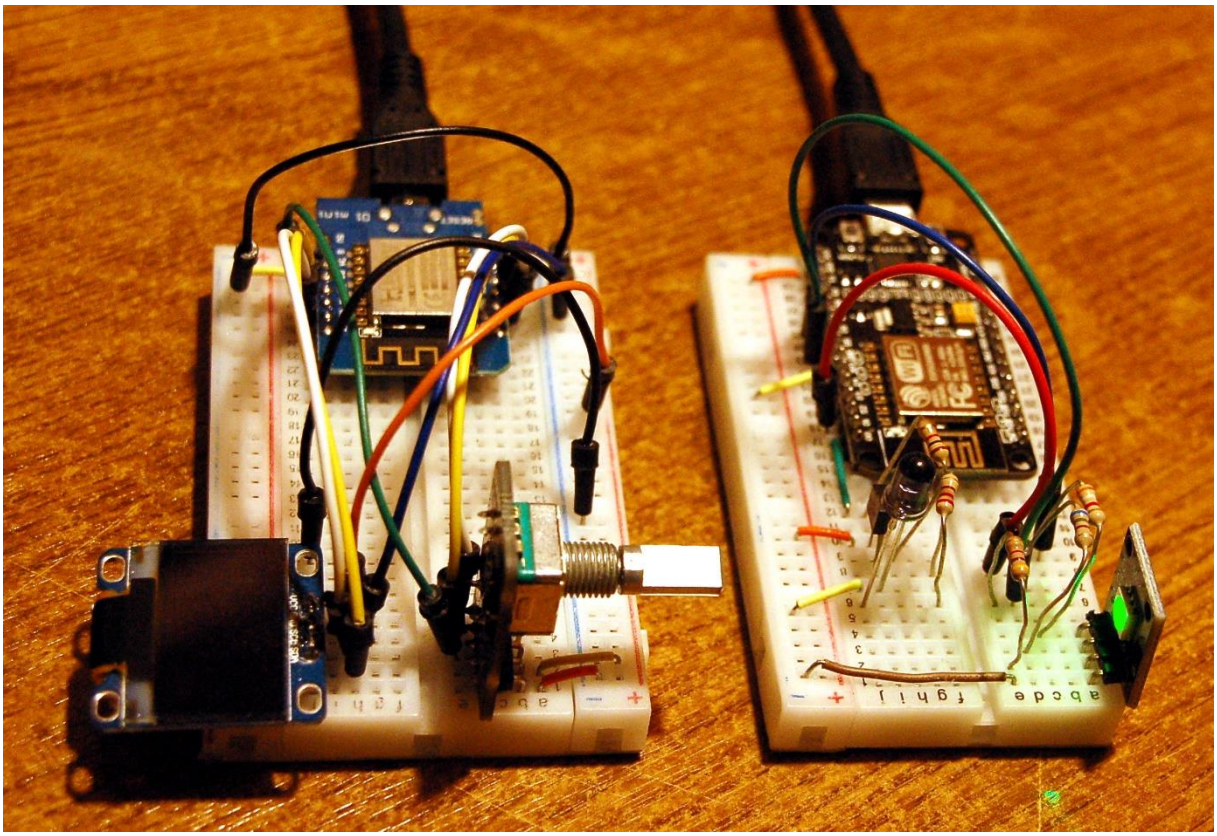
Die WLAN-Connection - UDP-Client und UDP-Server

Aus ein bis zwei Metern Entfernung klappt der IR-Auslöser für die Nikon ganz gut. Auch das Ablesen der Einstellung ist OK, sogar bei Dunkelheit, dank dem OLED-Display. Manchmal wäre aber ein viel größerer Abstand noch besser, zum Beispiel bei Aufnahmen scheuer Tiere.

Zur Vergrößerung der Reichweite teilen wir die Schaltung aus dem ersten Teil zum Thema Nikon-Timer einfach auf. Zeitauswahl und Display erledigt im Titelbild ein ESP8266 D1 mini, der IR-Strahler wir von einem zweiten solchen oder einem anderen Verwandten aus dem ESP-Clan bedient. Den Kontakt zwischen den beiden stellen wir über Funk her und zwar via WLAN und UDP. Und weil das auch in der Pampa funktionieren soll, brauchen wir eine Insellösung, ohne lokales Hausnetz. Das heißt, wir werden beim zweiten ESP8266, ich habe hier einen Amica in Betrieb, den systemeigenen Accesspoint aktivieren und einen UDP-Server draufsetzen. Der andere ESP8266 spielt den Client und wählt sich beim Server ein. Ganz nebenbei, gibt es wieder Einsteigerinformationen zu MicroPython. Damit willkommen bei einer neuen Folge aus der Reihe

MicroPython auf dem ESP32 und ESP8266

heute

Foto-Timer mit Fernsteuerung

Wie man eine IR-Fernsteuerung auslesen kann, habe ich im ersten Teil ausführlich beschrieben, ebenso das Nachbilden der IR-Impulse durch Software. Das Programm aus diesem Post wird heute zwischen Client und Server aufgeteilt. Das geschieht in der Form, dass auch weitere Ideen in Folgeprojekten möglichst nahtlos umgesetzt werden können. Beginnen wir mit der Hardware. Alle Teile aus Folge 1 sind wieder im Einsatz. Dazu gekommen sind ein weiterer ESP32 oder ESP8266, zwei Widerstände und ein Kleinleistungs-PNP-Transistor.

Hardware

2	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	KY-040 Drehwinkelgeber Drehgeber Rotary Encoder Modul
1	KY-005 IR Infrarot Sender Transceiver Modul
1	0,96 Zoll OLED SSD1306 Display I2C 128 x 64 Pixel
1	KY-022 Set IR Empfänger Infrarot Receiver CHQ1838
1	Widerstand 100 Ω
1	Widerstand 1kΩ
1	PNP-Transistor BC558 ähnlich
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
1	Logic Analyzer optional

ESP32 oder ESP8266 sind beide mit einer Ausnahme gleichermaßen für dieses Projekt geeignet. Die Ausnahme ist der ESP8266-01, weil der einfach zu wenig herausgeführte GPIO-Leitungen hat.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
SALEAE – Logic-Analyzer-Software für Windows 8, 10, 11

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder
ESP32 mit 4MB Version 1.18 Stand 25.03.2022
Die Versionsnummer ist entscheidend für die Umsetzung des Projekts.

Die MicroPython-Programme zum Projekt:

oled.py OLED-Frontend
ssd1306.py OLED-Treibermodul
rotary.py Treiber für Winkel-Encoder portübergreifend
rotary_irq_esp.py Treiber für ESP32/ESP8266
ir_ausloeser.py Testprogramm für die Auslösesequenz
nikon_timer.py Betriebssoftware
nikon_timer_remote.py Betriebssoftware Server
nikon_rc.py Betriebssoftware Client

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 05.02.2022) auf den ESP-Chip gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm kompilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Makro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen.

Autostart

Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP-Chip hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton,

oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein kompiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA oder … enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer so, wie hier beschrieben.

Kontaktaufnahme mit (Micro)-Python

In der letzten Folge habe ich ein paar grundsätzliche Datentypen vorgestellt, Ganzzahlen (Integer oder int), Fließkommazahlen (float), Zeichenketten (Strings) und den Exoten None. Sie haben auch schon mit den seriellen Datentypen Liste und Tuple Bekanntschaft gemacht.

Datentypen

Heute wird es in der Funkabteilung um einen weiteren Datentyp gehen, den ich Ihnen zunächst ohne Bezug auf das Programm nahebringen möchte, weil er etwas gewöhnungsbedürftig ist. In diesem Zusammenhang werden wir uns auch mit Typumwandlungen beschäftigen, denn auch daran kommen wir später nicht vorbei. Schließlich wird im Programm noch ein Datentyp auftauchen, der entfernte Ähnlichkeiten mit einer Liste hat, das assoziative Array (Dictionary oder kurz Dict).

Datentyp bool

Beginnen möchte ich aber mit einem ganz einfachen Datentyp, den wir schon mehrfach implizit benutzt haben, also ohne uns dessen bewusst zu werden. Ich meine den Typ bool. Instanzen dieses Typs können nur die Werte True (wahr) und False (falsch) annehmen. Vorgekommen sind die boolschen Werte in if-Konstrukten.

```
if chip == 'esp8266':
    SCL=Pin(5) # S01: 0
    SDA=Pin(4) # S01: 2
elif chip == 'esp32':
    SCL=Pin(21)
    SDA=Pin(22)
else:
    raise OSError
```

Ausführlich gelesen hört sich das so an:

wenn die Aussage "chip ist gleich 'esp8266' " wahr ist, dann setze SCL dem Pin-Objekt Pin(5) gleich …

Wahrheitswerte kann man auch verknüpfen, so wie man 3 + 2 berechnen kann. Im Programm kommt die UND-Verknüpfung vor. Natürlich gibt es dafür andere Vorschriften wie für die Berechnung der Summe 3 + 2. Nehmen wir zwei Vergleiche her, und prüfen wir nach, ob für verschiedene Zahlen Vergleich A (i ist kleiner als 7) und zugleich Vergleich B (i ist größer als 5) wahr sind.

```
i < 7 und zugleich i > 5
if i < 7 and i > 5:
    print("i ist 6!")
```

Für i = 3 ist A wahr und B falsch
Für i = 11 ist A falsch und B wahr
Nur für i = 6 ist A und zugleich B wahr und nur dann wird der Text "i ist 6!" ausgegeben. Das führt auf folgende Wahrheitstabelle

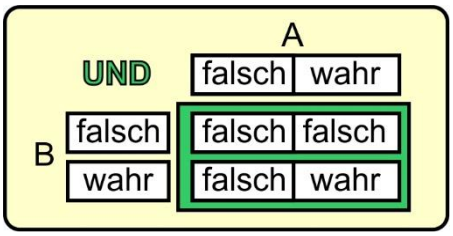

Abbildung 1: Wahrheitstabelle UND

Auch für die anderen Basistypen ist jeweils ein Wert definiert, der als False interpretiert wird, bei Integer ist es die 0, bei float ist es 0.0, bei str "" und None wird natürlich auch als False gewertet, ebenso wie die leere Liste [ ] oder das leere tuple ( ). Alles andere gilt als True.

Das Dictionary (Dict)

Das leere **Dict** liefert übrigens auch **False**. Womit wir beim nächsten Thema wären. Listen und Tuples sind sequentielle Datentypen, die Auflistung erfolgt so, wie die Strukturen definiert wurden. Bei Dicts kann man sich darauf nicht verlassen. Ein Dictionary besteht aus Schlüssel-Wert-Paaren. Schlüssel und Wert sind durch einen Doppelpunkt getrennt und das Ganze ist durch geschweifte Klammern eingeschlossen. Im Programm habe ich ein Dict verwendet, um den Zahlencodes, welche die WLAN-Funktion **status**() zurückgibt, Klartextwerte zuzuordnen.

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

>>> connectStatus
{201: 'NO AP FOUND', 1000: 'STAT_IDLE', 1010: 'STAT_GOT_IP', 202: 'STAT_WRONG_PASSWORD', 1001: 'STAT_CONNECTING'}

Der Zugriff auf einen Wert erfolgt, ähnlich wie bei Listen und Tuples über den Index, über den Schlüssel.

>>> connectStatus[1001]
' STAT_CONNECTING'

Die Typen str, bytes und bytearrays

Wir werden in den beiden Programmen zur Datenübertragung das UDP-Protokoll nutzen. Die Socket-Methoden **sendto**() und **recvfrom**() verwenden zum Transfer bytes-Objekte. Das trifft weder für den Typ **int** und auch nicht für **float** zu, für Listen und Tuples schon gleich gar nicht. Am ehesten trifft das auf den Typ **str** zu, zumindest beim Versenden. Was von **recvfrom**() zurückgegeben wird ist jedenfalls kein String, sondern ein bytes-Objekt. Dieser Typ dient der internen Verarbeitung und dem Transport von Daten als binäre Bytesequenzen, ohne einen Bezug zu Codetabellen. **str**-Objekte dienen der Darstellung von Zeichen und können neben den normalen ASCII-Zeichen auch Sonderzeichen enthalten, wie die deutschen Umlaute. Nehmen wir einmal nur die normalen ASCII-Zeichen, die aus einer 7-Bit-Codierung resultieren, dann gibt es zwischen einem **str**-Objekt und einem **bytes**-Objekt diverse Ähnlichkeiten aber auch Unverträglichkeiten. Beide lassen sich zum Beispiel als Zeichenketten interpretieren, liefern bei der Darstellung aber ein unterschiedliches Erscheinungsbild.

>>> a="Nikon"
>>> a
'Nikon'
>>> b=b"Nikon"
>>> b
b'Nikon'

Aber während das gut funktioniert,

>>> a+a
'NikonNikon'

liefert das einen Fehler.

>>> a+b
Traceback (most recent call last):
* File "<stdin>", line 1, in <module>*
TypeError: can't convert 'bytes' object to str implicitly

Außerdem sieht so etwas nicht besonders ansprechend aus.

>>> print(b,"-Kamera")
b'Nikon' -Kamera

Für das Versenden von Daten, aber noch mehr für den Empfang und die Darstellung, brauchen wir also eine Umwandlung von bytes-Folgen in str-Objekte und umgekehrt. Das liefern die Instanzmethoden encode() und decode(). Für die Umwandlung ist die internationale Codepage UTF-8 Standard und muss daher nicht angegeben werden.

>>> print(b.decode("utf8"))
Nikon

>>> a="Nikon"
>>> a.encode()
b'Nikon'

Interessant ist auch folgender Unterschied zwischen bytes-Objekten und str-Objekten. Die Elemente von str-Strings werden als Zeichen interpretiert, die von bytes-Strings als Zahlen.

>>> a[2]
'k'
>>> b[2]
107

Die Methoden derI2C-Klasse erlauben neben str- und bytes-Objekten zum Transfer auch den Datentyp **bytearray**, der ebenfalls auf dem Buffer-Protokoll basiert. Welcher Datentyp verwendet wird, ergibt sich aus der jeweiligen Situation. Beispiele finden sich in der Klasse SSD1306_I2C im Modul ssd1306.py.

Die folgenden Typumwandlungen tauchen auch im Programm auf. Die Methode decode() macht zwar aus einem bytes-Objekt einen String. Wenn dieser eine Zahl enthält mit der gerechnet werden soll, dann muss der String zuerst in eine Zahl umgewandelt werden.

>>> rec="1234"
>>> 5+rec
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported types for __add__: 'int', 'str'

>>> 5+int(rec)
1239

Das geht auch umgekehrt. Wir zerlegen eine Zahl in ihre Ziffern:

>>> for i in range(len(str(w))):
    print(str(w)[i])

1
2
3
4
5
6
7
8

Andere Typumwandlungen erfolgen transparent, int + float -> float.

>>> 23+5.9
28.9

>>> print("23 + 5.9 = {}".format(23+5.9))
23 + 5.9 = 28.9

Die str-Methode **format**() wandelt das Ergebnis von 23+5.9 in einen String um und fügt diesen an Stelle der geschweiften Klammern in die Zeichenkette ein.

Nach den Grundlagen wenden wir uns den beiden Programmen zu.

Der UDP-Client

Wie eingangs erwähnt, zerpflücken wir Schaltung und Programm aus der ersten Folge und peppen die Teile durch die WLAN-Fähigkeit auf. Für den Client als Sender bedeutet das, dass er über die Anzeige und den Winkelencoder verfügen muss. Alles was nichts damit zu tun hat, kann aus der Schaltung und dem Programm nikon_timer.py entfernt werden.

Die Schaltung

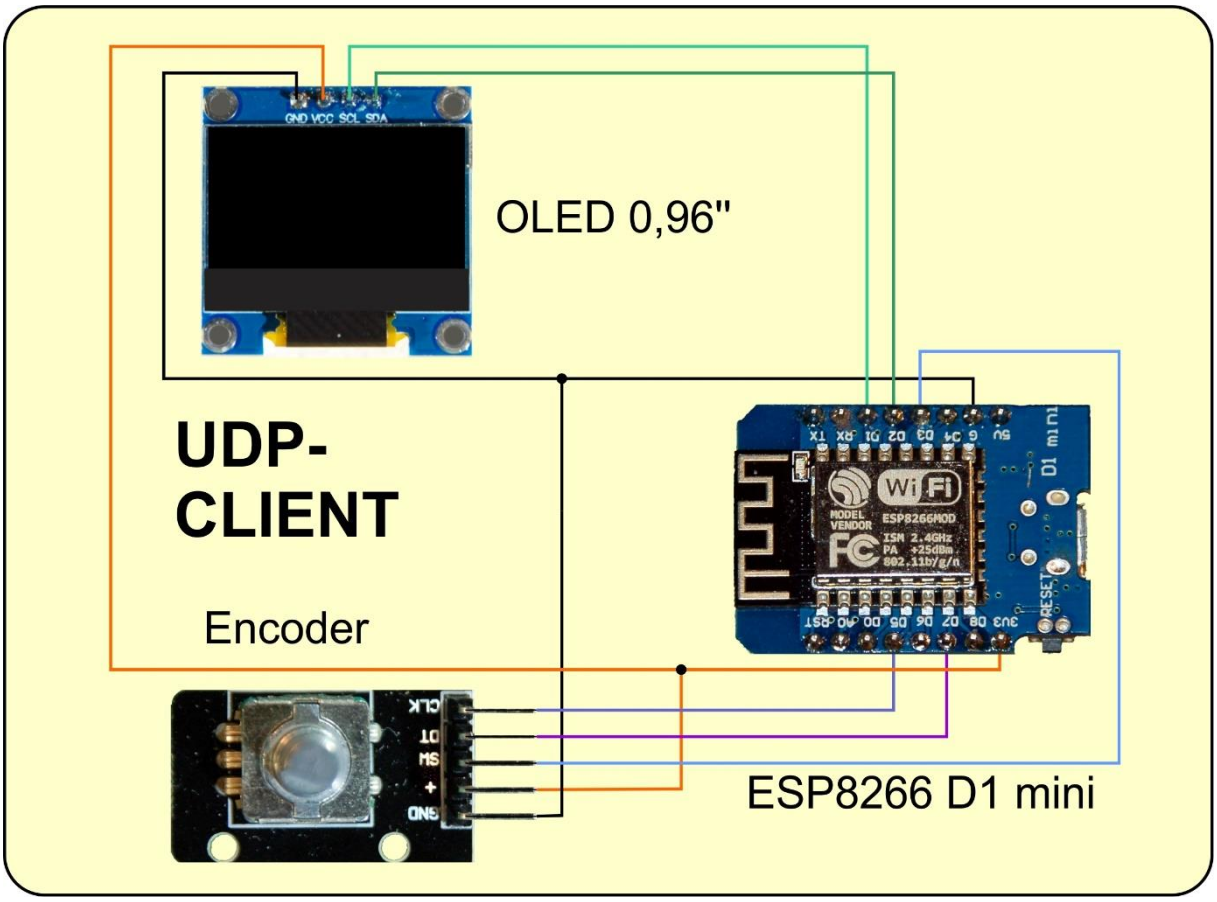

Abbildung 2: UDP-Client

Das Client-Programm

Der Import im neuen Programm sieht dann so aus.

```
# nikon_remote.py
# Nach dem Flashen der Firmware im Terminal eingeben:
# import webrepl_setup
# > d fuer disable
# Dann RST; Neustart!
#
import sys
from time import sleep_ms, sleep, ticks_ms
from machine import SoftI2C, Pin
from rotary_irq_esp import RotaryIRQ
from oled import OLED
from ssd1306 import SSD1306_I2C
import network
import socket
```

Es folgen die Daten für den Aufbau der Funkverbindung und des UDP-Sockets. Die WLAN-Verbindung entspricht etwa dem Kabel einer USB-Verbindung und der Socket ist das Gegenstück zum COM-Interface.

```
# **********Objekte und Variablen deklarieren ***************
mySSID = 'foto_shoot'
myPass = 'guest'
myNetwork = "10.1.1."
myIP=myNetwork+"94"
myGW=myNetwork+"96"
myDNS=myNetwork+"96"
myPort=9009
remoteIP="10.1.1.96"
remotePort=9009
target=(remoteIP,remotePort)
```

Entsprechend dem Controllertyp setze ich die GPIO-Pins für die I2C-Schnittstelle und initialisiere diese. Dann instanziiere ich damit das OLED-Objekt und lösche die Anzeige.

```
chip=sys.platform
if chip == 'esp8266':
    # Pintranslator fuer ESP8266-Boards
    # LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
    # ESP8266 Pins 16  5  4  0  2 14 12 13 15
    #                 SC SD
    SCL=Pin(5) # S01: 0
    SDA=Pin(4) # S01: 2
elif chip == 'esp32':
    SCL=Pin(21)
    SDA=Pin(22)
```

```
else:
    raise OSError ("Unbekannter Port")

i2c=SoftI2C(SCL,SDA)
d=OLED(i2c)
d.clearAll()
```

Die Ein- und Ausgänge für den Winkelencoder werden definiert, dann erzeuge ich damit die Encoder-Instanz. Werte zwischen 0 und 25 sollen damit eingestellt werden können.

Die Liste delay enthält alle vorgesehenen Eckwerte für Zeitintervalle. Die Verlagerung dieser Liste in den Client lässt aber auch die Variante offen, beliebige andere Werte einstellen und senden zu können.

```
delay=[
    86400,
    5,    10,    15,    20,    30,    40,    50,   60,
    90,  120,   180,   240,   300,   360,   480,  600,
    900,1200,  1800,  2400,  3000,  3600,  5400, 7200,
    ]
```

Über das Dict connectStatus habe ich schon weiter oben alles Wesentliche gesagt. Die Codenummer für ein und denselben String unterscheiden sich für ESP32 und ESP8266.

```
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",
    }
```

Die Funktion hexMac übersetzt das bytes-Objekt, das in byteMac übergeben wird, in einen zweckdienlichen String, bei dem die hexdezimal codierten Bytewerte mit einem "-" getrennt werden.

```
def hexMac(byteMac):
    macString =""
    for i in range(0,len(byteMac)):
```

```
        macString += hex(byteMac[0])[2:].upper()
        if i <len(byteMac)-1 :
            macString +="-"
    return macString
```

Was da genau abläuft zeigt dieser Terminalauszug für das erste Byte der MAC-Adresse. Ich rufe den MAC-bytes-String ab, gebe ihn aus und forme dann das erste Byte Schritt für Schritt in die zweistellige hexadezimale Notation um.

>>> byteMac=nic.config('mac')
>>> byteMac
b'\xec\xfa\xbcn\xf7\x08'
>>> len(byteMac)
6
>>> byteMac[0]
236
>>> hex(byteMac[0])
'0xec'
>>> hex(byteMac[0])[2:]
'ec'
>>> hex(byteMac[0])[2:].upper()
'EC'

Hier wird zwar nur ein Sendeauftrag erteilt, es könnten aber auch mehrere Anweisungen sein, die in diesem Zusammenhang umzusetzen sind. Daher wurde dafür eine eigene Funktion definiert.

```
def transmit(cmd,val):
    s.sendto(cmd+str(val),target)
```

Mit Hilfe der Funktion **TimeOut**() erzeuge ich einen nichtblockierenden Softwaretimer für die Intervallsteuerung. Dazu übergebe ich die Zeitdauer in Millisekunden. In der lokalen Variablen **start** wird der aktuelle Zeitpunkt in Millisekunden abgelegt. TimeOut gibt statt eines Zahlenwerts eine Referenz auf die lokal definierte Funktion **compare**() zurück. **compare**() ist eine sogenannte Closure und die bewirkt, dass die Variable **start** und der Parameter **t** auch nach dem Verlassen der umgebenden Funktion **TimeOut**() bis zu weiteren Aufrufen von **compare**() erhalten bleiben. Weiter unten sehen Sie die Anwendung. Normalerweise, werden ja lokal erzeugte Objekte nach dem Verlassen der Funktion eingestampft, aber eben nicht so bei einer Closure.

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

Der nächste Programmabschnitt ist etwas umfangreicher. Er richtet die Verbindung zum Accesspoint in der Servereinheit ein. Die Kommentare sind sehr umfangreich und die Befehlszeilen sprechen weitgehend für sich. Deshalb stelle ich die Sequenz ohne weitere Erläuterung dar.

```
# ******************** WLAN einrichten *********************
# WLAN-Verbindung zum Nikon-remote-AP aufbauen
# Unbedingt das eigene AP-Interface ausschalten
nac=network.WLAN(network.AP_IF)
nac.active(False)
nac=None

# Wir erzeugen eine Netzwerk Station-Interface-Instanz
nic = network.WLAN(network.STA_IF)
# und deaktivieren sie erst einmal
nic.active(False)

# Wir geben die MAC-Adresse des Accesspoints bekannt
MAC = nic.config('mac')
myID=hexMac(MAC)
print("Client-ID",myID)

# Wir aktivieren das Netzwerk-Interface
nic.active(True)

# Aufbau der Verbindung
# Wir setzen eine statische IP-Adresse
nic.ifconfig((myIP,"255.255.255.0",myGW,myDNS))

# Anmelden am WLAN-Router
nic.connect(mySSID, myPass)
nmax=10
if not nic.isconnected():
    # warten bis die Verbindung zum Accesspoint steht
    n=0
    while not nic.isconnected() and n < nmax:
        print("{}.".format(nic.status()),end='')
        d.writeAt("."*(n+1),0,0)
        n+=1
        sleep(1)

# Wenn verbunden, zeige Verbindungsstatus & Config-Daten
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
if nic.isconnected():
    # War die Konfiguration erfolgreich? Kontrolle
    STAconf = nic.ifconfig()
    print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
          STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
    d.writeAt("SENDING ON:",0,0,False)
    d.writeAt(STAconf[0],0,1)
    sleep(3)
    d.clearAll()
```

```
else:
    print("No AP found")
    d.writeAt("NO ACCESSPOINT",0,0,False)
    d.writeAt("FOUND",0,1)
    while 1:
        pass
```

Die nächsten 6 Zeilen richten das Kommunikations-Interface ein, den UDP-Socket. Wir benutzen die IPv4-Familie (AF.INET) und zwar auf der Basis von Datagrammen (SOCK.DGRAM), was eben dem UDP-Protokoll entspricht. Gerade während der Entwicklung muss das Programm oft neu gestartet werden. Damit dann vor dem Neustart des Sockets ein Kaltstart mit Reset erfolgen muss, erklären wir dem System, dass die vorhergehenden Socket-Einstellungen wiederverwendet werden sollen (SO.REUSEADDR). Wir binden die Schnittstelle an die oben vergebene IP-Adresse und die in myPort angegebene Portnummer. Ein timeout von 50ms für die Empfangsschleife sorgt dafür, dass diese die Hauptschleife nicht blockiert, damit andere Aktionen ausgeführt werden können. In target legen wir die Socketadresse des UDP-Servers mit der IR-Einheit fest.

```
# ******************* Socket einrichten ********************
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', myPort))
print("sending on port",myPort)
s.settimeout(0.05)
target=(remoteIP,remotePort)
```

Dann lasse ich die Anzeige löschen und die Startmeldung ausgeben. Die Position des Winkelencoders wird als Startwert gesetzt. Wir deklarieren schon einmal vorab die Funktion clear() über TimeOut(), damit die Referenz bekannt ist, wenn wir die Funktion in der Hauptschleife aufrufen lassen. wipe ist eine Hilfsvariable, die im Zusammenwirken mit clear() die Rückmeldung vom Server in der anzeige löscht.

```
d.writeAt("NIKON-REMOTE",2,0,False)
d.writeAt("press button",2,4,False)
d.writeAt("to expose!",4,5)

indexOld = r.value()
clear=TimeOut(10)
wipe=False
```

Dann betreten wir die Hauptschleife.

```
while True:
    index = r.value()
```

Sie startet mit dem Einholen eines neuen Werts vom Winkelencoder.

Unterscheidet der sich vom alten, wurde am Encoder gedreht. Wir lesen die Indexzeit aus der Liste decode, geben sie aus und senden sie an den Timer.

```
    if indexOld != index:
        indexOld = index
        d.clearFT(0,2,d.width-1,2,False)
        d.writeAt("Pause:{}s".format(delay[index]),0,2)
        transmit("time:",delay[index])
```

Wurde die Taste am Encoder gerückt, lösen wir ein Foto aus. 200 Millisekunden entkoppeln die Aktion vom nachfolgenden Tastendruck.

```
    if taste.value() == 0:
        transmit("shot:",0)
        print("Foto")
        sleep_ms(200)
```

Dann schauen wir nach, ob eine Nachricht vom Server eingetroffen ist. In diesem Fall liefert recvfrom() die bytes-Folge der Nachricht in rec und adr enthält die Socket-Adresse des Absenders. Die bytes-Folge decodieren wir als String und entfernen Zeilenvorschub (\n) und Wagenrücklauf (\r). Die Antwort wird im Terminal und in der Anzeige ausgegeben und der Löschtimer clear() auf 1 Sekunde eingestellt. Wir setzen wipe auf True, damit nach einer Sekunde die Rückmeldung vom Server gelöscht wird.

```
    try:
        # receive response
        rec,adr=s.recvfrom(150)
        rec=rec.decode().strip("\r\n")
        # decodieren
        print(rec,adr)
        d.writeAt(rec.upper(),6,3)
        clear=TimeOut(1000)
        wipe=True
        rec=""
```

Eine timeout-Exception wird übergangen, für andere Fehler erfolgt eine Fehlermeldung am Display.

```
    except OSError:
        pass # timeout uebergehen
    except:
        d.clearFT(0,2,show=False)  # bei sonstigen Fehlern
        d.writeAt("E R R O R",3,3)
        d.blinkDisplay(3)
```

Die Rückmeldung des Servers wird gelöscht, wenn wipe True ist und der Timeout abgelaufen ist. wipe setzen wir dann auf False. Erst wenn eine neue Rückmeldung vom Server eingetroffen ist, erhält wipe wieder den Wert True nachdem der Timer

erneut gestartet wurde. Wir warten noch 500 ms, dann bricht eine neue Runde der Mainloop an.

```
    if wipe and clear():
        d.clearFT(0,3,d.width-1,3)
        wipe = False

    sleep_ms(500)
```

Die Serverseite

Die Schaltung

Auf der Serverseite bleiben von der Peripherie eigentlich nur die IR-LED und der Vorwiderstand übrig.

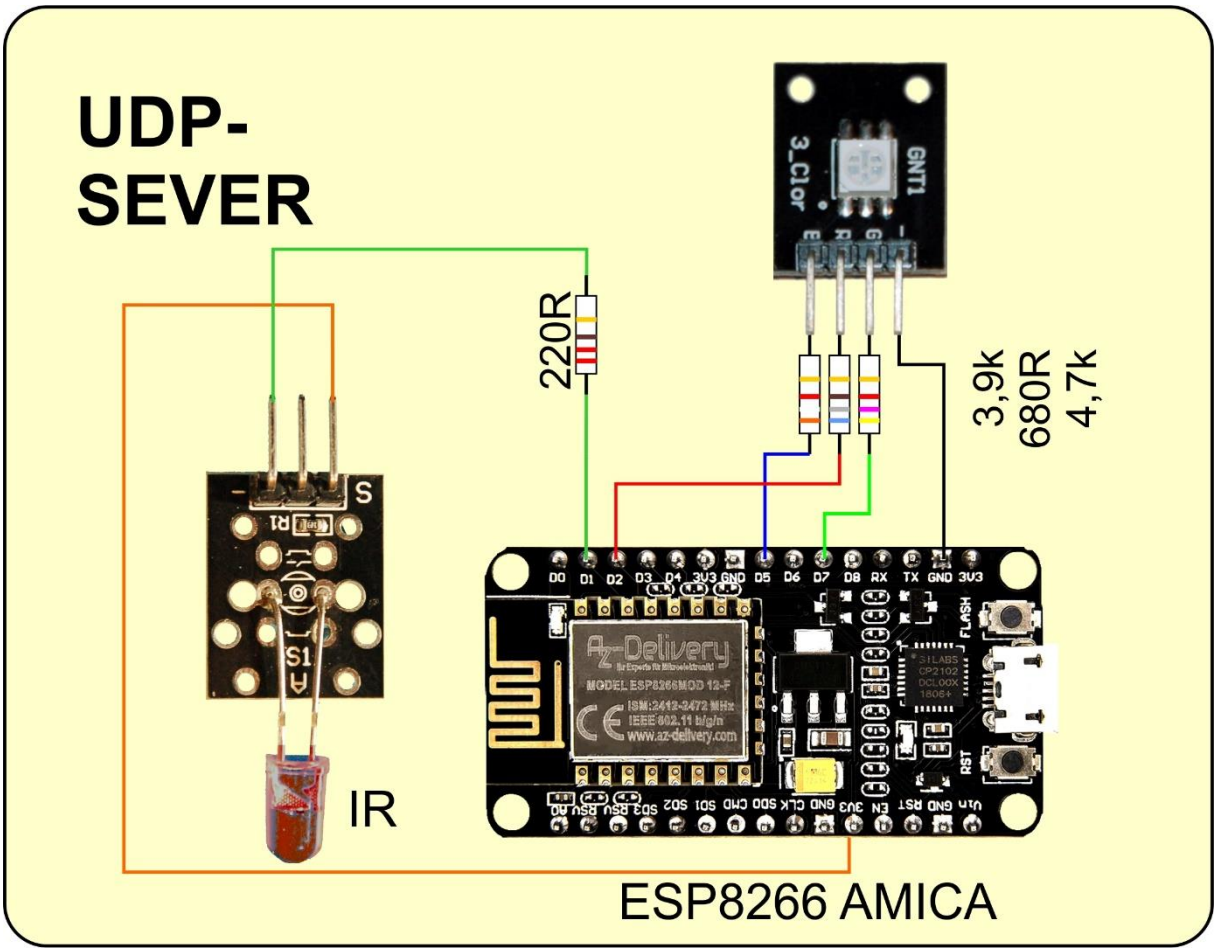

Abbildung 3: UDP-Server ohne Transistorstufe

Der Widerstand in Abbildung 3 ist so berechnet, dass ein Strom von maximal 10mA fließen kann. Der Ausgangspin darf höchstens mit 12mA belastet werden, wir sind also schon in der Grenzregion.

Wer die Reichweite vergrößern möchte, kann das mit einer Transistorstufe und einem kleineren Vorwiderstand tun. Bei 3,3V fließen jetzt bei eingeschalteter LED

20mA. Weil die LED nur für 13ms an ist, könnte man den Widerstand auch noch weiter bis auf 56Ω verringern. Die Pulsstromstärke liegt dann bei ca. 40mA.

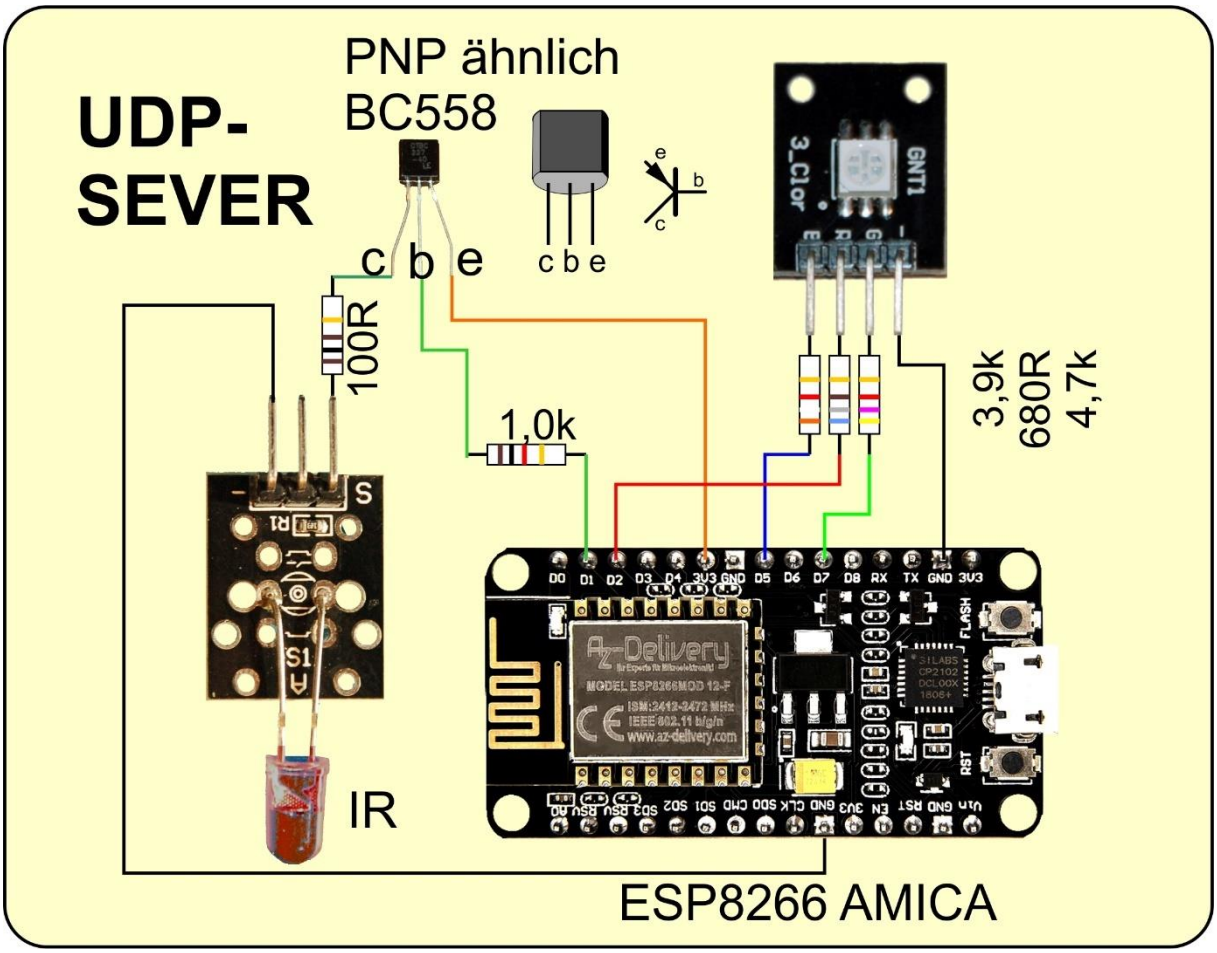

Abbildung 4: UDP-Server mit IR Booster

Eines gilt es allerdings zu beachten, Der Ruhezustand an D1 = GPIO5 ist durch das Programm auf 3,3V festgelegt. Durch die LED soll in diesem Zustand kein Strom fließen. Damit man im Programm keine Änderungen durchführen muss, habe ich für die Schaltung einen PNP-Transistor ausgewählt. Wenn D1 auf logisch 1, also 3,3V liegt, befindet sich die Basis des BC558 auf Emitter-Potenzial, und der Transistor sperrt. Durch den 100Ω-Widerstand und die IR-Diode fließt kein Strom. Sobald der Pegel an D1 auf 0V fällt, liegt die Basis auf Kollektor-Potenzial und der Transistor geht in die Sättigung, die IR-LED leuchtet. Mit regulären Schaltsymbolen sieht die Booster-Stufe so aus.

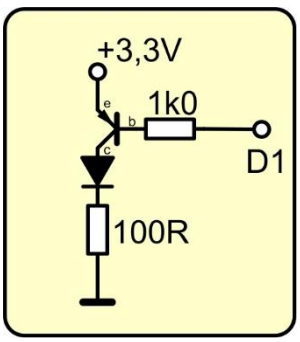

Abbildung 5: Boosterstufe

Das Server-Programm

Die Restmenge von Programmanweisungen aus nikon_timer.py findet sich in nikon_remote.py wieder. Dazu kommen auch hier die Module **network** und **socket**.

```
import sys
from time import sleep_ms, sleep_us, ticks_ms
from machine import Pin, bitstream
import network
import socket
```

Weil die Anzeige aus größerer Entfernung nicht ablesbar ist, spendiere ich für die Darstellung der Transferzustände eine RGB—LED. Neben den bereits bekannten Variablen und Objekten definiere ich die GPIO-Nummern für die drei Farben.

```
#********************Variablen deklarieren ******************
redLed = 4  # D2
greenLed = 13  # D5
blueLed = 14  # D7

taste=Pin(0,Pin.IN)  #  D3
out=Pin(5,Pin.OUT,value=1)  # D4
mySSID = 'foto_shoot'  # Ihre Wahl
myPass = 'guest'  # beliebig + notwendig, nicht verwendet
myIP="10.1.1.96"
myGW=myIP
myDNS=myIP
myPort=9009

errorLed=Pin(redLed,Pin.OUT,value=0)
onairLed=Pin(blueLed,Pin.OUT,value=0)
statusLed=Pin(greenLed,Pin.OUT,value=0)
led=[errorLed,onairLed,statusLed ]
red,green,blue=0,1,2
```

Zur übersichtlichen und flexiblen Handhabung bekommen die LEDs Namen. Eine Liste **led** erlaubt das Ansteuern durch Indizes.

Das Dict connectStatus sowie die Funktionen ausloesen und hexMac Kennen sie auch bereits, weshalb ich sie nicht noch einmal wiedergebe.

Neu ist die Funktion blink. pulse und wait definieren die Zeiten für die Leucht- und Dunkelphase. col ist eine der oben definierten Farbnummern red, green, blue. inverted=False steht für eine LED, die vom GPIO-Pin gegen GND geschaltet ist. Liegt die LED gegen +Vcc, dann muss inverted=True gesetzt werden. cnt gibt die Anzahl Blinkvorgänge an.

```
# Zeiten in Millisekunden
def blink(pulse,wait,col,inverted=False,cnt=1):
    for i in range(cnt):
```

```
        if inverted:
            led[col].off()
            sleep_ms(pulse)
            led[col].on()
            sleep_ms(wait)
        else:
            led[col].on()
            sleep_ms(pulse)
            led[col].off()
            sleep_ms(wait)
```

```
# ********************* AP einrichten ***********************
# Constructoraufruf erzeugt WiFi-Objekt nic
nic = network.WLAN(network.AP_IF)
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')  # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)        # in eine Hexziffernfolge umgewandelt
print("AP MAC: \t"+myMac+"\n") # ausgeben
#

# konfiguriere das Interface mit den oben definierten Werten
nic.ifconfig((myIP,"255.255.255.0",myGW,myDNS))
print(nic.ifconfig())

# MicroPython akzeptiert nur Authmodus 0, kein Passwort!
nic.config(authmode=0)
print("Authentication mode:",nic.config("authmode"))

# config Strings fuer SSID _UND_ Passwort
nic.config(essid=mySSID, password=myPass)

# wir warten auf die Aktivierung des Interfaces
while not nic.active():
  print(".",end="")
  blink(500,500,blue)
print("NIC active:",nic.active())
```

Für die Einrichtung des UDP-Sockets sind keine neuen Anweisungen hinzugekommen.

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

Mit der Intervallzeit für einen Tag initialisieren wir den Timer **jetzt**, dann geht es in die Mainloop. Mit **recvfrom**() rufen wir die Empfangsschleife auf und lesen bis zu 150 Zeichen. Sind welche angekommen, dann dröseln wir das zurückgegebene Tuple gleich in die Nachricht und die Absenderadresse auf. Das Bytesobjekt in **rec** decodieren wir zum String und entfernen davon Wagenrücklauf (0x0D) und Zeilenvorschub (0x0A).

```
delay=86400
jetzt=TimeOut(delay*1000)
# Serverschleife
while 1:
    gc.collect()
    try:
        # Nachricht empfangen
        rec,adr=s.recvfrom(150)
        rec=rec.decode().strip("\r\n")
        print(rec)
        # Nachricht parsen und
        # Aktionen ausloesen
```

Die Rückantwort belege ich schon mal vor, falls in der Nachricht ein Fehler entdeckt wird. Das ist zum Beispiel der Fall, wenn kein ":" in **rec** gefunden wird und die Methode **find**() deshalb **-1** zurückgibt.

Wurde ein Doppelpunkt gefunden, splitten wir den String daran auf in **cmd** und **val**. cmd kann nur "shot" oder "time" enthalten.

shot hat eine doppelte Bedeutung. Es wird ein Einzelschuss ausgelöst, aber gleichzeitig auch die Sequenz mit der aktuell eingestellten Impulsdauer gestartet. Als Antwort erhalten wir im Clientdisplay "DONE".

Das Kommando "time" stellt eine neue Intervallzeit ein und sendet als Antwort "GOT IT".

```
        answer="FALSCHE SYNTAX"
        # parsen und ausfuehren
        if rec.find(":") != -1:
            cmd,val = rec.split(":")
            if cmd == "shot":
                jetzt=TimeOut(delay*1000)
                ausloesen()
                print("Foto")
                blink(195,5,red)
                answer="DONE"
            if cmd == "time":
                delay=int(val)
                answer = "GOT IT"
```

Der Antwortstring wird als Bytes-Objekt encodiert und an den Absender zurückgeschickt. An dieser Stelle könnten weitere Sendebefehle an andere Empfänger eingefügt werden, zum Beispiel zu Debugging- Zwecken.

```
        # Ergebnisse encodiert oder als String senden
        # es kann an mehrere Adressen gesendet werden
        reply=answer.encode()
        s.sendto(reply,adr)
        rec=""
```

Eine Timout-Exception wird übergangen. Die wird geworfen, wenn keine neue Nachricht vorliegt.

```
    except OSError:
        pass
```

Dennoch könnten weitere Ausnahmefehler passieren. In diesem Fall lassen wir die rote LED kurz aufblinken.

```
    except:
        blink(50,950,red)   # timeout uebergehen
```

Falls der Timer **jetzt**() gestartet wurde, ist er irgendwann abgelaufen. Dann liefert **jetzt**() ein **True** zurück. Jetzt ist der Timer neu zu starten, und eine Aufnahme ist auszulösen. Ein längeres rote Blinksignal erzählt uns von dem Ereignis.

```
    if jetzt():
        jetzt=TimeOut(delay*1000)
        ausloesen()
        s.sendto("DONE",adr)
        print("Foto")
        blink(195,5,red)
```

Falls die Flashtaste am ESP8266 gedrückt wurde, empfangen wir am Client die Nachricht "CANCELLED", und die Servereinheit beendet das Programm.

```
    if taste.value()==0:
        print("CANCELLED")
        s.sendto("CANCELLED",adr)
        sys.exit()
    blink(50,950,green)
```

Damit beide Einheiten autonom arbeiten und nach dem Einschalten der Versorgungsspannung automatisch durchstarten, müssen die Programme nikon_rc.py und nikon_timer_remote.py als **boot.py** in den Flash des jeweiligen Controllers hochgeladen werden. Die korrekte Arbeitsweise des Servers kann zum Beispiel auf der GPIO5-Leitung mit Hilfe des Logic Analyzers überprüft werden.

Ausblick

Die Auslösung von Aktionen durch Ereignisse fast beliebiger Art ist durch die vielfältigen Sensor-Module, die an einen ESP32 oder einen ESP8266 angekoppelt werden können, möglich geworden. Die Auslösung einer fotografischen Aufnahme ist dafür nur eine Variante. In der nächsten Folge werde ich einige dieser "Auslöser" untersuchen.

Bis dann!