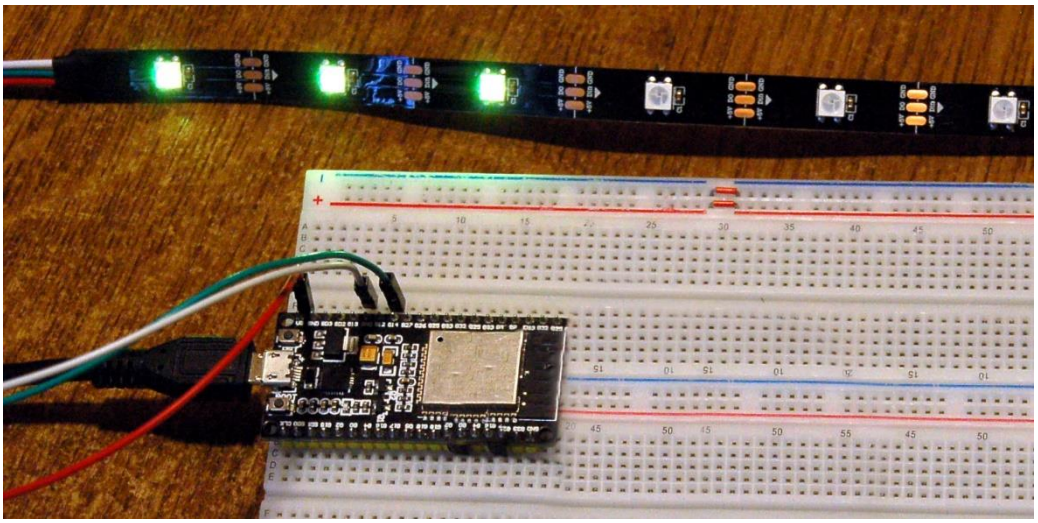

LED-Streifen-Uhr_Ausbaustufe 1

Diesen Beitrag gibt es auch als PDF-Dokument.

Die Schaltung ist ausgesprochen einfach aufgebaut und daher für Anfänger bestens geeignet. In Ausbaustufe 1 stellt auch die Programmierung kein nennenswertes Problem dar. Im Folgebeitrag werden wir ein OLED-Display und einen BME280 hinzufügen, was die Schaltung nur unwesentlich aufmufft. Zur Präzision der Zeitangaben werden wir die ESP32-eigene RTC (aka Real Time Clock = Echtzeituhr) durch eine Synchronisation via NTP (Network Time Protokoll) aufbessern.

Neopixel-LED-Streifen kann man nicht nur als Selbstzweck zur Beleuchtung verwenden. Interessant wird die Beleuchtung, wenn sie auch noch einen weiteren Zweck erfüllt. So kam ich auf die Idee, mit so einem Streifen eine Uhr zu bauen. Weil die Streifen sich aber nicht zu einem Kreis biegen lassen, so dass alle Pixel in dieselbe Richtung zeigen, wurde daraus eine lineare Zeitanzeige.

Einen Streifen mit 60 LEDs (pro Meter) könnte man als Abwicklung eines Zifferblatts mit 60 Minuten ansehen. Die Zeigerpositionen ließen sich dann durch eine leuchtende LED angeben, die Minuten zum Beispiel in Blau, die Stunden in Rot. Aber eine Uhr mit einem Meter Länge ist schon etwas unhandlich. Es geht tatsächlich auch mit knapp der halben Anzahl, mit 26 LEDs lassen sich mit einem Trick sogar Stunden, Minuten und Sekunden anzeigen. Das funktioniert so ähnlich wie das Rechnen mit dem Abakus. Die Stunden werden mit sechs LEDs, Minuten und Sekunden mit je 10 LEDs dargestellt. Wie das im Einzelnen funktioniert, das erfahren Sie in der neuen Folge aus der Reihe

MicroPython auf dem ESP32, dem ESP8266 und dem Raspberry Pi Pico

heute

Die Abakus-Uhr

Der vorliegende Entwurf gehört in jedem Fall zu den "etwas anderen" Uhren. Mitdenken beim Ablesen ist erforderlich. Wie funktioniert das Ganze?

Vom LED-Streifen mit 30 LEDs pro Meter werden drei Teile abgeschnitten. Sechs LEDs für die Stunden, und je 10 LEDs für die Minuten und Sekunden werden an den dafür vorgesehenen Stellen abgetrennt und von oben nach unten auf einen glatten Untergrund als Träger geklebt. Die Ausgänge habe ich über dreiadrige Leitungen mit den darauffolgenden Eingängen des nächsten Streifens verbunden. Den Eingang des Stunden-Streifens habe ich an den 5V-Pin des ESP32 gelegt, der als Koordinator des Aufbaus verwendet wird. Die Eingangs-Steuerleitung des Streifens liegt an GPIO14 und GND kommt an einen GND-Anschluss des ESP32. Der Schaltplan zeigt die Einzelheiten, das Ganze ist echt easy!

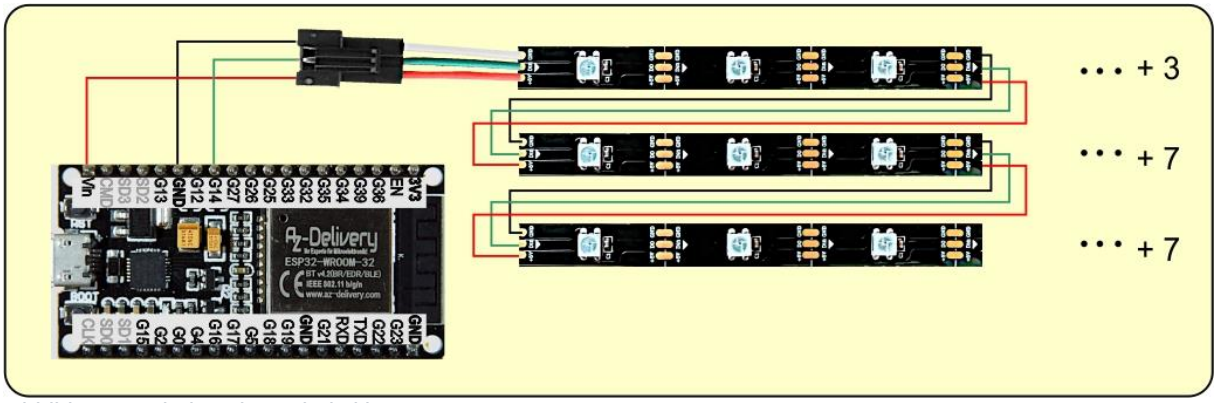

Abbildung 1: Abakusuhr_Schaltskizze

Was hat das aber jetzt mit einem Abakus (aka Rechenbrett) zu tun? Der chinesische Abakus, genannt Suanpan, besitzt zwei Felder mit 2 beziehungsweise 5 Perlen auf jedem Stäbchen. Die fünf Perlen im unteren Feld haben den Wert 1, die beiden im oberen Feld den Wert 5. Die Stäbchen stellen von rechts nach links die Stufenwerte des 10-er-Systems dar, 1-er, 10-er, 100-er … Gewertet werden die zur Trennwand geschobenen Perlen. In Abbildung 2 wird die Zahl 598 dargestellt.

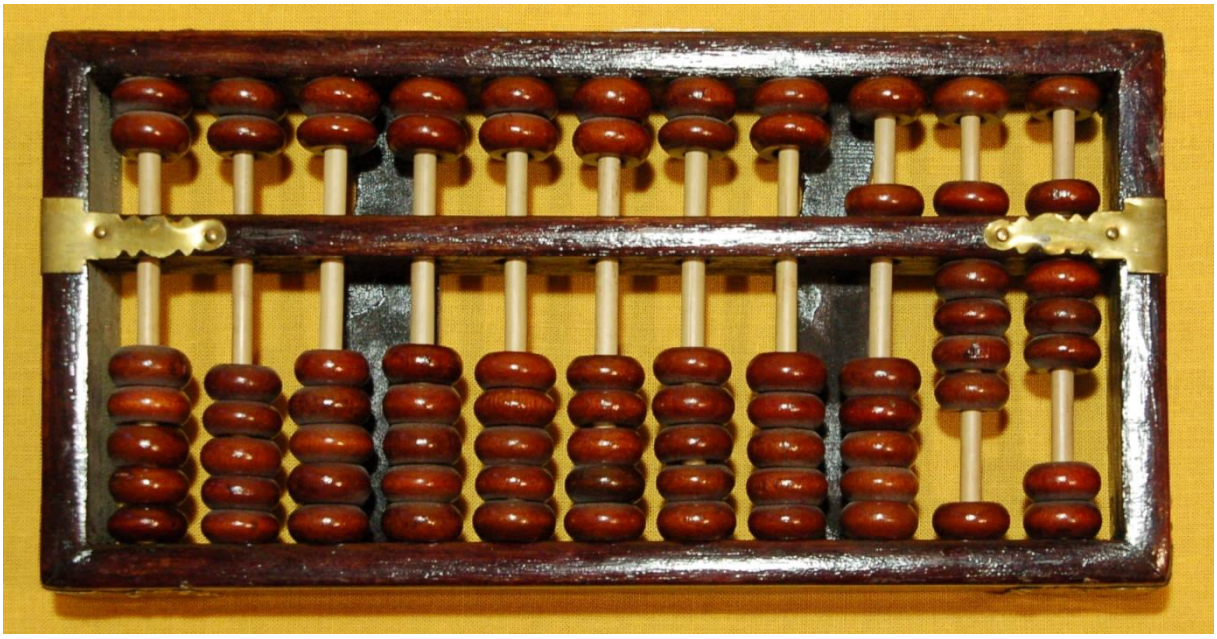

Abbildung 2: Chinesischer Abakus zeigt die Zahl 598

Für meinen Ansatz habe ich das System ein wenig abgeändert. Schließlich muss der ESP32 in dem Projekt ja nicht rechnen, sondern nur zählen. In der oberen Reihe werden 12 Stunden dargestellt, 0 bis 6 in Pink, 7 bis 11 in gelb.

Die Stufenwerte der beiden unteren Reihen sind auch nicht 10, sondern 60 wie bei den Babyloniern und entsprechen den 60 Minuten und ganz unten den Sekunden.

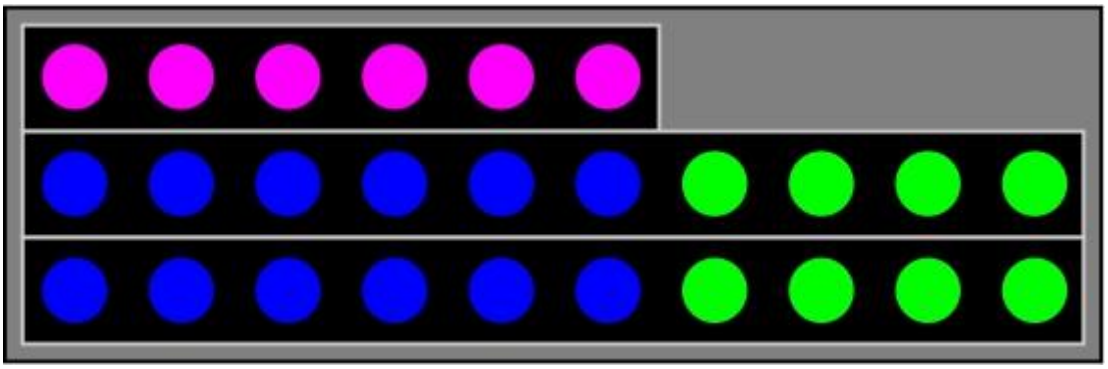

Abbildung 3: Abakusuhr_alle_LEDs

Die grünen LEDs stehen für die Einer, die blauen LEDs symbolisieren die Fünfer wie beim Abakus. Wenn wir bis 5 gezählt haben, gehen die grünen LEDs aus und dafür eine weitere blaue an. Mit diesem System kommen wir bis zur Darstellung der Zahl 34. Den Übergang zur linken Seite des Zifferblatts haben wir damit bereits überschritten, die nächste Zahl wäre 35: ein weiterer Fünfer, keine Einer. Die Reihe der Fünfer für 35, 40, 45, 50 und 55 werden durch dieselben LEDs dargestellt, jetzt aber in Rot. Durch die Verwendung von Neopixels stellt das keine große Herausforderung dar. Die Situation in Abbildung 4 zeigt also die Zeit 03:21:44.

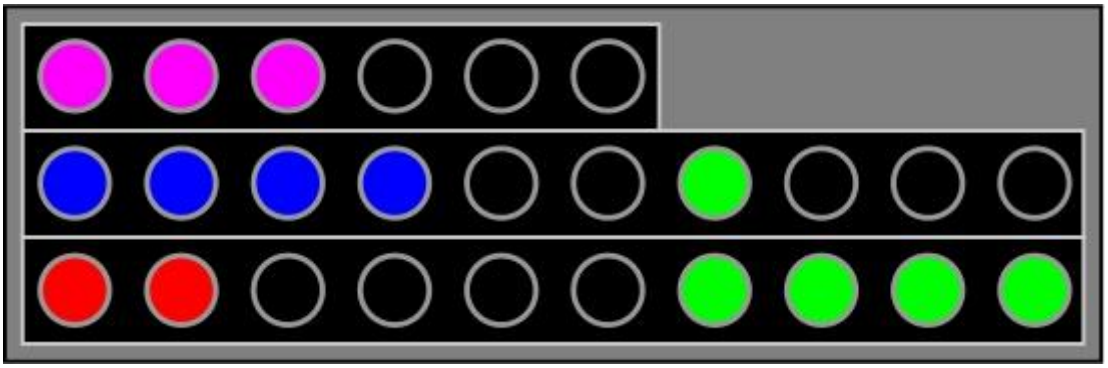

Abbildung 4: Abakusuhr_03-21-44

Die Stunden sind im 12-er System zu zweimal sechs Stunden organisiert. Bei 00:xx leuchtet keine Stunden LED, von 01:xx bis 06:xx sind sie Pink und von 07:xx bis 11:xx gelb. AM und PM könnte man leicht durch eine weitere LED symbolisieren, was ich hier aber nicht umgesetzt habe.

Nachdem das System soweit klar ist, gehen wir an die Umsetzung. Abbildung 5 zeigt den Aufbau. Am LED-Streifen lesen wir 9 Stunden (7, 8, 9, weil gelb) ab.

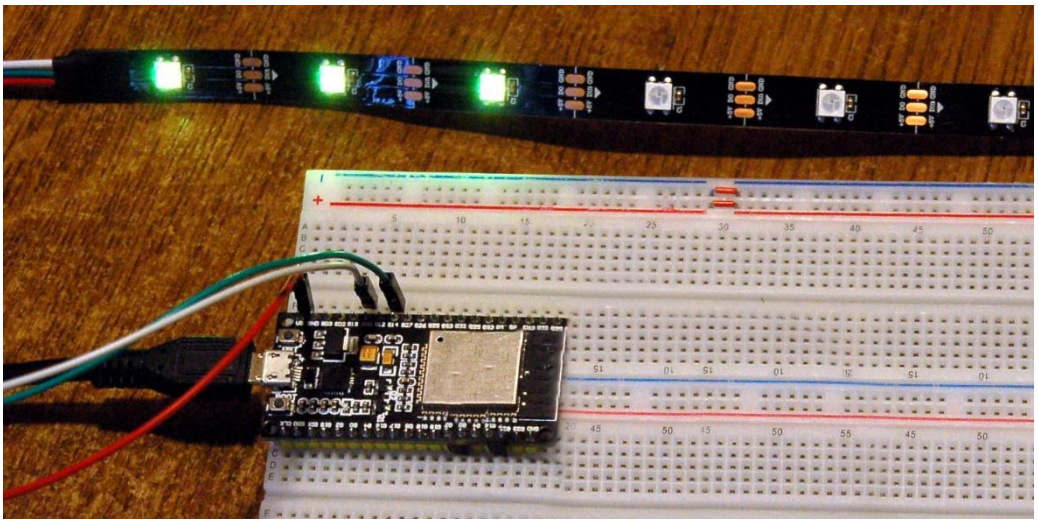

Abbildung 5: LED-Streifen-Uhr_Ausbaustufe 1

Die Hardwareliste ist nicht sehr lang.

Hardware

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
1	Oder Raspberry Pi Pico W RP2040 Mikrocontroller-Board
1	Oder D1 Mini NodeMcu with ESP8266-12F WLAN module compatible with Arduino
1	WS2812B 30 LEDs/m 30 Pixels/m LED Strip RGB adressierbarer LED Streifen mit 5050 SMD LEDs IP20 Schwarz nicht Wasserdicht
1	Optional für ESP8266 D1 mini und Raspberry Pi Pico Taste zum Beispiel KY-004 Taster Modul
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
Optional	Logic Analyzer
Optional	Charger Doctor

Für die Ausbaustufe 1 würde auch ein ESP8266 ausreichen. Im Hinblick auf die Stufe 2, in der sich ein BME280 dazugesellt, ist der Kleine aber leider zu schwach auf der Speicherbrust, weshalb bereits hier ein ESP32 eingesetzt wird. Wahlweise kann auch ein Raspberry Pi Pico (W) die Steuerung übernehmen. WLAN-fähig sollte er deshalb sein, weil wir in Stufe 2 eine Verbindung zu einem NTP-Server im Internet brauchen.

Mit einem Logic Analyzer lassen sich die Signale auf dem NeoPixel-Bus sehr schön sichtbar machen. Das ist besonders hilfreich, wenn Probleme auftreten oder wenn man wie in diesem Beitrag, das Verhalten von NeoPixel-Arrays studieren möchte.

Der Charger Doctor ermöglicht die Überwachung/Messung von Spannung und Stromstärke am USB-Bus.

Hier wären die Schaltbilder für die anderen beiden Controller-Familien.

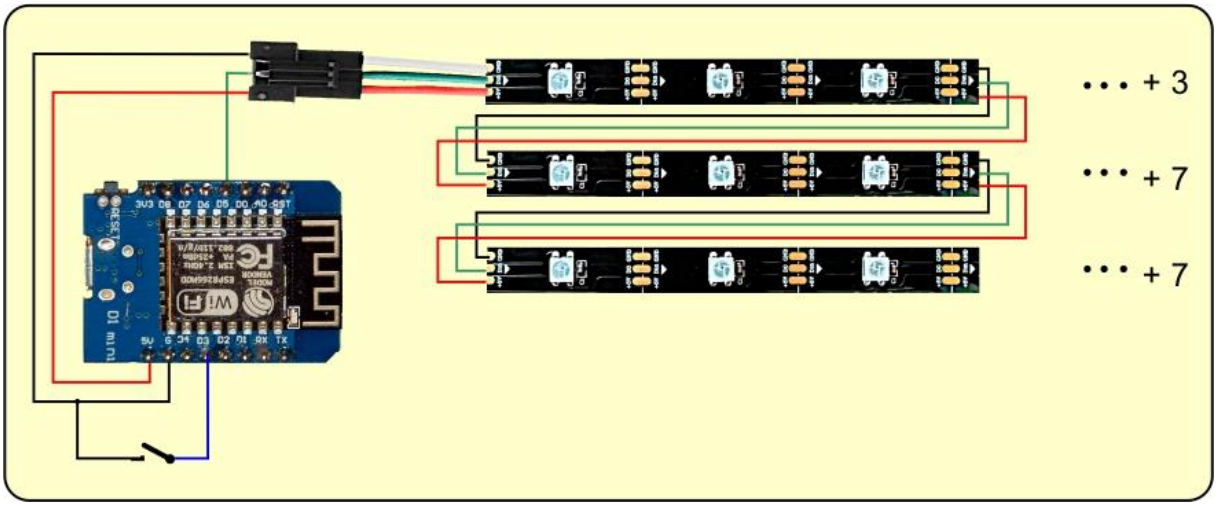

Abbildung 6: Abakusuhr mit ESP8266 D1 mini

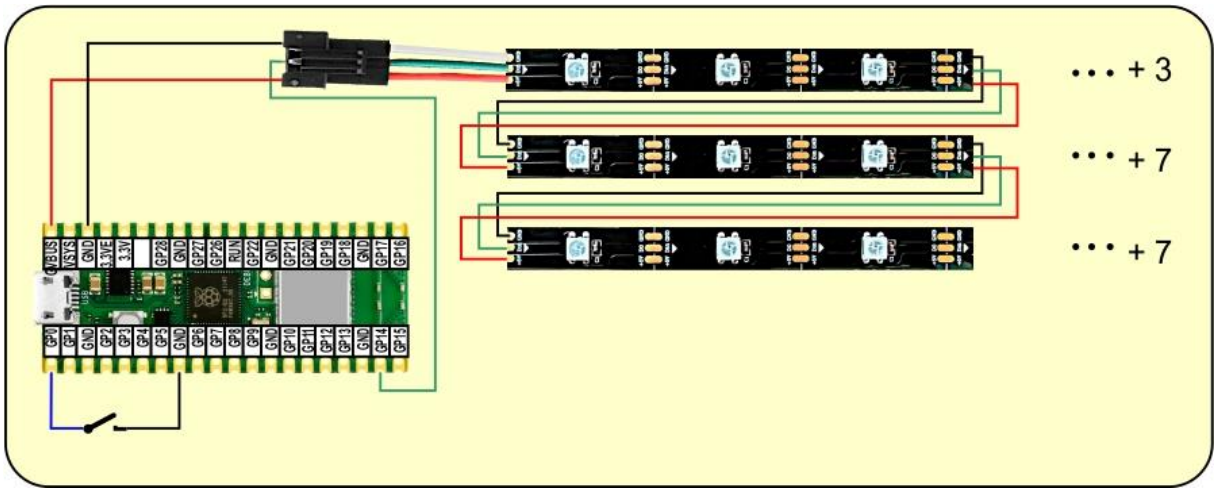

Abbildung 7. Abakusuhr mit Raspberry Py Pico W

Die Steuerleitung für die Neopixels ist in allen Fällen GPIO14, sodass diesbezüglich am Programm, das wir in Kürze besprechen werden, keine Änderungen nötig sind.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Zum Darstellen von Bussignalen

SALEAE – Logic-Analyzer-Software (64 Bit) für Windows 8, 10, 11

Verwendete Firmware für den ESP8266:

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

Verwendete Firmware für den ESP32:

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

Verwendete Firmware für den Raspberry Pi Pico (W):

RPI_PICO_W-20240602-v1.23.0.uf2

Die MicroPython-Programme zum Projekt:

uhr.py Betriebsproramm für die Uhr

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 25.01.2024) 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.

Die Zutaten

Die lokale System-Zeit

Auf dem **ESP32/ESP8266** erfolgt die Zeitrechnung in Sekunden seit Beginn der Epoche oder Ära. Das ist bei den ESPs unter MicroPython der 01.01.2000. Sie können das verifizieren, indem Sie im Terminalfenster von Thonny aus dem Modul **time** die Funktion **localtime**() mit 0 Sekunden aufrufen.

>>> import time
>>> time.localtime(0)
(2000, 1, 1, 0, 0, 0, 5, 1)

Diese Funktion gibt das Datum, 2000-01-01, und die Uhrzeit (00:00:00) zurück. Es folgen der Wochentag (5 = Samstag) und die Nummer des Tages im Jahr (1). Die Werte sind in einem Tupel (eng. Tuple) zusammengefasst. Die Reihenfolge der einzelnen Angaben ist also wie folgt. DOW steht für Day of Week, DOY für Day of the Year.

Jahr, Monat, Tag, Stunden, Minuten, Sekunden, DOW, DOY

Beim **Raspberry Pi Pico (W)** beginnt die Epoche am 01.01.1970.

>>> import time
>>> time.localtime(0)
(1970, 1, 1, 0, 0, 0, 3, 1)

Dazu noch ein Beispiel, das die Zusammenhänge verdeutlicht. Die Funktion **time**() des Moduls **time** liefert die Anzahl Sekunden seit der Epoche. Die Funktion **localtime**() wandelt diesen Wert in die Datumswerte des Tupels um.

>>> time.time()
796996679

>>> time.localtime(time.time())
(2025, 4, 3, 11, 58, 25, 3, 93)

Es ist also der 03.04.2025, 11:58:25 Uhr. Der 93. Tag im Jahr ist ein Donnerstag. Die Wochentage werden von 0 = Montag bis 6 = Sonntag durchgezählt.

Die Real Time Clock – das Modul RTC

Die Klasse **RTC** wohnt bei den ESPs wie auch beim Raspberry Pi Pico im Modul **machine**.

>>> from machine import RTC
>>> RTC
<class 'RTC'>

Sie stellt die Methode **datetime**() zur Verfügung, welche beim Aufruf ohne Parameter ebenfalls ein Tupel mit Datums- und Zeitwerten zurückgibt. Die Reihenfolge der Parameter weicht allerdings von der der Funktion **time.localtime**() ab. Der Wochentag folgt unmittelbar auf das Datum und der letzte Parameter ist nicht der Tag im Jahr, sondern liefert die Sekundenbruchteile.

>>> rtc=RTC()
>>> rtc.datetime()
*(2025, 4, 3, **3**, 12, 53, 36, **400**)*

Übergibt man der Methode **datetime**() dagegen ein Tupel dieser Form, dann wird die Angabe transparent in einen Sekunden-Timestamp umgeformt und die RTC danach gestellt. Wochentag und Sekundenbruchteile werden als 0 übergeben.

>>> rtc.datetime((2022, 12, 4, 0, 9, 6, 41, 0))
>>> rtc.datetime()
(2022, 12, 4, 6, 9, 6, 42, 295)

Um das RTC-Format in das Format des Moduls **time** umzuformen und umgekehrt, habe ich zwei Funktionen geschrieben. Die man als Modul rtc_time.py in Programme importieren kann. Die Ein- und Ausgabe erfolgt in Form des entsprechenden Tupels.

```
def time2rtc(t):
    Jahr,Monat,Tag,Stunden,Minuten,Sekunden,DOW,DOY = t
    return (Jahr,Monat,Tag,DOW,Stunden,Minuten,Sekunden,0)

def rtc2time(r):
    Jahr,Monat,Tag,DOW,Stunden,Minuten,Sekunden,ms = r
    s=mktime((Jahr,Monat,Tag,Stunden,Minuten,Sekunden,0,0))
    t=gmtime(s)
    return t
```

In **time2rtc**() wird das das Tupel **t** mit der lokalen Zeit in einzelne Variablen entpackt. Der Wochentag (DOW) wird umsortiert und als Anzahl der Millisekunden wird einfach eine 0 zurückgegeben. RTC kann mit der Tagesnummer DOY, bezogen auf das Jahr, nix anfangen, also landet DOY im Nirwana. Die Anzahl Millisekunden im RTC-

Tupel kann nicht aus dem Hut gezaubert werden und wird daher also einfach auf 0 gesetzt.

rtc2time() ist etwas aufwändiger. Das RTC-Tupel wird ebenfalls zuerst zerpflückt. Dann bauen wir aus den Daten ein Tupel der lokalen Zeit zusammen, wobei wir den Wochentag und den Tag im Jahr auf 0 setzen. Dieses Tupel übergeben wir der Methode **mktime**() aus dem Modul **time**, die daraus die Sekunden der Epoche berechnet. Die Methode **gmtime**() ermittelt dann daraus wieder ein time-Tupel, in dem jetzt aber Wochentag und Tag im Jahr korrekt erscheinen.

Wird das Modul als Hauptprogramm gestartet, dann sorgt der nachfolgende Teil für die Demonstration der Anwendung. Im Attribut __**name**__ steht dann der String **"__main__"**, was zur Ausführung des Codes im if-Block führt.

```
if __name__ == "__main__":
    from machine import RTC
    from time import localtime
    r=RTC()
    zeit = localtime()
    print("lt",zeit)
    rt=time2rtc(zeit)
    print("rt",rt)
    tm=rtc2time(rt)
    print("LT",tm)
```

Wir importieren das Modul **RTC** und instanziieren ein Objekt **r**. Die lokale Zeit wird geholt. Das Tupel landet in **zeit**, wird ausgegeben und an die Methode **time2rtc**() weitergereicht, die ein RTC-Tupel im RTC-Format zurückgibt. Auch das lassen wir in REPL ausgeben bevor wir es an **rtc2time**() weiterreichen. Die Methode ergänzt die zwischenzeitlich verloren gegangenen Werte für DOW und DOY in dem time-Tupel, welches zurückkommt.

lt (2025, 4, 3, 13, 22, 56, 3, 93)
rt (2025, 4, 3, 3, 13, 22, 56, 0)
LT (2025, 4, 3, 13, 22, 56, 3, 93)

Für den Sekundentakt unserer Uhr verwenden wir einen Timer. Der soll der Hauptschleife sagen, wann es Zeit ist, eine neue Konstellation an den LED-Streifen zu senden. Hier stoßen wir auf den ersten Unterschied zwischen den drei Controller-Familien. Der ESP32 verfügt über vier Hardwaretimer, die unabhängig von der Software arbeiten. Beim ESP8266 können quasi beliebig viele Software-Timer generiert werden, die auf der Basis des Betriebssystems RTOS arbeiten. Ähnliches gilt für den Raspberry Pi Pico, dessen beliebig viele Software-Timer auf einen gemeinsamen Hardwaretimer zurückgreifen.

In allen Fällen ist das Modul **Timer** aus dem Modul **machine** zu importieren. Während bei den ESP-Familien der Konstruktor der Timer-Klasse mit einer Nummer aufgerufen wird, führt das beim Raspberry Pi Pico zu einer Fehlermeldung. Hier ist keine Nummer anzugeben. Wir berücksichtigen diesen Umstand, indem wir durch Vergleich mit der Variablen **platform** die Art des Controllers feststellen.

Der Neopixelstreifen

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 30 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(14,Pin.OUT) # D5
>>> neo=NeoPixel(np,30)
>>> 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\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.

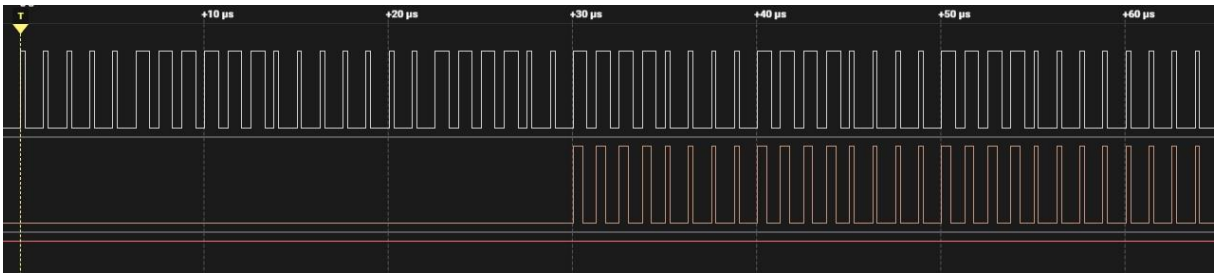

Abbildung 1: Pulsfolge für RGB = 0xE0, 0x07, 0x3C

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 Uhr zuwenden.

Das Programm uhr.py

Wir besprechen jetzt die einzelnen Teile des Programms **uhr.py**. Wir brauchen, wie schon angesprochen, die Module **Pin**, **RTC** und Timer sowie **NeoPixel**. Die Funktion **exit**() erlaubt uns das geordnete Verlassen des Programms vor allem während der Entwicklung.

```
from machine import Pin,RTC,Timer
from neopixel import NeoPixel
from time import sleep_ms
from sys import exit, platform
```

Dann stellen wir die Controller-Familie fest und richten entsprechend den Timer ein.

```
if platform == "esp32":
    t0=Timer(0)
elif platform == "esp8266":
    t0=Timer(0)
elif platform == "rp2":
    t0=Timer()
else:
    print("Nicht unterstützter Controller")
    exit()
```

Weitere Objekte werden eingerichtet. Der GPIO14-Anschluss für die Neopixel-Steuerleitung wird als Ausgang programmiert. Um wirklich alle LEDs während der Entwicklung ansteuern zu können erzeugen wir ein NeoPixel-Objekt mit 30 LEDs. Dadurch wird im Speicher des Controllers ein Puffer mit 30 mal 3 Bytes für rot, grün und blau angelegt, den wir später entsprechend befüllen und den Inhalt dann in einem Rutsch an den Streifen ausgeben können. Ferner legen wir ein RTC-Objekt und den GPIO0-Eingang für den Anschluss der Taste an.

```
np=Pin(14, Pin.OUT)
n=NeoPixel(np,30)
```

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

Als Nächstes schaffen wir die Rahmenbedingungen für die Bearbeitung der Zeitinformationen. Die Variable **dt** nimmt ein RTC-Tupel auf. Die Indizes auf die Stunden- Minuten- und Sekunden-Positionen werden gesetzt und die Farbcodes festgelegt. Das Flag **ticked** bekommt bei abgelaufenem Timer **t0** den Wert True, bis dahin bleibt es auf False.

```
dt=rtc.datetime()
hor=const(4)
mnt=const(5)
sec=const(6)
high=(0x40,0,0)
low=(0,0,0x40)
lowH=(0x40,0,0x40)
highH=(0x40,0x40,0)
single=(0,0x40,0)
ticked=False
```

tick() ist die Callback-Routine, die beim Ablauf des Timers **t0** aufgerufen wird und die das Flag **ticked** auf True setzt. Damit die Änderung nach dem Beenden der ISR (Interrupt Service Routine) im Hauptprogramm verfügbar wird, muss **ticked** global deklariert werden. Da eine ISR so kurz wie möglich gehalten werden soll, wird das Updaten des LED-Streifens dem Hauptprogramm überlassen. Im Parameter t wird vom Betriebssystem die Nummer des Timers an die Routine übergeben. Der Parameter ist daher zwingend anzugeben, auch wenn wir keinen Gebrauch davon machen.

```
def tick(t):
    global ticked
    ticked = True
```

Eine weitere Funktion **ledsOff**() wird deklariert, sie löscht **nbr** LEDs des Streifens. Dazu werden die RGB-Tupel jeder LED auf (0,0,0) gebracht. **write**() schiebt den Puffer zum Streifen, wenn die Funktion mit **immediate = True** aufgerufen wird. Wird das optionale Argument weggelassen, dann erhält **immediate** den Defaultwert **None**, welcher bei der if-Abfrage als **False** gewertet wird.

```
def ledsOff(nbr,immediate=None):
    for i in range(nbr):
        n[i]=(0,0,0)
    if immediate:
        n.write()
```

Bevor es in die Hauptschleife geht, löschen wir alle 30 LEDs mit sofortiger Wirkung und initiieren den Timer mit der Ablaufzeit von 1 Sekunde = 1000 ms für den Dauerlauf. Der Timer ruft beim Ablauf die Funktion **tick**() auf und startet erneut.

```
ledsOff(30, True)
t0.init(period=1000,mode=Timer.PERIODIC,callback=tick)
while 1:
```

Die Hauptschleife läuft ständig durch und hat zwei Aufgaben zu erfüllen, das Update der LEDs und die Prüfung, ob die Taste gedrückt ist. In letzterem Fall sind die LEDs auszuschalten und die Schleife muss verlassen werden. Würde das Programm durch STRG+C verlassen, dann bleiben in der Regel die LEDs an, weil der Ausstieg an einer zufälligen Stelle im Programm passiert. Genau das verhindern wir mit der Taste. Vor dem Programmende werden die LEDs ausgeschaltet.

Was passiert, wenn die Hauptschleife einen Timerinterrupt über das Flag **ticked** feststellt?

```
    if ticked:
        ticked = False
        dt=rtc.datetime()
```

Dann muss **ticked** auf False gesetzt werden und wir müssen die RTC auslesen. Der aktuelle Timestamp wird in der Variablen **dt** gespeichert.

Wir kümmern uns dann um die Anzahl der Tagesstunden, die wir erst einmal auf 12 Stunden reduzieren, Tagesstunden modulo 12 bestimmt den 12-er-Teilungsrest des 24-Stundenwerts, der von der RTC geliefert wird. Tag und Nacht können wir ja wohl noch selbst auseinanderhalten.

```
        hour=dt[hor] % 12
        sColor = lowH if hour <= 6 else highH
        ledsOff(30)
```

Dann geht es um die Feststellung der ersten oder zweiten 6-Stündigen Runde, welche die LED-Farbe der Darstellung festlegt. Für jede Runde löschen wir die LEDs des Streifens, um die Konfiguration neu aufzubauen.

Wir beginnen mit den Stunden. Für 00:xx darf keine der Stunden-LEDs leuchten. Von 01:00 bis 06:59 müssen die LEDs an Position 0 bis 5 in der Farbe **lowH** für den unteren Bereich aufleuchten, für 07:00 bis 11:59 in der Farbe **highH** für den oberen Stundenbereich. Weil die Position einer einzelnen LED schwer zu orten ist, werden die LEDs bis zur gewünschten Position angemacht. Zur vierten Stunde leuchten also 4 LEDs in pink (1, 2, 3, 4) und während der neunten Stunde drei LEDs in gelb (7, 8, 9). Diesen Job erledigen die for-Schleifen. Die Stunden-LEDs haben die Wertigkeit 1. Bei den gelben LEDs muss sechs addiert werden.

```
        if 1 <= hour <= 6:
            for i in range(1,hour+1):
                n[i-1] = sColor
        elif 7 <= hour <= 11:
            for i in range(7,hour+1):
                n[i-7] = sColor
```

Minuten und Sekunden sind im Abakus-Schema codiert. Blaue und rote LEDs haben die Wertigkeit 5, die Grünen haben die Wertigkeit 1. Blau codiert die Werte 5, 10, 15, 20, 25, 30, Rot die Werte 35, 40, 45, 50, 55. Die grünen LEDs geben die Minutenwerte modulo 5 an, zählen also von 1 bis 4. Beim Erreichen des

Zählerstands 5 gehen alle 4 grünen LEDs aus und dafür eine blaue oder rote an. Das entspricht dem Hin- und Herschieben der Perlen auf dem Abakus beim Hochzählen.

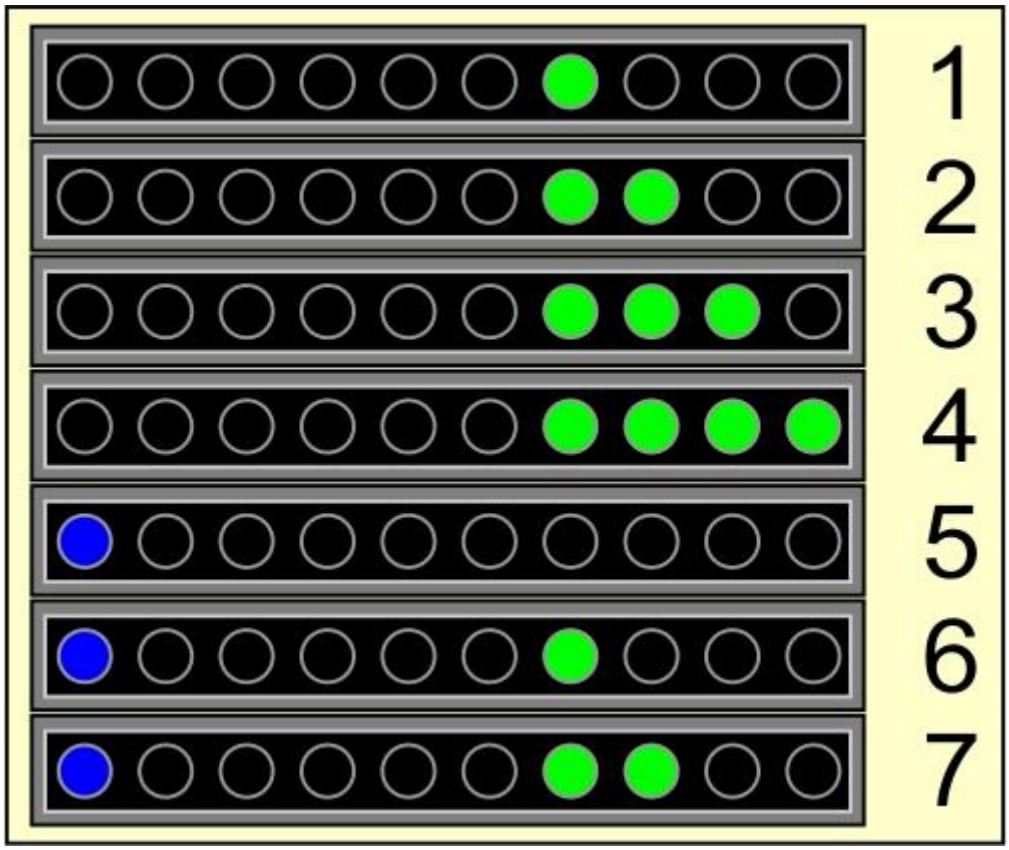

Abbildung 8: Zählweise beim Abakus

Die folgende Programmsequenz modelliert dieses Verhalten. Vom RTC-Tupel holen wir den Minutenwert. Die Ganzzahldivision liefert die Anzahl der 5-Minuten-Pakete, während der ganzzahlige Teilungsrest modulo 5 die Einzelminuten 1 bis 4 liefert. Über die Anzahl der 5-Minuten-Pakete stellen wir die Leuchtfarbe ein, kleiner oder gleich 6 stellt Blau ein, sonst Rot. Die Minutenausgabe erfolgt ab der Pixelposition 6 mit den 5-er-Gruppen, Die Einzelminuten folgen ab Position 12.

```
        FifeMin=dt[mnt] // 5
        Minutes=dt[mnt] % 5
        sColor = low if FifeMin <= 6 else high
        if FifeMin <= 6:
            for i in range(FifeMin % 7):
                n[i+6]=sColor
        else:
            for i in range(FifeMin - 6):
                n[i+6]=sColor
        for i in range(1,Minutes+1):
            n[i+5+6]=single
```

In analoger Weise funktioniert die Ausgabe der Sekunden, die ab Position 16 folgt.

```
        FifeSec=dt[sec] // 5
        Seconds=dt[sec] % 5
        sColor = low if FifeSec <= 6 else high
```

```
        if FifeSec <= 6:
            for i in range(FifeSec % 7):
                n[i+16]=sColor
        else:
            for i in range(FifeSec - 6):
                n[i+16]=sColor
        for i in range(1,Seconds+1):
            n[i+5+16]=single
```

Damit der Puffer, den wir bisher gelöscht und dann befüllt haben, auch an den LED-Strang geschickt wird, müssen wir das Kommando **n.write**() geben. Zur Kontrolle lassen wir uns die Zeit in REPL anzeigen. Diese Anweisung können wir im Produktionsbetrieb getrost streichen.

```
        n.write()
        print(dt[hor],dt[mnt],dt[sec])
```

Damit sind wir fast fertig, fehlt nur noch die Tastenabfrage.

```
    if taste() == 0:
        ledsOff(30,True)
        t0.deinit()
        break
```

Wird die Taste gedrückt, zieht der Kontakt den Eingang GPIO0 auf GND-Potenzial. Ist die Taste nicht gedrückt, dann zieht der interne Pullup-Widerstand den Eingang auf +3,3V. Beim ESP32 brauchen wir keine extra Taste, da verwenden wir die Flash-Taste, die an GPIO0 liegt. Wir schalten dann alle LEDs umgehend aus, deaktivieren den Timer und verlassen mit break die while-Schleife.

Wenn alles angeschlossen und das Programm uhr.py eingetippt ist, kann der erste Test starten. Ich habe zu diesem Zweck den LED-Streifen noch in einem Stück belassen. Weil die Stunden- und die 5-Minuten-LEDs bei 06:xx direkt aneinander liegen, habe ich für die Stunden andere Farben gewählt. Wenn die geteilten Streifen so wie in Abbildung 3 oder 4 übereinander liegen, kann man natürlich auch für die Stunden blau und rot im Programm einstellen.

Die LED-Helligkeit ist bei rot, grün und blau mit 0x40 auf 25% reduziert. Das führt zu einer Stromstärke von bis zu 130 mA. Ein Charger Doctor- Modul war zur Bestimmung hilfreich. Bei Volllast zeigt das Messgerät 270mA an. Das lässt nun den Schluss zu, dass die Helligkeit nicht direktproportional den Einstellwerten folgt.

Ausblick

Die Auswahl an Sensoren für die drei Microcontroller lässt der Phantasie freien Lauf. Die Leuchtstärke der LEDs ließe sich zum Beispiel mit Hilfe eines LDR (Light Dependend Resistor = Photowiderstand) automatisch der Umgebungshelligkeit anpassen. Ebenso geeignet wäre ein Helligkeitssensor vom Typ **GY-302 BH1750**, der über den I2C-Bus an den Controller angeschlossen werden kann.

Wir werden in der nächsten Folge einen BME280 mit ins Boot holen, um damit Temperatur, Luftdruck und relative Luftfeuchte zu messen. Ferner werden wir die Uhr in bestimmten Abständen über das Internet mit einem NTP-Server synchronisieren. Das ist empfehlenswert, weil die RTC auf den Controllern nicht gerade das gelbe vom Ei darstellt, was die Ganggenauigkeit angeht.

Abbildung 9: Strommessung am USB mit dem Charger Doctor