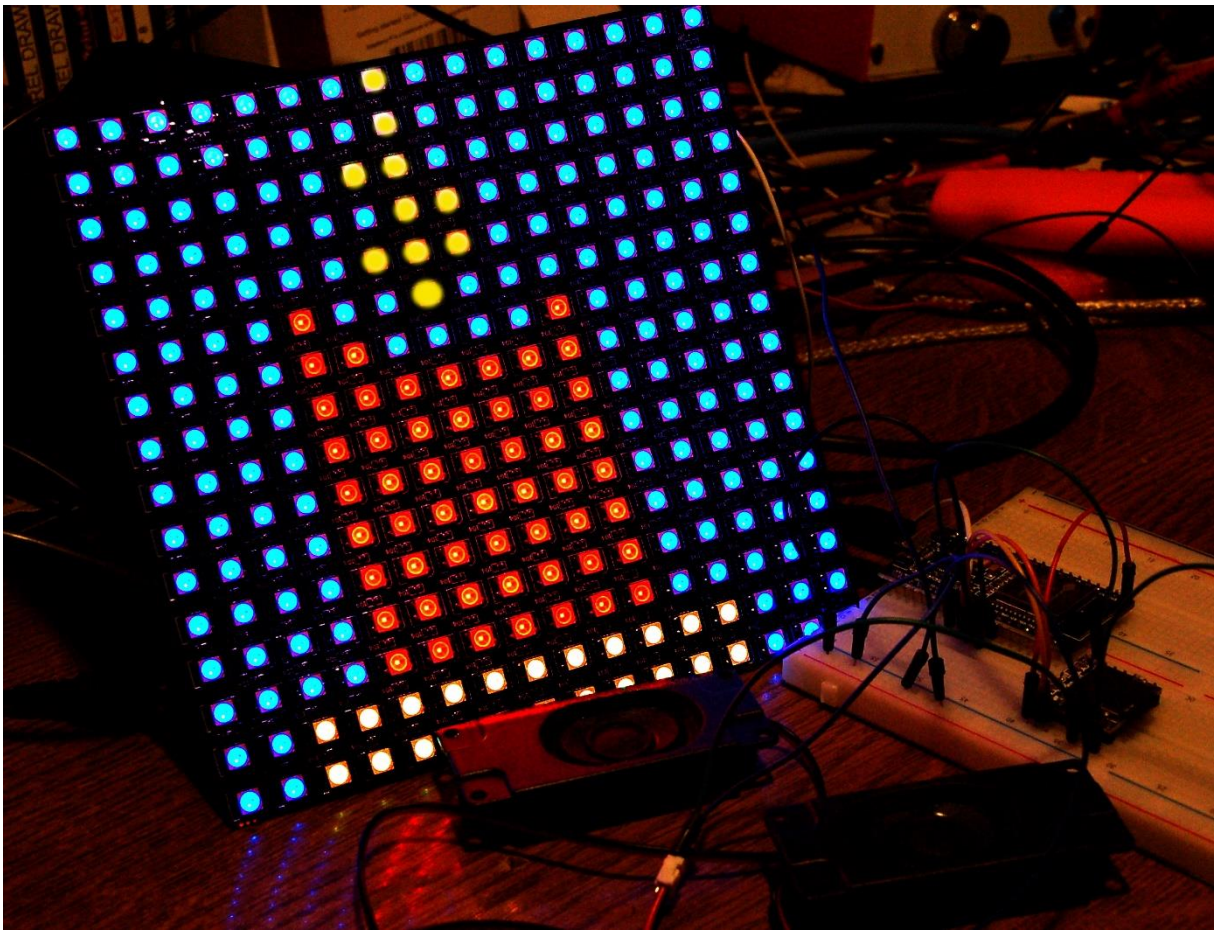

Adventskerze

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Mit dem ESP32 kann man malen. Mein Pinsel ist ein Programm, das in MicroPython geschrieben wurde, die Leinwand besteht aus einer LED-Matrix, bestückt mit 256 Neopixel-LEDs. Und beim Malen steuert der Controller ein Soundmodul an, das die Lieder von einer SD-Karte abspielt. Jede Stunde wird ein Song durch einen Zufallsgenerator ausgewählt. Damit die Zeit stimmt, synchronisiert der ESP32 seine Systemzeit mit einem Zeitserver im Internet. Neugierig geworden? Dann kommen Sie mit mir ins virtuelle Malstudio mit einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Adventliche Bilder und weihnachtliche Musik mit dem ESP32

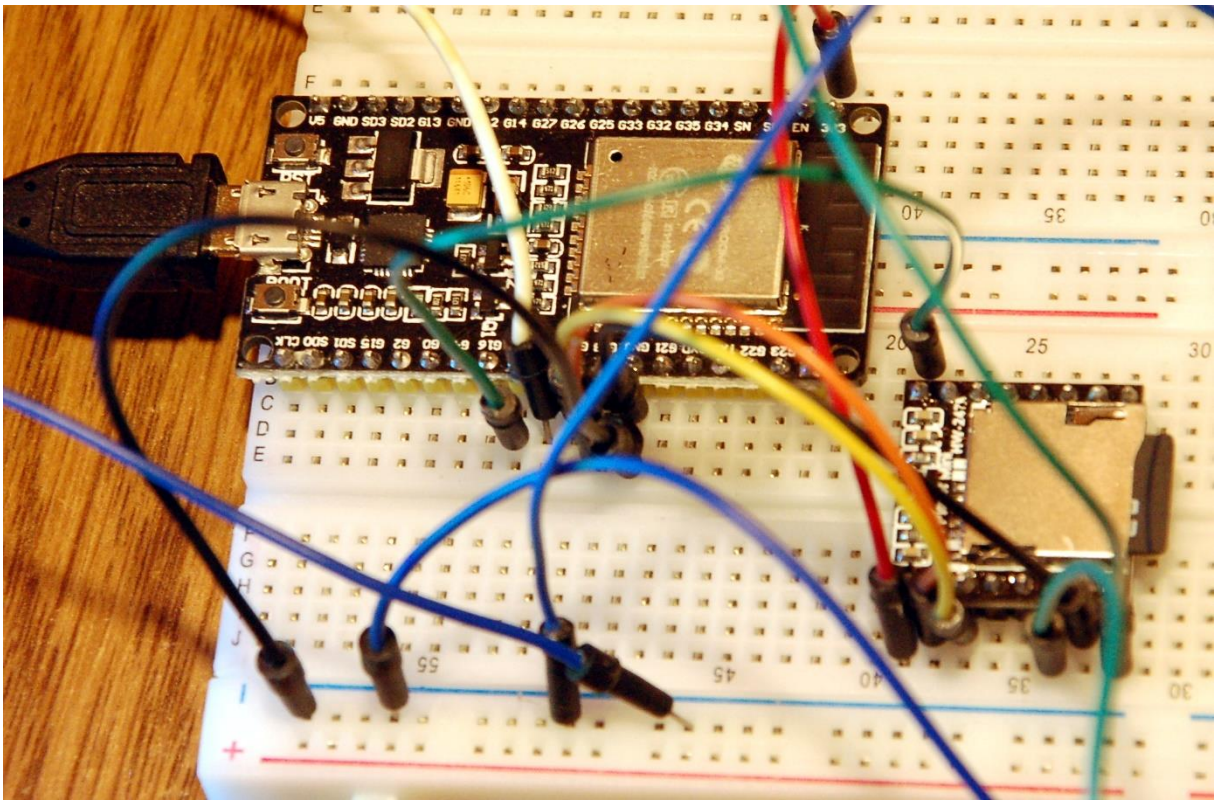

Abbildung 1: Minimalistische Schaltung

Die Schaltung kommt mit fünf Teilen aus, ist also sehr übersichtlich. Die beiden aktiven Bausteine sind ein ESP32 und ein DF-Player mini. Ersterer benötigt wegen seiner breiten Bauform zwei seitlich aneinander gesteckte Breadboards, damit man allen Beinchen des 38-Füßlers ein Zuhause geben kann. Versorgt wird der Controller über das USB-Kabel oder später dann von einem 5V-Steckernetzteil. Dieses sollte Stromstärken bis ca. 2 Ampere liefern können, weil die LEDs der Matrix bei voller Leuchtkraft der roten, grünen und blauen LED pro Stück ca. 30 mA ziehen. Sind alle 256 LEDs auf voller Leistung, dann schluckt das Panel satte 7,5A. Aber keine Sorge, die Bilder, die wir erzeugen werden, bleiben deutlich unter einem Ampere, wenn wir nur mit 20% Leistung fahren. Glauben Sie mir, das ist hell genug.

Der DF-Player wohnt auch auf einem der Breadboards. Die Lieder befinden sich als MP3- oder WAV-Dateien auf einer Mini-SD-Karte, die bis zu 4 GB an Speicherplatz haben darf. Darauf können in bis zu 100 Ordnern bis zu je 255 Dateien untergebracht und direkt adressiert werden. Die Namen der Ordner müssen von der Form 00 … 99 sein, die Dateinamen beginnen stets mit drei Ziffern. Weitere alphanumerische Zeichen können folgen. Ein Punkt trennt den Namen von der Namensergänzung. Hierfür kommen .MP3 oder .WAV in Frage. Ein gültiger Dateiname könnte zum Beispiel 003_Vom_Himmel_hoch.MP3 sein. Die Karte kann FAT16 oder FAT32 formatiert sein. Die Sampling-Rate der Files muss einer aus der folgenden Aufzählung entsprechen:

8KHz, 11.025KHz, 12KHz, 16KHz, 22.05KHz, 24KHz, 32KHz, 44.1KHz, 48KHz

Angesteuert wird der DF-Player über eine RS232-Verbindung. Die Standard-Einstellungen sind 9600 Baud, 8 Bit, keine Parität, ein Stopp-Bit. Für bidirektionalen Betrieb ist ein ESP32 nötig, weil der ESP8266 an UART1 nur eine TX-Leitung zur

Verfügung stellt. Außerdem ist der kleine Bruder des ESP32 recht schmalbrüstig, was den Systemspeicher angeht.

Das Datenblatt des DF-Players bedient den Leser mit einem recht eigenwilligen Englisch. Dort sind die meisten der Befehle, die an den Chip gesendet werden können aufgeführt. Ich habe daraus ein MicroPython-Modul gebastelt, welches Ihnen die Mühe abnimmt, die Befehle an den DF-Player selbst codieren zu müssen. Wenn Sie allerdings tiefer einsteigen möchten, dann rentiert sich ein Blick in die Datei **dfplayer.py** in jedem Fall.

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	RGB LED Panel WS2812B 16x16 256 LEDs Flexibel Led Matrix
1	Mini MP3 Player DFPlayer Master Module
1	2 Stück DFplayer Mini 3 Watt 8 Ohm Mini-Lautsprecher
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
Evtl.	Basisbrett 17cm x 17cm

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

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

Die MicroPython-Programme zum Projekt:

dfplayer.py Hardwaretreiber für den DF-Player mini
pictures.py Der Malermeister

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) auf den ESP-Chip gebrannt wird.

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.

Signale auf dem RS232- und Neopixel-Bus

Wie eine Übertragung auf dem RS232-Bus abläuft und wie die Signalfolge ausschaut, das können Sie durch ein interessantes kleines Tool erkunden, mit dem Sie die Bus-Signale auf Ihren PC holen und analysieren. Die Bedienung des Logic-Analyzers ist in mammutmatrix_2 beschrieben. Klemmen Sie die Messtrippen aber nicht an die SCL- und SDA- sondern an die RX- und die TX-Leitung. Natürlich lässt sich das Tool auch einsetzen, um die Signale sichtbar zu machen, die Sie an das LED-Panel senden. Beiden Bussystemen liegt jeweils ein bestimmtes Protokoll zugrunde. Die Funktionsweise von Neopixel-LEDs habe ich in Bandit – Spiele mit dem ESP32 in MicroPython schon einmal beschrieben. Über die RS232-Schnittstelle habe ich in dem Artikel "Chat mit dem PC - die RS232-Schnittstelle" Informationen zusammengestellt.

Der Malermeister

Das Programm zum Ansteuern der LEDs ist modular gehalten. Alle vier Bilder sind anhand von Funktionen realisiert. Dadurch ist es leicht, die vorhandenen Strukturen zu verändern oder auszubauen und neue hinzuzufügen. Unregelmäßige Bilder wie den Stern legt man am besten in Form von Listen an. Die Listenelemente ihrerseits sind Tuples mit den Positionsnummern der LEDs. Der sequenzielle Datentyp Liste wird durch eckige Klammern definiert, ein Tuple durch runde Klammern.

```
family=[
    (120,121,136,137), # 0
    (103,106,151,154), # 1
    (86,91,166,171), # 2
    (52,69,70,85,87,102,
     90,107,75,92,76,61,
     150,167,165,182,181,196,
     170,155,172,187,188,205), # 3
    (88,89,104,105,
     122,123,138,139,
     152,153,168,169,
     118,119,134,135), # 4
    (18,35,36,51,53,54,68,71,84,101,
     31,45,46,59,60,62,74,77,93,108,149,
     183,164,180,197,198,195,211,212,226,
     156,186,173,203,204,189,206,221,222,239), #5
    (40,56,57,72,73,
     124,125,126,140,141,
     184,185,200,201,217,
     116,117,131,132,133), # 6
    (1,2,17,19,20,34,37,38,50,55,67,83,100,
     58,43,44,29,30,15,16,32,47,63,78,94,109,
     157,174,190,207,223,240,256,255,237,238,220,219,202,
     199,213,214,227,228,241,242,225,210,194,179,163,148), # 7
    (8,24,25,41,
     127,128,142,143,
     216,232,233,249,
     114,115,129,130) # 8
    ]
```

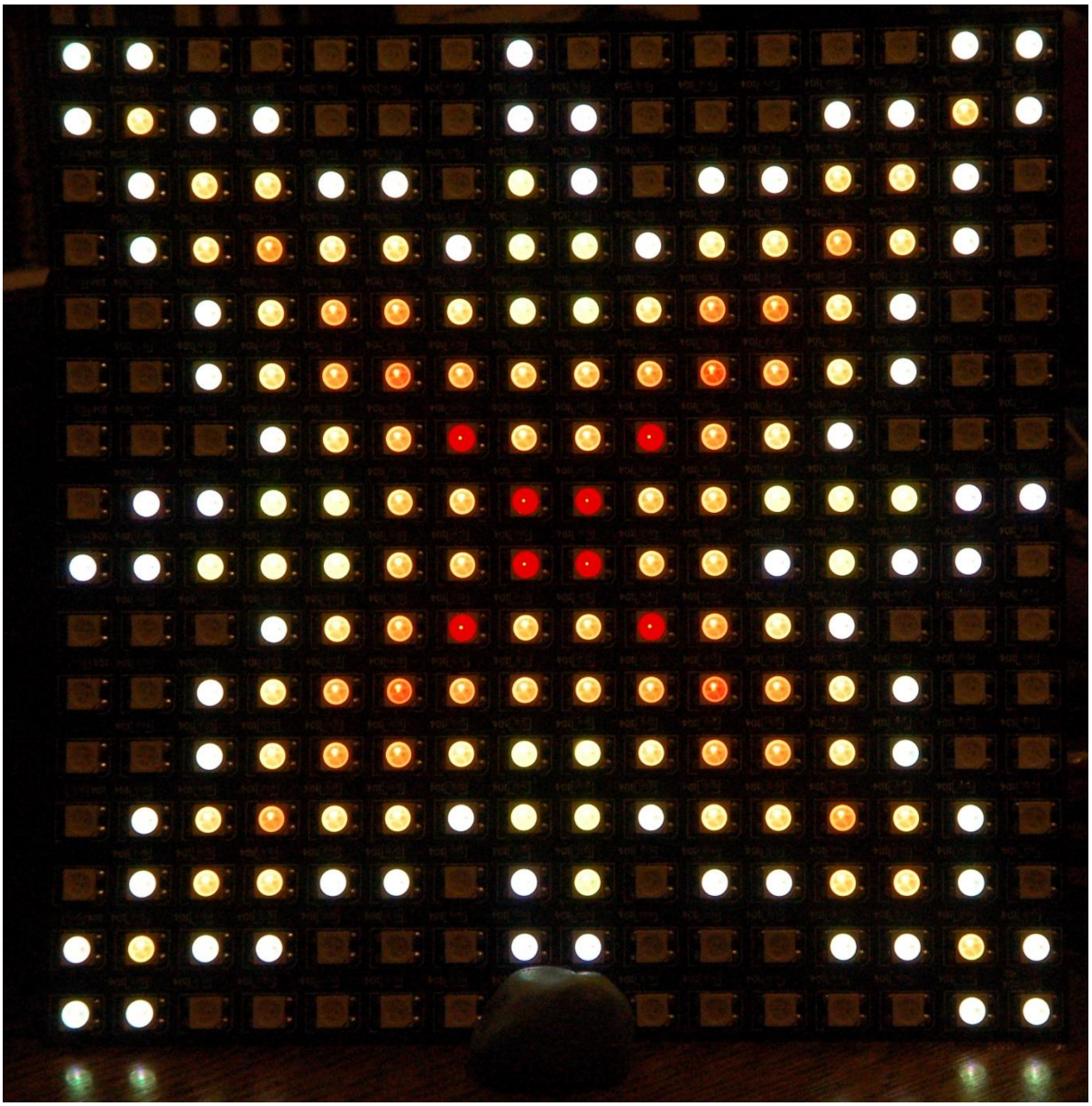

Abbildung 2: Weihnachtsstern

Auch die Farbpalette des Sterns ist eine Liste. Die Tuples enthalten die Farbanteile für rot, grün und blau.

```
faktor=0.5 # Helligkeits-Faktor
color=[(32,0,8),
       (64,0,2),
       (96,0,0),
       (128,16,0),
       (128,32,0),
       (128,48,0),
       (128,72,0),
       (128,128,0),
       (128,128,32),
       (128,128,64),
       (128,128,96),
       (128,128,128),
       (192,192,192)
      ]
```

Andere Bilder werden mit Hilfe von Funktionen aufgebaut, die Pixel setzen, Rechtecke zeichnen oder Füllungen erzeugen.

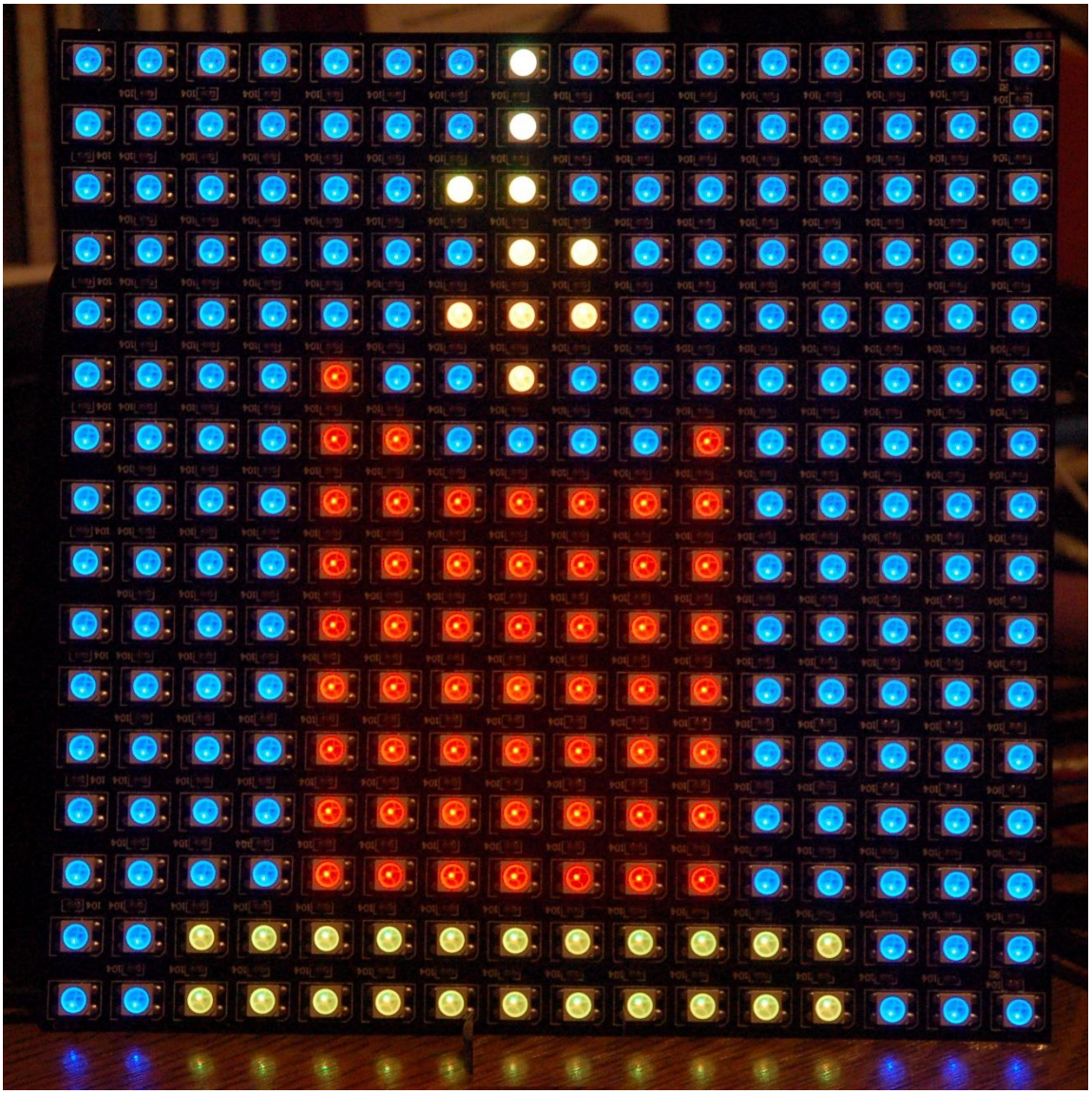

Abbildung 3: Adventskerze groß

Aber, alles der Reihe nach. Wie jedes MicroPython-Programm beginnt auch dieses mit ein paar Importen. Danach werden die Objekte erzeugt.

```
from machine import Pin, RTC
from time import sleep, time, localtime, ticks_ms
import ntptime
from neopixel import NeoPixel
from sys import exit
import network, socket
from dfplayer import DFPlayer
import random
```

Bis auf das Modul **dfplayer** sind alle anderen bereits im MicroPython-Paket enthalten. dfplayer.py muss extra auf den ESP32 hochgeladen werden, bevor man das Modul importieren kann. Laden Sie also die Datei aus dem Netz herunter und kopieren Sie sie ins Arbeitsverzeichnis von Thonny.

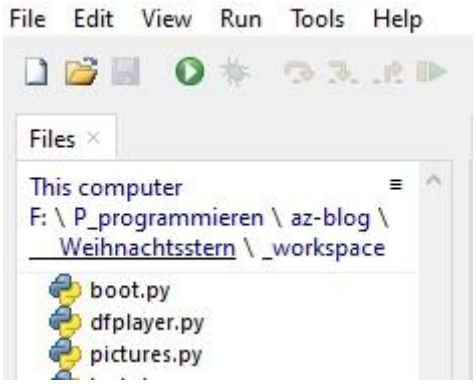

Abbildung 4: Dateien im Arbeitsverzeichnis

Klicken Sie **dfplayer**.py mit der rechten Maustaste und wählen Sie aus dem Kontextmenü **Upload to /**.

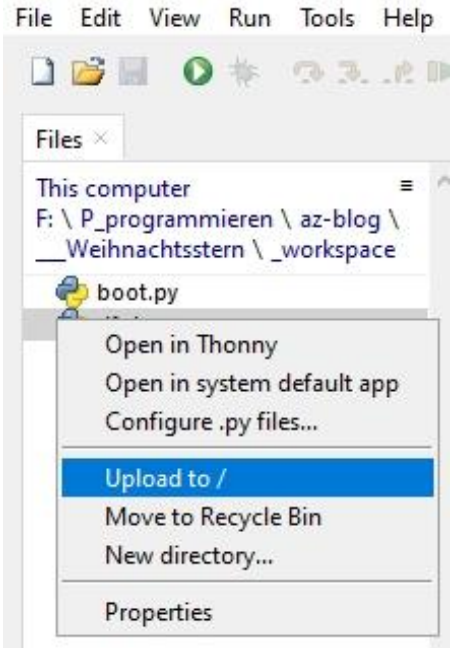

Abbildung 5: Datei hochladen

Jetzt sollte sich die Datei im Flash des ESP32 befinden.

Abbildung 6: Hochgeladene Datei

Ich erzeuge eine Neopixel-Instanz, die über GPIO5 das Panel steuert. Das Objekt stellt eine Liste als Datenpuffer zur Verfügung. Jedes Element enthält die drei Farbinformationen in Form eines Tuples.

```
neo=5 # D3
neoPin=Pin(neo,Pin.OUT)
neoCnt=256
np = NeoPixel(neoPin, neoCnt) #
```

Damit Datum und Uhrzeit stimmen, muss die interne Uhr, die Real-Time-Clock (RTC), des ESP32 regelmäßig mit der Zeitmarke eines Zeitservers im Internet synchronisiert werden. Die Zeitspanne in der das passiert, legt **syncIntervall** fest.

```
syncIntervall=600000 # ms
```

Wir befinden uns in Europa in der Zeitzone, in der eine Stunde zur UTC (Universal Time Coordinated) addiert wird. Die Stunden, in denen mein Panel zusammen mit dem Abspielen von Musik aktiv sein soll, lege ich durch eine Bereichsangabe fest, die ich in der Variablen **stunden** verankere.

```
timeZone=1
rtc=RTC()
rtcTag=(2022,11,27,6,8,0,0,0) # RTC-Tagesstempel Start
# Jahr,Monat,Tag,Stunde,Minute,Sekunde,Wochentag,Jahrestag
stunden=range(8,23)
```

Für den Zugriff auf einen NTP-Server brauchen wir das WLAN. Die Zugriffsparameter für die SSID und das Passwort richten sich nach den Einstellungen Ihres WLAN-Routers. Tragen Sie hier also unbedingt ihre eigenen Credentials ein. Die Portnummer ist dagegen fast frei wählbar und darf zwischen 1024 und 65535 liegen.

```
# **************WLAN-Zugriff definieren******************
#
mySSID="Here goes your SSID"
myPass="Here goes your password"
myPort=9009
```

Das Netzwerk-Interface (nic) des ESP32 gibt beim Verbindungsaufbau verschiedene Statusmeldungen zurück. Das Dictionary (aka Dict) **connectStatus** übersetzt die Zahlencodes in Klartext.

```
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }
```

Wie die Werte in einem Tuple sind die Schlüssel in einem Dictionary (kurz Dict) immutable, sie können nicht mehr geändert werden, nachdem die Struktur deklariert wurde. Während Schlüssel eines Tuples einzigartig sein müssen, dürfen die Werte auch mehrfach vorkommen. Dicts werden durch geschweifte Klammern eingefasst. Schlüssel und Wert werden durch einen Doppelpunkt getrennt. Zwischen den Paaren steht ein Komma, wie zwischen den Elementen einer Liste.

Ein Großteil der Programmarbeit wird durch Funktionen erledigt. Die Funktion **hexMac**() ist eine von ihnen und übersetzt das, von der Methode **nic.config()** gelieferte Bytes-Objekt, in eine menschenlesbare Zeichenkette. Dieser String besteht aus den üblichen Hexadezimalziffern 0-9 und A-F, durch welche die MAC-Adresse des Station-Interfaces (STA) des ESP32 festgelegt ist.

Dieses Six-Pack muss dem WLAN-Router bekanntgemacht werden, damit er dem ESP32 Zugang gewährt. Dazu ist die Adresse in die Liste der zugelassenen Geräte aufzunehmen. Die finden Sie in der Regel im Pflegemenü Ihres Routers unter dem Punkt WLAN – Sicherheit. Für das genaue Vorgehen ziehen Sie bitte das Handbuch Ihres Geräts zu Rate. Es ist übrigens aus Sicherheitserwägungen heraus keine gute Idee, allen sich anmeldenden Geräten, generell den Zugang zu erlauben, indem man die MAC-Filterung ausschaltet. Der nächste Hacker freut sich tierisch, wenn Sie ihm alle Türen offenhalten.

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

Die Kernzelle des Programms pictures.py ist das Setzen der Farbe eines Pixels, die Funktion **setPixe**l(). Sie nimmt neben der Nummer der LED-Einheit die Farbcodes für rot, grün und blau als Positionsparameter. Es folgt der Parameter f, welcher die Helligkeit des Pixels definiert. Der Wert reicht von 0.0 bis 1.0. Der Parameter **show** definiert mit dem Default-Wert **True**, dass die Änderung unmittelbar an das Panel übertragen wird. Wird er beim Aufruf der Funktion mit **False** angegeben, dann erfolgt die Änderung erst mit dem **True** eines nachfolgenden, entsprechenden Funktionsaufrufs oder durch Aufruf der Funktion **show**().

```
def setPixel(num,r,g,b,f,show=True):
    z=(num // 16) + 1
    n=num if z % 2 == 0 else z*16 - (num % 16 + 1)
    r=int(r*f)
    g=int(g*f)
    b=int(b*f)
    np[n]=(r,g,b)
    if show:
        np.write()
        sleep(0.01)
```

Hier ist ein Einschub erforderlich. Ich hatte ursprünglich angenommen, dass sich die Adressierung der Zellen des Panels von links nach rechts und zeilenweise von oben nach unten orientieren würde, entsprechend einem x-y-Koordinatensystem. Dem war leider nicht so. Der Durchlauf der Adressen passiert nach meiner Orientierung des Panels in der ersten Zeile von rechts oben, Pixel 0, bis links oben, Pixel 15, und dann in der zweiten Zeile von links, Pixel 16, bis rechts, Pixel 31, und der Schlangenlinie folgend nach unten.

Das Bild wurde aber eben nach dem x-y-System entworfen. Ich habe dazu Libre Office Calc verwendet. Die Abbildungen 7 und 8 zeigen den Zusammenhang auf, welcher in den folgenden Zeilen hergestellt wird.

```
    z=(num // 16) + 1
    n=num if z % 2 == 0 else z*16 - (num % 16 + 1)
```

Alle höheren Funktionen zur Erleuchtung von Pixeln greifen auf **setPixel**() zurück.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Abbildung 7: Libre Office x-y-System

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
143	142	141	140	139	138	137	136	135	134	133	132	131	130	129	128
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
175	174	173	172	171	170	169	168	167	166	165	164	163	162	161	160
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
207	206	205	204	203	202	201	200	199	198	197	196	195	194	193	192
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
239	238	237	236	235	234	233	232	231	230	229	228	227	226	225	224
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Abbildung 8: Libre Office Schlangen-System

Funktionen mit Breitenwirkung sind **showColors**() und vor allem **clearPanel**(). Erstere zeigt die Farben, die für den Stern zur Verfügung stehen, **clearPanel**() sollte eigentlich alle LEDs zum Erlöschen bringen. Es zeigte sich allerdings bereits in der frühen Entwicklungsphase des Programms ein seltsamer Effekt. Anstatt alle LEDs zu löschen, (Code = (0,0,0)) leuchteten einzelne in verschiedenen Farben und Helligkeitsstufen unvermittelt auf. Position und Helligkeit variierten und tun das auch jetzt noch. Welches Phänomen dahintersteckt konnte ich bis jetzt nicht entdecken. So fackeln also bei jedem Zugriff auf die Schreibfunktion **np.write**() unvermittelt völlig unbeteiligte LEDs auf. Falls Sie ein Gegenmittel dafür wissen, freue ich mich auf Ihren Kommentar zu diesem Beitrag.

```
def showColors(fak):
    clearPanel()
    for col in range(len(color)):
        r,g,b=color[col]
        setPixel(col,r,g,b,fak)

def clearPanel(show=True):
    for i in range(neoCnt):
        np[i]=(0,0,0)
    if show:
        np.write()
        sleep(0.01)
```

Die Funktion **fill**() lässt die Pixel von Adresse **von** bis Adresse **bis** in der angegebenen Farbe des Tuples **col** mit dem Helligkeitsfaktor **fak** erstrahlen.

```
def fill(von,bis,col,fak,show=True):
    for n in range(von,bis+1):
        setPixel(n,col[0],col[1],col[2],fak,show)
```

Eine ähnliche Funktion weist **rectangle**() auf. Sie füllt die Pixel ab der seriellen Position **n** in einer Breite von **b** und einer Höhe von **h** in der, in **col** codierten Farbe mit dem Faktor **fak** angepassten Helligkeit auf.

```
def rectangle(n,b,h,col,fak,show=True):
    red,green,blue=col
    for y in range(h):
        for x in range(b):
            setPixel(n+x+16*y,red,green,blue,fak,show=True)
```

Die Funktion **show**() bringt verdeckte Änderungen am Pufferspeicher des Objekts **np**, die nicht mit entsprechenden Funktionen initiiert wurden (show=False), zur Anzeige.

```
def show():
    np.write()
    sleep(0.01)
```

Die Funktion **kranzKerze**() baut eine Kerze für den Adventskranz auf, der in der Funktion **kranz**() definiert wird. Das Attribut **n** kennzeichnet das Pixel im Koordinatensystem an dem die linke untere Ecke der Kerze sein soll.

```
def kranzKerze(n,col,fak):
    f=n-32-2*16
    rectangle(n-2*16,2,3,col,fak)
    rectangle(f,2,2,(128,72,0),fak)
    setPixel(f-16,128,96,0,fak)
    setPixel(f-32+1,128,128,0,fak)
```

```
def kranz(fak):
    fill(0,223,(0,0,32),fak)
    fill(224,255,(16,16,0),fak)
    rectangle(177,14,3,(9,128,0),fak)
    for n in (162,166,169,172):
        kranzKerze(n,(128,0,0),fak)
        sleep(1)
```

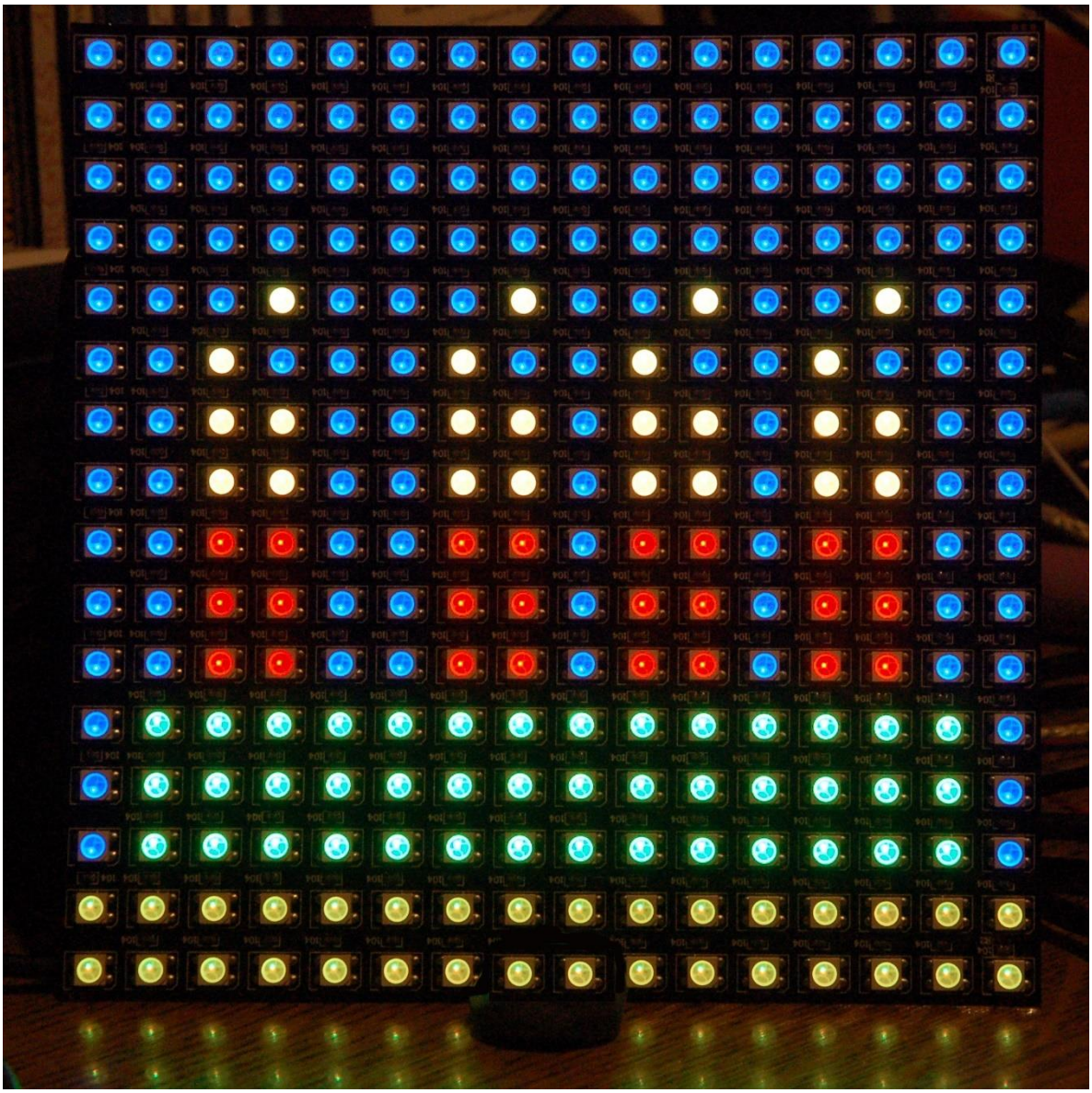

Abbildung 9: Adventkranz mit Kerzen

Die Funktion **kranzKerze**() wird von **kranz**() aufgerufen. **kranz**() erzeugt ein Rechteck, das mit Kerzen bestückt wird. Es erscheint vor einem blauen Hintergrund, der mit der Funktion **fill**() erzeugt wird.

Die Funktion **baum**() erzeugt das Abbild eines Weihnachtsbaums – mit Kerzen. Sie bedient sich der Funktion **schicht**(), welche die einzelnen Ebenen des Baums aufbaut. Dabei ist n hier die Nummer des Pixels in der Schicht links unten. Es folgen die Breite b dieser unteren Reihe und die Höhe h.

```
def schicht(n,b,h,fak):
    for y in range(h):
        for x in range(b-2*y):
            pos=n-y*16+y+x
            setPixel(pos,0,128,0,fak)
```

```
def baum(fak):
    fill(0,255,(0,0,32),fak)
    rectangle(231,2,2,(8,8,0),fak)
    schicht(210,12,4,fak)
    schicht(147,10,4,fak)
    schicht(84, 8,4,fak)
    schicht(37, 4,2,fak)
    for p in (40,54,89,74,118,152,181,202,211,215,220):
        baumKerze(p,(128,128,0),fak)
        sleep(1)
```

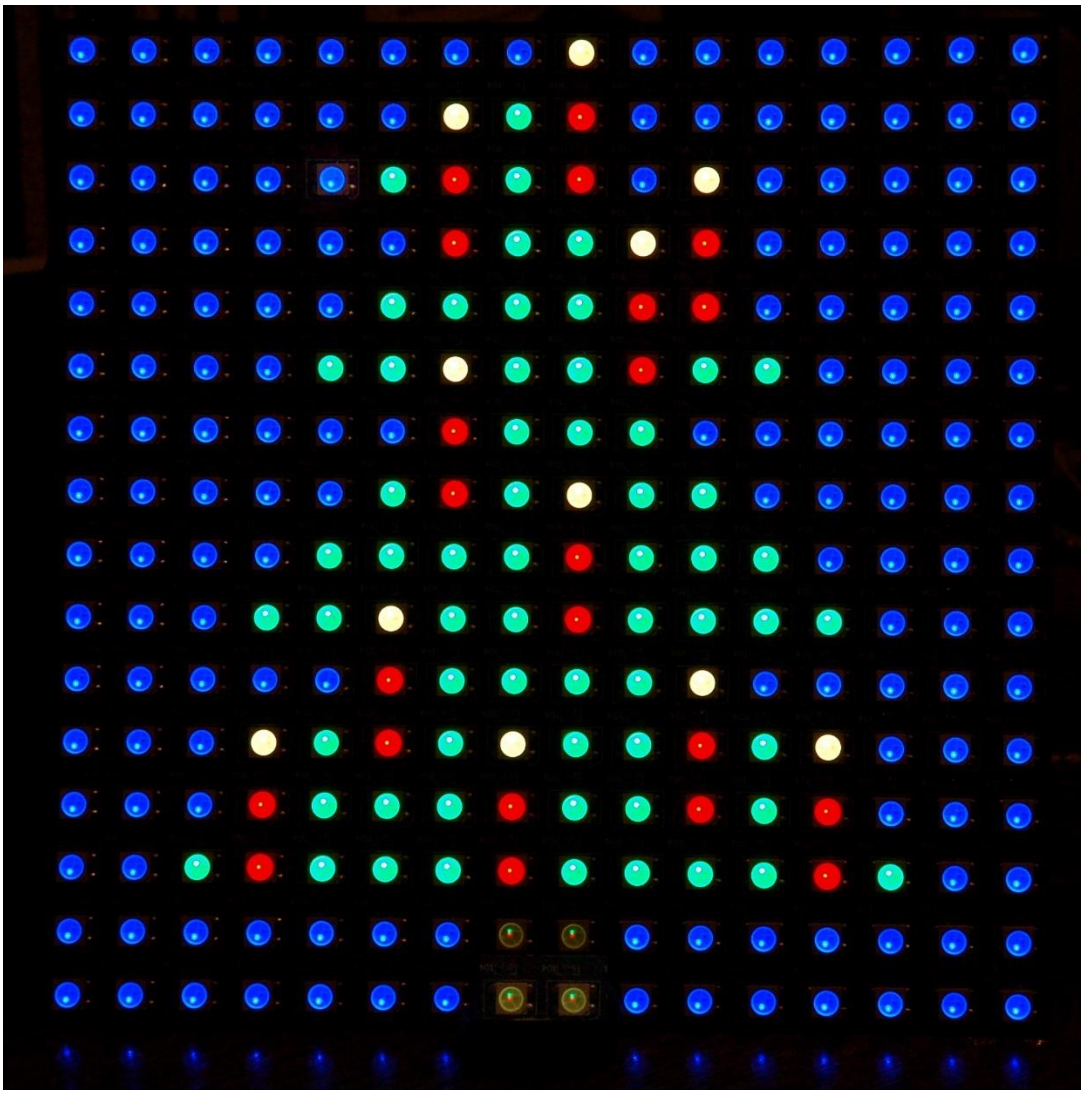

Abbildung 10: Baum mit Kerzen

Die Funktion **kerze**() baut aus den geometrischen Grundelementen **fill**(), **rectangle**() und **setPixel**() das Bild einer Kerze auf.

```
def kerze(fak):
    fill(0,255,(0,0,32),fak)
    rectangle(226,11,2,(16,16,0),fak)
    rectangle(116,7,7,(128,0,0),fak)
    for p in (84,100,101,106):
        setPixel(p,128,0,0,fak)
    for p in (84,100,101,106):
        setPixel(p,128,0,0,fak)
    for p in (70,71,72,87):
        setPixel(p,128,32,0,fak)
    for p in (55,56):
        setPixel(p,128,64,0,fak)
    for p in (38,39):
        setPixel(p,128,128,0,fak)
    for p in (7,23):
        setPixel(p,128,128,32,fak)
```

Die Struktur des Sterns ist erheblich komplexer als die bisherigen Bilder. Dennoch führt die Definition seines Outfits durch die Funktionen **stern**() und **showFamily**() über die Liste **family** zu einem überraschend kurzen Code in den Funktionen.

```
def showFamily(n,col,fak):
    for pos in family[n]:
        r,g,b=color[col]
        setPixel(pos-1,r,g,b,fak)
```

Der Stern ist wie ein Oger, und Oger sind wie Zwiebeln, sie haben Schalen (Ohh Shrek!). Die Schalen heißen bei mir Families. Die Funktion **showFamily**() bringt nun genau eine Schale, deren Nummer in **n** übergeben wird, zur Anzeige und zwar in der Farbe, die im Tuple **col** steckt.

Die Funktion **stern**() baut das Bild erst einmal auf, um nachfolgend die Farbe der Schalen noch einmal zu verändern.

```
def stern(fak):
    for fam in range(10):
        for frame in range(fam):
            showFamily(frame,fam-frame,fak)
    for col in range(9,13):
        for fam in range(9):
            showFamily(fam,13-(col-fam),fak)
```

Abhängig davon, ob ein WLAN zur Verfügung steht, gibt die Funktion **getDayTime**() ein Tuple mit dem Tagesdatum und der Uhrzeit zurück. Mit Netzzugang holt die Funktion sich die Information aus der Systemzeit des ESP32. Diese wird in Intervallen, welche die Variable **syncIntervall** in Millisekunden vorgibt, durch Kontaktaufnahme zu einem Zeitserver sychronisiert. Dazu gleich mehr.

```
syncIntervall=600000 # ms
```

Gibt es keinen Netzzugang, dann wird auf die Real Time Clock (RTC) des ESP32 zurückgegriffen. Natürlich muss die RTC erst einmal justiert werden. Dafür ist die Variable **rtcTag** zuständig.

```
rtcTag=(2022,12,14,8,23,12,0,0) # RTC-Tagesstempel Start
# Jahr,Monat,Tag,Stunde,Minute,Sekunde,Wochentag,Jahrestag
```

Das 8-Tuple wird vor dem Programmstart mit den Daten des Startzeitpunkts versorgt. Die Sortierung des RTC-Datensatzes ist eine andere als die der Systemzeit (warum auch immer?), und deswegen müssen die Felder umsortiert werden.

```
def getDayTime():
    if nic.status() in (5,1010):
        return localtime(time()+timeZone*3600)
    yr,mon,day,dow,hor,minute,sec,ms=rtc.datetime()
    return(yr,mon,day,hor,minute,sec,dow)
```

Die Funktion **TimeOut**() ist etwas ganz spezielles. Sie gibt keinen Wert zurück, was Funktionen mitunter tun (siehe **getDayTime**()), sondern die Funktion **compare**(), die im Funktionskörper definiert wird. Genau genommen wird nicht die Funktion, sondern eine Referenz darauf zurückgegeben. Alles, was innerhalb einer Funktion an Objekten und Variablen deklariert wird, ist im Bezug auf die Funktion lokal und außerhalb der Funktion nicht referenzierbar, also quasi nicht erreichbar. Wird die Funktion verlassen, sterben alle lokal definierten Objekte. Das wird bei **TimeOut**() dadurch umgangen, dass die innerhalb definierte Funktion **compare**() auf den Parameter **t** und die außerhalb von **compare**() definierte, zu **TimeOut**() lokale Variable **start** zugreift. Aus der Funktion **TimeOut**() wird damit eine sogenannte Closure.

Durch diesen Klimmzug ist es mir möglich, beliebig viele, einfach zu verwaltende Softwaretimer in meine Programme einzubauen. Jeder Timer arbeitet unabhängig von den anderen im Hintergrund, blockiert also den Programmablauf in keiner Weise. Erst beim Aufruf der Referenz auf die zurückgegebene Funktion erwacht diese aus ihrem Dornröschenschlaf und liefert die Information, ob der Timer abgelaufen ist oder nicht. Leider ist diese Art Timer nicht interruptfähig. Mit **TimeOut**() eingerichtete Timer können ein laufendes Programm nicht durch eine Unterbrechungsanforderung (IRQ = Interrupt Request) unterbrechen und müssen deshalb wiederholt abgefragt werden.

```
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare
```

Nach dem Bestücken der Werkzeugkiste mit Objekten und Funktionen geht es jetzt an die Arbeit. Wir versuchen erst einmal, den ESP32 ans Netz zu bringen. Dazu brauchen wir das Station-Netzwerk-Interface (STA). Damit das Accesspoint-Interface nicht stört, was beim ESP8266 gern der Fall ist, würge ich es erst einmal sicher ab. Dann instanziiere ich das STA-Interface und aktiviere es.

```
nic = network.WLAN(network.AP_IF)  # AP-Interface-Objekt
nic.active(False)                  # sicher ausschalten
nic = network.WLAN(network.STA_IF) # WiFi-Objekt erzeugen
nic.active(True)                   # STA-Objekt einschalten
```

Die Methode **config**() mit dem Argument **'mac'** (als String!) aufgerufen, liefert mir die MAC-Adresse des STA, die ich mir von **hexMac**() in Klartext übersetzen lasse. Die MAC-Adresse müssen Sie im Router eintragen, damit der dessen Türsteher den ESP32 dann später auch den Club betreten lässt.

```
MAC = nic.config('mac')    # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)          # in eine Hexziffernfolge umwandeln
print("STATION MAC: \t"+myMac+"\n") # ausgeben
```

Falls der ESP32 selbst, ohne unser Zutun, noch keine Verbindung zum WLAN-Router aufgebaut hat (der ESP8266 tut das gerne beim Booten ohne unser Zutun), leiten wir den Aufbau einer Verbindung ein. Haben Sie eingangs Ihre Credentials richtig gesetzt, die SSID des Routers und das Passwort? Hier wird jetzt beides gebraucht.

```
clearPanel()
if not nic.isconnected():
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySSID, myPass)
  # warten bis die Verbindung zum Accesspoint steht
  print("connection status: ", nic.isconnected())
  n=0
  while (nic.status() != network.STAT_GOT_IP) and (n < 10):
    print(".",end='')
    np[n]=color[n]
    np.write()
    sleep(1)
    n+=1
```

Ich lösche das LED-Panel, um damit den Aufbau der Verbindung zu verfolgen. Solange die Methode **status**() nicht den Wert **1010** = **STAT_GOT_IP** zurückgibt, wird im Sekundenabstand im Terminal ein Punkt ausgegeben und eine LED mehr im Panel eingeschaltet. Der Zähler **n** dient zur Adressierung und zur Farbwahl aus der Palette **color**.

Im Terminal lasse ich mir den Verbindungsstatus und die Verbindungsdaten ausgeben, nachdem wenigstens 10 Sekunden vorbei sind.

```
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
#STAconf = nic.ifconfig((myIP,"255.255.255.0",myGW,myDNS))
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",STAconf[1],\
      "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
```

Jetzt geht es ans Setzen von Datum und Uhrzeit. Wenn dem ESP32 vom DHCP-Server des Routers eine IP-Adresse zugewiesen wurde, besteht eine Netzwerkanbindung. Dann wird versucht, die Systemzeit mit dem Zeitserver zu synchronisieren. Schlägt das aus irgendeinem Grund fehl, stelle ich die RTC mit dem vorgegebenen Tuple **rtcTag** ein.

```
if nic.status() in (5,1010):
    try:
        ntptime.settime()
        print("Synchonized",localtime(time()+timeZone*3600))
    except:
        rtc.datetime(rtcTag)
        print("RTC-Time set",rtc.datetime())
```

Die Funktion **getDayTime**() übergibt das aktuelle Zeit-Tuple an **dayTime**. Das Feld mit dem Index 3 enthält die aktuelle Stunde. Ich gebe den Wert an **newHour** weiter. **oldHour** setze ich einfach auf 0. Die beiden Variablen dienen in der Hauptschleife dazu, einen Stundenwechsel zu erkennen, der unter normalen Bedingungen sicher nicht um Mitternacht stattfinden wird (oder sind Sie ein Workaholic wie ich?).

```
dayTime=getDayTime()
newHour=dayTime[3]
oldHour=0
# sleep(3)
clearPanel()
```

Der Timer für die Zeitsynchronisation wird scharf gemacht.

```
synchronize=TimeOut(syncIntervall)
```

synchronize referenziert die Funktion **compare**(), welche **True** zurückgibt, wenn das Zeitintervall abgelaufen ist.

In der Hauptschleife prüfe ich als Erstes, ob eine Netzwerkverbindung besteht und ob der Timer für die Zeit-Synchronisation abgelaufen ist. In diesem Fall versuche ich den Zeitserver abzufragen, lasse dann die Systemzeit zur Kontrolle ausgeben und stelle den Timer neu.

```
while 1:
    if nic.status() in (5,1010) and synchronize():
        try:
            ntptime.settime()
            print("Synchonized",localtime(time()+\
                                        timeZone*3600))
            synchronize=TimeOut(syncIntervall)
        except:
            pass
```

Die aktuelle Stunde wird ausgelesen. Ein Stundenwechsel hat stattgefunden, wenn die Werte von **newHour** und **oldHour** nicht übereinstimmen. Wenn jetzt der Stundenwert auch noch im Bereich **stunden** liegt, den ich bei den vorbereitenden Maßnahmen eingerichtet habe, und der DF-Player grade nichts zu tun hat, lasse ich einen Titel auswürfeln und beauftrage den Player denselben abzuspielen. **played** wird auf **True** gesetzt und **oldHour** wird aktualisiert.

```
    newHour=getDayTime()[3]
    if (newHour != oldHour) \
        and (newHour in stunden) \
        and not played:
        n = random.choice(titles)
        df. play(0,n)
        played=True
        oldHour = newHour
```

Die Variable **figur** hatte ich vor dem Eintritt in die Hauptschleife auf 0 gesetzt. Also wird beim ersten Durchlauf der Baum am Panel auftauchen. Drei Sekunden Pause, dann Panel löschen.

```
    if figur == 0:
        baum(0.2)
    elif figur == 1:
        kranz(0.2)
    elif figur == 2:
        kerze(0.2)
    elif figur == 3:
        stern(0.2)
    sleep(3)
    clearPanel()
```

Wenn nun irgendwann das Musikstück beendet ist, und **played** noch auf **True** steht, setze ich die Variable auf **False** und bereite damit das Abspielen einer weiteren Datei in etwa einer Stunde vor.

```
    if not df.isPlaying() and played:
        played=False
    figur += 1
    figur %= 4
```

Abschließend wird der Ringzähler **figur** erhöht oder auf 0 gesetzt.

Jetzt steht einem Test nichts mehr im Weg, und ich bin gespannt, ob Ihr LED-Panel genauso reagiert wie meines. Beim Erzeugen der Lichtbilder blitzen, wie weiter oben schon beschrieben, unvermittelt weitere LEDs kurz in wechselnden Farben und unterschiedlicher Helligkeit auf, so als würde es schneien. Das Endprodukt ist in den meisten Fällen dann OK und entspricht dem programmierten Bild. Was diesen Effekt hervorruft, konnte ich auch nach längeren Versuchsreihen nicht feststellen. Bei anderen Neopixel-Modulen trat er bisher nicht auf. Auch beim Löschen der Anzeige bleiben manchmal einzelne Pixel stehen, was nicht verwunderlich ist, da auch in diesem Fall alle LEDs angesteuert werden, halt mit den Tuple (0,0,0). Vielleicht ist das Leitergeflecht auf dem Montage-Untergrund nicht unschuldig daran. Schließlich funkt der ESP32 mit 800kHz, und das locker im Mittelwellenbereich. Somit könnte das Leitergeflecht des Panels Signale auffangen, die gar nicht für die LED gedacht sind, welche sie zufällig empfängt. Sei's drum.

Ich wünsche Ihnen viel Freude beim Erstellen neuer Figuren. Ein Tabellen-Programm wie Libre Office ist ein sehr brauchbares Werkzeug dafür. Ich habe die Felder durchnummerieren lassen und färbe sie dann einfach mit Hintergrundfarbe ein, um ein Muster zu entwerfen.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
79	78	77	76	75	74	73	72	71	70	69	68	67	66	65	64
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
111	110	109	108	107	106	105	104	103	102	101	100	99	98	97	96
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
143	142	141	140	139	138	137	136	135	134	133	132	131	130	129	128
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
175	174	173	172	171	170	169	168	167	166	165	164	163	162	161	160
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
207	206	205	204	203	202	201	200	199	198	197	196	195	194	193	192
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
239	238	237	236	235	234	233	232	231	230	229	228	227	226	225	224
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

Abbildung 11: Der Stern in Libre Office

Schöne Adventszeit und Frohe Weihnachten!