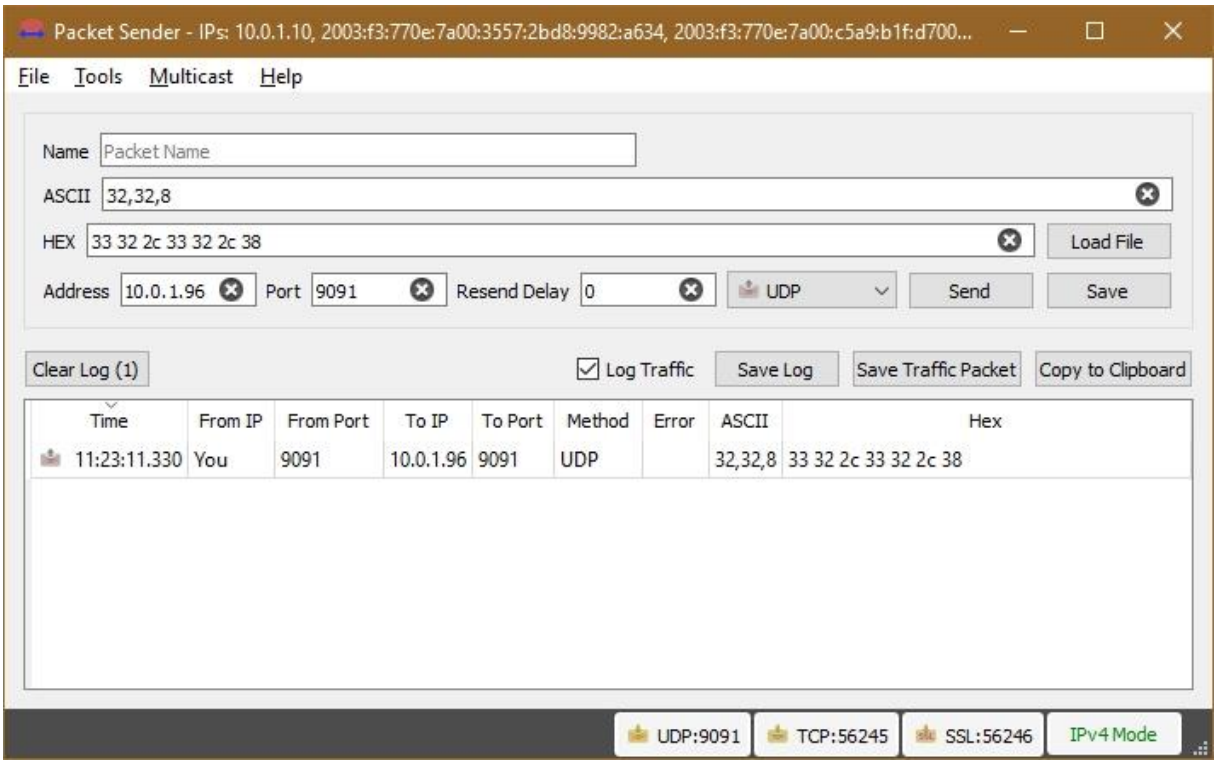

Abbildung 13: Packetsender-Fenster

Sind Daten angekommen, dann teilen wir das Tupel der folgenden Form in seine Bestandteile auf, das Bytes-Objekt und die Socketdaten des Senders, also des PCs. Das passiert durch Unpacking indem wir den Inhalt des Tupels zwei Variablen zuweisen.

>>> s.recvfrom(150)
(b'32,32,8', ('10.0.1.10', 9091))

Oder besser:
>>> rec,adr=s.recvfrom(150)

rec enthält nun die Zeichenfolge und **adr** die Socketdaten.

>>> rec;adr
b'32,32,8'
('10.0.1.10', 9091)

```
rec=rec.decode().strip("\r\n")
```
Das Bytes-Objekt wandeln wir in einen String um und entfernen Zeilenvorschub (\n) und Wagenrücklauf (\r).

Dann teilen wir den String an den Kommas in rot-grün-blau-Werte auf. Die entstehende Liste entpacken wir in die Variablen r, g und b.
```
r,g,b=rec.split(",")
```

Die in Ganzzahlen konvertierten String-Werte weisen wir dem Tupel **color** zu, **lum**() sendet sie an den Ring.

```
color=[int(r),int(g),int(b)]
lum()
```

Das Schlusslicht bildet die optionale Tastenabfrage. Ist eine Taste an GPIO0 verfügbar, dann kann die Hauptschleife sauber verlassen werden, nachdem die LEDs ausgeschaltet wurden.

```
    if taste.value()==0:
        aus()
        sys.exit()
```

Der Test

Für einen ersten ultimativen Test entferne ich jetzt das **sys.exit**() vor der Hauptschleife und starte das Programm neu – **F5**.

>>> %Run -c $EDITOR_CONTENT
STATION MAC: a4-cf-12-df-5f-68

Status: UNKNOWN
STA-IP: 10.0.1.96
STA-NETMASK: 255.255.255.0
STA-GATEWAY: 10.0.1.96

listening on port 9091

Über Packetsender kann man nun die Farbe der Lampe und deren Helligkeit durch Senden von Farb-Tupeln steuern.

In der nächsten Folge bauen wir eine Handy-App, die dann per Schieberegler die Steuerung der Lampe übernimmt.

Also – dranbleiben!