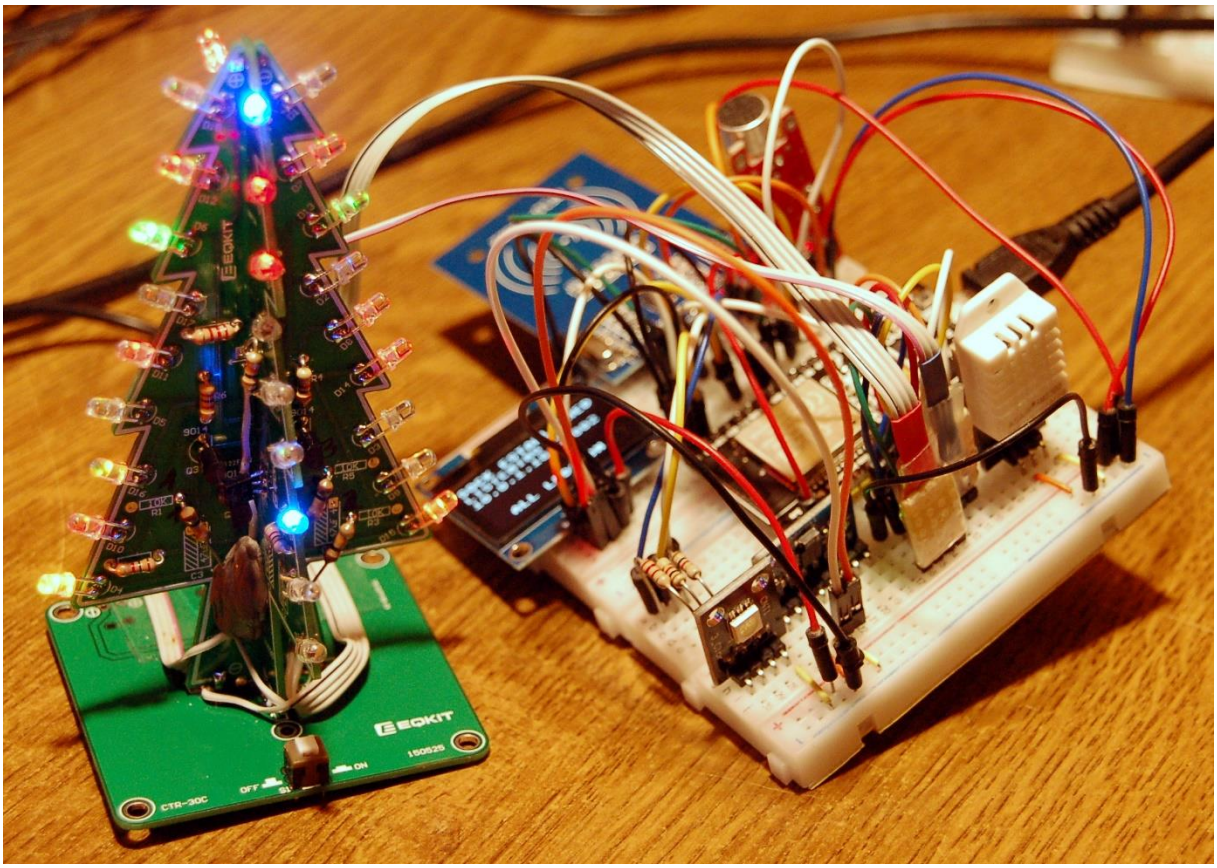

Abbildung 1: Die Baum-Geschichten

Diesen Beitrag gibt es auch als PDF in Deutsch.
This episode is also availlable as PDF-document in english.

Geschichten beginnen meistens mit "Es war einmal". Die folgenden Geschichten spielen heute und jetzt. Heute werden wir nämlich ein paar Projekte um ein Christbäumchen der besonderen Art entwickeln.

Gemeint ist ein Bäumchen von ca. 11,5 cm Höhe mit 36 (37) bunten Lichtern dran. Die blinken in allen Farben lustig durcheinander. Aber die tun das mehr oder weniger immer in derselben Art und Weise. Das brachte mich auf die Idee, durch ein paar zusätzliche Bauteile etwas Ordnung in das Chaos zu bringen. Herausgekommen ist eine Steuerung mit einem ESP32 und diversen Sensoren, welche die Illumination der LEDs in den Dienst verschiedener Messaufgaben stellen oder einfach nur für Verwunderung sorgen. Einen Teil dessen, was insgesamt möglich wäre, stelle ich Ihnen in den folgenden Kapiteln einzeln vor. Programmiert werden alle Anwendungen natürlich in MicroPython. Damit willkommen zu den

MicroPython-Geschichten um einen kleinen Weihnachtsbaum

Der Inhalt:

1. **Die Teileliste**
2. **Die Software**
3. **Wir bauen das Bäumchen und verkabeln es**
4. **Gezielte Illumination**
5. **Das OLED-Display für Klartext-Informationen**
6. **Gestuftes Leuchten**
7. **Das verzauberte Bäumchen**
8. **Wer macht denn so einen Lärm?**
9. **Dem Bäumchenklau auf der Spur**
10. **Angenehmen Aufenthalt wünschen ESP32 und DHT22/DHT11**
11. **Die etwas andere Weihnachtsverlosung**
12. **Die Weihnachtsbaum-App**

Die Teile-Liste

1	DIY LED Weihnachtsbaum Kit
1	KY-009 RGB LED SMD Modul Sensor oder KY-016 FZ0455 3-Farben RGB LED Modul 3 Color
1	1,3 Zoll OLED I2C 128 x 64 Pixel Display kompatibel mit Arduino und Raspberry Pi
1	DHT22 AM2302 Temperatursensor und Luftfeuchtigkeitssensor
1	KY-021 Magnet Schalter Mini Magnet Reed Modul Sensor
1	RFID Keycard Card 13,56MHz Schlüsselkarte Karte MF S50 (13,56 MHz) – 10x RFID Karte
1	RFID Kit RC522 mit Reader, Chip und Card für Raspberry Pi und Co. (13,56MHz)
1	Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins kompatibel mit Arduino und Raspberry Pi
1	KY-038 Klangerfassungsmodul Mikrofon Voice- Ton Sensor
1	ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102
1	GY-521 MPU-6050 3-Achsen-Gyroskop und Beschleunigungssensor alternativ Rüttelkontakt KY-020 oder KY-002 (*)
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
3	Widerstand 1,0kΩ

(*) Der Einsatz der Rüttelkontakte macht eine abweichende Programmierung erforderlich.

2. Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender zum Testen des ESP32/ESP8266 als UDP-Server
Browser: Opera oder Chrome

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

MIT-Lizenz-Text
Gerätetreiber:
gy521rc.py
mfrc522.py
sh1106.py
oled.py

Projektdateien:
movementalarm.py
noisy.py
rfid.py
roomclimate.py
steigerung.py
verzaubert.py
webcontrol.py

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung, wie die MicropythonFirmware 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, bevor 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 compilieren 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 Macro 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 compiliertes 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.

3. Wir bauen das Bäumchen und verkabeln es

Für den Zusammenbau des Bäumchens gibt es ein Video. Natürlich kann man sich nach dem Zusammenbau des Bäumchens entspannt zurücklehnen und sein Werk bewundern. Zusätzlichen Spaß bereitet es, wenn wir den Aufbauvorgang an einigen Stellen abändern. An drei Stellen im Video müssen wir für unser Projekt anders verfahren. Die dort genannten 4,7kΩ Widerstände sind im Teilepäckchen solche mit 10kΩ. Und diese drei je Platine A und B werden nur an den Stellen auf der Platine verlötet, wie es in den Abbildungen Abb.2 und Abb.3 gezeigt ist, das andere Ende dieser Widerstände **bleibt vorerst frei**. An diese freien Enden löten wir später dünne Kabelchen (zum Beispiel Flachband) für die Verbindung zum ESP32. Das gilt für beide Platinen, A und B. Die **Elektrolytkondensatoren bleiben ganz weg**.

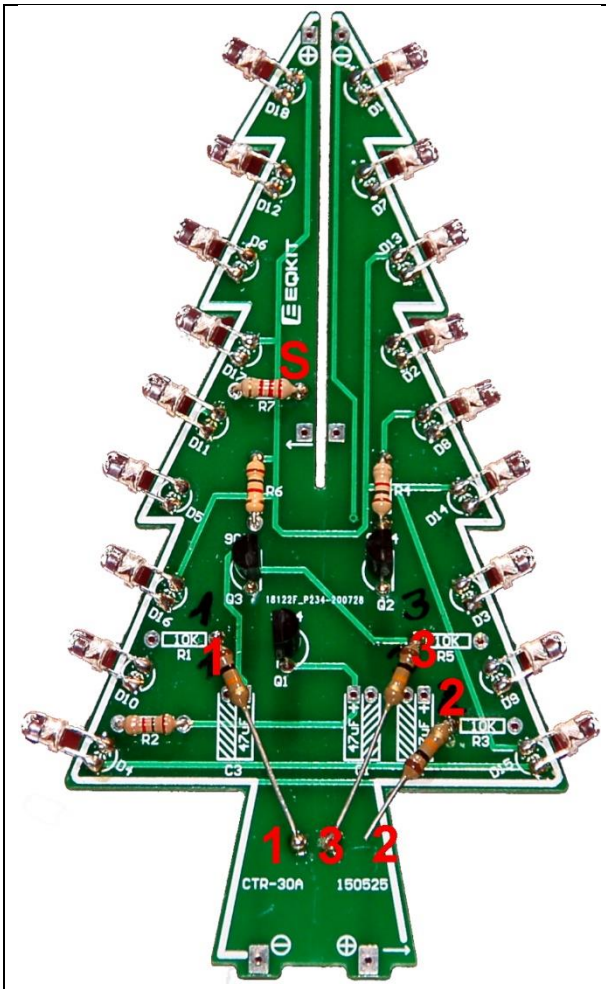

Abbildung 2: Teil_A

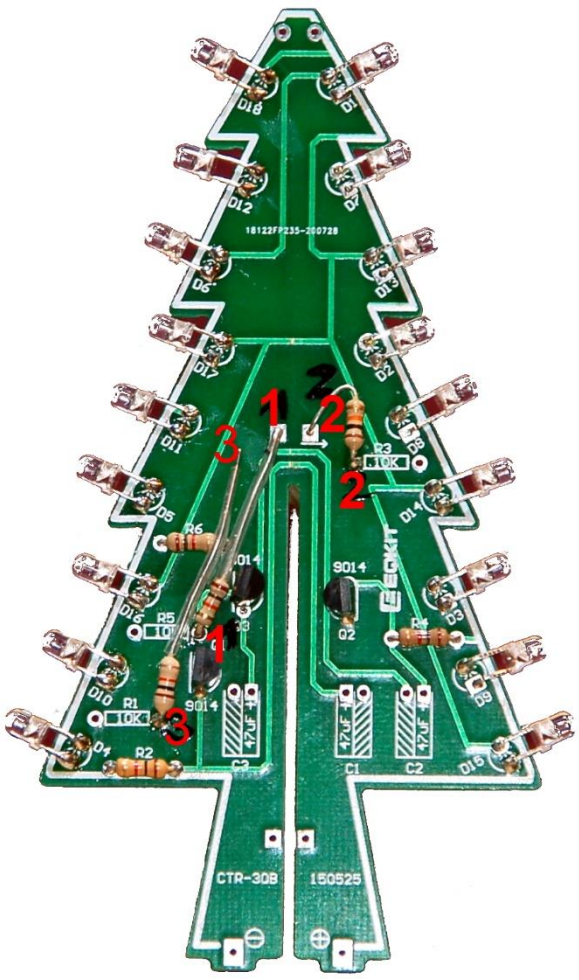

Abbildung 3: Teil_B

Der restliche Aufbau kann genau nach der Videovorlage erfolgen. Wenn auch die Bodenplatte dran ist, löten wir die Kabelchen an die freien Enden der 10kΩ Widerstände. Die Länge sollte so zwischen 25 und 30 cm betragen. Ans andere Kabelende löten wir, damit die Sache steckbar wird, jeweils ein Stück Stiftleiste.

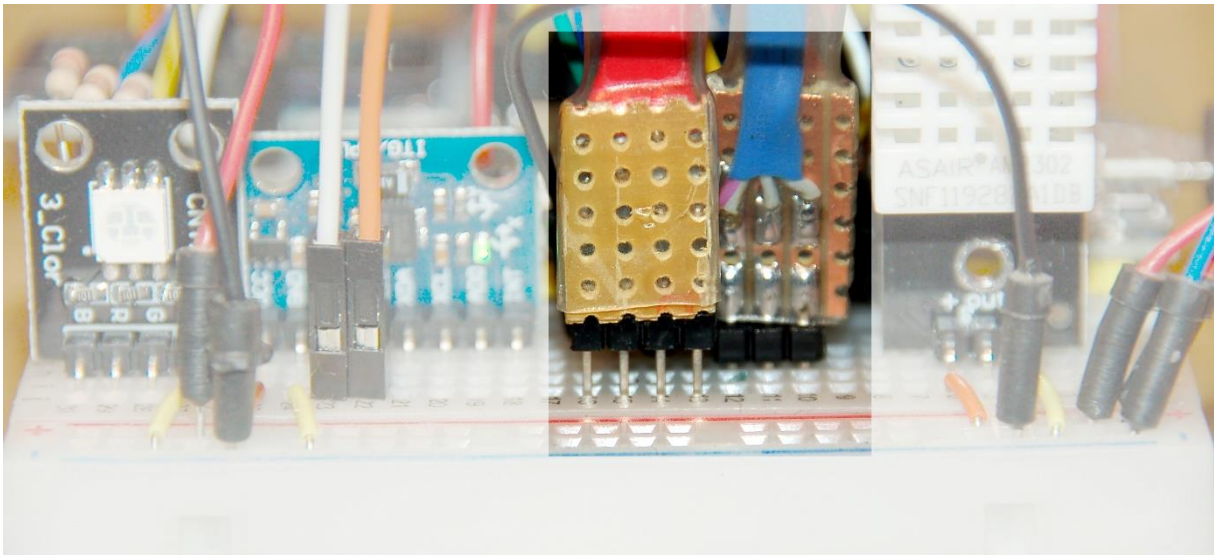

Abbildung 4: The Tree-Connection

Die Zuordnung der Anschlüsse am Bäumchen zu den GPIOs des ESP32 ist aus der Tabelle 1 ersichtlich. Der Index bezieht sich auf die Liste der Pin-Objekte. Diese Liste mit dem Namen **schicht** dient der Adressierung der LED-Schichten durch Schleifen, wie wir noch sehen werden. Die Anschlüsse sind so verteilt, dass auf einen geradzahligen Index immer die gleichlagige LED-Schicht auf der Platine B folgt. Natürlich sind beliebige Umbesetzungen jederzeit möglich.

Bäumchen	A1	B1	A2	B2	A3	B3
GPIO	32	26	33	27	25	12
Index	0	1	2	3	4	5

Tabelle 1: Verbindungen zwischen Bäumchen und ESP32

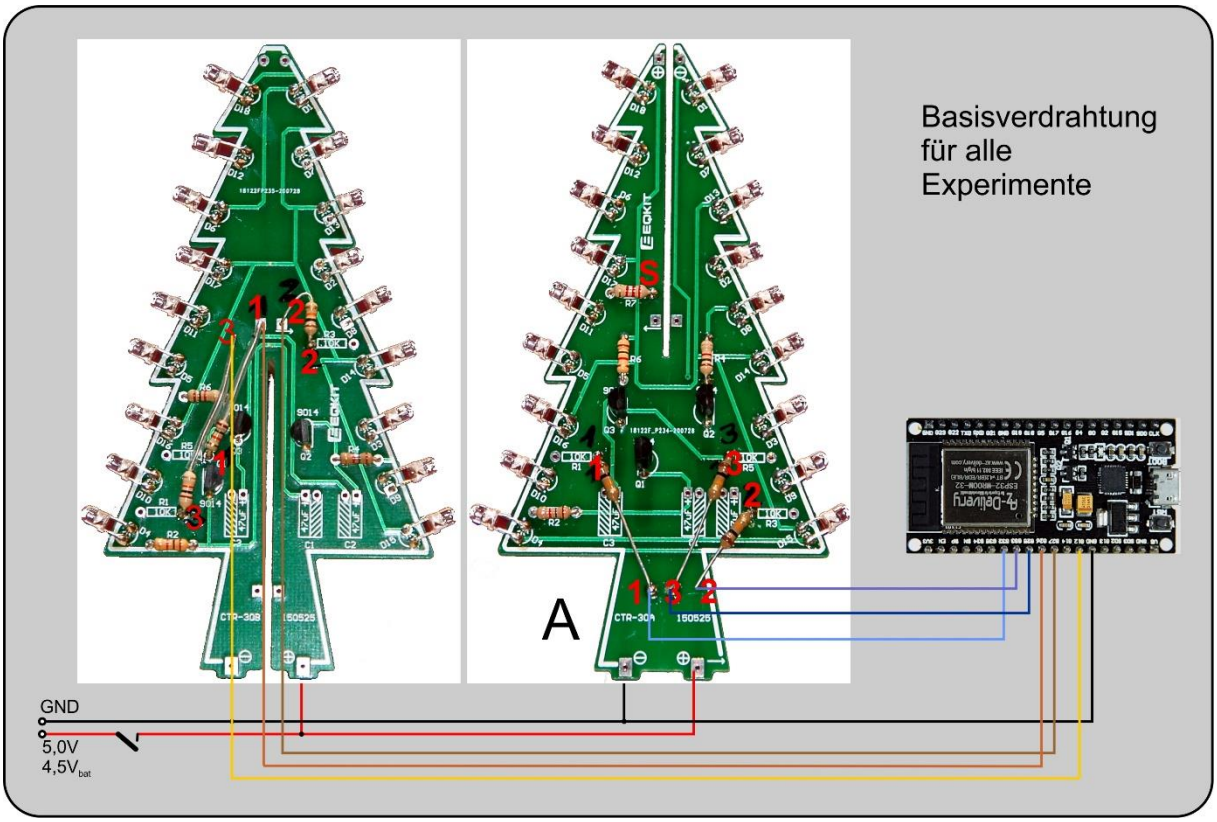

Abbildung 5: Basisverdrahtung

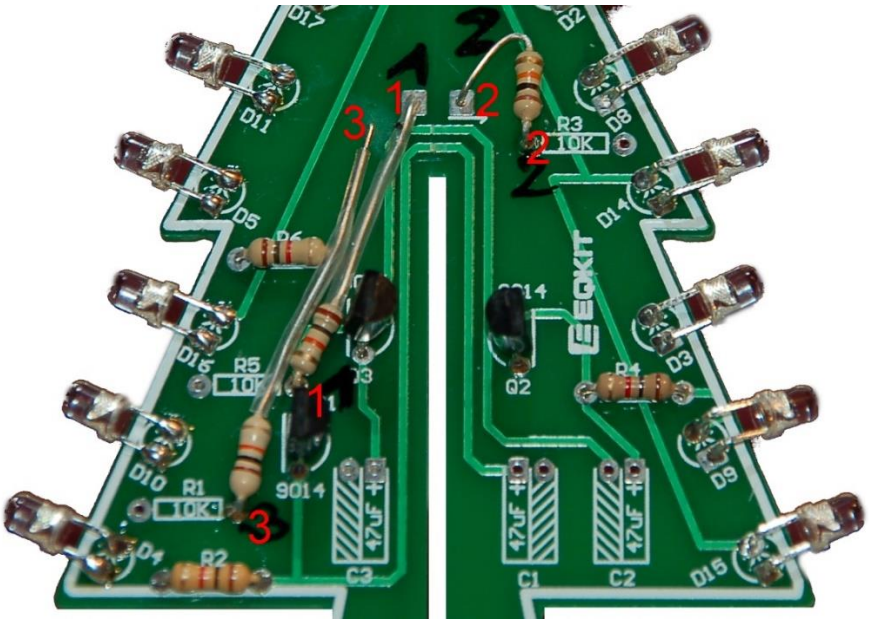

Abbildung 6: Basiswiderstände an Teil B – Detail, freie Enden liegen oben

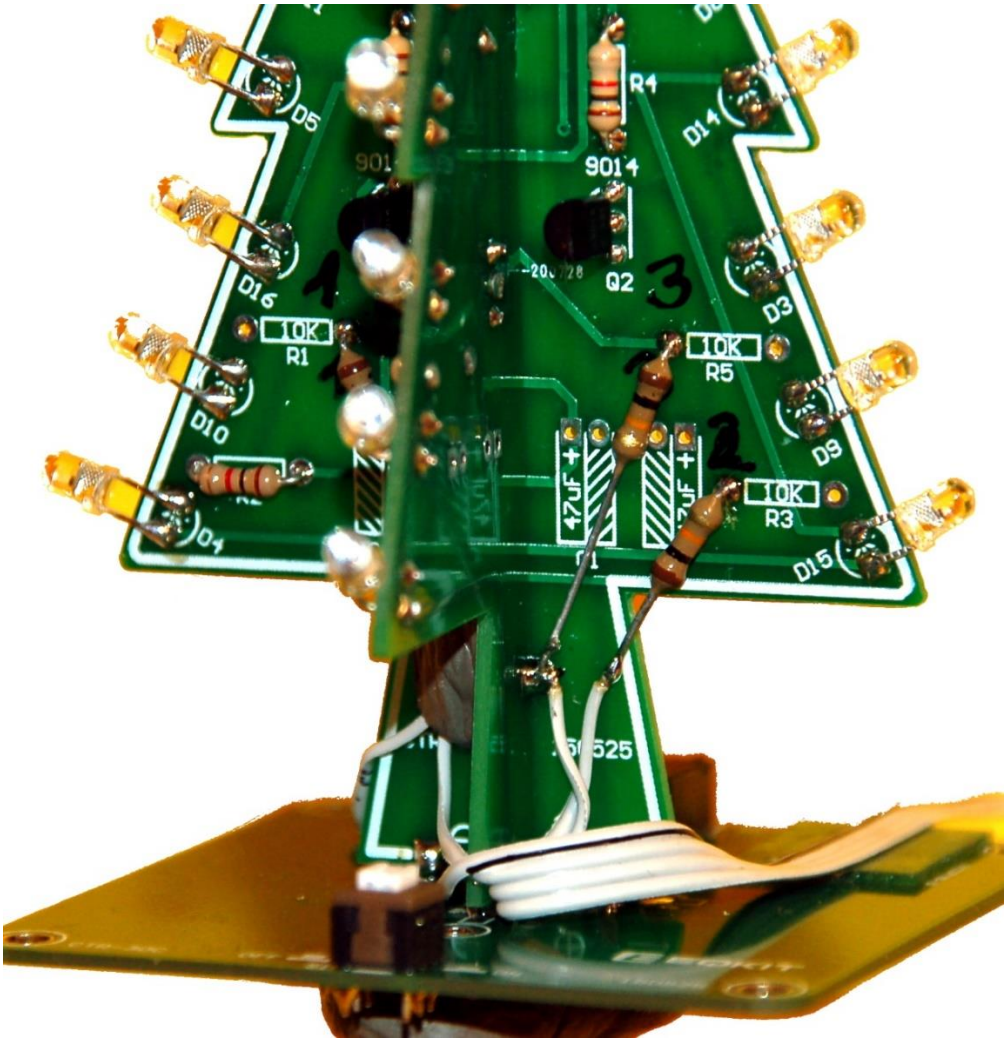

Abbildung 7: Verkabelt an Teil A

4. Gezielte Illumination

Die Verkabelung ist fertiggestellt? Dann wollen wir die LEDs am Bäumchen schon einmal entfachen. Wir versorgen das Bäumchen entweder über die Batterien oder über das mitgelieferte Kabel von einem USB-Anschluss. Nach dem Einschalten bleibt es dunkel. Klar, denn die Basisanschlüsse der Transtoren liegen frei, somit kann kein Basisstrom fließen und weil dann auch kein Kollektorstrom fließt, bleiben die LEDs dunkel.

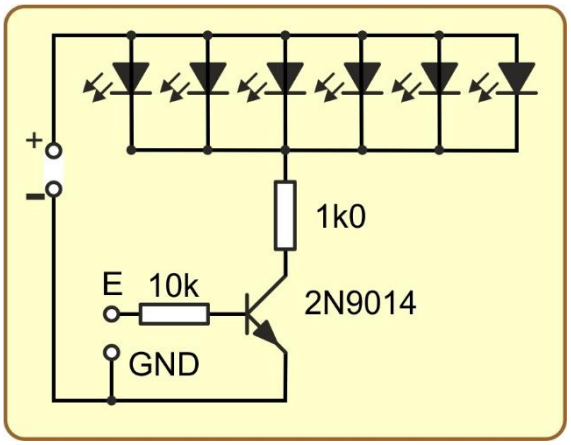

Abbildung 8: Eine von 6 Transistorstufen

Das ändert sich, wenn die GPIOs als Ausgänge programmiert werden und der Pegel von GND-Potential auf 3,3V angehoben wird. Wir erreichen das, indem wir eine 1 als Wert zuweisen. Im Terminal von Thonny geben wir folgende Zeilen ein.

```
>>> from machine import Pin
>>> a1=Pin(32,Pin.OUT,value=0)
>>> a1.value(1)
```

Bei korrekter Verdrahtung fangen nun die LEDs der Ebene A1 zu leuchten an, um nach der Eingabe von

```
>>> a1.value(0)
```

zu erlöschen. Im Gegensatz zur Vorgängerversion des Bäumchen-Kits ist die neue Version mit Flacker-LEDs ausgestattet. Vorher waren es einfache bunte LEDs. Das hat einen gewissen Nachteil, weil ein Dimmen der "Flashing LEDs" nicht mehr möglich ist. Trotzdem macht es Spaß damit zu experimentieren. Über die sechs Transistoren sind wir jetzt in der Lage alle 6 Ebenen genau nach unseren Wünschen zu starten oder abzuschalten. Damit beeinflussen wir auch die Gesamthelligkeit.

Die Anordnung Der Lichter zeigt die Abb. 9. Sie gilt sowohl für den Teil A als auch für den Teil B.

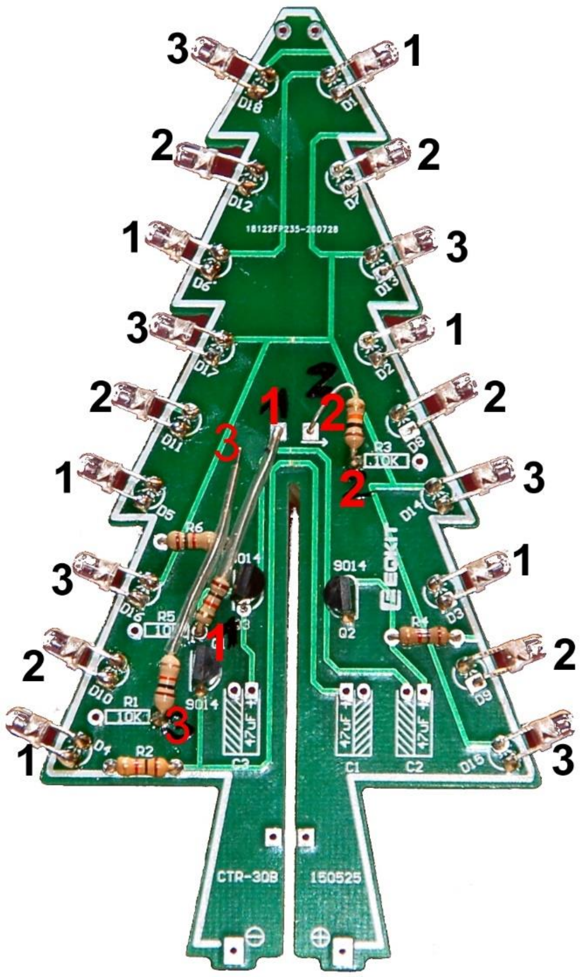

Abbildung 9: Verteilung der LEDs – erfolgt reihum

Die Anordnung der LEDs, die Verdrahtung zum und die Anschlüsse am ESP32 werden für alle weiteren Versuche als gegeben vorausgesetzt. Sie tauchen daher in den Beschreibungen und Schaltbildern der Teilschaltungen nicht mehr explizit auf.

5. Das OLED-Display

Das OLED-Display kann uns mit Klartextinformationen versorgen, es kann aber auch Grafiken in schwarz-weiß darstellen. Die Programmierung ist einfach, wenn wir die zugehörigen MicroPython-Software-Module verwenden. Der Hardwaretreiber **SH1106** ist direkt für das 1,3"-Display zuständig und auch nur hierfür zu gebrauchen. Das in MicroPython-Kern integrierte Modul **framebuf** stellt einfache Grafik- und Textbefehle zur Verfügung und das Modul **oled**.py gibt uns komfortable Befehle für die Textausgabe an die Hand.

Die Ansteuerung des Displays erfolgt nur über die beiden Leitungen des I2C-Busses. Wir erzeugen ein I2C-Objekt und übergeben es an den Konstruktor der Klasse **OLED**. Die Hardware-Geräteadresse des Displays ist fest vorgegeben und in OLED als Konstante verankert. Dennoch schauen wir zunächst nach, was auf dem Bus so alles vorhanden ist. Dann löschen wir den Bildschirm und geben ein paar Zeilen aus.

Die Grafik in Abb. 10 und das folgende Programm demonstrieren die Handhabung. Das Programm geben wir im Thonny-Editor ein, speichern es ab und starten es dann mit der Funktionstaste F5.

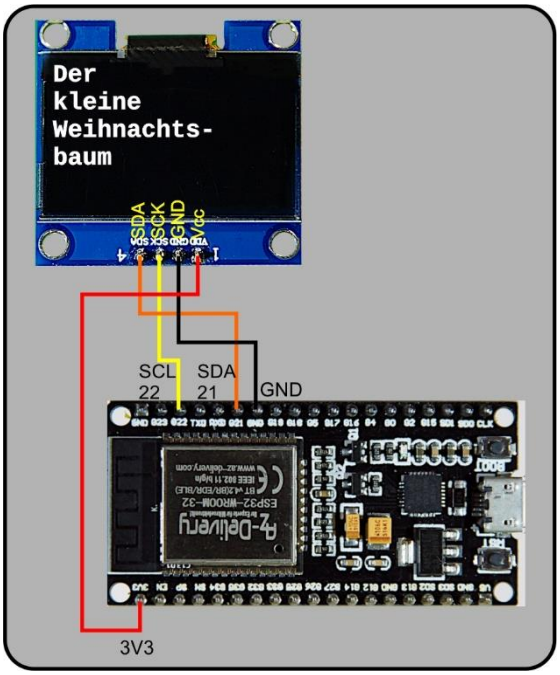

Abbildung 10: Das OLED am ESP32

oledtest.py

```
# OLED-Display-Demo
#
from machine import Pin, I2C
from time import sleep
from oled import OLED

# Initialisieren der Schnittstelle **********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
print(i2c.scan())
d=OLED(i2c)

d.clearAll()
d.writeAt("Der",0,0,False)
d.writeAt("kleine",0,1,False)
d.writeAt("Weihnachts-",0,2,False)
d.writeAt("baum",0,3)
sleep(4)
d.clearFT(0,2,15,2,False)
d.writeAt("Christ-",0,2)
```

Ausgabe:
[60]
this is the constructor of OLED class
Size:128x64

Die Geräteadresse des Displays ist dezimal 60 oder 0x3C hexadezimal. Der Konstruktor der OLED-Klasse weiß auch, dass das Display 128 x 64 Pixel besitzt.

Nach der Ausgabe der vier Zeilen wird 4 Sekunden später "Weihnachts-" durch "Chist-" ersetzt. Zuvor müssen wir diese Zeile natürlich löschen. Probieren Sie die einzelnen Befehle ruhig auch einzeln über REPL, die Terminal-Console von Thonny aus.

6. Gestuftes Leuchten

Da wir die einzelnen Ebenen der LEDs am Bäumchen getrennt ansteuern können, nutzen wir das doch gleich einmal aus, um von der untersten Stufe – AUS – bis zur maximalen Helligkeit hoch und wieder herunter zu fahren. An der Schaltung brauchen wir nichts zu ändern. Das Display informiert uns über die gerade aktive Stufe.

steigerung.py

```
# steigerung.py
#
import sys
from machine import Pin, I2C
from oled import OLED
from time import sleep, sleep_ms

# Initialisieren der Schnittstellen **********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

# LED-Schichten einrichten *******************************
#schichtPin = [32,33,25,27,26,12]  # sortiert
schichtPin = [32,26,33,27,25,12]   # verteilt
schicht=[0]*6
for i in range(6): # Ausgaenge erzeugen und auf 0
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)

# Funktionen defnieren ***********************************
def  switch(n,val):  # Ebene n ein-/ausschalten
    schicht[n].value(val)

def stop():  # alle LED-Ebenen aus
    d.writeAt("GOOD BYE",4,3)
    for i in range(6):
        switch(i,0)

def alle():  # alle LED-Ebenen ein
    for i in range(6):
        sleep_ms(300)
        switch(i,1)

# Hauptprogramm ******************************************
d.clearAll()
d.rect(4,16,123,40,1)  # Rechteck in Pixelwerten
for j in range(3):
```

```
    for i in range(6):
        d.writeAt("Ebene: {} ein".format(i),2,3)
        switch(i,1)
        sleep_ms(3000)
    for i in range(5,-1,-1):
        d.writeAt("Ebene: {} aus".format(i),2,3)
        switch(i,0)
        sleep_ms(3000)
d.clearFT(2,3,14,3,False)
stop()
```

Die Reihenfolge der Ebenen legen wir in der Liste **schichtPin** fest. Nach diesem Muster werden in der folgenden for-Schleife die Pinobjekte erzeugt. Die Funktionen **switch**(), **stop**() uns **alle**() helfen uns, das Programm übersichtlicher zu gestalten. Außerdem werden wir sie in den folgenden Kapiteln mehrfach einsetzen.

Im Hauptprogramm löschen wir den Bildschirm und zeichnen einen Rahmen. 4 und 16 sind die Pixel-Koordinaten der linken oberen Ecke, 123 und 40 die Breite und Höhe in Pixeln und 1 die Farbe weiß, mehr Farben gibt's nicht. Die äußere for-Schleife zählt die Gesamtdurchgänge. Die erste innere for-Schleife zählt i in Intervallen von 3 Sekunden hoch und schaltet die Ebenen ein. Die zweite Schleife zählt rückwärts und löscht die LEDs wieder.

Die letzte Ausgabe wird entfernt, und die Funktion stop() löscht zuverlässig alle LEDs und verabschiedet sich mit einem freundlichen "GOOD BYE".

Über die Intervalllänge und die Anzahl der Durchläufe können wir das Verhalten der LEDs voll selber vorgeben.

7. Das verzauberte Bäumchen

Da könnte ja jeder daherkommen und unser Bäumchen einschalten wollen. Aber nix da, das geht nur durch unsere Zauberhände. Wir sagen natürlich nicht, dass wir in jeder Hand einen kleinen Neodym-Magnetstab verborgen haben. Wozu wir den brauchen? Zum "Zaubern" eben. Denn wir haben unsere Schaltung inzwischen umgebaut. Am GPIO-Pin 13 ist jetzt ein Reed-Kontakt gegen Masse angeschlossen. In dem Glasröhrchen befindet sich ein Schaltkontakt, der beim Annähern eines Magneten schließt.

Achtung:
Das Glas ist sehr spröde und die Drähte sind sehr steif. Nicht dran herumbiegen, sonst splittert das Glas und man kann das Bauteil begraben.

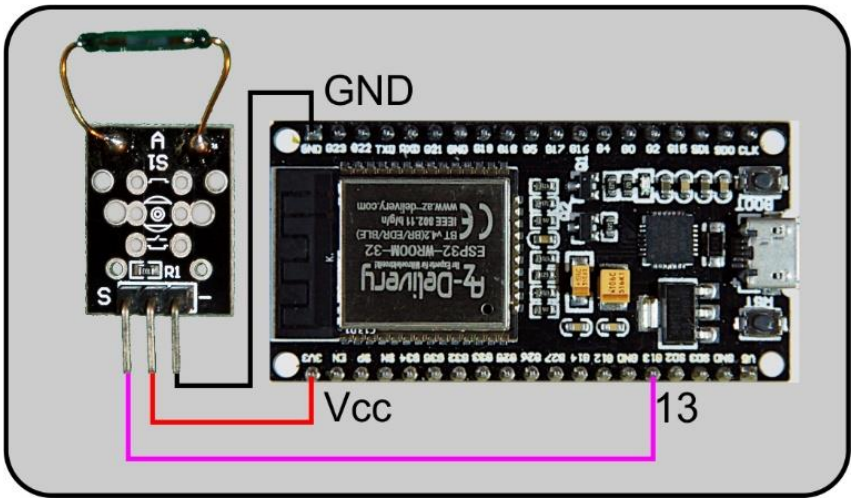

Abbildung 11: Reedkontakt hilft zaubern

Etwas anderes dürfen wir nicht vergessen, nämlich, dass OLED und Bäumchen wie oben beschrieben angeschlossen bleiben.

verzaubert.py

```
from os import uname
import sys
from machine import  Pin, I2C
from oled import OLED
from time import sleep_ms

# Initialisieren der Schnittstellen ***********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

taste=Pin(0,Pin.IN,Pin.PULL_UP)
reed=Pin(13,Pin.IN,Pin.PULL_UP)

# LED-Schichten einrichten ********************************
schichtPin = [32,33,25,26,27,12]
schicht=[0]*6
for i in range(6):
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)

# Funktionen defnieren ************************************
def  switch(n,val):
    schicht[n].value(val)

def stop():
    d.writeAt("   MUGGLE     ",1,3)
    for i in range(6):
        switch(i,0)

def alle():
    d.writeAt("  DUMBLEDOR   ",1,3)
    for i in range(6):
```

```
        sleep_ms(300)
        switch(i,1)

# Hauptprogramm ****************************************
d.clearAll()
d.writeAt("Kannst du ...",0,0)
d.writeAt("ZAUBERN???",0,1)
while 1:
    if reed()==0:
        alle()
    else:
        stop()
```

Der Reed-Kontakt muss so angebracht werden, dass wir mit unserem Magneten gut in seine Nähe kommen, wenn wir Bäumchen oder Breadboard auf die Handfläche legen. Ein dünner schwarzer Handschuh hilft uns dabei den Magneten unsichtbar zu halten. Wahrscheinlich sind alle in Ihrer Umgebung Muggles.

Das Programm ist sehr einfach. Bis zum Hauptprogramm kenn wir schon alles. Die while-Schleife läuft endlos bis zum Stromabschalten. Ist der Kontakt in der Nachbarschaft des Magneten geschlossen, dann liegt GPIO13 auf GND-Potenzial und alle Lichter gehen an. Andernfalls zieht der im Modul eingebaute Widerstand den GPIO13 auf Vcc=3,3V und die Lichter gehen aus.

Damit der Zauber besser gelingt, sollte das Bäumchen mit Breadboard von der Batterie betrieben werden. Der Plus-Anschluss der Batterie muss dazu mit dem Pin Vin / 5V des ESP32 verbunden werden. Ferner muss das Programm dann als boot.py auf dem ESP32 hochgeladen werden, damit der Controller nach dem Einschalten autonom durchstartet. Wie das geht ist in Kapitel 2 – Autostart genau beschrieben.

8. Advent und Weihnachten, die "staade" Zeit.

"Staad" in den allgemeinen deutschen Sprachgebrauch übersetzt heißt so viel wie "ruhig", "besinnlich". Der Alltag lehrt uns aber, dass es auch in der Vorweihnachtszeit schon mal heftig her gehen kann. Wenn es dann zu turbulent wird, mahnt das Bäumchen dazu, ein paar Dezibel zurückzuschalten. Wie es das macht? Nun es gibt ein Soundmodul, welches Schall aufnimmt und das digitalisierte Signal am ESP32 abliefert.

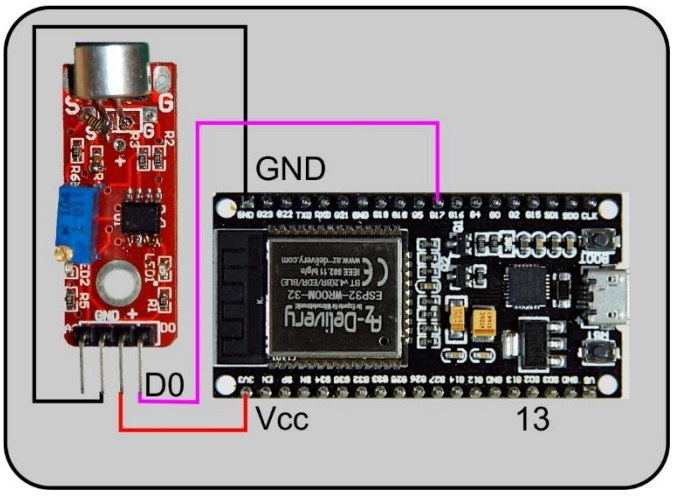

Abbildung 12: Soundmachine am ESP32

noisy.py

```
# noisy.py
import esp32
from os import uname
from machine import Timer, Pin, I2C
from oled import OLED
from time import time, sleep,


i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

# IRQ-Steuerung durch Soundmodul
ST=Timer(1)
sound=Pin(17,Pin.IN)
#
# LED-Schichten einrichten ********************************
schichtPin = [32,33,25,27,26,12]
L=len(schichtPin)
schicht=[0]*L
for i in range(L):
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)

# Funktionen defnieren ***********************************
def  switch(n,val):
    schicht[n].value(val)

def stop():
    d.clearAll()
    d.writeAt("ALLES GUT",4,2)
    for i in range(L):
        switch(i,0)

def alle():
    d.clearAll()
    d.writeAt("ZU LAUT!!",0,0)
    for i in range(L):
        sleep(0.5)
        switch(i,1)

def soundDetected(pin):
    global n
    if pin==Pin(17):
        sound.irq(handler=None)
        if n:
            return
        n=True
        ST.init(period=15000, mode=Timer.ONE_SHOT,\
                callback=soundDone)
        print("begin",time())
        alle()
```

```
def soundDone(t):
    global n
    n=False
    print("ende",time())
    stop()
    sound.irq(handler=soundDetected, trigger=Pin.IRQ_FALLING)

# Hauptprogramm ************************************
n=False
sound.irq(handler=soundDetected, trigger=Pin.IRQ_FALLING)
```

Schall wird durch schnelle Druckschwankungen der Luft übertragen. Mit ca. 340 m/s breitet sich ein Schallsignal aus. In einem Zimmer praktisch ohne wahrnehmbare Verzögerung. Das Mikrofon im Soundmodul setzt die Druckschwankungen in ein elektrisches Signal um. Anders als beim Reedkontakt kann man diese Schwingungen aber nicht mehr durch Abfragen des GPIO-Ports erkennen, dieses Verfahren ist zu langsam. Deshalb verwenden wir hier eine andere Technik, die Interrupt-Programmierung. Ein Interrupt ist die Unterbrechung eines Programms durch ein bestimmtes Ereignis. Wir werden zwei verschiedene Unterbrechungsquellen benutzen. Die eine löst eine Unterbrechung aus, wenn sich der Pegel an einem GPIO-Pin ändert, von 0 auf 1 oder umgekehrt. Die andere IRQ-Quelle ist ein Hardwaretimer des ESP32. Er triggert den IRQ, wenn der Wecker schellt.

Beide spielen sich nun abwechselnd den Ball zu. Der GPIO17 wartet auf ein Signal vom Soundmodul. Trifft eine fallende Flanke ein, startet die Funktion **soundDetected**() und prüft als Erstes, ob sie aufgrund des übergebenen Parameters pin gemeint ist.. Ist **n True**, dann ist bereits ein Zyklus am Laufen, und es gibt nichts weiter zu tun. Ist **n** dagegen **False**, dann ist es ein frischer Auftrag. Der Pin-Change-IRQ wird abgeschaltet, und **n** wird auf **True** gesetzt, um unmittelbar nachfolgende Impulse an GPIO17 zu unterdrücken. Dann wird der Timer gestartet, der die Laufzeit der Baumbeleuchtung vorgibt. Die Beleuchtung wird durch Aufruf von **alle**() eingeschaltet.

Ist der Timer abgelaufen, wird der zugehörige Interrupt ausgelöst, der die Funktion **soundDone**() startet. **n** wird auf **False** gesetzt, die Lichter gehen aus, und der Pin-Change-IRQ wird erneut scharf geschaltet.

Das Hauptprogramm besteht grade mal aus zwei Zeilen. **n** wird auf **False** gesetzt, damit der danach aktivierte Pin-Change-IRQ ausgelöst werden kann.

Das Interessante daran ist, dass die IRQs weiterhin aktiv sind, auch wenn das Hauptprogramm bereits beendet ist. Um das abzuschalten muss der ESP32 mit dem STOP/RESTART-Button zurückgesetzt werden.

9. Dem Bäumchenklau auf der Spur

Da soll es doch Leute geben, die ihren Weihnachtsbaum klauen – aus dem Wald. Nun, liebe Förster, macht es doch wie wir, und baut in eure Bäumchen auch so einen Wächter ein wie den, den wir gleich beschreiben.

OK, das wird zugegebenermaßen sicher ebenso schwierig, wie bei der Überwachung anderer Verbote, wenn das Personal fehlt. Wozu verbietet man dann überhaupt etwas, wenn man's nicht kontrollieren kann? Sei's drum.

Unser Bäumchen bekommt einen Überwacher – nämlich sich selbst! Dabei hilft ihm ein Sensor, der aus einer ganz anderen Ecke kommt. Das eingesetzte GY-521-Modul mit dem Baustein MPU6050 ist ein Accelerometer mit Gyroskop. Damit kann man Beschleunigungen, Kräfte und Rotationen messen. Ja und wenn man etwas wegnehmen möchte, muss man es anheben und in Bewegung versetzten. In beiden Fällen beschleunigt man den Gegenstand. Bereits ganz geringe Ortsänderungen bringen Kräfte hervor und damit unseren Sensor zum Ansprechen. Der Rest ist einfach, auf die Auslösung folgt die Beleuchtung des Bäumchens und der potentielle Dieb sucht hoffentlich das Weite. Für die Dauer des Alarms ist übrigens wieder ein Timer-Interrupt zuständig.

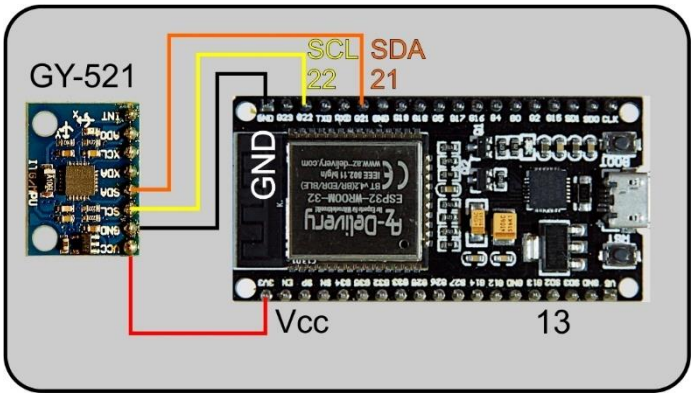

Abbildung 13: Antiklau-Einheit

alarm.py

```
# alarm.py
# RED-ALLERT by movement
import esp32
from os import uname
from machine import Timer, Pin, I2C
from oled import OLED
from time import sleep,ticks_ms, ticks_us, sleep_ms
from gy521rc import GY521

# Initialisieren der Schnittstellen **********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

AT=Timer(0)
acc=GY521(i2c)
limit=36
dauer=5000
```

```
schichtPin = [32,33,25,27,26,12]
L=len(schichtPin)
schicht=[0]*L
for i in range(L):
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)

# Funktionen defnieren **********************************
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare

def  switch(n,val):
    schicht[n].value(val)

def stop():
    d.clearAll()
    d.writeAt("ALLES GUT",4,2)
    for i in range(L):
        switch(i,0)

def alle():
    d.clearAll()
    d.writeAt("DIEBSTAHL",0,0)
    for i in range(L):
        sleep(0.5)
        switch(i,1)

def hasMoved(delay):
    xs,ys,zs=0,0,0
    for i in range(100):
        x,y,z=acc.getXYZ()
        xs+=x
        ys+=y
        zs+=z
    xs//=100
    ys//=100
    zs//=100
    #print(xs,ys,zs)
    n=0
    while 1:
        x,y,z=acc.getXYZ()
        x=abs(xs-x)
        y=abs(ys-y)
        z=abs(zs-z)
        #print(x,xs//limit)
        if x > abs(xs//limit) :
            print("*******",n)
            n+=1
            alle()
```

```
            # Optional Nachricht via UDP
            AT.init(period=delay, mode=Timer.ONE_SHOT,\
                    callback=alertDone)
        sleep(0.3)

def alertDone(t):
    stop()

print("Diebstahlschutz gestartet")
hasMoved(dauer)
```

Wieder gibt es gute Bekannte in dem Programm. Neu ist aber die Initialisierung des **GY521**. Für den Baustein müssen wir ein weiteres Modul zum ESP32 hochladen, **gy521rc**.py. Die darin enthaltene Klasse heißt genauso wie der Baustein.

Wie das OLED-Display wird auch der GY521 über den I2C-Bus bedient. Wir übergeben dem Konstruktor dasselbe I2C-Objekt, legen die Schwelle für das Auslösen des Alarms und dessen Dauer in Millisekunden fest.

Die Schwelle ist der absolute Betrag der Abweichung des Messwerts vom Mittelwert der Beschleunigungsmessung in x-Richtung. Der Sensor ist so ausgerichtet, dass die positive x-Achse senkrecht nach oben zeigt. Der Messwert liegt um die 16000 Counts und entspricht in diesem Fall der Erdbeschleunigung g=9,81m/s².

Die Funktion **hasMoved**() stellt hier die Hauptschleife dar. Beim Eintritt wird durch 100 Messungen der Mittelwert bestimmt. Klar, dass sich dabei der Sensor nicht bewegen darf.

Dann geht's in die Hauptschleife. die aktuelle Beschleunigung wird gemessen und die Abweichungen zu den Mittelwerten berechnet. Überschreitet die Differenz das vorgegebene Limit, wird Alarm ausgelöst und der Timer aktiviert. Alarm heißt, das Bäumchen geht auf volle Helligkeit.

Die Service-Routine des Timer-IRQs löscht die Lichter. Die Lösung über den IRQ sorgt dafür, dass sofort nach Alarmauslösung die Schaltung wieder scharf ist. Würde die Alarmdauer durch einen sleep-Befehl in der Hauptschleife vorgegeben, dann wäre die Schaltung für diese Dauer tot.

Die in der Teileliste genannten Rüttelkontakte würden ähnlich wie der Reedkontakt am ESP32 angeschlossen, erlauben aber keine Einstellung der Sensitivität.

10. Angenehmen Aufenthalt wünschen ESP32 und DHT22

Zur Festtagsstimmung gehört ein angenehmes Raumklima. Nun kann der ESP32 in dieser einfachen Anwendung das Raumklima nicht ändern, doch er kann darüber berichten. Die genauen Werte für Temperatur und Luftfeuchte werden auf dem OLED-Display angezeigt, die groben Werte sagt uns das Bäumchen. In 2-Gradstufen berichtet es durch verschieden viele eingeschaltete LED-Ebenen die Werte der Raumtemperatur.

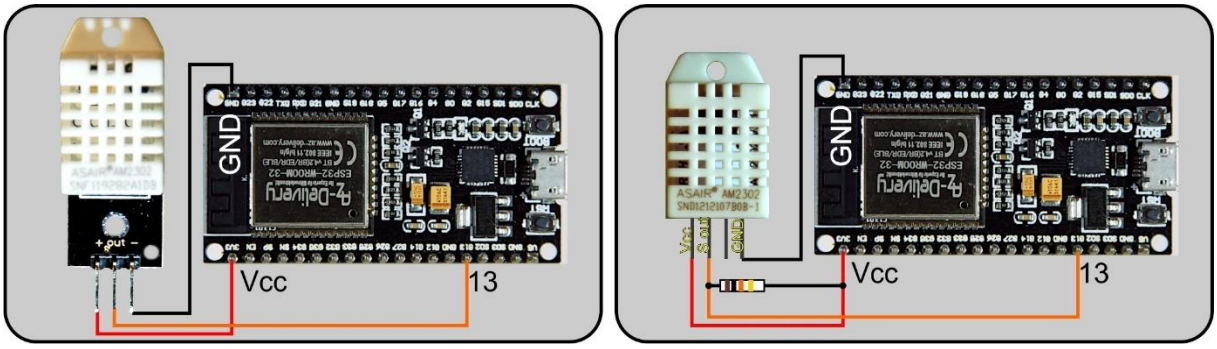

Abbildung 14: Temperatur- und Luftfeuchtemessung in einem Modul

Von DHT22, alias AM2302, gibt es 2 Varianten. Das Modul in der linken Abbildung enthält bereits den nötigen Pullupwiderstand für den One-Wire-Bus, der übrigens nicht mit dem System des Dallas- Bausteins DS18B20 verwechselt werden darf. Der Dallas-Bus hat ein ganz anderes Timing. Für die nackte Version in der rechten Abbildung muss ein 4,7kΩ bis 10kΩ Widerstand gegen Vcc eingebaut werden.

Die Bedienung im Programm ist denkbar einfach. Die drei nötigen Befehle stellt das in MicroPython bereits integrierte Modul **dht** zur Verfügung.

roomclimate.py

```
# roomclimate.py
import esp32, dht
from os import uname
import sys
from machine import Pin, I2C
from oled import OLED
from time import sleep

# Initialisieren der Schnittstellen **********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

dhtPin=Pin(13)
dht22=dht.DHT22(dhtPin)

# LED-Schichten einrichten *******************************
schichtPin = [32,33,25,26,27,12]
schicht=[0]*6
for i in range(6):
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)
```

```
# Funktionen defnieren **********************************
def  switch(n,val):
    schicht[n].value(val)

def stop():
    d.writeAt("TEMP TO LOW",4,2)
    for i in range(6):
        switch(i,0)

def alle():
    for i in range(6):
        sleep_ms(300)
        switch(i,1)

def tree(n):
    for i in range(6):
        if i <=n:
            switch(i,1)
        else:
            switch(i,0)
# Hauptprogramm *****************************************
d.clearAll()
d.writeAt("***RAUMKLIMA***",0,0)
while True:
    sleep(0.3)
    dht22.measure()
    t=dht22.temperature()
    h=dht22.humidity()
    d.rect(0,10,126,38,1)
    d.clearFT(1,2,14,3)
    d.writeAt("TEMP: {:.1f} *C".format(t),1,2)
    d.writeAt("HUM : {:.1f} %".format(h),1,3)
    tree(int(((t-15)//2)%6))
    sleep(2.7)
```

Neben den üblichen Verdächtigen bietet das Programm an Neuem nur den Import des Moduls **dht**, die Instanziierung des Objekts **dht22** und die Haupschleife mit dem Messauftrag **dht22.measure**() und dem Einlesen von Temperatur- und Feuchtewert. Die Ausgabe am Display und die Baumanzeige kennen wir schon. Interessant und unscheinbar ist vielleicht die Umrechnung der Temperatur von °C in den Index der Beleuchtungsstufe. durch den Term int(((t-15)//2)%6). Vom Quotientenwert der Ganzzahldivision der Abweichung der Temperatur von 15 °C nach oben und 2 wird der 6-er-Teilungsrest bestimmt und sicherheitshalber als Ganzzahl dargestellt. Noch einmal ganz langsam.

Beispiel: t = 18 °C
18-15 =3
3//2 = 1
1 % 6 =1 also Stufenindex 1

für 28°C käme heraus: 28-15=13; 13//2=6; 6%6 = 0; Der letzte Schritt ist nötig, weil es keine Stufe mit der Nummer 6 gibt.

11. Die (etwas andere) Weihnachtsverlosung

Ich kenne das noch aus der Schulzeit. Jeder brachte in der Adventszeit ein Päckchen mit und in der Woche vor den Freien wurde die Tombola gestartet – jedes Los gewinnt.

Als recyclebare Lose habe ich neutrale RFID-Karten gewählt, Die Auslosung übernimmt der ESP32 zusammen mit dem RFID-Kit. Nur um die Gewinne müssen Sie sich selber kümmern. Natürlich ist auch das Bäumchen dabei. Durch seine Leuchtkraft verkündet es dem jeweiligen Mitspieler seinen Gewinn. Damit keine Zweifel bei der Deutung aufkommen, nennt das Display unzweifelhaft den Ort jeder Ziehung: Freiburg, Berlin, Hamburg …. Sechs Loskarten und eine Masterkarte werden gebraucht.

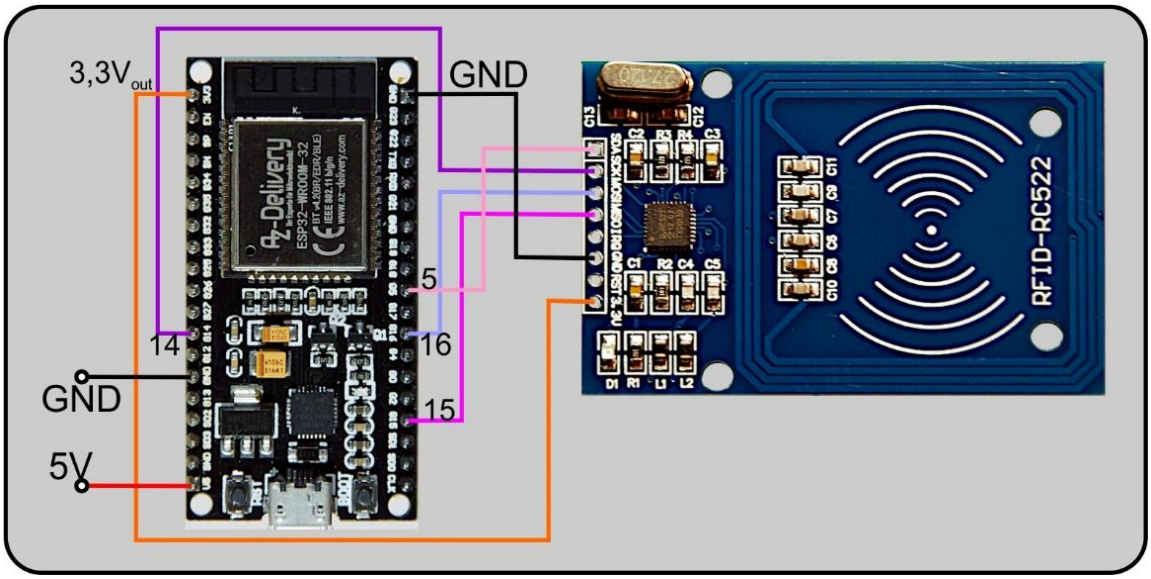

Abbildung 15: RFID-Karten-Leser

Durch den SPI-Bus ist die Verdrahtung etwas aufwendiger als beim I2C-Bus mit seinen 2 Leitungen. SPI-Bus-Geräte haben keine Hardware-Geräteadresse, dafür haben sie einen Chip-select-Anschluss, der auf LOW liegen muss, wenn das Gerät angesprochen werden soll. Auch der Datentransfer läuft etwas anders, es wird stets gleichzeitig gesendet und empfangen. Das nähere Procedere zu klären ist hier nicht notwendig, denn die Klasse **MFRC522** erledigt das für uns. Dem Konstruktor teilen wir nur die Anschlussbelegungen und die Übertragungsgeschwindigkeit mit. Der Transfer arbeitet mit flotten 3,2MHz. Zum Vergleich, I2C arbeitet auf 400kHz.

Die Funktion **readUID**() liest die eindeutige Kennung der Karte aus und gibt sie zurück und zwar als Hexadezimalwert und als Dezimalzahl. Die Karten werden über das OLED-Display angefordert. Damit die Funktion nicht den gesamten Ablauf blockiert, sorgt ein Timeout für einen geordneten Rückzug. In diesem Fall wird statt der Karten-ID der Wert None zurückgegeben.

Abbildung 16: RFID-Karten und Chip

Damit die Loskarten ins Spiel kommen, brauchen wir eine Masterkarte. Dazu nehmen wir eine beliebige Karte oder einen Chip aus dem Stapel, lesen die ID aus und belegen damit gleich am Programmbeginn die Variable mit dem Dezimalwert: MasterID=4217116188.

Beim ersten Start stellt der ESP32 fest, dass noch keine Datei mit den Los-Karten-Daten besteht und verlangt die Masterkarte. Nachdem diese erkannt wurde, wird eine Loskarte angefordert. Nach dem Auslesen der ID wird diese in die Datei geschrieben und erneut die Masterkarte verlangt. Das Einlesen wird bis zur letzten Loskarte fortgesetzt. Wird nach der Anforderung der Masterkarte 10 Sekunden lang keine Loskarte angeboten, startet das System sich selbst neu. Voraussetzung dazu ist, dass das Programm **rfid**.py als **boot**.py zum ESP32 geschickt wurde. Im Kapitel 2 - Autostart ist genau erklärt, wie das geht. Um komplett von vorne zu beginnen, können wir die Datei **slavecards.txt** mit den Los-Karten-IDs über die Thonny-Console löschen. Nach einem Reset können dann die Los-Karten erneut eingelesen werden.

rfid.py

```
# rfid.py
# workes with RC522 13,2MHz
import mfrc522
import esp32, dht
from os import uname
from machine import Timer, Pin, I2C, ADC, reset
from oled import OLED
from time import sleep,ticks_ms, ticks_us, sleep_ms
from gy521 import GY521

# Initialisieren der Schnittstellen **********************
if uname()[0] == 'esp32':
    #                    sck, mosi, miso, cs=sda
    rdr = mfrc522.MFRC522(14,  16,   15,   5,
baudrate=3200000)
elif uname()[0] == 'esp8266':
    #                    sck, mosi, miso, cs=sda
```

```
    #                       D3   D4    D2    D5
    rdr = mfrc522.MFRC522(0,   2,    4,    14,
baudrate=100000)
else:
    raise RuntimeError("Unsupported platform")
MasterID=4217116188  #  0XFB5C161C

i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

schichtPin = [32,33,25,27,26,12]
schicht=[0]*6
for i in range(6):
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)
gewinn=[
        "Freiburg",
        "Berlin",
        "Hamburg",
        "Augsburg",
        "Ratzeburg",
        "Erfurt",
        "Essen",
        "Bonn",
       ]
# Funktionen defnieren ***********************************
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare

def readUID(display,kartentyp,timeout):
    display.clearFT(0,1,15,show=False)
    display.writeAt("Put on "+kartentyp,0,1)
    readTimeOut=TimeOut(timeout)
    while not readTimeOut():
        (stat, tag_type) = rdr.request(rdr.REQIDL)
        if stat == rdr.OK:
            (stat, raw_uid) = rdr.anticoll()
            if stat == rdr.OK:
                display.clearFT(0,2,15,show=False)
                display.writeAt("Card OK",0,2)
                sleep(1)
                userID=0
                for i in range(4):
                    userID=(userID<<8) | raw_uid[i]
                userIDS="{:#X}".format(userID)
                print(userIDS)
                return userID,userIDS
    return None
```

```
def addUID(display):
    display.clearAll()
    m=readUID(display,"Master",3000)
    if m is not None:
        mid,_= m
        if mid==MasterID:
            sleep(3)
            u=readUID(display,"Slavecard",3000)
            if u is not None:
                uid,uids=u
                if uid is not None and uid != MasterID:
                    with open("slavecards.txt","a") as f:
                        f.write("{}\n".format(uids))
                        display.writeAt("New slave
written",0,3)
                        sleep(3)
                        return True
            else:
                display.writeAt("ERROR!!!",0,3)
                display.writeAt("Card not added!",0,4)
                return False
        else:
            display.writeAt("ERROR!!!",0,3)
            display.writeAt("Not mastercard",0,4)
            sleep(3)
    return False

def  switch(n,val):
    schicht[n].value(val)

def stop():
    d.writeAt("GOOD BYE",4,2)
    for i in range(6):
        switch(i,0)

def alle():
    for i in range(6):
        sleep_ms(300)
        switch(i,1)

def tree(n):
    for i in range(6):
        if i <=n:
            switch(i,1)
        else:
            switch(i,0)

# ****************** Hauptprogramm *********************
d.clearAll()
d.writeAt("*XMAS  LOTTERIE*",0,0)
d.rect(0,20,127,28,1)
cards=[]
```

```
try:
    with open("slavecards.txt","r") as f:
        for line in f:
            cards.append(line.strip("\n"))
    closed=TimeOut(60000)
    while not closed():
        u=readUID(d,"LOSKARTE",5000)
        d.clearFT(1,3,14,4,False)
        if u is not None:
            uid,uids=u
            try:
                n=cards.index(uids)
                d.writeAt("TREFFER {}".format(n),1,3, False)
                d.writeAt(gewinn[n],1,4)
            except ValueError as e:
                d.writeAt("TROSTPREIS",1,3)
                n=-1
            tree(n)
            closed=TimeOut(60000)
            sleep(10)
            stop()
except OSError as e:
    print("keine Datei, keine Daten!")
    allRead=TimeOut(10000)
    while not allRead():
        if addUID(d):
            allRead=TimeOut(10000)
    print("Alle Karten eingelesen und gespeichert")
    d.clearFT(0,3,15,4,False)
    d.writeAt(" ALL CARDS READ",0,3)
    d.writeAt("**R E B O O T**",0,4)
    reset()
d.clearFT(0,1,15,3,False)
d.writeAt("Lotterie neu",0,2)
d.writeAt("starten",0,3)
```

Für einen Spielzyklus werden 6 Gewinne bestimmt, die 6 Karten gemischt und verteilt und die neue Runde mit der PROG-Taste am ESP32 gestartet.

12. Das Bäumchen im LAN/WLAN

Machen wir das Dutzend voll, und bringen wir das Bäumchen ans Netz. Denn, wenn schon ein ESP32 zur Steuerung verwendet wird, dann muss auch ein LAN- oder WLAN-Zugriff auf die Steuerung her. Ich habe mich für die Realisierung eines Webservers auf dem ESP32 entschieden, weil die Ebenen des Bäumchens dann mit fast jedem Browser kontrolliert werden können. Als Alternative wäre ein UDP-Server auf dem Controller und eine Handy-App in Frage gekommen. Das hätte aber den Rahmen dieses Blogs gesprengt und deshalb habe ich davon Abstand genommen. Für Interessierte, eine solche Art der Steuerung habe ich bereits in anderen Beiträgen beschrieben, zum Beispiel hier und hier.

Für die Schaltung wird der Aufbau von Kapitel 5 gebraucht, den wir um eine RGB-LED und drei 1,0 kΩ-Widerstände erweitern.

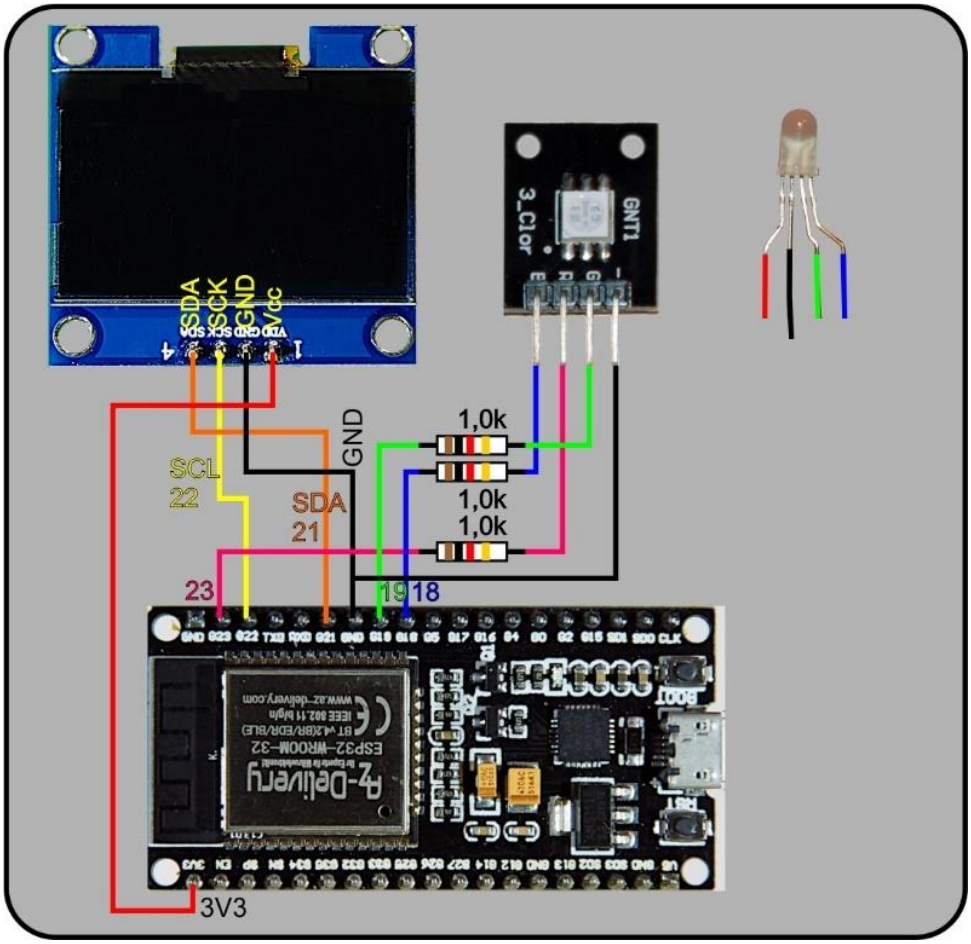

Abbildung 17: Web-Aufbau

Nach dem Import der nötigen Module definieren wir die Pins für die RGB-LED, welche uns den Netzwerkstatus auf weite Entfernung sichtbar verrät. Es folgt die Auswahl der Netzwerkbetriebsart, WLAN oder ESP32-eigener Accesspoint. WLAN ist standardmäßig voreingestellt. Für den Zugriff auf den WLAN-Router müssen hier dann auch die Zugangsdaten angegeben werden. Die Schichtendefinition wird um drei Listen erweitert, Klartext für an/aus, Hintergrundfarbe für die Tabelle in der Webseite und die Schaltzustände der Ebenen.

Bei den Funktionen wurden die blockierenden entfernt, **schwellen**(), **abschwellen**(), **welle**() und **baum**(). Neu dazugekommen sind **hexMac**(), **blink**(), **ledsOff**() und **web_page**(). **hexMac** gibt die MAC-Adresse des ESP32 im Station-Mode aus, **blink**() signalisiert die Netzwerk und Serverzustände. Mit **ledsOff**() wird die RGB-LED ausgeschaltet und **web_page**() zerpflückt Anfragen vom Browser, führt die Aufträge aus und gibt eine Antwort als Webseitentext zurück.

Die Anfrage des Browsers geht als Query-String an den Server. Der String hat die Form **?a**, **?p** oder **?e=x&v=y**. Darin stehen x für die Ebenennummer und y für den Schaltzustand, 0 oder 1.

web_page() wandelt die Anfrage in Großbuchstaben, prüft zunächst auf "A" und "P". Wenn die Anfrage mehr als 2 Zeichen enthält, wird versucht, die Ebene und den

Schaltzustand zu ermitteln. Passiert dabei ein Fehler, wird keinerlei Aktion ausgelöst und die nackte Startseite aufgerufen. Das passiert auch, wenn kein Querystring angegeben wurde. Danach erfolgt der Aufbau der Webseite als String und dessen Rückgabe an die Hauptschleife.

Nach den Funktionsdefinitionen folgt der Aufbau der Netzverbindung, entweder als eigener Accesspoint oder als Verbindung zum WLAN-Router. Gesteuert wird das durch die beiden Variablen **ownAP** und **WLANconnect**. In beiden Fällen wird eine feste IP-Adresse (10.0.1.181) vergeben, da es sich ja um einen Server handelt. Dynamische Adressen vom WLAN-Router sind ungeeignet, da sie von mal zu mal wechseln können. Der Verbindungsaufbau zum Router wird durch das Blinken der blauen LED markiert. Das Display informiert uns über, wenn die Verbindung steht und auch der Verbindungs-Socket s bereit ist, Anfragen entgegenzunehmen.

In der Hauptschleife wartet die Empfangsschleife der Methode **accept**() auf eine Anfrage. Kommt bis zum Timeout nichts an, wirft **accept**() eine exception, die wir mit dem vorangehenden try abfangen.

Liegt eine Anfrage vor, dann liefert **accept**() einen Kommunikationssocket **c** und die Adresse der anfragenden Maschine zurück. c dient zum Abwickeln des Datenaustauschs zwischen Client und Server, während s wieder frei wird, um weitere eingehende Anfragen anzunehmen. Die Methode **c.recv**() liefert den Text der Anfrage, von dem uns allerdings nur die ersten paar Zeichen interessieren. In der Entwicklungsphase kann man zum Testen des Parsers **web_page**() Anfragen von Hand eingeben. **ownAP** und **WLANconnect** müssen dann beide auf **False** gesetzt sein.

Das Bytesobjekt **request** des empfangenen Texts wird nun in einen String **r** dekodiert, der sich leichter handhaben lässt. Wir suchen nach einem "GET /" ganz am Anfang des Strings r und nach der Position, an der " HTTP" folgt. Wird beides gefunden, dann isolieren wir den Text nach dem "/" von "GET" bis vor das Leerzeichen von " HTML" und senden ihn als Querystring an den Parser **web_page**(). Dessen Antwort empfangen wir in der Variablen **response**. Danach senden wir den HTML-Header und den Text der HTML-Seite mit der enthaltenen Antwort an den Aufrufer zurück. Die nachfolgenden beiden **else** und das **except** dienen zum Abfangen und Behandeln von eventuellen Fehlern. Wichtig ist das abschließende **c.close**(), welches den Kommunikationssocket **c** schließt.

Nach einer Tastenabfrage zum Programmabbruch zeigt uns die grüne LED durch ihr kurzes Aufblitzen als Heartbeat an, dass das System noch lebt.

webcontrol.py

```
# webcontrol.py
# Fernsteuerung vom Browser via TCP
# (C) 2021 Jürgen Grzesina
# released under MIT-License (MIT)
# http://www.grzesina.de/az/weihnachtsbaum/MIT-License.txt
#
from machine import Pin, I2C
from oled import OLED
```

```
# ******************** Network stuff ********************
from time import sleep,ticks_ms, sleep_ms
try:
  import usocket as socket
except:
  import socket
import ubinascii
import network

statusLed=Pin(18,Pin.OUT,value=0) # blau=2
onairLed=Pin(19,Pin.OUT,value=0)  # gruen=1
errorLed=Pin(23,Pin.OUT,value=0)  # rot=0
led=[errorLed,onairLed,statusLed ]
red,green,blue=0,1,2
request = bytearray(50)
response=""
taste=Pin(0,Pin.IN,Pin.PULL_UP)

# Auswahl der Betriebsart Netzwerk oder Tastatur:
# -----------------------------------------------------
# Netzwerk: Setzen Sie genau !_EINE_! Variable auf True
WLANconnect=True  # Netzanbindung ueber lokales WLAN
ownAP=False       # Netzanbindung ueber eigenen Accesspoint
# beide False ->> Befehlseingabe ueber PC + USB in Testphase
# Falls WLANconnect=True:
# Geben Sie hier die Credentials Ihres WLAN-Accesspoints an
mySid = 'YOUR_SSID'; myPass = "YOUR_PASSWORD"
myIP="10.0.1.181"
myPort=9002

# Initialisieren der Schnittstellen *********************
i2c=I2C(-1,scl=Pin(22),sda=Pin(21))
d=OLED(i2c)

#schichtPin = [32,33,25,27,26,12]  # sortiert
schichtPin = [32,26,33,27,25,12]   # verteilt
schicht=[0]*6
for i in range(6):
    schicht[i]=Pin(schichtPin[i],Pin.OUT)
    schicht[i].value(0)
zustand=["aus","an "]
color=["red","lightgreen"]
eState=[0,0,0,0,0,0]

connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "GOT_IP"
    }
```

```
# Funktionen defnieren ***********************************
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare

def  switch(n,val):
    schicht[n].value(val)

def stop():
    d.writeAt("ALL LEDS OFF",2,5)
    for i in range(6):
        switch(i,0)

def alle():
    d.writeAt("ALL LEDS ON ",2,5)
    for i in range(6):
        sleep_ms(300)
        switch(i,1)

def tree(n):
    d.writeAt("TREE PROGR. ",2,5)
    for i in range(6):
        if i <=n:
            switch(i,1)
        else:
            switch(i,0)

def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
  entgegen und bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):     # Fuer alle Bytewerte
    macString += hex(byteMac[i])[2:]  # ab Position 2 bis Ende
    if i <len(byteMac)-1 :            # Trennzeichen
      macString +="-"
  return macString

def blink(pulse,wait,col,inverted=False):
    if inverted:
        led[col].off()
        sleep(pulse)
        led[col].on()
        sleep(wait)
    else:
        led[col].on()
        sleep(pulse)
        led[col].off()
```

```
        sleep(wait)

def ledsOff():
    for i in range(3):
        led[i].value(0)

def web_page(q):
    global eState
    q=q.upper()
    print("Anfrage: ",q)
    if q=="?A":
        alle()
        for i in range(6):
            eState[i]=1
    elif q=="?P":
        stop()
        for i in range(6):
            eState[i]=0
    elif len(q)>2:
        try:
            ebene,state=q[1:].split("&")
            _,ebene= ebene.split("=")
            _,state= state.split("=")
            ebene=(int(ebene) if 0<=int(ebene)<=5 else 0)
            state=(int(state) if 0<=int(state)<=1 else 0)
            switch(ebene,state)
            eState[ebene]=state
        except:
            pass
    else:
        pass
    antwort="<tr>"
    for i in range(6):
        h="<td bgcolor={}><H3>E{}
{}.</H3></td>".format(color[eState[i]],i, zustand[eState[i]])
        antwort=antwort+h
    antwort=antwort+"</tr>"
    html1 = """<html>
    <head>
    <meta name="viewport" content="width=device-width,
initial-scale=1">
    </head>
    <body>
    <h2>Hallo, <br>ich bin dein
Weihnachtsb&auml;umchen</h2>"""
    html2="""<table border=2 cellspacing=2>
    """
    html3="""
    <tr>
    <td>
    <a href='http://10.0.1.181:9002/?e=0&v=1'><H3>E0 An </H3>
</a>
```

```
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=1&v=1'><H3>E1 An </H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=2&v=1'><H3>E2 An </H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=3&v=1'><H3>E3 An </H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=4&v=1'><H3>E4 An </H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=5&v=1'><H3>E5 An </H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?a'><H3>ALLE AN </H3> </a>
    </td>
    </tr>
    <tr>
    <td>
    <a href='http://10.0.1.181:9002/?e=0&v=0'><H3>E0 Aus</H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=1&v=0'><H3>E1 Aus</H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=2&v=0'><H3>E2 Aus</H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=3&v=0'><H3>E3 Aus</H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=4&v=0'><H3>E4 Aus</H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?e=5&v=0'><H3>E5 Aus</H3>
</a>
    </td>
    <td>
    <a href='http://10.0.1.181:9002/?p'><H3>ALLE AUS</H3> </a>
```

```
    </td>
    </tr>
    """
    html9 = "</table> </body> </html>"
    html=html1+html2+antwort+html3+html9
    return html

if taste.value()==0:
    print("Mit Flashtaste abgebrochen")
    ledsOff()
    d.writeAt("Abbruch d. User ",0,5)
    sys.exit()

# *******************************************************
# Netzwerk einrichten
# *******************************************************
# Eigener ACCESSPOINT
# *******************************************************
if ownAP and (not WLANconnect):
    #
    nic = network.WLAN(network.AP_IF)
    nic.active(True)
    ssid="christbaum"
    passwd="don't_care"

    # Start als Accesspoint
    nic.ifconfig((myIP,"255.255.255.0",myIP,\
                  myIP))

    print(nic.ifconfig())

    # Authentifizierungsmodi ausser 0 werden nicht
unterstuetzt
    nic.config(authmode=0)

    MAC=nic.config("mac") # liefert ein Bytes-Objekt
    # umwandeln in zweistellige Hexzahlen
    MAC=ubinascii.hexlify(MAC,"-").decode("utf-8")
    print(MAC)
    nic.config(essid=ssid, password=passwd)

    while not nic.active():
      print(".",end="")
      sleep(0.5)

    print("Unit1 listening")
# ***************  Setup accesspoint end *****************

# *******************************************************
# WLAN-Connection
# *******************************************************
if WLANconnect and (not ownAP):
```

```
    nic = network.WLAN(network.STA_IF) # erzeuge WiFi-Objekt
    nic.active(True)  # Objekt nic einschalten
    #
    MAC = nic.config('mac')  # binaere MAC-Adresse abrufen +
    myMac=hexMac(MAC)        # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    # Verbindung mit AP im lokalen Netzwerk aufnehmen,
    # falls noch nicht verbunden, dann
    # connect to LAN-AP
    if not nic.isconnected():
      nic.connect(mySid, myPass)
      # warten bis die Verbindung zum Accesspoint steht
      print("connection status: ", nic.isconnected())
      while not nic.isconnected():
        blink(0.8,0.2,0)
        print("{}.".format(nic.status()),end='')
        sleep(1)
    # zeige Verbindungsstatus & Config-Daten
    print("\nconnected: ",nic.isconnected())
    print("\nVerbindungsstatus: ",connectStatus[nic.status()])
    print("Weise neue IP zu:",myIP)
    nic.ifconfig((myIP,"255.255.255.0",myIP, \
                  myIP))
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')

# ***********  Setup Router connection end **************

# ******************************************************
# TCP-Web-Server
# ******************************************************
# ----------------- Server starten -------------------------
if WLANconnect or ownAP:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    s.bind(('', myPort))
    print("Socket established, waiting on port",myPort)
    d.clearAll()
    #          0123456789012345
    d.writeAt("SOCK ESTABLISHED",0,0)
    d.writeAt("LISTENING AT",0,1)
    d.writeAt(myIP+":"+str(myPort),0,2)
    s.settimeout(0.9)
    s.listen(2)

if taste.value()==0:
    print("Mit Flashtaste abgebrochen")
    ledsOff()
    d.writeAt("Abbruch d. User ",0,5)
    sys.exit()
```

```
# ------------------- Serverschleife ----------------------
while True:
    try:  # wegen timeout
        r=""
        if WLANconnect or ownAP:
            c, addr = s.accept()
            print('Got a connection from {}:{}\n'.\
                  format(addr[0],addr[1]))
            request=c.recv(1024)
        else:
            request=input("Kommando:")
            addr="999.999.999.999:99999"
        try:  # decodieren und parsen
            r=request.decode("utf8")
            getPos=r.find("GET /")
            if r.find("favicon")==-1:
                print("************************************")
                print("Position:",getPos)
                print("Request:")
                print(r)
                print("************************************")
                pos=r.find(" HTTP")
                if getPos == 0 and pos != -1:
                    query=r[5:pos] # nach ? bis HTTP
                    print("*********QUERY:{}*********\n\n".\
                          format(query))
                    response = web_page(query)
                    print("---------------\n",response,\
                          "\n----------------")
                    c.send('HTTP/1.1 200 OK\n'.encode())
                    c.send('Content-Type: text/html\n'\
                           .encode())
                    c.send('Connection: close\n\n'.encode())
                    c.sendall(response.encode())
                else:
                    print("##########\nNOT HTTP\n###########")
                    c.send('HTTP/1.1 400 bad request\n'\
                           .encode())
            else:
                print("favicon request found")
                c.send('HTTP/1.1 200 OK\n'.encode())
        except:  # decodieren und parsen
            request = rawRequest
            c.send('HTTP/1.1 200 OK\n'.encode())
        c.close()
    except:  # wegen timeout
        pass

    if taste.value()==0:
        print("Mit Flashtaste abgebrochen")
        ledsOff()
        d.writeAt("Abbruch d. User ",0,5)
```

```
        sys.exit()
    blink(0.05,0.05,1)
```

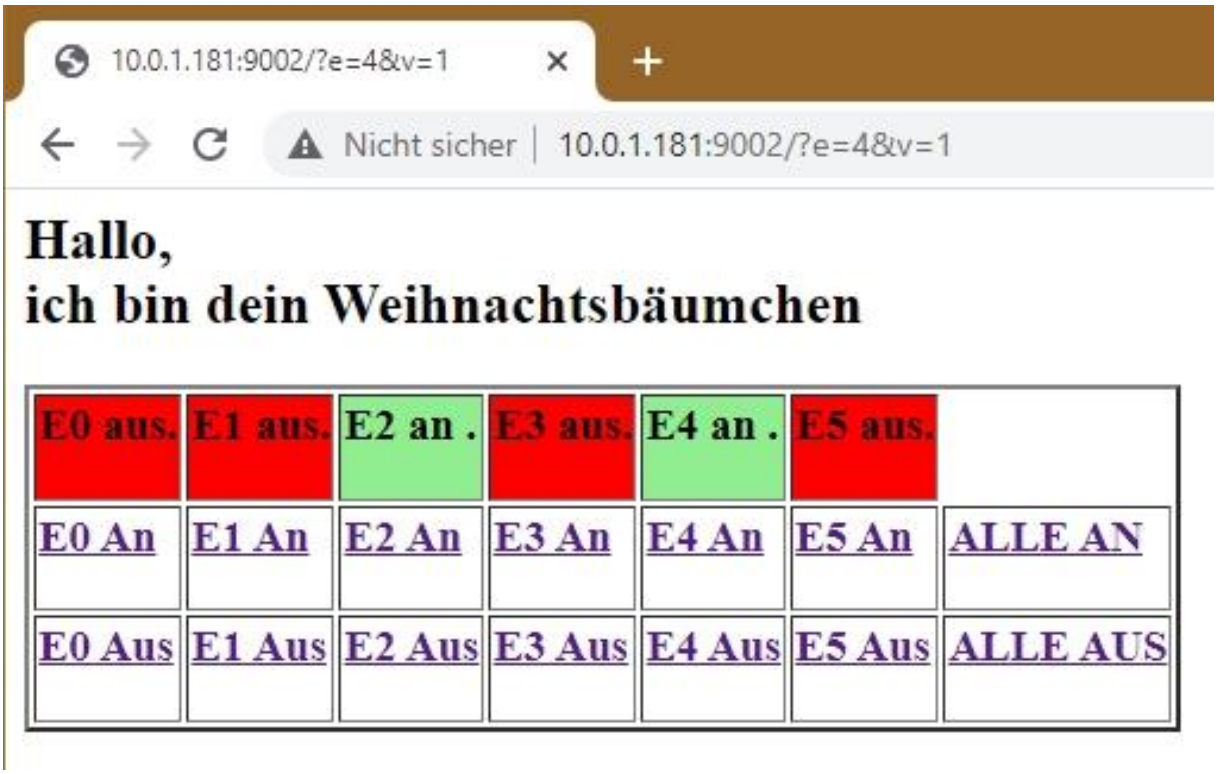

Abbildung 18: Live aus dem Browser

So sieht die Webseite in der Realität auf Google Chrome aus. Opera bietet ein ähnliches Bild nach der Eingabe der URL 10.0.1.181:9002. Firefox macht Zicken, weil die Macher sich in den Kopf gesetzt haben, die User gängeln zu müssen, indem ihr Browser nur https-Adressen akzeptiert. Aber es gibt ja Alternativen. Wenn es ganz schlimm kommt, könnte man sich mit CPython sogar ein eigenes Frontend für den PC schreiben.

Nun, ich denke, jetzt haben Sie bis Weihnachten genug zu tun mit den Bäumchen-Projekten. Sicher ist für jeden das Eine oder Andere dabei. Wichtig ist, dass Sie Freude an der Umsetzung haben und dass ich Ihr Interesse wecken konnte. Ich wünsche Ihnen jedenfalls eine schöne Adventszeit.