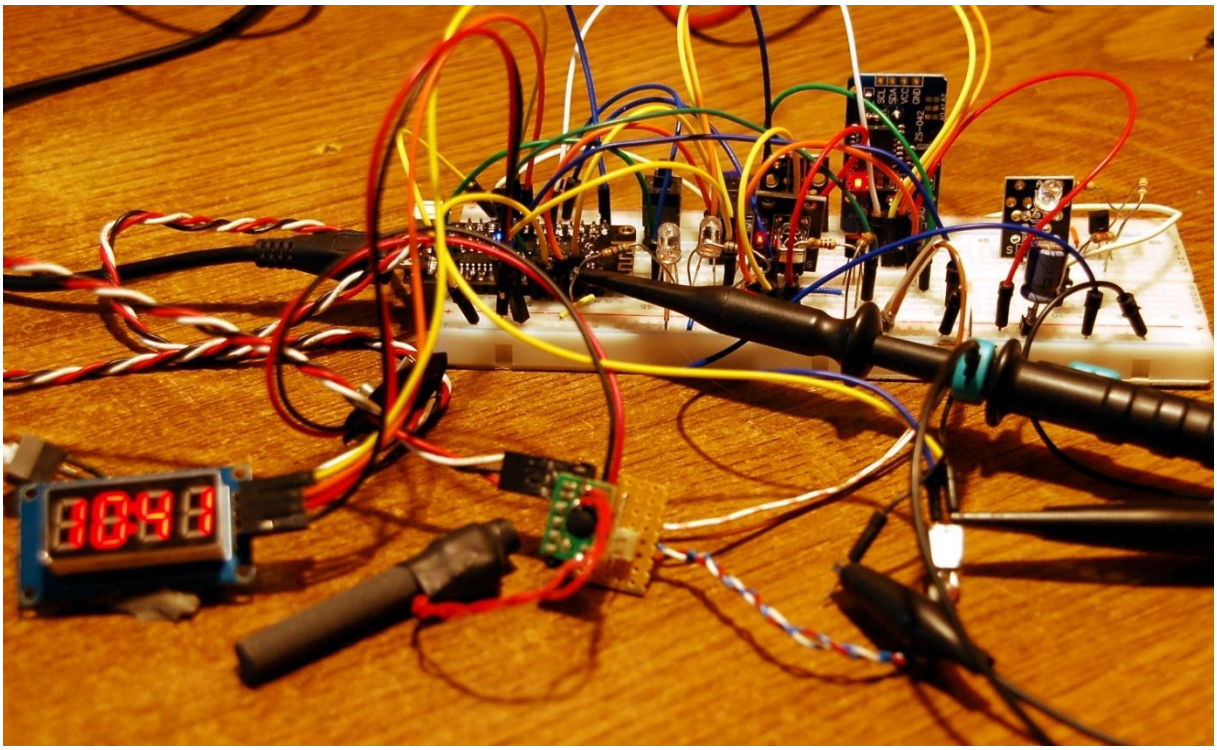

DCF77-Empfangsmodul am ESP32

Diesen Beitrag gibt es auch als PDF-Dokument zum Download.

Willkommen zu einer neuen Folge von

MicroPython auf dem ESP32 und ESP8266

heute

Ein Funkwecker, der RC5 spricht

Wer die vorangegangenen Beiträge verfolgt hat, weiß was heute auf der Tagesordnung steht. Für alle anderen hier eine kleine Zusammenfassung.

In der ersten Folge haben wir dem ESP32 beigebracht, eine RC5-IR-Fernsteuerung auszulesen. Im zweiten Beitrag, sendet der Controller selbst RC5-IR-Code . Dann lernte der ESP32, wie er eine PS/2-Tastatur, abfragen kann. Eine RTC, mit guter Ganggenauigkeit bekam der ESP32 in einer weiteren Folge spendiert, nachdem die bordeigene Real Time Clock alles andere als exakt läuft. Ein großes LED-Display kam in Folge 5. Zuletzt sorgten wir mit einem DCF77-Modul für den Kontakt zum Zeitzeichensender der PTB (Physikalisch Technische Bundesanstalt).

Ja, und heute werden wir die ganzen erarbeiteten Module zu einem Wecker zusammenbauen, der sich automatisch mit der amtlichen Zeit synchronisiert, der den RC5-Code unserer Infrarot-Remote-Control auslesen und in einer Datei abspeichern und nach dem Empfang eines Befehls vom Handy aussenden kann. Das Handy ersetzt somit auch die RC. Und der Wecker klingelt nicht, sondern schaltet zum Wecken die Stereoanlage ein. Natürlich ist es auch denkbar, dass ein

ESP32/ESP8266 als Empfänger der RC5-Codes aufgebaut wird und über ein Relais weitere Geräte schaltet.

Hardware

Da zur Hardware keine weiteren Teile hinzugekommen sind, habe ich die Liste von der letzten Folge übernommen.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	KY-022 Set IR Empfänger
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Breadboard Kit - 3x Jumper Wire m2m/f2m/f2f + 3er Set MB102 Breadbord kompatibel mit Arduino und Raspberry Pi - 1x Set
1	KY-004 Taster Modul
diverse	Jumper Wire Kabel 3 x 40 STK
1	Real Time Clock RTC DS3231 I2C Echtzeituhr
1	TM1637 4 Digit 7-Segment LED-Display Modul
1	KY-018 Foto LDR Widerstand Photo Resistor Sensor
1	DCF77-Empfänger-Modul
2	NPN-Transistor BC337 oder ähnlich
1	Widerstand 1,0 kΩ
1	Widerstand 10 kΩ
1	Widerstand 330 Ω
1	Widerstand 47Ω
1	Widerstand 560Ω
1	LED (Farbe nach Belieben)
1	Adapter PS/2 nach USB oder PS/2-Buchse
1	Logic Analyzer
1	PS/2 - Tastatur

Abbildung 1 zeigt die Schaltung mit allen Teilen.

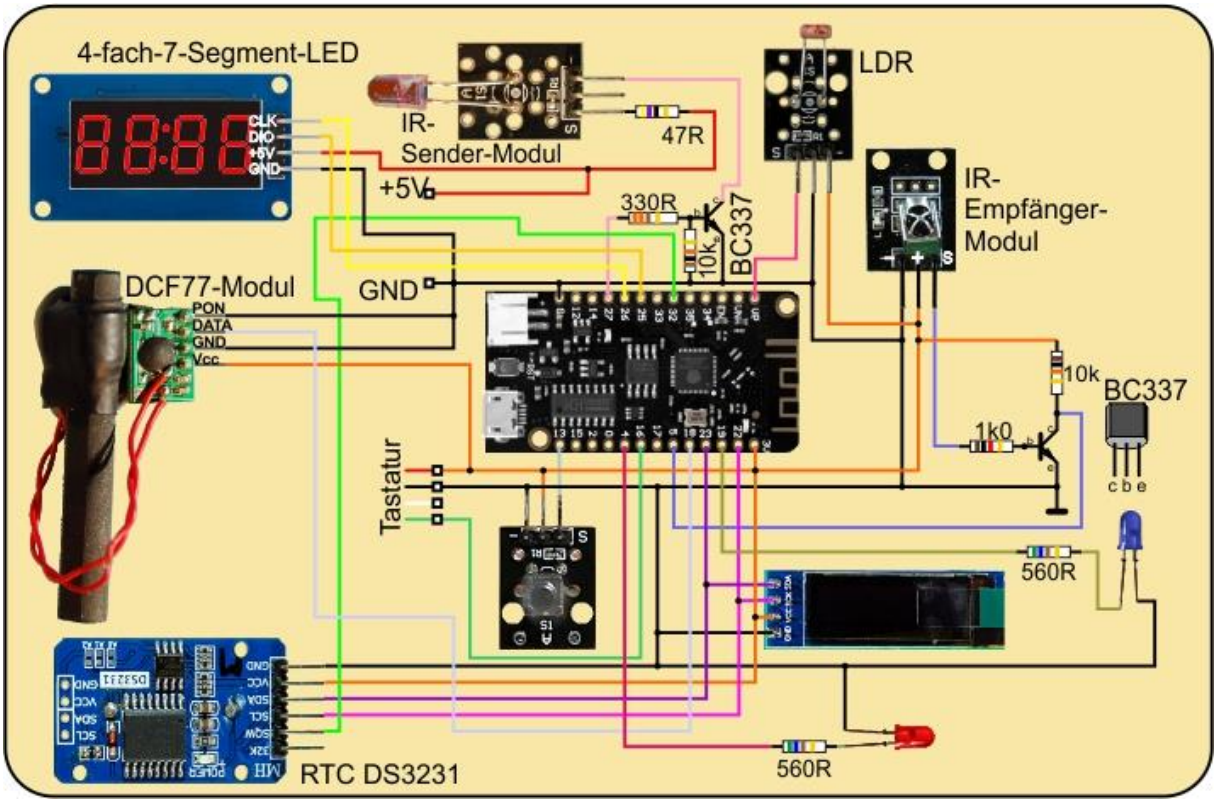

Abbildung 1: Alles zusammen = Funkuhr mit IR-RC-Ambitionen

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft
Betriebs-Software Logic 2 von SALEAE
packetsender.exe: Netzwerkterminal für TCP und UDP

Verwendete Firmware für den ESP32:

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

Verwendete Firmware für den ESP8266:

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

Die MicroPython-Programme zum Projekt:

tm1637_4.py: API für die 4- und 6-stellige 7-Segment-Anzeige mit dem TM1637
ds3231.py: Treiber-Modul für das RTC-Modul
oled.py: OLED-API
ssd1306.py: OLED-Hardware-Treiber
buttons.py: Modul zur Tastenabfrage

dcf77.py: Treiber für das DCF77-Modul
ir_rx-small.zip: Paket zum IR-Empfangs-Modul
irsend.py: IR-Sende-Modul
timeout.py: Softwaretimer
sync_it.py: Programm zum Synchronisieren mit DCF77
sekundenalarm.py: Demoprogramm zum Alarm auslösen
lern.py: Auslesen von RC5-Codes

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.

Das Wecker-Programm

Neben den in den, in den vergangenen Folgen, besprochenen Modulen importieren wir **network** und **socket**, die beiden brauchen wir für den WLAN-Verkehr mit dem Handy, das wir momentan noch mit dem Programm Packetsender auf dem PC ersetzen.

```
# wecker.py
#
from dcf77 import DCF77
from ds3231 import DS3231
from machine import Pin,SoftI2C, Timer
from oled import OLED
from tm1637 import TM1637
from sys import exit
from irsend import IRSEND
from esp32 import RMT
import network, socket
from time import sleep
```

Für den Fall, dass unser Wecker später als Accesspoint arbeiten soll vergeben wir folgende Interfacedaten.

```
interface="AP"
if interface=="AP":
    # ***************** als Accesspoint ******************
    client=("192.168.0.2",9091) # Handy
    myIP="192.168.0.1"
    myPort=9091
    server=(myIP,myPort) # ESP32
    # **** Geben Sie hier Ihre eigenen Zugangsdaten an ****
    mySSID = 'ANTARES99' # Credentials ESP32
    myPass = 'greenCacadu'
```

Man kann aber dann nicht mit **packetsender** darauf zugreifen. Deshalb benutzen wir in der Testphase erst einmal den WLAN-Router zum Aufbauen eines Kontakts. Denken Sie bitte daran, den ESP32 mit dessen MAC-Adresse am Router anzumelden.

```
elif interface=="STA":
    # ******************* als Station *********************
    myIP="10.0.1.96"
    myPort=9091
    # **** Geben Sie hier Ihre eigenen Zugangsdaten an ****
```

```
    mySSID = 'EMPIRE_OF_ANTS' # Credentials Router
    myPass = 'nightingale'
```

Wir erzeugen ein I2C-Objekt und legen die Frequenz auf sichere 100kHz fest. Das Flag **alarmTrigger** wird später abgefragt, wir deklarieren es bereits hier und setzen es auf False. An **GPIO13** liegt unsere Taste, die diversen Zwecken dient. Auch die Liste für den Timestamp deklarieren wir hier schon indem wir eine List-Comprehension verwenden. Sie erzeugt eine Liste mit 7 Nullen.

```
i2c=SoftI2C(scl=Pin(22),sda=Pin(23),freq=100000)
alarmTrigger=False
taste=Pin(13,Pin.IN,Pin.PULL_UP)
dt=[0 for _ in range(7)]
```

Mit der I2C-Instanz erzeugen wir ein OLED-Objekt und ein DS3231-Objekt, das unsere RTC (Real Time Clock) verkörpert.

```
d=OLED(i2c,heightw=32)
rtc=DS3231(i2c)
```

Das DCF77-Objekt wird seine Signale an **GPIO18** senden und die LEDs an **GPIO4** (rot) und **GPIO19** (blau) nutzen.

```
dcf=DCF77(dcf=18,sec=4,wait=19)
```

Die LED-Anzeige wird über **GPIO26** (clk) und **GPIO25** (dio) angesteuert. Ordentlich, wie wir sind, befreien wir das Display von Altlasten.

```
tm=TM1637(Pin(26),Pin(25))
tm.clearDisplay()
```

An **GPIO27** liegt wird das RMT-Modul seine 38-kHz-Bursts abgeben, um damit die IR-Sende-LED über den Transistor anzusteuern. Der Dutycycle (Puls-Periodendauer-Verhältnis) ist 50%, Puls und Pause haben also die gleiche Länge.

```
rmtPin = Pin(27, Pin.OUT, value = 0)
cfreq=38000
duty=50
rmt = RMT(0, pin=rmtPin,
          idle_level=0,
          clock_div=80,
          tx_carrier = (cfreq, duty, 1))
ir=IRSEND(rmt)
codes={}
```

Die Parameter, die wir dem Konstruktor **RMT**() übergeben sind im Beitrag ir-sender_ger.pdf erklärt. Insgesamt entsteht am Kollektor des BC337 ein Signal mit 3,3V-Ruhepegel, dessen Einsen durch 38kHz-Down-Bursts codiert sind.

Mit **codes** bereiten wir den Container für die RC5-Codes vor, die später von der Datei **befehle.cfg** eingelesen werden. Entstanden ist diese Datei im Flash des ESP32 in der ersten Folge durch den Lauf des Programms **lern.py**.

Wir wollen Befehle per UDP-Protokoll an den ESP32 senden. Dafür brauchen wir eine WLAN-Verbindung. Während der Entwicklungszeit der Handy-App läuft das optimal über den WLAN-Router des Heimnetzes. Später darf der ESP32 selbst Accesspoint in einem Inselnetz spielen.

Über den Status des Verbindungsaufbaus informiert uns das Dictionary **connectstatus**, das die Nummerncodes der Schnittstelle in Klartext übersetzt. Sie ist sowohl für ESP32 als auch ESP8266 geeignet.

```
connectStatus = {
    1000: "STAT_IDLE",  # ESP32
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "UNKNOWN",
    0: "STAT_IDLE",  # ESP8266
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
    4:  "STAT_CONNECT_FAIL",
    }
```

Für den Kontakt zum WLAN-Router ist in der Regel die MAC-Adresse des Station-Interfaces des ESP32 vonnöten. Das ist dann der Fall, wenn der Router mit **MAC-Filtering** arbeitet. Dann muss die MAC des ESP32 in der Filtertabelle des Routers eingetragen sein, damit der Türsteher unseren Controller reinlässt. Meist ist der Eintrag über den Menüpunkt **WLAN – Sicherheit** des Routers möglich. Die Funktion **hexMac**() sagt uns die MAC im Klartext.

```
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
    val="{:02X}".format(byteMac[i])
    macString += val
    if i <len(byteMac)-1 :            # Trennzeichen
      macString +="-"                 # bis auf letztes Byte
  return macString
```

Der Parameter **byteMac** bekommt die MAC-Adresse als bytes-Objekt, das recht kryptisch aussehen kann. Wir legen einen leeren String vor und klappern dann **byteMac** Byte für Byte in der for-Schleife ab.

Mit Hilfe des Formatstrings **{:02X}** wandeln wir den Bytewert in einen zweistelligen Hexadezimalzahlen-String um, den wir an den bisherigen Ergebnisstring dranpappen. Bis auf die letzte Stelle fügen wir noch ein "-" als Trennzeichen dazu.

Den Verbindungsaufbau habe ich durch Funktionen codiert. Damit werden die Anwendung und der Austausch flexibler. Beginnen wir mit dem Accesspoint. Wir erzeugen ein Interface-Objekt und aktivieren es.

```
def setAccessPoint():
    # ************** Accesspoint aufsetzen
    nic=network.WLAN(network.AP_IF)
    nic.active(True)
```

Dann fragen wir die MAC-Adresse ab und lassen sie ausgeben. Die Pause vor dem Konfigurieren der Schnittstelle hat sich als notwendig herausgestellt, um Fehlfunktionen zu vermeiden.

```
    MAC = nic.config('mac')# binaere MAC-Adresse abrufen und
    myMac=hexMac(MAC)      # in Hexziffernfolge umwandeln
    print("STATION MAC: \t"+myMac+"\n") # ausgeben
    sleep(1)
    nic.ifconfig((myIP,"255.255.255.0",myIP,myIP))
    nic.config(essid=mySSID, password=myPass)
```

Dann bereiten wir einen String mit 10 Punkten vor und setzen den Durchlaufzähler **n** auf 1. Mit der while-Schleife warten wir bis das Accesspoint-Interface als aktiv gemeldet wird und geben im Sekundenabstand jeweils einen Punkt mehr in REPL und im Display aus.

```
    points="..........."
    n=1
    while not nic.active():
        print(".",end='')
        d.writeAt(points[0:n],0,2)
        n+=1
        sleep(1)
    print("NIC active:",nic.active())
    d.clearAll()
```

Hat alles geklappt, fragen wir die Konfiguration ab und lassen sie anzeigen.

```
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1], "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    print()
    d.writeAt(STAconf[0],0,0,False)
```

```
    d.writeAt(STAconf[1],0,1,False)
    d.writeAt(STAconf[2],0,2)
    return nic
```

Damit das Hauptprogramm auf das Interface-Objekt zugreifen kann, geben wir es zurück.

Das Erzeugen eines Station-Interface-Objekts läuft ähnlich ab. Damit das Accesspoint-Interface uns nicht in die Suppe spuckt, schalten wir es definitiv aus.

```
def connect2router():
    # ************** Zum Router verbinden
    nic=network.WLAN(network.AP_IF)
    nic.active(False)
```

Was dann folgt, ist mit dem Accesspoint-Interface nahezu identisch.

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

Ein Socket-Objekt ist auf Netzwerkebene das, was ein UART-Objekt für die serielle Datenverbindung via RS232 ist, quasi das Tor, durch welches der Datenaustausch stattfindet. Damit verbunden ist eine Empfangsschleife, die ankommende Zeichen in einen Empfangspuffer schiebt, aus dem wir sie dann abholen können.

Da unser Transfer auf dem UDP-Protokoll basieren soll, sagen wir dem Konstruktor des Socket-Objekts, dass wir die IP-V4-Familie (AF_INET) verwenden wollen, und dass Datagramme (SOCK_DGRAM) ausgetauscht werden sollen.

Für die Übertragung von Webseiten würde das TCP-Protokoll eingesetzt, das auf Datenstreams setzt. Für diesen Fall steht die auskommentierte Zeile zur Verfügung.

Während der Entwicklungsphase muss das Programm öfters hintereinander gestartet werden, möglichst ohne den ESP32 neu zu booten. Damit das jedes Mal ohne Fehlermeldung geschehen kann, sagen wir dem Socket-Objekt, dass wir die Wiederverwendung von IP-Adresse und Portnummer wünschen (**SO_REUSEADDR**).

```
def setSocket():
    # ***************** Socket aufsetzen
#     sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    sock.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
    sock.bind(('', myPort))
    print("sending on port",myPort)
    sock.settimeout(0.1)
    return sock
```

Dann binden wir die Portnummer an die bereits vergebene IP-Adresse. Damit die Empfangsschleife des Sockets uns nicht die Hauptprogrammschleife blockiert, vereinbaren wir einen Timeout von 0,1 Sekunden. Dadurch wird die Empfangsschleife zwar sicher aufgerufen, wenn allerdings keine Daten eingetrudelt sind, wird das Warten auf solche nach 100ms abgebrochen. Natürlich muss auch das Socket-Objekt zurückgegeben werden, weil es im Hauptprogramm gebraucht wird.

An den ESP32 sollen verschiedene Nachrichten oder Befehle versandt werden. Der Parser, die Funktion **parse**(), versucht die Syntax abzuklappern und in Folge geeignete Aktionen auszulösen. Sollte etwas schiefgehen, setzen wir die Nachricht, die zurückgegeben wird, vorsichtshalber schon mal auf "Fehler".

```
def parse(rec):
    msg="Fehler"
```

Zwei Befehlsformate sind implementiert.

RC:code
Hole den RC5-Code, der dem Schlüssel **code** entspricht aus dem Befehls-Dictionary **codes** und sende es über die IR-LED

ALARM:60
Setze jede volle Minute einen Alarm ab.

ALARM:minute
Setze jede Stunde zur selben Minute einen Alarm ab.

ALARM:stunde,minute
Setze täglich zur selben Zeit einen Alarm ab.

Wir stellen erst einmal fest, ob der Befehlsstring einen Doppelpunkt enthält. Ist das der Fall, dann meldet **find**() seine Position zurück, andernfalls -1. Wir splitten den Record am Doppelpunkt auf in Kommando **cmd** und Datenteil **data**. Beides wird in Großbuchstaben konvertiert, von **data** entfernen wir ein fakultatives Newline oder Carriage return.

```
    if rec.find(":") != -1:
        cmd,data=rec.split(":")
        cmd=cmd.upper()
        data=data.upper().strip("\n\r")
```

Das Kommando prüfen wir auf "RC". Ist bei Übereinstimmung auch noch der Inhalt von **data** in **codes** zu finden, dann holen den RC5-Code ab und senden ihn zweimal an die IR-LED.

```
        if cmd=="RC":
            if data in codes:
                code=codes[data]
                sendeRCcode(code,2)
                return "gesendet"
```

Wurde "ALARM" gesendet, dann müssen verschiedene Fälle unterschieden werden. Wir suchen zuerst einmal nach einem Komma. Ist keines in **data** enthalten, dann kann **data** den String **OFF** enthalten, wonach ein gesetzter Alarm2 disabled wird.

```
        elif cmd=="ALARM":
            if data.find(",") == -1:  # nur ein Byte
                print(data)
                if data == "OFF":
                    rtc.AlarmAus(2) # Alarm2 disable
                    msg= "disabled"
                    print("Alarm2 aus")
                    return msg
                zeit=int(data)
```

Wir wandeln den String in **data** in eine Zahl um und weisen diese der Variablen **zeit** zu. Ist **zeit** gleich 60, wird der Alarm zu jeder vollen Minute gesetzt.

```
                if zeit == 60:
                    rtc.Alarm2(0,0,0,rtc.jedeMinute)
                    msg= "jede Minute"
```

Liegt **zeit** im Bereich von 0 bis 59 incl., dann wird der stündliche Alarm zu dieser Zeit gesetzt.

```
                elif zeit in range(60):
                    rtc.Alarm2(0,0,int(zeit),
                               rtc.MinutenAlarm) #stuendlich
                    msg= "stuendlich bei {} Min.".\
```

```
                            format(zeit)
            else:
```

Wurde ein Komma in **data** entdeckt, dann liegt eine Angabe von Stunden und Minuten vor. Die Anteile bekommen wir durch Splitten am Komma. Die Integerwerte daraus senden wir an die RTC für einen täglichen Alarm zur festgesetzten Zeit.

```
                h,m=data.split(",")
                rtc.Alarm2(0,int(h),int(m),
                           rtc.StundenAlarm) #taeglich
                msg= "taeglich um {}:{}.".format(h,m)
            rtc.ClearAlarm(2)
            return msg
    elif rec.find("-") != -1:
        data=rec[1:].strip("\n\r")
        code=codes[data]
        sendeRCcode(code,2)
        return "sent" + data
```

Wenn ein Alarm ausgelöst werden soll, braucht der ESP32 einen GPIO-Eingang für die Leitung SQW vom DS3231. Für diese Leitung wird ein IRQ freigegeben, dessen ISR **alarmCallback**() ist. Der Trigger wird auf fallende Flanke gesetzt. Im Handler wird **alarmTrigger** als global declariert, damit der auf True gesetzte Wert an das Hauptprogramm weitergegeben werden kann. Eine ISR kann keinen Wert mit **return** zurückgeben, an wen denn auch? Die Routine wurde ja von keinem Programmteil, sondern von der Hardware aufgerufen.

```
def alarmCallback(pin):
    global alarmTrigger
    alarmTrigger=True

rtcIRQ=Pin(32, Pin.IN)
rtcIRQ.irq(handler=alarmCallback, trigger=Pin.IRQ_FALLING)
```

Die Ansteuerung des LED-Displays erledigt die Funktion **timeOutput**(). Sie liest die RTC aus, gibt die Liste aus und pickt sich dort Stunde und Minute heraus. Die Werte werden in Dezimalziffern zerlegt und die korrespondierenden Segmentmuster zu einer Liste zusammengesetzt. Die Elemente dieser List senden wir in der for-Schleife an das entsprechende Segment.

```
def timeOutput():
    global dt
    dt=rtc.DateTime()
    print(dt)
    hour=dt[4]
    minute=dt[5]
    zeit=[tm.Segm[hour//10],tm.Segm[hour%10]|0x80,
          tm.Segm[minute//10],tm.Segm[minute%10],]
    for i in range(4):
        tm.segment(zeit[i],i)
```

Die Vorgänge einer Zeitsynchronisation mit dem DCF77 fasst die Funktion **sync**() zusammen. Damit das Multiplexing der Anzeige den Empfang nicht stört, löschen wir das Display und rufen dann **dcf.synchronize**() auf. Nach spätestens 2 Minuten sollte der Vorgang abgeschlossen sein. Den empfangenen Timestamp vom Sender schreiben wir in die RTC und geben die Zeit am Display aus.

```
def sync():
    print("warte auf Minutenstart")
    tm.clearDisplay()
    dt=dcf.synchronize()
    rtc.DateTime(dt)
    timeOutput()
    print("synchronisiert")
```

Kommando und Adresse eines RC5-Codes müssen an **ir.transmit**() diskret übergeben werden. Deshalb wird in **sendeRCcode**() das Code-Tupel aus dem Dictionary **codes** in seine Einzelteile aufgeteilt und dann gesendet.

```
def sendeRCcode(code,rep):
    addr,cmd,_=code
    ir.transmit(int(addr),int(cmd),rep)
```

In der ersten Folge dieser Reihe hatten wir die Datei **befehle.cfg** im Flash des ESP32 angelegt. Die zapfen wir jetzt an, um damit das Dictionary **codes** zu füllen. Wir legen ein leeres Dictionary vor und geben im OLED-Display kund und zu wissen, was gerade läuft.

```
def readData():
    codes={}
    d.clearAll()
    d.writeAt("TRY READING",0,0,False)
    d.writeAt("befehle.cfg",0,1)
```

Die Dateioperationen kapseln wir in einer try–except-Struktur, um eine Fehlermeldung zu erhalten, falls etwas schiefläuft. Wir öffnen die Datei unter dem Handle **f** in einem with-Block, das spart ein **close**(f) am Schluss. Durch with wird die Datei automatisch beim Verlassen des Blocks geschlossen.

Zeile für Zeile wird eingelesen und von Steuerzeichen am Zeilenende befreit. Alle wesentlichen Angaben zu einem Code sind in einer Zeile untergebracht und durch Kommas getrennt. Dort trennen wir den eingelesenen String in seine Bestandteile auf. Daten, Adresse und Kontrollbyte werden unter dem Schlüssel in **key** als Tupel an **codes** angehängt.

```
    try:
        with open("befehle.cfg","r") as f:
            for line in f:
                line=line.strip("\n\r")
```

```
                key,data,addr,ctrl=line.split(",")
                codes[key]=(data,addr,ctrl)
        d.writeAt("GOT KEY-CODES",0,1)
        sleep(3)
        d.clearAll()
        return codes
    except OSError as e:
        d.writeAt("NOT FOUND",0,2)
        sleep(3)
        d.clearAll()
        return None
```

Wir nähern uns der Hauptschleife und treffen die letzten Vorbereitungen. Die Uhr wird beim Start des Programms synchronisiert, wenn wir das Kommentarzeichen in der nächsten Zeile entfernen. Ich habe während der Entwicklung diese Zeile auskommentiert, um nicht jedes Mal zwei Minuten warten zu müssen, bis ich weitermachen konnte.

```
# sync()
if interface=="AP":
    nic=setAccessPoint()
elif interface=="STA":
    nic=connect2router()
sleep(3)
```

Je nach der Belegung von **interface** stellen wir jetzt das entsprechende Interface bereit. Drei Sekunden zum Lesen der Meldung im Display. Dann setzen wir den Socket auf und lesen die RC5-Code-Staffel ein. Ausgabe in REPL und Zeitanzeige im LED-Display.

```
s=setSocket()
codes=readData()
print(codes)
timeOutput()
```

Alarmtimer 1 bringt jede volle Minute die aktuelle Uhrzeit auf das LED-Display. Alarm 2 schalten wir kurz an, um ihn danach sofort wieder zu deaktivieren.

```
rtc.Alarm1(0,0,0,0,DS3231.SekundenAlarm) # zur vollen Minute
rtc.Alarm2(0,12,5,DS3231.MinutenAlarm)
rtc.AlarmAus(2)
rtc.ClearAlarm(1)
rtc.ClearAlarm(2)
print("Alarm:",rtc.TellAlarmStatus())
alarmTrigger=False
```

Die Alarmflags beider Alarme werden zurückgesetzt, um einen neuen Alarm sicher zu gewährleisten. Die Leitung SQW des DS3231 geht damit auf 3,3V. Den Stand der Flags lassen wir uns anzeigen. Das Flag **alarmTrigger** setzen wir auf False und steigen in die Hauptschleife ein.

```
rtc.Alarm1(0,0,0,0,DS3231.SekundenAlarm) # zur vollen Minute
rtc.Alarm2(0,12,5,DS3231.MinutenAlarm)
rtc.AlarmAus(2)
rtc.ClearAlarm(1)
rtc.ClearAlarm(2)
print("Alarm:",rtc.TellAlarmStatus())
alarmTrigger=False
while 1:
```

Ein Alarm wurde ausgelöst, wenn **alarmTrigger** auf True steht. Wir holen uns den Alarmstatus, geben ihn aus und setzen das Flag zurück.

```
    if alarmTrigger:
        status=rtc.TellAlarmStatus()
        print("triggered:",rtc.TellAlarmStatus())
        alarmTrigger=False
```

Wir haben einen Minutenalarm, wenn das Bit 0 im Status gesetzt ist. Jetzt muss ein Zeitupdate im LED-Display erfolgen. Zuvor setzen wir das Interruptflag des DS3231 zurück. Die Ausgabe zeigt an, ob das Interruptflag 1 gelöscht wurde.

```
        if status & 0x01:
            rtc.ClearAlarm(1)
            print("Alarm1:",rtc.TellAlarmStatus())
            timeOutput()
```

Ein Stunden- oder Tagesalarm liegt vor, wenn das Bit 1 im Status gesetzt ist. Nach dem Rücksetzen des IRQ-Flags sende ich hier das Kommando 1 an die Adresse 0 zweimal an mein RC5-Gerät, welches dadurch eingeschaltet wird. Die Ausgabe zeigt an, ob das Interruptflag 2 gelöscht wurde.

```
        if status & 0x02 & rtc.TellAlarmEnabled():
            rtc.ClearAlarm(2)
            print("Alarm2:",rtc.TellAlarmStatus())
            sendeRCcode((0,1,0),2) # ((addr,cmd), repeat)
```

Einmal pro Tag wird die RTC synchronisiert und zwar um 03:01 Uhr.

```
    if dt[4:6] == [3,1]:
        sync()
```

Der Empfang von UDP-Nachrichten muss in try-except gekapselt werden. Wurden durch **recvfrom**() keine Zeichen registriert, dann wird die Empfangsschleife verlassen und eine Exception geworfen, die wir abfangen müssen.

```
    try:
        rec,adr=s.recvfrom(150)
        rec=rec.decode().strip("\r\n")
```

```
            reply=parse(rec)
            print(rec,adr)
            s.sendto(reply,adr)
        except:
            pass
```

Ist ein Befehl vom Handy angekommen, dann liefert **recvfrom**() einen Record, mit der Nachricht und den IP-Socket des Absenders. **rec** enthält ein bytes-Objekt, das wir mit **decode**() in einen String umwandeln, von dem die Steuerzeichen entfernt werden. Den aufbereiteten Text schicken wir zum Parser, der einen Kommentarstring zurückgibt. Diesen schicken wir an den Absender zurück. Wir werden als Absender gleich **packetsender.exe** auf dem PC für einen ersten Test verwenden.

Die nächsten Zeilen teilen uns mit, dass ein Alarm durch den Timer2 ausgelöst wurde.

```
    if rtc.TellAlarmStatus() & 0x02:
        print("Alarm2 triggered")
        rtc.ClearAlarm(2)
```

Um das Programm sauber verlassen zu können, fragen wir die Taste ab. Wurde sie gedrückt, dann setzen wir den IRQ-Handler des rtc-Interrupts auf **None** und beenden das Programm mit **exit**().

```
    if taste.value() == 0:
        rtcIRQ.irq(handler=None)
        exit()
```

Der Test mit packetsender

Netzwerkverbindungen lassen sich mit packetsender.exe sehr gut testen. Das kostenlose Programm kann UDP- oder TCP-Nachrichten versenden und empfangen. Man muss lediglich einen lokalen Port für den PC festlegen, die IP-Adresse wird automatisch von der Netzwerkkarte übernommen. Dann gibt man noch die Socketdaten der Gegenseite ein und wählt das Protokoll. Die eigegebene Nachricht wird durch Klick auf **Send** übermittelt und eine etwaige Antwort angezeigt.

Starten Sie jetzt das Programm wecker.py im Editorfenster von Thonny und **packetsender** auf dem PC. Abbildung 2 zeigt die Einstellungen und die Antwort auf den Befehl **ALARM:OFF**.

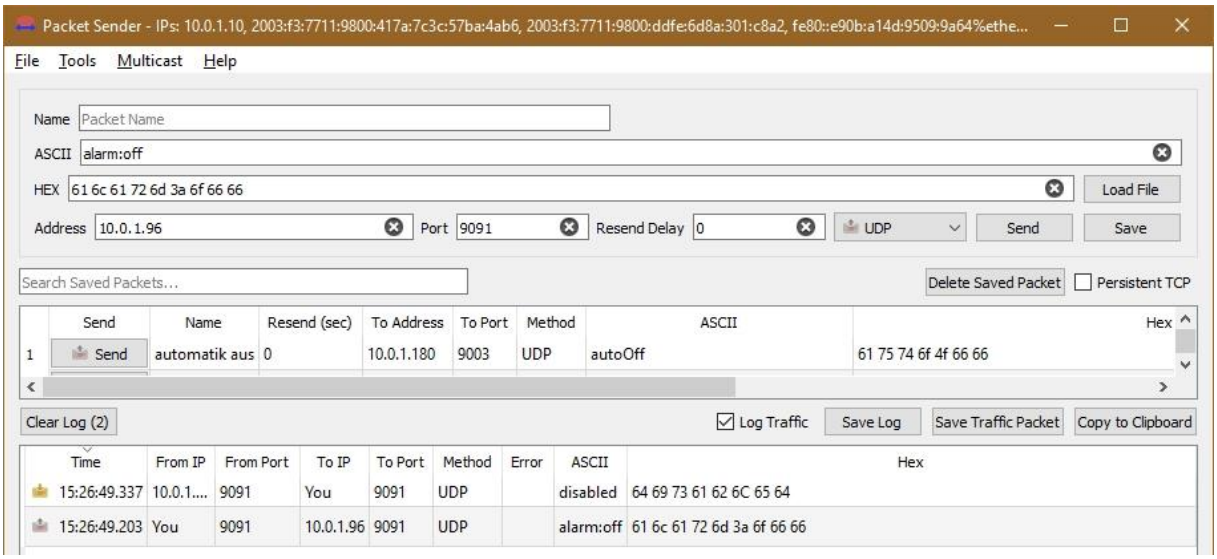

Abbildung 2: Das Fenster von packetsender

Auf die gleiche Weise können Sie jetzt auch einen Alarm setzen.
ALARM:15,33

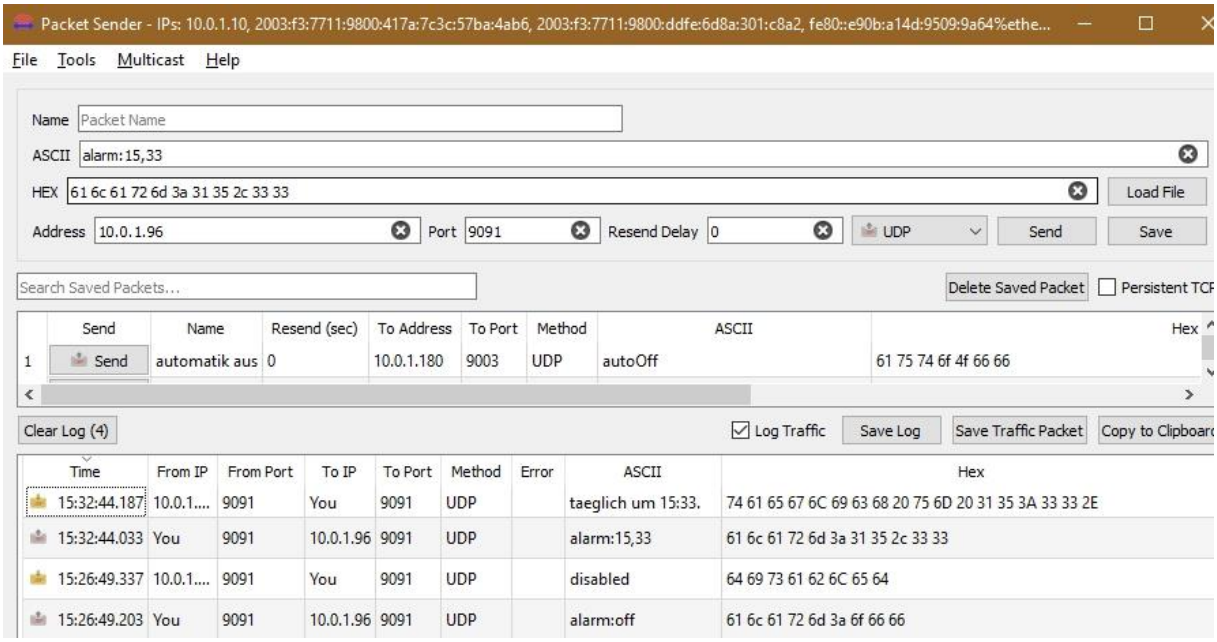

Abbildung 3:Alarm2 setzen

Hier die REPL-Ausgabe von Thonny.

Alarm: 0
OFF
Alarm2 aus
alarm:off ('10.0.1.10', 9091)
…
alarm:15,33 ('10.0.1.10', 9091)
…
triggered: 3
Alarm1: 2

[2023, 9, 6, 3, 15, 33, 0]
Alarm2: 0

Senden Sie ruhig mit packetsender auch einen RC5-Code an Ihr Gerät und prüfen Sie, ob es richtig reagiert.

RC:OFF

Hat es sich ausgeschaltet? Bestens! Eine stärkere Sende-Diode finden Sie übrigens bei Reichelt: GRV IR TRANS. Sie hat eine angegebene Reichweite von 10 Metern.

In der nächsten Blogfolge basteln wir dann eine Android-App mit der wir den ESP32 steuern können.

Bleiben Sie dran!