Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

Zuletzt ging es um das Senden von Textnachrichten über das Mobilfunknetz via SMS. Auch in dieser Folge kommen ESP32 und SIM808 wieder zum Zug. Sie bilden eine Art Server oder Relaisstation, an die zur Steuerung Textnachrichten gesendet werden können.

Der Controller ist aber auch in der Lage, selbst Nachrichten zeit- oder eventgesteuert abzusetzen. Das Handy oder der PC, der Raspi, etc ist dann Schalt- und Empfangszentrale, SMS-Funktionalität vorausgesetzt. Meist wird es das Handy oder Tablett sein und weil die Strecke zwischen SIM808 und Handy keine Rolle spielt, gibt es auch kein Entfernungslimit, sofern ausreichende Netzabdeckung gegeben ist. Das LCD-Keypad wird eigentlich überflüssig, weil eine direkte Steuerung der entfernten ESP32-Einheit durch die Tasten sowieso nicht in Frage kommt. Dennoch leistet ein Display gute Dienste beim Start der Relaiseinheit und für weitere Statusmeldungen. Und – eine Taste ist essentiell, die verwende ich als Notbremse! Damit herzlich willkommen zum vierten Teil von

GSM und GPS mit MicroPython auf dem ESP32

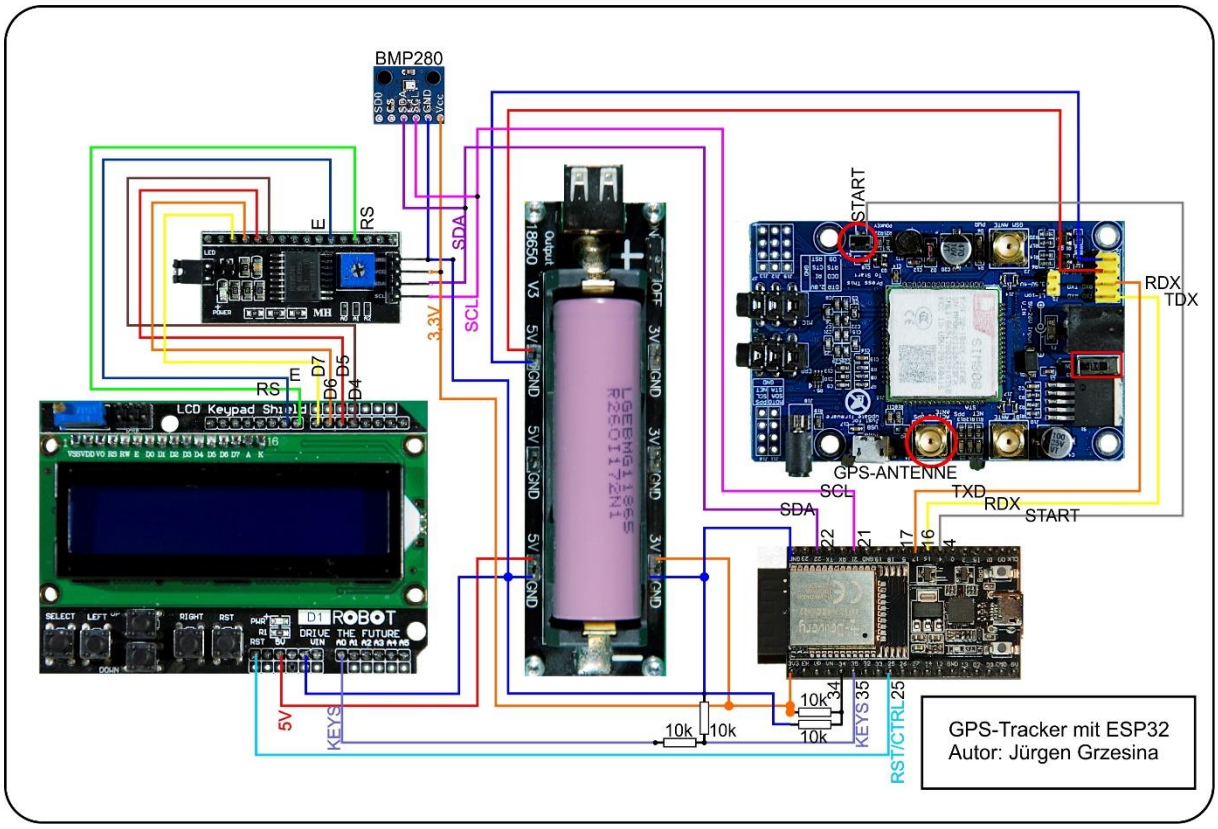
Hardware – etwas Zuwachs

An Hardware kommt im Vergleich zu Teil 3 ein wenig dazu. Wir wollen ja immerhin wenigstens eine externe Funkeinheit ins Projekt integrieren. Gut, in der folgenden Liste finden Sie alle Teile für das aktuelle Projekt. Fast alles davon wurde bereits in Teil 1 , Teil 2 und Teil 3 eingesetzt und dort natürlich auch ausführlich beschrieben. Diese Bauteile verwenden wir selbstverständlich wieder. Im 3. Teil der Serie ist eine SIM-Karte hinzugekommen, denn ohne diese werden Sie keine SMS-Nachricht versenden oder empfangen können. Und nun werden wir die Funktionalität der Relaisstation um eine weitere Option ergänzen. Zu diesem Zweck habe ich als Beispiel einen ESP8266 zusammen mit einem LDR-Widerstand und einem weiteren BMP280 als Funksensor vorgesehen. Anstatt des LDR kann natürlich ein beliebiger anderer Sensor dienen, der analoge Signale liefern kann. Auf der externen Funkeinheit wird nämlich in diesem Beispiel eine analoge Spannung am A0-Pin des ESP8266 abgetastet. Andere Sensoren wie DS18B20 (Temperatur), DHT22 AM2302(Feuchte und Temperatur), GY-302 BH1750 (Licht Sensor), etc. können natürlich dank der diversen MicroPython-Module, die es dafür gibt, ebenso gut hergenommen werden. Das Modul zum BMP280 hatten wir schon in Episode 3 im Einsatz. In der Regel sind die Module leicht an eigene Bedürfnisse anzupassen. Zur Not wird ein Modul aus einer Library der Arduino-IDE für MicroPython umgearbeitet. Meist kommt dabei sogar ein lesbarer Quelltext heraus und die Klassenmethoden machen durch die Anpassung genau das, was man sich vorstellt.

Hier ist die Liste der Zutaten für die neue Kochsession. Schließlich gibt es dieses Mal zwei Hauptgänge.

1	ESP32 Dev Kit C V4 unverlötet oder ähnlich
1	LCD1602 Display Keypad Shield HD44780 1602 Modul mit 2x16 Zeichen
1	SIM 808 GPRS/GSM Shield mit GPS Antenne für Arduino
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
1	Li-Akku Typ 18650
1	I2C IIC Adapter serielle Schnittstelle für LCD Display 1602 und 2004
4	Widerstand 10kΩ
2	GY-BMP280 Barometrischer Sensor für Luftdruckmessung
1	SIM-Karte (beliebiger Anbieter)
1	NodeMCU Lua Lolin V3 Module ESP8266 ESP-12F WIFI Development Board unverlötet
1	Foto Widerstand Photoresistor Licht Sensor Modul LDR5528 oder KY-018 Foto LDR Widerstand Diode Photo Resistor Sensor für Arduino

Die Schaltung für die Relaisstation wird erst einmal 1:1 von Teil 2 übernommen. Später entscheiden Sie selbst, welche Teile Sie weglassen, ersetzen oder neu hinzunehmen. Das Projekt ist in jeder Richtung skalierbar. Die programmtechnischen Grundlagen für die Umsetzung finden Sie in diesem Beitrag.



Ein besser lesbares Exemplar der Darstellung in DIN A4 bekommen Sie mit dem Download der PDF-Datei .

Die Software

Verwendete Software:
Fürs Flashen und die Programmierung des ESP:
Thonny oder
µPyCraft
ncat als UDP-Server für Windows
paketsender zum Testen des ESP8266 als UDP-Server

Verwendete Firmware:
MicropythonFirmware

MicroPython-Module und Programme
GPS-Modul für SIM808 und GPS6MV2(U-Blocks)
LCD-Standard-Modul
HD44780U-I2C-Erweiterung zum LCD-Modul
Keypad-Modul
Button Modul
BMP208-Modul
i2cbus-Modul für standardisierten Zugriff auf den Bus
Das erweiterte Hauptprogramm relais.py
server.py das Programm auf der externen Sensoreinheit mit dem ESP8266

Tricks und Infos zu MicroPython

In diesem Projekt wird die Interpretersprache MicroPython benutzt. Der Hauptunterschied zur Arduino-IDE ist, dass Sie die MicroPython-Firmware auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang im ersten Teil des Blogs zu diesem Thema beschrieben.

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Bei der Entwicklung der Software für diesen Blog habe ich von dem direkten Dialog zum ESP32 wieder reichlich Gebrauch gemacht. Das Spektrum reicht von einfachen Tests der Syntax bis zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen. Zu diesem Zweck erstelle ich auch gerne, wie schon bei den vorangegangenen Folgen, kleine Testprogramme. Sie bilden eine Art Macro, weil sie wiederkehrende Befehle zusammenfassen.

Gestartet werden solche Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5. Das geht schneller als der Mausklick auf den Startbutton oder über das Menü **Run**. Die Installation von Thonny habe ich ebenfalls im ersten Teil genau beschrieben.

Greifen wir doch gleich noch ein wenig in die Trickkiste von MicroPython. Wie funktioniert das mit dem UDP-Transfer von Informationen?

Wenn Sie sich über die USB-Leitung mit Ihrem ESP32 unterhalten, verwenden Sie das Prinzip der Nachrichtenübermittlung. Eingaben über die Tastatur werden vom PC zum ESP32 gesendet. REPL, der Kommandozeileninterpreter von MicroPython, empfängt die Nachricht am ESP32 und decodiert sie. Dann schaut der Interpreter des ESP32, was er tun soll. Die Information wird geparst, also auf den Sinninhalt abgeklopft. Daraufhin wird die gewünschte Aktion ausgelöst, deren Ergebnis vom ESP32, implizit als print-Befehl, an den PC zurückgegeben wird. Der empfängt diese Nachricht und stellt sie im Terminalfenster dar.

UDP-Transfers laufen im Prinzip ähnlich. Es gibt aber zwei Unterschiede. Die Übertragungsstrecke ist statt dem Kabel die Funkverbindung, und die Übertragung ist bei UDP nicht gesichert. Das bedeutet, dass das UDP-Protokoll weder sicherstellt, dass die Nachricht angekommen ist, noch dass die Nachricht unverfälscht oder vollständig übertragen wurde. Es ist ferner nicht garantiert, dass die Informationspakete (aka Datagramme) in der Reihenfolge ankommen, in der sie gesendet wurden. Handelt es sich um Informationen mit wenig Inhalt, dann spielt die Reihenfolge schon mal keine Rolle, weil der Inhalt in einem Paket übertragen wird. Das ist bei unserem Projekt der Fall. Bei der Übermittlung von Messwerten spielt es in der Regel auch keine Rolle, wenn bei rascher Abtastung einmal ein Messwert falsch, unvollständig oder gar nicht übertragen wird.

Dafür gibt es einen entscheidenden Vorteil, UDP ist schnell und einfach zu handhaben. Und das Protokoll kann, genau wie TCP, ebenfalls Daten in beide Richtungen zwischen Client und Server übertragen.

Wenn wir voraussetzen, dass die WLAN-Verbindungen bereits bestehen, beschränkt sich die Programmierung von UDP-Client und -Server auf wenige Zeilen.

Client:

```
# clientexample.py
#import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9181)) # IP comes from boot-section
s.settimeout(5.0)
while 1:
    s.sendto("get:poti",("10.0.3.99",8888))
    response, addr = s.recvfrom(256)
    print("Antwort von {} empfangen:{}".format(addr,response))
    # weitere Schleifenbefehle
```

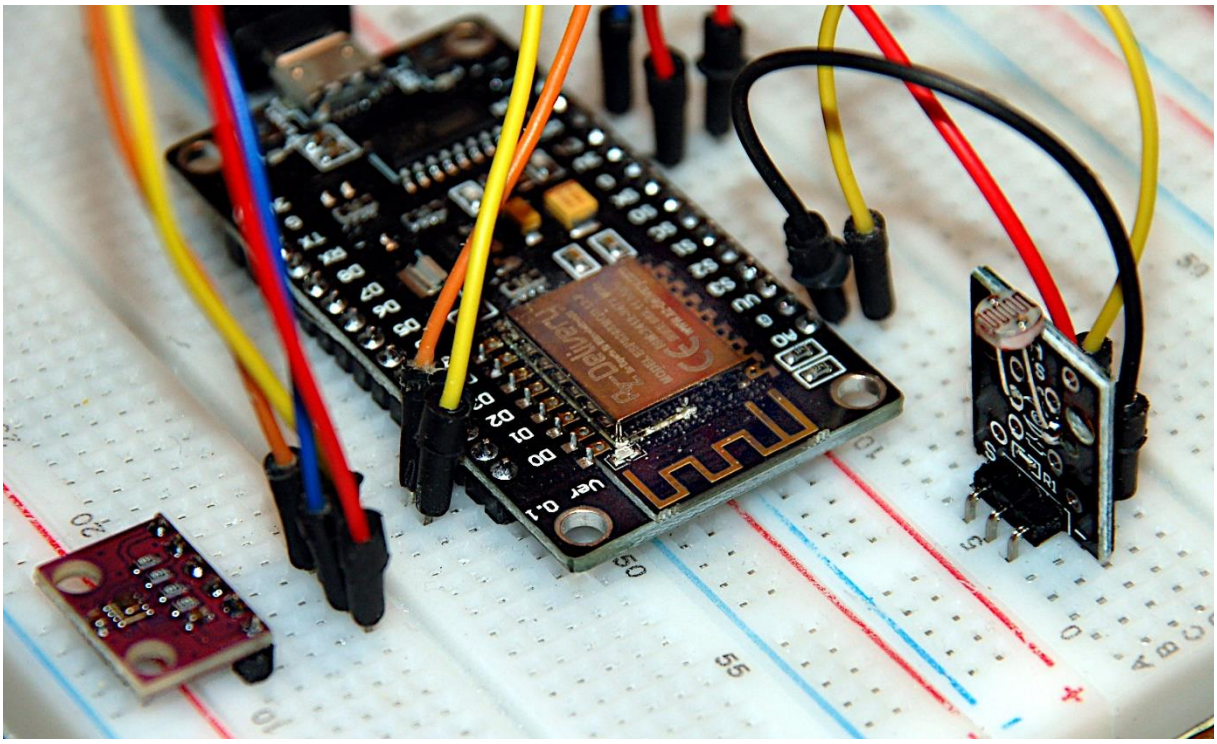
Server:

```
# serverexample.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 8888)) # IP comes from boot-section
s.settimeout(5.0) # Blockadebrecher
while 1:
    request, addr = s.recvfrom(256)
    print("Auftrag von {} empfangen:{}".format(addr,request))
    # Auftrag decodieren, parsen, Aktion auslösen
    ergebnis="irgend ein Wert"
    s.sendto("Messergebnis {}".format(ergebnis),addr)
    # weitere Schleifenbefehle
```

sendto() ist der Ersatz für **print**() und der **input**-Befehl wird durch **receivefrom**() ersetzt.

Die am Server in der Variable **request** ankommende Information ist als Bytes-Objekt codiert und sollte vor der weiteren Verarbeitung zunächst in einen String decodiert werden, der ist leichter zu parsen. Der Absender steht in der Variable **addr**. Der Server parst die Anfrage, erledigt die angefragte Aktion und sendet das Ergebnis an den Absender zurück.

So, wie das hier dargestellt ist, funktioniert auch die Datenübermittlung zwischen externer Sensoreinheit (ESP8266+LDR) und ESP32 im SMS-Relais. Der Umfang der tatsächlichen Programmteile ist größer, weil eben das Parsen einen großen programmtechnischen Aufwand erfordert. Ich habe deshalb, wegen der klareren Darstellung des Kerns der UDP-Übertragung, diesen Teil an den Anfang gestellt. Später werfen wir noch einen genaueren Blick auf die entsprechenden Programmteile. Hier auch schon mal vorab ein Foto von einer Servereinheit. Links unten der BMP280 und rechts der LDR. Am ESP8266 sind noch 5 Pins frei für weitere Sensoren oder Aktoren.



Per SMS externe Sensoren ansprechen

Die Installation von Thonny, der Programmierumgebung wurde in Teil 1 der Reihe behandelt, ebenso das Flashen der Firmware MicroPython. Die entsprechende Firmware muss natürlich auch auf den ESP8266 geflasht werden. Wir haben heute zwei getrennte Einheiten zu programmieren und zu testen. Das muss nach einander geschehen. Zunächst bereiten wir den ESP32 vor.

Der Haupteinsatzzweck des SIM808 in dieser Blogfolge ist die Relaisfunktion. Ich gehe deshalb auch nicht mehr extra auf das Thema GPS ein, das war das Hauptthema in Teil 1 und Teil 2. Wir wollen aber dennoch einmal kurz die Funktion des GSM-Boards ansehen, denn wir brauchen es ja jetzt wegen seiner Fähigkeit SMS-Nachrichten zu versenden.

Weil die UART0-Schnittstelle für REPL reserviert ist, muss eine zweite Schnittstelle für die Konversation mit dem SIM808 vorhanden sein. Der ESP32 stellt eine solche als UART2 bereit. Die Anschlüsse für RXD (Empfang) und TXD (Sendung) können sogar frei gewählt werden. Für einen Vollduplexbetrieb (senden und empfangen gleichzeitig) müssen die Anschlüsse RXD und TXD vom ESP32 zum SIM808 gekreuzt werden. Sie können das am Schaltplan nachvollziehen. Die Defaultwerte am ESP32 sind RXD=16 und TXD=17. Die Organisation des Anschlusses übernimmt das Modul gps.py.

Das beginnt mit dem Einschalten des SIM808. Wenn Sie meiner Empfehlung gefolgt sind und ein Kabel an den Einschalttaster gelötet haben, können Sie das SIM808 jetzt mit folgendem Befehl einschalten, vorausgesetzt, dass dieses Kabel am Pin 4 des ESP32 liegt und die Instanz g der Klasse gps.GSM bereits erzeugt wurde.

```
>>> from gps import *

>>> g=GSM(4)
    I (157300) uart: ALREADY NULL
    GPS initialized, Position:49.28869,11.47506
    SIM808 initialized
    GSM module initialized
>>> g.simOn()
>>>
```

Befehle an das SIM808 werden im AT-Format übermittelt. Es gibt eine riesige Anzahl von Befehlen, die in einer PDF-Datei nachgelesen werden können. Aber keine Sorge, für unser Projekt reichen wenige dieser Befehle. Zwei davon sind in den Methoden **init808**() und **deinit808**() zusammengefasst.

```
    def init808(self):
    self.u.write("AT+CGNSPWR=1\r\n")
    self.u.write("AT+CGNSTST=1\r\n")

     def deinit808(self):
    self.u.write("AT+CGNSPWR=0\r\n")
    self.u.write("AT+CGNSTST=0\r\n")
```

AT+CGNSPWR=1 schaltet die Stromversorgung zum GPS-Modul ein und AT+CGNSTST=1 aktiviert die Übertragung der NMEA-Sätze zum ESP32 über die serielle Schnittstelle UART2. Der Controller empfängt die Informationen des SIM808 und stellt sie in der oben beschriebenen Weise via Terminal und LCD bereit.

Zum genaueren Studium des gps-Moduls folgt nun das Listing. Die drei enthaltenen Klassen GPS, SIM808(GPS) und GSM(SIM808) bauen durch Vererbung einen einheitlichen Namensraum auf, den wir mit **from gps import *** importiert haben. Daher sind alle Methoden in dem Objekt g der Klasse GSM verfügbar. Wird GSM nicht gebraucht (kein SMS-Transfer), dann kann man auch über die Klasse SIM808 einsteigen, wie es im Teil 2 gemacht wurde.

Die Klasse SIM808 kümmert sich um die Hardwaresteuerung und um den Datentransfer zum und vom ESP32. Die Klasse GPS enthält Methoden zur Decodierung der NMEA-Sätze vom SIM808, zur deren Darstellung auf dem Display und zur Kursberechnung. Die GSM-Klasse schließlich stellt die Methoden für SMS-Transfer und Verwaltung zur Verfügung. Genau damit werden wir uns nun beschäftigen und außerdem im Anschluss den UDP-Datenverkehr näher anschauen.

```
"""
1234567890123456789012345678901234567890123456789012345678901234567890123456
File: gps.py
Author: J. Grzesina
Rev. 1.0: AVR-Assembler
Rev. 2.0: Adaption auf Micropython
------------------
Die enthaltenen Klassen sprechen einen ESP32 Controller an.
Dieses Modul beherbergt GPS, GPS6MV2 und SIM808 und GSM.
```

```
GPS stellt Methoden zur Decodierung u. Verarbeitung der
NMEA-Saetze $GPGAA und $GPRMC bereit, welche die wesentlichen
Infos zur Position, Hoehe und Zeit einer Position liefern.
Sie werden dann angezeigt, wenn die Datensaetze als "gueltig"
gemeldet werden. Eine Skalierung auf weitere NMEA-Sätze ist
jederzeit möglich.
GPS6MV2 und SIM808 beziehen sich auf die entsprechende Hardware.
GSM bietet Methode für das Senden von SMS-Nachrichten.
"""
from machine import UART,I2C,Pin
import sys
from time import sleep, time, ticks_ms
from math import *

# ********************** Beginn GSM **********************
class GPS:
    #
    gDeg=const(0)
    gFdeg=const(1)
    gMin=const(1)
    gFmin=const(2)
    gSec=const(2)
    gFsec=const(3)
    gHemi=const(4)

    #DEFAULT_TIMEOUT=500
    #CHAR_TIMEOUT=200

    def __init__(self,disp=None,key=None):  # disp mit OLED-API
        self.u=UART(2,9600)
        # u=UART(2,9600,tx=19,rx=18)  # mit alternativen Pins
        self.display=disp
        self.key=key
        self.timecorr=2
        self.Latitude=""
        self.Longitude=""
        self.Time=""
        self.Date=""
        self.Height=""
        self.Valid=""
        self.Mode="DMF" # default
        self.AngleModes=["DDF","DMS","DMF"]
        self.displayModes=["time","height","pos"]
        self.DMode="pos"
        # DDF = Degrees + DegreeFractions
        # DMS = Degrees + Minutes + Seconds + Fractions
        # DMF = Degrees + Minutes + MinuteFraktions
        self.DDLat=49.28868056  # aktuelle Position
        self.DDLon=11.47506105
        self.DDLatOld=49.3223 # vorige Position
        self.DDLonOld=11.5000
        self.zielPtr=0
```

```
        self.course=0
        self.distance=0
        print("GPS initialized, Position:{},{}".format\
              (self.DDLat,self.DDLon))

    def decodeLine(self,zeile):
        latitude=["","","","","N"]
        longitude=["","","","","E"]
        angleDecimal=0
        def formatAngle(angle):  # Eingabe ist Deg:Min:Fmin
            nonlocal angleDecimal
            minute=int(angle[1])     # min als int
            minFrac=float("0."+angle[2]) # minfrac als float
            angleDecimal=int(angle[0])+(float(minute)+minFrac)/60
            if self.Mode == "DMS":
                seconds=minFrac*60
                secInt=int(seconds)
                secFrac=str(seconds - secInt)
                a=str(int(angle[0]))+"*"+angle[1]+"'"+\
                   str(secInt)+secFrac[1:6]+'"'+angle[4]
            elif self.Mode == "DDF":
                minutes=minute+minFrac
                degFrac=str(minutes/60)
                a=str(int(angle[0]))+degFrac[1:]+"* "+angle[4]
            else:
                a=str(int(angle[0]))+"*"+angle[1]+"."+\
                   angle[2]+"' "+angle[4]
            return a

        # GPGGA-Fields
        nmea=[0]*16
        name=const(0)
        time=const(1)
        lati=const(2)
        hemi=const(3)
        long=const(4)
        part=const(5)
        qual=const(6)
        sats=const(7)
        hdop=const(8)
        alti=const(9)
        auni=const(10)
        geos=const(11)
        geou=const(12)
        aged=const(13)
        trash=const(14)
        nmea=zeile.split(",")
        lineStart=nmea[0]
        if lineStart == "$GPGGA":
            self.Time=str((int(nmea[time][:2])+\
                           self.timecorr)%24)+":"+\
                           nmea[time][2:4]+":"+nmea[time][4:6]
```

```
            latitude[gDeg]=nmea[lati][:2]
            latitude[gMin]=nmea[lati][2:4]
            latitude[gFmin]=nmea[lati][5:]
            latitude[gHemi]=nmea[hemi]
            longitude[gDeg]=nmea[long][:3]
            longitude[gMin]=nmea[long][3:5]
            longitude[gFmin]=nmea[long][6:]
            longitude[gHemi]=nmea[part]
            self.Height,despose=nmea[alti].split(".")
            self.Latitude=formatAngle(latitude)
            self.DDLat=angleDecimal
            self.Longitude=formatAngle(longitude)
            self.DDLon=angleDecimal
        if lineStart == "$GPRMC":
            date=nmea[9]
            self.Date=date[:2]+"."+date[2:4]+"."+date[4:]
            try:
                self.Valid=nmea[2]
            except:
                self.Valid="V"

    def waitForLine(self,title,delay=2000):
        line=""
        c=""
        d=delay
        if delay < 1000: d=1000
        start = ticks_ms()
        end=start+d
        current=start
        while current <= end:
            #print(end-current)
            if self.u.any():
                c=self.u.read(1)
                if ord(c) <=126:
                    c=c.decode()
                    if c == "\n":
                        test=line[0:6]
                        if test==title:
                            #print(line)
                            return line
                        else:
                            line=""
                    else:
                        if c != "\r":
                            line +=c
            current = ticks_ms()
            sleep(0.05)
        return ""

    def showData(self):
        if self.display:
            if self.DMode=="time":
```

```
                self.display.writeAt("Date:{}".format\
                                     (self.Date),0,0)
                self.display.writeAt("Time:{}".format\
                                     (self.Time),0,1)
            if self.DMode=="height":
                self.display.writeAt("Height: {}m   ".format\
                                     (self.Height),0,0)
                self.display.writeAt("Time:{}".format\
                                     (self.Time),0,1)
            if self.DMode=="pos":
                self.display.writeAt(self.Latitude+" "*\
                                     (16-len(self.Latitude)),0,0)
                self.display.writeAt(self.Longitude+" "*\
                                     (16-len(self.Longitude)),0,1)

    def printData(self):
        print(self.Date,self.Time,sep="_")
        print("LAT",self.Latitude)
        print("LON",self.Longitude)
        print("ALT",self.Height)

    def showError(self,msg):
            if self.display:
                self.display.clearAll()
                self.display.writeAt(msg,0,0)
            print(msg)

    def storePosition(self):  # aktuelle Position als DD.dddd
merken
        lat=str(self.DDLat)+","
        lon=str(self.DDLon)+"\n"
        try:
            D=open("stored.pos","wt")
            D.write(lat)
            D.write(lon)
            D.close()
            if self.display:
                self.display.clearAll()
                self.display.writeAt("Pos. stored",0,0)
                sleep(3)
                self.display.clearAll()
        except (OSError) as e:
            enumber=e.args[0]
            if enumber==2:
                print("Not stored")
                if self.display:
                    self.display.clearAll()
                    self.display.writeAt("act. Position",0,0)
                    self.display.writeAt("not stored",0,0)
                    sleep(3)
                    self.display.clearAll()
```

```
    def chooseDestination(self, wait=3):
        if not self.display: return None
        self.display.clearAll()
        self.display.writeAt("ENTER=RST-Button",0,0)
        n="positions.pos"
        try:
            D=open(n,"rt")
            ziel=D.readlines()
            D.close()
            i = 0
            while 1:
                lat,lon=(ziel[i].rstrip("\r\n")).split(",")
                self.display.clearAll()
                self.display.writeAt("{}. {}".format(i,lat),0,0)
                self.display.writeAt("   {}".format(lon),0,1)
                sleep(wait)
                if self.key.value()==0: break
                i+=1
                if i>=len(ziel): i=0
            self.zielPtr=i
            self.display.clearAll()
            self.display.writeAt("picked: {}".format(i),0,0)
            sleep(wait)
            self.display.clearAll()
            lat,lon=ziel[i].split(",")
            lon=lon.strip("\r\n")
            print("{}. Lat,Lon: {}, {}".format(i,lat,lon))
            lat=float(lat)
            lon=float(lon)
            return (lat,lon)
        except (OSError) as e:
            enumber=e.args[0]
            if enumber==2: print("File not found")
            self.display.clearAll()
            self.display.writeAt("There is no",0,0)
            self.display.writeAt("Positionfile",0,1)
            sleep(3)
            self.display.clearAll()
            return (0,0)

    def calcNewCourse(self,delay=3): # von letzter Position bis
hier
        lat,lon=self.chooseDestination(delay)
        if lat==0 and lon==0: return
        dy=(lat-self.DDLat)*60*1852
        dx=(lon-self.DDLon)*60*1852*cos(radians(self.DDLatOld))
        print("Start {},{}".format(self.DDLat,self.DDLon),\
              "  Ziel {},{}".format(lat,lon))
        #print("Ziel-Start",self.DDLon-self.DDLonOld,\
        #self.DDLat-self.DDLatOld)
        return self.calcCourse(dx,dy)
```

```
    def calcLastCourse(self): # von letzter Position bis hier
        try:
            D=open("stored.pos","rt")
            lat,lon=(D.readline()).split(",")
            D.close()
            self.DDLatOld=float(lat)
            self.DDLonOld=float(lon)
        except:
            self.DDLatOld=49.3223
            self.DDLonOld=11.50
        dy=(self.DDLat-self.DDLatOld)*60*1852
        dx=(self.DDLon-self.DDLonOld)*60*1852*\
            cos(radians(self.DDLatOld))
        print("Start {},{}".format\
              (self.DDLonOld,self.DDLatOld),\
              "  Ziel {},{}".format(self.DDLon,self.DDLat))
        #print("Ziel-Start",self.DDLon-self.DDLonOld,\
        #self.DDLat-self.DDLatOld)
        return self.calcCourse(dx,dy)

    def calcCourse(self,dx,dy): # von letzter Position bis hier
        course=0
        distance=0
        #print(dx,dy,degrees(atan2(dy,dx)))
        if abs(dx) < 0.0002:
            if dy > 0:
                course=0
                #print("Trace: 1")
            if dy < 0:
                course=180
                #print("Trace: 2")
            if abs(dy) < 0.0002:
                course=None
                #print("Trace: 3")
        else:  # dx >= 0.0002
            if abs(dy) < 0.0002:
                if dx > 0:
                    course=90
                    #print("Trace: 4")
                if dx < 0:
                    course=270
                    #print("Trace: 5")
            else:  ## dy > 0.0002
                course=90-degrees(atan2(dy,dx))
                #print("Trace: 6")
                if course > 360:
                    course -= 360
                    #print("Trace: 7")
                if course < 0:
                    course += 360
                    print("Trace: 8")
        self.course=int(course)
```

```
        self.distance=int(sqrt(dx*dx+dy*dy))
        print("Distance: {}, Course: {}".format\
              (self.distance,self.course))
        return (self.distance,self.course)

# ********************** Ende GPS **************************

# ******************** Beginn SIM808 **********************
class SIM808(GPS):
    DEFAULT_TIMEOUT=const(500)
    CHAR_TIMEOUT=const(100)
    CMD=const(1)
    DATA=const(0)

    def __init__(self,switch=4,disp=None,key=None):
        self.switch=Pin(switch,Pin.OUT)
        self.switch.on()
        super().__init__(disp,key)
        self.display=disp
        self.key=key
        print("SIM808 initialized")

    def simOn(self):
        self.switch.off()
        sleep(1)
        self.switch.on()
        sleep(3)

    def simOff(self):
        self.switch.off()
        sleep(3)
        self.switch.on()
        sleep(3)

    def simStartPhone():
        pass

    def simGPSInit(self):
        self.u.write("AT+CGNSPWR=1\r\n")
        self.u.write("AT+CGNSTST=1\r\n")

    def simGPSDeinit(self):
        self.u.write("AT+CGNSPWR=0\r\n")
        self.u.write("AT+CGNSTST=0\r\n")

    def simStopGPSTransmitting(self):
        self.u.write("AT+CGNSTST=0\r\n")

    def simStartGPSTransmitting(self):
        self.u.write("AT+CGNSTST=1\r\n")

    def simFlushUART(self):
```

```
        while self.u.any():
            self.u.read()

    # Wartet auf Zeichen an UART -> 0: keine Zeichen bis Timeout
    def simWaitForData(self,delay=CHAR_TIMEOUT):
        noOfBytes=0
        start=ticks_ms()
        end=start+delay
        current=start
        while current <= end:
            sleep(0.1)
            noOfBytes=self.u.any()
            if noOfBytes>0:
                break
        return noOfBytes

    def simReadBuffer(self,cnt,tout=DEFAULT_TIMEOUT,\
                      ctout=CHAR_TIMEOUT):
        i=0
        strbuffer=""
        start=ticks_ms()
        prevchar=0
        while 1:
            while self.u.any():
                c=self.u.read(1)
                c=chr(ord(c))
                prevchar=ticks_ms()
                strbuffer+=c
                i+=1
                if i>=cnt: break
            if i>= cnt:break
            if ticks_ms()-start > tout: break
            if ticks_ms()-prevchar > ctout: break
        return (i,strbuffer) # gelesene Zeichen

    def simSendByte(self,data):
        return self.u.write(data.to_bytes(1,"little"))

    def simSendChar(self,data):
        return self.u.write(data)

    def simSendCommand(self,cmd):
        self.u.write(cmd)

    def simSendCommandCRLF(self,cmd):
        self.u.write(cmd+"\r\n")

    def simSendAT(self):
        return self.simSendCmdChecked("AT","OK",CMD)

    def simSendEndMark(self):
        self.simSendChar(chr(26))
```

```
    def simWaitForResponse(self,resp,typ=DATA,tout=\
                           DEFAULT_TIMEOUT,ctout=CHAR_TIMEOUT):
        l=len(resp)
        s=0
        self.simWaitForData(500)
        start=ticks_ms()
        prevchar=0
        while 1:
            if self.u.any():
                c=self.u.read(1)
                if ord(c) < 126:
                    c=c.decode()
                    prevchar=ticks_ms()
                    s=(s+1 if c==resp[s] else 0)
                if s == l: break
            if ticks_ms()-start > tout: return False
            if ticks_ms()-prevchar > ctout: return False
        if type==CMD:
            self.simFlushUART()
        return True

    def simSendCmdChecked(self,cmd,response,typ,tout=\
                          DEFAULT_TIMEOUT,ctout=CHAR_TIMEOUT):
        self.simSendCommand(cmd)
        return self.simWaitForResponse(response,typ,tout,ctout)

# ********************** Ende  SIM808 ************************

# ************************ Beginn GSM ************************

class GSM(SIM808):

    def __init__(self, switch=4, disp=None, key=None):
        super().__init__(switch,disp,key)
        try:
            self.gsmInit()
            print("GSM module initialized")
        except:
            raise OSError("GSM-Init failed")

    def gsmInit(self):
        if not self.simSendCmdChecked("AT\r\n","OK\r\n",CMD):
            return False
        if not self.simSendCmdChecked("AT+CFUN=1\r\n",\
                                      "OK\r\n",CMD):
            return False
        if not self.gsmCheckSimStatus():
            return False
        return True

    def gsmIsPowerUp(self):
```

```
        return self.simSendAT()

    def gsmPowerOn(self):
        self.switch.off()
        sleep(1)
        self.switch.on()
        sleep(3)

    def gsmPowerReset(self):
        self.switch.off()
        sleep(3)
        self.switch.on()
        sleep(3)
        self.switch.off()
        sleep(1)
        self.switch.on()
        sleep(3)

    def gsmCheckSimStatus(self):
        n=0
        a=""
        while n < 3:
            self.simFlushUART()
            self.simSendCommand("AT+CPIN?\r\n")
            self.simWaitForData()
            a=self.simReadBuffer(50)
            #print(a)
            if "+CPIN: READY" in a[1]:
                break
            n+=1
            sleep(0.3)
        if n == 3:
            return False
        return True


    def gsmSendSMS(self,phoneNbr,mesg):
        if not self.simSendCmdChecked\
                ("AT+CMGF=1\r\n","OK\r\n"\
                                        ,CMD):
            print("SMS-Mode not selcted")
            return False
        sleep(0.5)
        self.simFlushUART()
        if not self.simSendCmdChecked('AT+CMGS="'+phoneNbr+\
                                        '"\r\n',">",CMD):
            print("Phonenumber Problem")
            return False
        sleep(1)
        self.simSendCommand(mesg)
        sleep(0.5)
        self.simSendEndMark()
```

```
        sleep(1)
        return self.simWaitForResponse(mesg,CMD)
        #return self.simReadBuffer(50)

    def gsmAreThereSMS(self,stat):
        buf=""
        # SMS-Modus aktivieren
        self.simSendCmdChecked("AT+CMGF=1\r\n","OK\r\n",CMD)
        sleep(1)
        self.simFlushUART()
        # ungelesene SMS listen ohne Statusänderung ",1"
        #print("SMS-Status",stat)
        self.simSendCommand('AT+CMGL="{}",1\r\n'.format(stat))
        sleep(2)
        # OK findet sich in den ersten 30 Zeichen nur, wenn
        # keine ungelesene SMS vorliegt
        a=self.simReadBuffer(30)[1]
        #print(30,a)
        if "OK" in a:
            sleep(0.1)
            return 0
        else:
            # restliche Zeichen im UART-Buffer entsorgen
            self.simFlushUART()
            # erneuter Aufruf zum Einlesen
            self.simSendCommand('AT+CMGL="{}",1\r\n'.format\
                                (stat))
            sleep(2)
            a=self.simReadBuffer(48)[1]
            #print(48,a)
            # suche nach der Position von "+CMGL:"
            p=a.find("+CMGL:")
            if p != -1:
                pkomma=a.find(",",p)
                #print("gefunden",a[p+6:pkomma])
                return int(a[p+6:pkomma])
            else:
                #print("CMGL not found", a)
                return None
        #print("Kein 'OK'")
        return None

    def gsmReadAll(self,stat,cnt=500):
        # SMS-Modus aktivieren
        self.simSendCmdChecked("AT+CMGF=1\r\n","OK\r\n",CMD)
        sleep(1)
        self.simFlushUART()
        # SMS listen ohne Statusänderung ",1"
        self.simSendCommand('AT+CMGL="{}",1\r\n'.format(stat))
        sleep(2)
        a=self.simReadBuffer(cnt)
        #print("BUFFER:\n",a[1],"\n")
```

```
        return a

    def gsmFindSMS(self,stat="ALL",cnt=150):
        Liste=[]
        i=0
        p0=0
        again=-1
        self.simFlushUART()
        m,a=self.gsmReadAll(stat,cnt)
        while again == -1:
            again=a.find("OK")  # OK gefunden? letzter Durchgang
            #print("again",again) # only for debugging
            #print("@@@@",a,"@@@@") # only for debugging
            while 1:
                merker=p0
                #print("merker: ",merker) # only for debugging
                p0=a.find("+CMGL:",p0)
                #print("While1_p0",p0)
                if p0 == -1:
                    p0=merker
                    #print("@p0_break",p0) # only for debugging
                    break
                p1=a.find(",",p0)
                #print("p0 und p1: ",p0,p1) # only for debugging
                if p1 == -1:
                    p0=merker
                    #print("@p1_break",p0) # only for debugging
                    break
                n=int(a[p0+6:p1])
                #print("Liste: ",Liste,"neu: ",n)# for debugging
                Liste.append(n)
                p0=p1
            m,x=self.simReadBuffer(cnt)
            #print(p0,": Buffer: \n",a[p0:],"*****",x,"<<<<<") \
            # only for debugging
            a=a[p0:]+x
            p0=0
            if m == 0:
                break
        return Liste

    def gsmReadSMS(self,index,mode=0):
        # mode=1: Status nicht veraendern
        # SMS-Modus einschalten
        self.simSendCmdChecked("AT+CMGF=1\r\n","OK\r\n",CMD)
        sleep(1)
        self.simFlushUART()
        try:
            if mode==1:
                self.simSendCommand("AT+CMGR={},1\r\n".format\
                                    (index))
            elif mode==0:
```

```
            self.simSendCommand("AT+CMGR={}\r\n".format\
                                (index))
        sleep(1)
        a=self.simReadBuffer(250) # a =(Anzahl, Zeichen)
        #print(a[1]) # only for debugging
        if a[0] != 0:
            a=a[1].split(',',4+mode)
            #print(a)  # only for debugging
            p0=a[0+mode].find('"')
            p1=a[0+mode].find('"',p0+1)
            Status=a[0+mode][p0+1:p1]
            Phone=a[1+mode].strip('"')
            Date=a[3+mode].lstrip('"')
            Time=a[4+mode].split('"')[0]
            Message=a[4+mode].split('"')[1].strip\
                    ("\r\n").rstrip("\r\nOK")
            return (Status,Phone,Date,Time,Message)
        else:
            return None
    except (IndexError,TypeError) as e:
        print("Index out if range",e.args[0])
        return None

def gsmShowAndDelete(self,stat,disp=None,delete=None):
    SMSlist=self.gsmFindSMS(stat, cnt=100)
    while 1:
        if len(SMSlist)==0:
            print("nothing to do")
            break
        for i in SMSlist:
            response=self.gsmReadSMS(i,mode=1)
            print(i,response[0],"\n",response[4])
            if disp:
                disp.clearAll()
                disp.writeAt("{}. {}".format\
                             (i,response[0]),0,0)
                disp.writeAt(response[4],0,1)
                sleep(2)
            if delete or not(response[1]==\
                             '+4917697953675'):\
                #evtl. weitere PhoneNbr + stati
                if self.gsmDeleteSMS(i):
                    print("!!! deleted",response[1])
        first=SMSlist[0]
        SMSlist=self.gsmFindSMS(stat,cnt=100)
        if len(SMSlist)==0 or SMSlist[0] == first: break
    if disp: disp.clearAll()

def gsmDeleteSMS(self,index):
    print("SMS[{}] deleted".format(index))
    return self.simSendCmdChecked\
    ("AT+CMGD={},0\r\n".format\
```

```
                                        (index),"OK\r\n",CMD)

# ********************** Ende GSM ************************

# ******************** Beginn GPS6MV2 **********************
class GPS6MV2(GPS):
    # Befehlscodes setzen
    GPGLLcmd=const(0x01)
    GPGSVcmd=const(0x03)
    GPGSAcmd=const(0x02)
    GPVTGcmd=const(0x05)
    GPRMCcmd=const(0x04)

    def __init__(self,delay=1,disp=None,key=None):
        super().__init__(disp,key)
        self.display=disp
        self.delay=delay  # GPS sendet alle delay Sekunden
        period=delay*1000
        SetPeriod=bytearray([0x06,0x08,0x06,0x00,period&0xFF,\
                        (period>>8)&0xFF,0x01,0x00,0x01,0x00])
        self.sendCommand(SetPeriod)
        self.sendScanOff(bytes([GPGLLcmd]))
        self.sendScanOff(bytes([GPGSVcmd]))
        self.sendScanOff(bytes([GPGSAcmd]))
        self.sendScanOff(bytes([GPVTGcmd]))
        self.sendScanOn(bytes([GPRMCcmd]))
        print("GPS6MV2 initialized")

    def sendCommand(self,comnd):  # comnd ist ein bytearray
        self.u.write(b'\xB5\x62')
        a=0; b=0
        for i in range(len(comnd)):
            c=comnd[i]
            a+=c  # Fletcher Algorithmus
            b+=a
            self.u.write(bytes([c]))
        self.u.write(bytes([a&0xff]))
        self.u.write(bytes([b&0xff]))

    def sendScanOff(self,item):  # item ist ein bytes-objekt
        shutoff=b'\x06\x01\x03\x00\xF0'+item+b'\x00'
        self.sendCommand(shutoff)

    def sendScanOn(self,item):
        turnon=b'\x06\x01\x03\x00\xF0'+item+b'\x01'
        self.sendCommand(turnon)
```

Zu den AT-Befehlen für das Einschalten der GPS-Einheit und das Öffnen der UART-Verbindung auf dem SIM808 gesellen sich für den SMS-Betrieb einige weitere, die wir jetzt in Augenschein nehmen.

Grundsätzlich sind AT-Befehle mit einem \r\n (Wagenrücklauf und Zeilenvorschub = \x0D und \x0A) abzuschließen, damit sie vom SIM808 als solche erkannt werden. Diese Zeichen werden zum Beispiel auch erzeugt, wenn Sie die Enter-Taste drücken. Bei den AT-Befehlen müssen die Zeichen aber separat angegeben werden, wie in den weiter unten folgenden Beispielen.

Das SIM808 sendet Antworten oder Ergebnisse ebenfalls mit \r\n zurück. Außer bei Nutzdaten, wie Nachrichtentexten, ist es für uns nur wichtig, ob die Antwort der erwarteten Zeichenfolge entspricht. Es gibt daher eine Methode, die genau diese Überprüfung durchführt, **simSendCmdChecked**(). Sie gibt **True** als Ergebnis zurück, falls die Überprüfung erfolgreich war, andernfalls **False**. Als Parameter nimmt die Methode den AT-Kommandostring, die erwartete Antwort, den Befehlstyp (Kommando oder Daten) sowie zwei optionale Timeoutwerte. Der erste davon limitiert die Zeitdauer für die gesamte Rückantwort vom SIM808, der zweite legt das Zeitlimit pro Zeichen fest. Beide verhindern so, dass sich die Methode festfrisst und das ganze Programm blockiert. Das Klassenattribut CMD (Wert = 1) kennzeichnet über den typ-Parameter den AT-Befehl als Kommando. Bei Antworten auf einen Befehl wird nach dem Erkennen der vorgegebenen Antwort der Rest des UART-Puffers auf dem SIM808 geflusht, also geleert, was bei einer Textantwort natürlich nicht gewünscht ist. Das Einlesen und Überprüfen der Antwort vom SIM808 übernimmt die Methode **simWaitForResponse**().

Hier gleich ein Beispiel. Damit SMS-Betrieb möglich ist, muss der 7-Bit-Textmodus eingeschaltet werden. War die Aktion erfolgreich, dann enthält die Antwort ein 'OK', auf dessen Existenz wir prüfen müssen.

```
simSendCmdChecked("AT+CMGF=1\r\n","OK\r\n",CMD)
```

AT+CMGF=1: schaltet auf 7-Bit-Textbetrieb um, und ermöglich so den Klartext-SMS-Betrieb. Mit AT+CMGF=0 arbeitet das SIM808 im PDU-Betrieb (von **P**hysical **D**ata **U**nit). In diesem Modus werden die 7-Bit-Zeichen in einen 8-Bit-Datenstrom eingebettet, der keinen Klartext, sondern nur Hieroglyphen in der Ausgabe ergibt.

Der Status von SMS-Nachrichten erlaubt deren grobe Auswahl zum Beispiel beim Listbefehl. Die Variable **stat** enthält dafür, neben weiteren möglichen, einen der folgenden Strings: "ALL", "REC UNREAD" oder "REC READ". Dieser Wert wird durch die Formatanweisung statt der geschweiften Klammern in den Befehl eingebaut und durch die beiden doppelten Anführungszeichen eingeschlossen. Der gesamte AT-Befehl steht hier in einfachen Anführungszeichen.

```
simSendCommand('AT+CMGL="{}",1\r\n'.format(stat))
```

Die '**1**' sorgt dafür, dass sich beim Auflisten der Status der gelisteten Nachrichten nicht ändert. Beachten Sie bitte, dass der Statusstring in Anführungszeichen stehen muss. Folgende Zeichenfolge wird durch den obigen Befehl an das SIM808 gesendet:

```
AT+CMGL="REC UNREAD",1,\r\n
```

Beim Versenden von SMS-Nachrichten muss dem Befehl als erstes die Nummer des Anschlusses mitgeteilt werden. Das SIM808 antwortet darauf mit einem ">".

```
simSendCmdChecked('AT+CMGS="'+phoneNbr+'"\r\n',">",CMD)
```

Wurde ">" erkannt, kann die Nachricht gesendet werden.

```
simSendCommand(mesg)
```

Das Ende der Nachricht wird der Gegenstation durch Senden eines Textendezeichens (chr(26) = \x1A = Strg+Z) bekannt gegeben.

Um eine SMS-Nachricht aus dem Speicher des SIM808 zu lesen, muss deren Index bekannt sein. Nach Absetzen des Lesebefehls, kann der UART-Puffer des SIM808 ausgelesen werden. Er enthält den Text der Nachricht. Wir warten bis wenigstens ein Zeichen vorliegt und lesen dann **anzahlZeichen** Bytes ein. Der Wert dieser Variable sollte an die Länge der Nachricht angepasst werden. Berücksichtigen Sie aber, dass neben dem Text auch weitere Angaben wie Datum, Zeit und Status zur Gesamtlänge beitragen. Die '**1**' sorgt dafür, dass sich beim Auflisten der Status nicht ändert.

```
simSendCommand("AT+CMGR={},1\r\n".format(index))
simWaitForData()
simReadBuffer(anzahlZeichen)
```

Nachrichten können durch Angabe des Index auch gelöscht werden.

```
simSendCmdChecked("AT+CMGD={},0\r\n".format(index),"OK\r\n",CMD)
```

Erst nachdem eine SIM-Karte erkannt wurde, ist der SMS-Betrieb möglich. Der Befehl

```
simSendCommand("AT+CPIN?\r\n")
```

überprüft das.

Die Methoden der Klasse GSM bedienen sich zur Ausführung der Befehle aus der SIM808-Klasse. Folgende Liste stellt die SMS-Methoden zusammen.

GSM	Positions-Parameter: - optionale Parameter: switch=sp Rückgabe: - sp = Nummer des Schaltausgangs am ESP32
gsmInit	Positions-Parameter: optionale Parameter: Rückgabe: Bemerkung: SIM808 eingeschaltet, Volle Funktionalität an SIM-Karte vorhanden?
gsmIsPowerUp	Positions-Parameter: optionale Parameter: Rückgabe: True/False Bemerkung: sendet AT\r\n testet auf OK

gsmPowerOn	Positions-Parameter: optionale Parameter: Rückgabe: Bemerkung: Board anschalten
gsmPowerReset	Positions-Parameter: - optionale Parameter: - Rückgabe: - Bemerkung: Board reset
gsmCheckSimStatus	Positions-Parameter: - optionale Parameter: - Rückgabe: +CPIN: READY Bemerkung: Prüft aus SIM-Karte
gsmSendSMS	Positions-Parameter: HandyNummer, Nachricht optionale Parameter: - Rückgabe: Nachricht Bemerkung: Sendet Nachricht an HandyNummer
gsmAreThereSMS	Positions-Parameter: stat optionale Parameter: Rückgabe: Index der ersten gefundenen Nachricht Bemerkung: Sucht nach Nachrichten mit dem Status in stat
gsmReadAll	Positions-Parameter: stat optionale Parameter: cnt=anzahlZeichen Rückgabe: höchstens cnt Zeichen aus dem UART-Puffer Bemerkung:
gsmFindSMS	Positions-Parameter: optionale Parameter: stat, cnt Rückgabe: Liste der Indizes der Nachrichten mit dem Status in stat Bemerkung: Es werden nur so viele Indizes erfasst, wie SMS-Inhalte in den UART-Buffer passen
gsmReadSMS	Positions-Parameter: index optionale Parameter: mode=0 Rückgabe: (Status, Phone, Date, Time, Message) Bemerkung: Holt die Nachricht mit der Nummer index aus dem SIM808-Speicher und gibt den Inhalt als Tupel zurück. mode=1 ändert den Status der Nachricht im SIM808-Speicher nicht. Eine 0 setzt den Status auf "REC READ".

gsmShowAndDelete	Positions-Parameter: stat optionale Parameter: disp=None, delete=None Rückgabe: Bemerkung: Erstellt eine Liste der Indizes der SMS vom Status in stat.Optionale Ausgabe auf dem LCD/OLED; fremde Mails und werden stets gelöscht,eigene nur, wenn delete=True
gsmDeleteSMS	Positions-Parameter: index optionale Parameter: Rückgabe: True/False Bemerkung: Löscht die angegebene Nachricht

Ein Teil der GSM-Methoden dient der internen Verarbeitung und Steuerung. Das Hauptprogramm relais.py zeigt die Anwendung der Klasse.

Das Anwendungsprogramm relais.py ist im Vergleich zur reinen GPS-Anwendung aus Teil3 erheblich umfangreicher geworden. Das liegt vor allem an der Einbindung der WLAN-Funktionalität, aber auch an dem hinzugefügten Beispiel für die Abfrage des Funksensors über UDP. Das Programm hat schon einiges zu bieten. Im Einzelnen demonstriert es:

- zeitgesteuerte SMS (alle x Stunden, Minuten…)
- eventgesteuerte SMS (Temperatur außerhalb eines Bereichs)
- SMS on demand (Antwort auf eine SMS)
- GPS-Tracker (Entfernung überschritten oder Wegpunktübermittlung)
- Messwert übermitteln
- Funksensor via UDP abfragen

Der letzte Punkt ist das Kernthema dieses Beitrags, alles andere wurde schon in Episode 3 besprochen. Deshalb schauen wir uns jetzt die entsprechenden Programmsnippets an. Danach beleuchten wir das Serverprogramm auf dem ESP8266.

Die Netzwerkverbindung kann entweder über den Accesspoint des WLAN-Routers oder direkt über den Accesspoint des ESP32 erfolgen. Natürlich ist nicht nur ein Funksensor ansprechbar. Nur eine Sache muss man beachten: Die ESPs müssen alle im gleichen WLAN-Teilnetz liegen. Dieses wird durch den WLAN-Router vorgegeben, falls ein solcher benutzt werden soll. Unbedingt nötig ist das nicht, denn der ESP8266 kann einen eigenen Accesspoint zur Verfügung stellen. Beide Ansätze haben ihre Vor- und Nachteile.

Mit WLAN-Router als Accesspoint:
Sowohl der ESP32, mit dem SIM808 im Anhang, als auch der/die ESP8266 starten im Station-Modus. Auf einfache Weise können mehrere ESP8266, lediglich durch Angabe der IP-Adresse, vom ESP32 angesprochen werden. Eine weitere Differenzierung bieten verschiedene Portnummern. Eine Adressierung über Broadcast ist möglich.

Mit ESP8266-eigenem Accesspoint:
Der ESP32 muss sich beim Start mit einem einzelnen Accesspoint verbinden. Nur der Server auf dieser Station kann demnach angesprochen werden. Damit weitere ESP8266 angesprochen werden können, müsste der ESP32 die erste WLAN-Verbindung beenden, um danach eine andere aufzubauen. Das ist aber wesentlich aufwändiger.

Weil sich außerdem so die Testumgebung einfacher gestaltet, habe ich die erste Lösung umgesetzt. Dennoch kann auf die zweite umgeschaltet werden. Am ESP32 geschieht das durch die Taste RST am LCD-Keypad beim Start. Das Display informiert über den Zeitpunkt. Wird die Taste nicht betätigt, startet die zweite Lösung, und der ESP32 versucht, den Accesspoint des im Programm angegebenen ESP8266 zu finden.

Auf dem ESP8266 ist die erste Lösung auskommentiert. Durch entkommentieren dieses Bereichs und Auskommentieren der Routerlösung, kann auch hier die Betriebsart umgestellt werden. Natürlich können Sie auch eine ähnliche Steuerung mit Taste, wie auf dem ESP32, etablieren.

Die folgenden Codeteile finden Sie in der Datei relais.py. Sie sind einzeln nicht lauffähig, sondern stellen nur die Grundlage zu ihrer Besprechung dar. Für ein genaueres Studium schlage ich vor, das Programm herunterzuladen und parallel zur Besprechung durchzugehen.

Diese Import-Zeilen, gleich am Anfang, bereiten den Netzwerkzugriff vor.

```
#
import ubinascii
import network
try:
import usocket as socket
except:
import socket
import ubinascii
```

Ein weiteres Steuerflag aktiviert die Abfrage des Funksensors

```
queryFlag   =0b00010000 # Abfragen externer Sensoren per UDP
```

Die Zeitsteuerung für die Fernabfrage wird eingerichtet

```
queryEnde=time()+5
```

Eine Struktur und eine Funktion für die Herstellung der WLAN-Verbindung werden definiert.

```
# Die Dictionarystruktur (dict) erlaubt spaeter die
Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
```

```
    201:  "NO AP FOUND",
    5:    "GOT_IP"
    }

#********************Funktionen deklarieren
**********************
def hexMac(byteMac):
  """
  Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode entgegen
und
  bildet daraus einen String fuer die Rueckgabe
  """
  macString =""
  for i in range(0,len(byteMac)):
    macString += hex(byteMac[i])[2:]
    if i <len(byteMac)-1 :
      macString +="-"
  return macString
```

Die Job-Routine für die Konversation mit der Funkeinheit leitet auch deren Antwort als Nachricht an die angegebene Handynummer weiter.

```
def queryJob():
    global errorCnt
    print("Nachricht versandt: 'get:poti'")
    s.sendto("get:poti",receiver)
    response, addr = s.recvfrom(256)
    print("Antwort von {} empfangen".format(addr))
    # Bei Bedarf ergÃ¤nzen: antortet der richtige Server?
    #
    # entspricht die Antwort den Erwartungen?
    try:
        r=response.decode("utf8")
        if r.find(":"):
            device,value=r.split(":")
            antwort="Potiwert ist:{}".format(value)
            print(antwort)
            g.gsmSendSMS("+49xxxxxxxxxxx",antwort)
            errorCnt=0
        else:
            print("Antwort nicht gueltig")
            errorCnt+=1
    except UnicodeError as e:
        print("Decodierfehler")
        errorCnt+=1
    if errorCnt>=10:
        g.gsmSendSMS("+49xxxxxxxxxxx","Sensorfehler: LDR war 10-
mal nicht auslesbar!")
```

Herstellen der WLAN-Verbindung

```
# ******************   Prepare Webcontact   *********************
# ************************* Get connected
*****************************
# Netzwerk-Interface-Instanz erzeugen und ESP32-Stationmodus
aktivieren;
# moeglich sind network.STA_IF und network.AP_IF beide
gleichzeitig,
# wie in LUA oder AT-based oder Arduino-IDE ist in MicroPython
nicht moeglich
# Create network interface instance and activate station mode;
# network.STA_IF and network.AP_IF,both at the same time,
# as in LUA or AT-based or Arduino-IDE is not possible in
MicroPython

#request = bytearray(100)
#act=bytearray(10)
nic = network.WLAN(network.STA_IF)  # Constructoraufruf erzeugt
WiFi-Objekt nic
nic.active(True)  # Objekt nic einschalten
sleep(3) #
MAC = nic.config('mac')             # # binaere MAC-Adresse
abrufen und
myMac=hexMac(MAC)                   # in eine Hexziffernfolge
umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben
#
# Verbindung mit AP aufnehmen, falls noch nicht verbunden
# connect to WLAN-AP or robot car directly
d.clearAll()
d.writeAt("RST Start WLAN",1,0)
sleep(3)
e=t.waitForTouch(rst,delay=3)
print("RST-Taste:",e)
d.clearAll()
ct=("10.0.1.199","255.255.255.0","10.0.1.20","10.0.1.100")
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySid = 'Your_SSID_goes_here'
myPass = "Put_your_password_here"
if e==1:
    targetIP="10.0.1.101"
    targetPort=9000
else:
    targetIP="10.0.2.101"
    targetPort=9000
    ct=("10.0.2.199","255.255.255.0","10.0.2.20","10.0.2.100")
    mySid = "unit1"; myPass = "uranium238"
print(targetIP)
if not nic.isconnected():
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySid, myPass)
  # warten bis die Verbindung zum Accesspoint steht
  #print("connection status: ", nic.isconnected())
```

```
  d.clearAll()
  d.writeAt("CONNECTING TO",0,0)
  d.writeAt(mySid,0,1)
  while not nic.isconnected():
    #pass
    print("{}.".format(nic.status()),end='')
    sleep(1)
# Wenn bereits verbunden, zeige Verbindungsstatus und Config-Daten
#print("\nconnected: ",nic.isconnected())
#print("\nVerbindungsstatus: ",connectStatus[nic.status()])
nic.ifconfig(ct)
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-
NETMASK:\t",STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
#
# Write connection data to OLED-Display
d.clearAll()
d.writeAt("CONNECTED AS:",0,0)
d.writeAt(STAconf[0],0,1)
sleep(3)
d.clearAll()
d.writeAt(STAconf[1],0,0)
d.writeAt(STAconf[2],0,1)
sleep(3)
d.clearAll()
```

Sobald die WLAN-Verbindung steht, wird der UDP-Client gestartet. Wir erzeugen einen Socket für die IP-V4-Familie (AF_INET) und legen den Datenaustausch über Datagramme fest, das bedeutet, dass wir UDP als Übertragungsprotokoll vereinbaren. Dann binden wir den Socket an die oben festgelegte IP-Adresse und die Portnummer 9181. Die beiden Angaben müssen als Tupel übergeben werden, deshalb die beiden öffnenden und schließenden Klammern. Das innere Paar markiert das 2er-Tupel, das äußere Paar die Parameterliste der Funktion.

Es ist wichtig, einen Timeout zu setzen, denn sonst bleibt das Programm im Befehl s.recvfrom() in der queryJob()-Funktion kleben. Auch die angegebene Zeit in Sekunden ist wichtig und muss so gewählt werden, dass in dieser Periode auch wirklich eine Antwort vom Funksensor eintreffen kann. Auch die Zieladresse in **receiver** muss als Tupel angegeben werden. Das Display informiert über die Verbindungsdaten und dann befinden wir uns auch schon in der Hauptschleife.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9181)) # IP comes from boot-section
s.settimeout(5.0)
d.clearAll()
d.writeAt("Sock established",0,0)
sleep(2)
d.writeAt("TARGET IS AT:",0,0)
d.writeAt("{}:{}".format(targetIP,targetPort),0,1)
print("Socket established, waiting...")
receiver=(targetIP,targetPort) # Address has to be a tuple
```

In der Mainloop wird eine if-Sequenz für die Steuerung ergänzt. Die Position ist nicht relevant, ich habe den Code ans Ende der while-Schleife gesetzt. Die Abfrage wird aufgerufen, wenn das durch das queryFlag angefordert wird und wenn die Wartezeit dafür abgelaufen ist. Ohne die zweite Bedingung würden pausenlos Nachrichten gesendet, was sicherlich nicht in Ihrem Sinn ist.

```
    if (Trigger & queryFlag)==queryFlag and time() >= queryEnde:
        queryJob()
        queryEnde=time()+queryBase
```

Wahrscheinlich wollen Sie jetzt gleich einmal den UDP-Client des Programms testen. Natürlich wollen Sie das, also legen wir los! Haben Sie den ESP32 mit seiner MAC am WLAN-Router angemeldet? Wenn nein, dann holen Sie das bitte jetzt nach, denn wenn der Router den ESP32 nicht kennt, lässt er ihn sehr wahrscheinlich nicht ins Netz. Zumindest sollte ein gut erzogener Router sich so verhalten – aus Sicherheitsgründen!

Wenn sich die eingangs verlinkten Module bmp280.py, button.py, gps.py, hd44780u.py, i2cbus.py, keypad.py und lcd.py im Flasch des ESP32 befinden, kann der UDP-Client getestet werden. In relais.py sollte die Variable Trigger in Zeile 72 auf 0 gesetzt sein. Starten Sie das Programm relais.py mit F5 im Editorfenster und lassen Sie den Startprozess mit gedrückter RST-Taste am LCD-Keypad durchlaufen. Nach diversen Statusmeldungen erscheint im LCD 'TARGET IS AT' mit der IP der ESP8266-Station und im Terminalfenster wird 'Socket established, waiting...' ausgegeben. Danach erscheint der REPL-Prompt '>>>'.

Jetzt brauchen Sie einen UDP-Server, an den sich der ESP32 wenden kann. Das Paket nmap stellt mit der Anwendung ncat einen zur Verfügung. Die Netz-Katze macht ihren PC zum UDP-Server. Laden Sie die Freeware herunter und installieren Sie diese. Im Installationsverzeichnis nmap finden Sie die Datei ncat.exe. Öffnen Sie eine Powershell, wechseln Sie in das Installationsverzeichnis und rufen Sie die Datei ncat folgendermaßen auf. Die IP 10.0.1.107 ist die von meiner Windowskiste, ersetzen Sie sie bitte durch die IP Ihrer Arbeitsstation.

```
ncat -vv -l 10.0.1.107 9000 -u
```

Tipp:
Um schnell eine Powershell im Installationsverzeichnis zu öffnen, suchen Sie im Explorer mit ein paar Klicks das Verzeichnis auf. Rechtsklicken Sie jetzt mit gedrückter Shift-Taste auf den Verzeichniseintrag und wählen Sie aus dem Kontextmenü "Powershell-Fenster hier öffnen".

Wenn ncat läuft, wechseln Sie jetzt ins Terminalfenster von Thonny und geben dort folgende Anweisung ein:

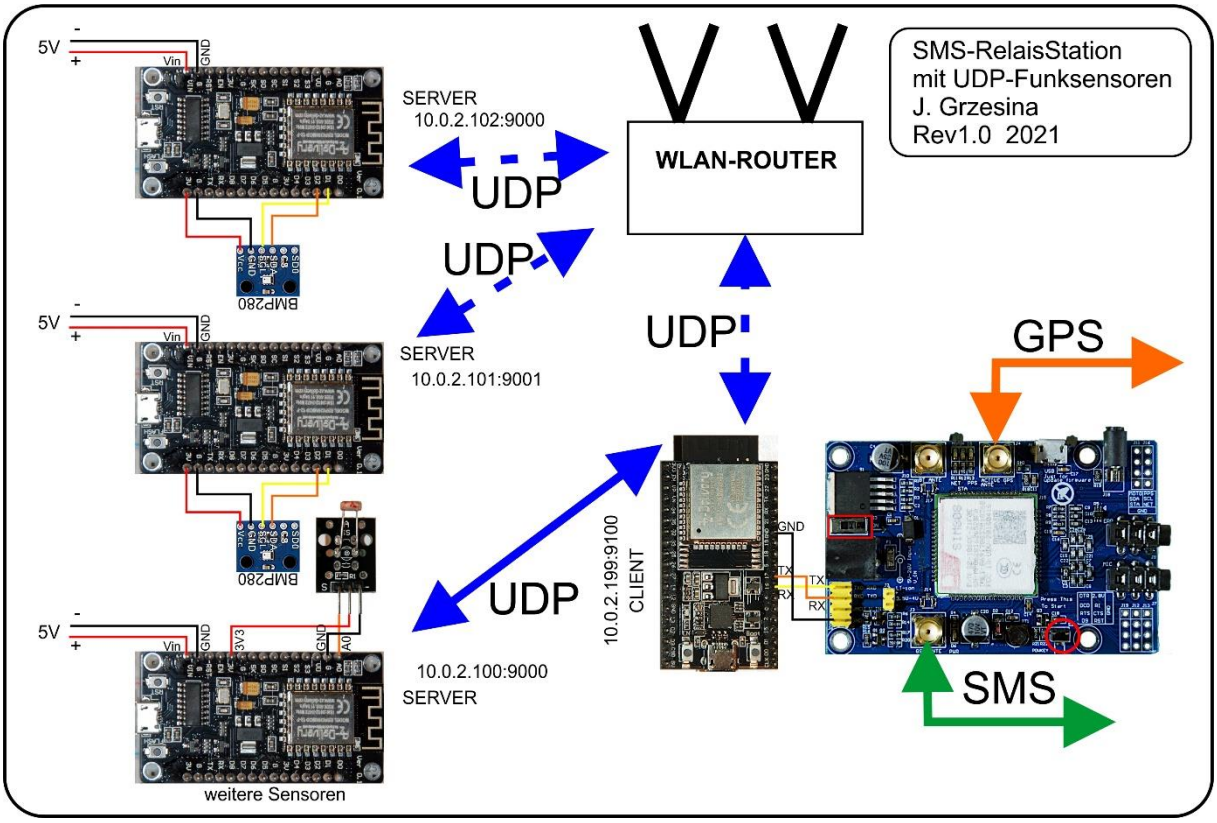
```
>>> s.sendto("Hallo Server",("10.0.1.107",9000))
12
>>>
```

REPL sagt Ihnen, dass 12 Zeichen gesendet wurden und wenn jetzt auch noch im ncat-Fenster 'Hallo Server' erscheint, haben Sie gesiegt.

```
Nmap
PS C:\Program Files (x86)\Nmap> ncat -vv -l 10.0.1.107: 9000 -u
Ncat: Version 7.91 ( https://nmap.org/ncat )
Ncat: Listening on 10.0.1.107:9000
Ncat: Connection from 10.0.1.199.
Hallo Server
```

UDP-Server als Messknechte

Nun sind Sie sicher neugierig, wie das Programm auf dem ESP8266 den Funksensor steuert. Wie das mit dem UDP-Transfer grundsätzlich funktioniert habe ich ja schon anfangs dargestellt. Für die Abfrage von Sensoren kommen die üblichen dedizierten Module zum Einsatz, das ist nichts Neues. Die Grafik verdeutlicht die Gesamtsituation. Gestrichelte Pfeile deuten den Weg über einen Router an, der durchgezogene Pfeil steht für die Direktverbindung.



Die Funkübertragung basiert auf einem einfachen, selbstgestrickten Protokoll, das Sie natürlich beliebig abwandeln und zum Beispiel noch durch eine Prüfsumme erweitern können.

Befehle an den ESP8266 haben das Format action:device[:magnitude]. Die Antwort sieht so aus: magnitude:value. Man hat also die Möglichkeit, in eine Funkeinheit mehrere Sensoren oder auch Aktoren einzubauen und via device und magnitude darauf zuzugreifen. Sie werden sehen, dass die WLAN-Teile des Programms ähnlich wie die vom Steuerprogramm auf dem ESP32 strukturiert sind. Weil aber der ESP8266 kein Display besitzt, werden Statusmeldungen per Blinklicht abgegeben. Dazu verwende ich die

lowaktive, blaue LED an Pin 2, die die Funktion **blink**() mit unterschiedlichen Zeiten und invertierbarem Level ansteuert. Ein Dict(ionary) und eine weitere Funktion sorgen beim Verbindungsaufbau wieder für lesbare Klartextausgaben im Terminalfenster.

Der Aufbau eines Accesspoints auf dem ESP8266 zwischen den Zeilen **Setup accesspoint** und **Setup accesspoint end** ist auskommentiert, weil das Beispiel über den Accesspoint des WLAN-Routers laufen soll. Möchten Sie das nicht, dann entfernen Sie bitte die Kommentarzeichen in diesem Bereich und versehen dafür den Bereich zwischen **Setup Router connection** und **Setup Router connection end** damit. Das geht ganz schnell, wenn Sie den Bereich markieren und mit Alt+4 die Kommentarzeichen entfernen oder mit Alt+3 welche setzen.

Vergessen Sie nicht, Ihre SSID und das Passwort anzugeben, sonst wird's nix mit der Verbindung. Möglicherweise müssen Sie auch den ESP8266 wie seinen Bruder zuerst am Accesspoint als zulässiges Gerät anmelden, damit es akzeptiert wird. Natürlich müssen auch die IP-Adresse, Netzwerkmaske, Gateway- und DNS-Adresse zu Ihrem Netzwerk passen. Bis die Verbindung zum Router steht, blinkt die LED im Sekundentakt.

Der Serverstart ist die nächste Station im Programm. Den Abschluss der Vorbereitungen zeigt ein Blinksignal durch **lang – kurz – kurz – kurz** an.

In der while-Schleife wird zunächst nach einer Aktion gesucht. Bei **get** oder **set** kommt die nächste Stufe, wo nach einem gültigen **device** gefahndet wird. Hierdurch können verschiedene Sensoren berücksichtigt werden.

Wenn nur ein Wert abfragbar ist, wird dieser ermittelt, wie im Beispiel **Poti**. Kann das Device mit mehreren Werten aufwarten, wie der BMP280, dann muss noch der Name der zu ermittelnden Messgröße folgen. Ähnlich verhält es sich im **set**-Zweig, der hier aber nicht weiter ausgeführt ist. Natürlich ist es nützlich, wenn, zum Beispiel nach dem Umschalten eines Relais, der Schaltzustand zurückgesandt wird. Für die Abfrage eines BMP280 müssen die Module bmp280.py und i2cbus.py in dessen Flashspeicher geladen worden sein. Wenn alles zu Ihrer Zufriedenheit läuft, kopieren Sie den Text des Programms server.py in die Datei boot.py. Nach dem Hochladen zum ESP8266 startet dieser, wie der ESP32, autonom. An den Blinksignalen während der Startphase und am Heartbeat sehen Sie, ob das Programm richtig läuft.

Nach jedem Messauftrag wird entweder das Ergebnis oder, bei Fehlschlag, eine Errormessage an den ESP32 geschickt. Der muss nun entscheiden, ob er die Meldung als SMS-Nachricht weiterleitet oder für sich behält und entsorgt. Jede Option der Steuerung auf dem ESP32 kann natürlich auch selbst einen Funkauftrag vergeben. Es kommt nur darauf an, was sie programmieren und durch die Flags freigeben, Sie sind der Chef!

```
#server.py
# ********************** Importgeschaeft ********************
import esp
esp.osdebug(None)
import os
import gc               # Platz fuer Variablen schaffen
gc.collect()
try:
  import usocket as socket
```

```
except:
  import socket
import ubinascii
import network
from machine import ADC,Pin,I2C
from time import sleep,time
import sys
from bmp280 import BMP280
from i2cbus import I2CBus
adc=ADC(0)
print("ADC Initialized: ",adc.read())
taste=Pin(0,Pin.IN)
blinkLed=Pin(2,Pin.OUT)
request = bytearray(160)
act=bytearray(30)
response=""
# 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  FL L
i2c=I2C(-1,scl=Pin(5),sda=Pin(4))
b=BMP280(i2c)

#********************Variablen deklarieren *******************
# Die Dictionarystruktur (dict) erlaubt  die Klartextausgabe
# des Verbindungsstatus anstelle der Zahlencodes
connectStatus = {
    1000: "STAT_IDLE",
    1001: "STAT_CONNECTING",
    1010: "STAT_GOT_IP",
    202:  "STAT_WRONG_PASSWORD",
    201:  "NO AP FOUND",
    5:    "GOT_IP"
    }

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

def blink(pulse,wait,inverted=False):
    if inverted:
        blinkLed.off()
        sleep(pulse)
```

```
        blinkLed.on()
        sleep(wait)
    else:
        blinkLed.on()
        sleep(pulse)
        blinkLed.off()
        sleep(wait)


# # *****************  Setup accesspoint ***************
# #
# nic = network.WLAN(network.AP_IF)
# nic.active(True)
# ssid="unit1"
# passwd="uranium238"
#
# # Start als Accesspoint
#
nic.ifconfig(("10.0.2.101","255.255.255.0","10.0.2.101","10.0.2.10
1"))
#
# print(nic.ifconfig())
#
# # Authentifizierungsmodi ausser 0 werden nicht unterstuetzt
# nic.config(authmode=0)
#
# MAC=nic.config("mac") # liefert ein Bytes-Objekt
# # umwandeln in zweistellige Hexzahlen ohne Prefix und in String
decodieren
# MAC=ubinascii.hexlify(MAC,"-").decode("utf-8")
# print(MAC)
# nic.config(essid=ssid, password=passwd)
#
# while not nic.active():
#   print(".",end="")
#   sleep(0.5)
#
# print("Unit1 listening")
# # *****************  Setup accesspoint end ***************

# # ****************  Setup Router connection ***************
mySid = "your_SSID"; myPass = "your_password"
nic = network.WLAN(network.STA_IF)  #  erzeuge WiFi-Objekt nic
nic.active(True)  # Objekt nic einschalten
#
MAC = nic.config('mac')    # # binaere MAC-Adresse abrufen und
myMac=hexMac(MAC)          # in eine Hexziffernfolge umgewandelt
print("STATION MAC: \t"+myMac+"\n") # ausgeben
# Verbindung mit AP im lokalen Netzwerk aufnehmen,
# falls noch nicht verbunden
# connect to LAN-AP
if not nic.isconnected():
```

```
  # Geben Sie hier Ihre eigenen Zugangsdaten an
  # Zum AP im lokalen Netz verbinden und Status anzeigen
  nic.connect(mySid, myPass)
  # warten bis die Verbindung zum Accesspoint steht
  print("connection status: ", nic.isconnected())
  while not nic.isconnected():
    blink(0.8,0.2,True)
    print("{}.".format(nic.status()),end='')
    sleep(1)
# Wenn bereits verbunden, zeige Verbindungsstatus & Config-Daten
print("\nconnected: ",nic.isconnected())
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
print("Weise neue IP zu:","10.0.1.101")
nic.ifconfig(("10.0.1.101","255.255.255.0","10.0.1.20", \
             "10.0.1.100"))
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
      STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
#
# # *************  Setup Router connection end ***************


# ----------------- Server starten -------------------------
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9000))
print("Socket established, waiting...")
s.settimeout(2.0) # timeout, damit 'while True' durchläuft
blink(2,0.5,True)
blink(0.3,0.8,True)
blink(0.3,0.8,True)
blink(0.3,1.5,True)
# ------------------ Serverschleife ----------------------
while True:
    try:
        # recvfrom wird nach 2 Sec. mit einer Exception
abgebrochen
        # Zwischenzeitlich eintreffende Zeichen bleiben bis zum
        # nächsten Durchlauf im Empfangsbuffer und werden dann
        # abgeholt.
        request, addr = s.recvfrom(256)
        r=request.decode("utf8")
        print('from {}\nContent = {}'.format(addr,r))
        if r.find(":") != -1:
            act,rest=r.split(":",1)
            print('Action={}, Rest={}'.format(act,rest))
            if act.upper()=="SET":
                # control action devices
                pass
                # end if "SET"
            elif act.upper()=="GET":
                # query sensor devices
                print("GET handler")
```

```
                if rest.find(":") != -1:
                    device,rest=rest.split(":",1)
                    if device.upper() == "BMP280":
                        # Aufteilung in einzelne Größen
                        print("BMP280")
                        if rest.upper()== "TEMP":
                            # Wertabrufen
                            w=b.calcTemperature()
                            response="TEMP:{}\n".format(w)
                            pass
                        elif rest.upper()=="PRES":
                            w=b.calcPressureNN()
                            response="PRES:{}\n".format(w)
                            pass
                        elif rest.upper()=="BOTH":
                            t=b.calcTemperature()
                            p=b.calcPressureNN()

response="TEMP:{}\nPRES:{}\n".format(t,p)
                            pass
                        else:
                            response="ERROR:invalid quantity
call\n"
                    # elif ... further devices
                    else: #  device mit mehreren Größen
                        response="ERROR:invalid device call\n"
                else: #  device als eigene Größe
                    device=rest
                    if device.upper()=="POTI":
                        poti=str(adc.read())
                        response=device.upper()+":"+poti+"\n"
                    # elif ... further single quantity devices
                    else:
                        response="ERROR:invalid device call\n"
            else:
                response="ERROR:invalid action call\n"
        else:
            response="ERROR:invalid command"
        s.sendto(response,addr)
    except OSError as e:
        #print(e.args[0])
        pass
    if taste.value()==0:
        print("Mit Flashtaste abgebrochen")
        sys.exit()
    blink(0.1,0.9, inverted=True)
    #sleep(1)
```

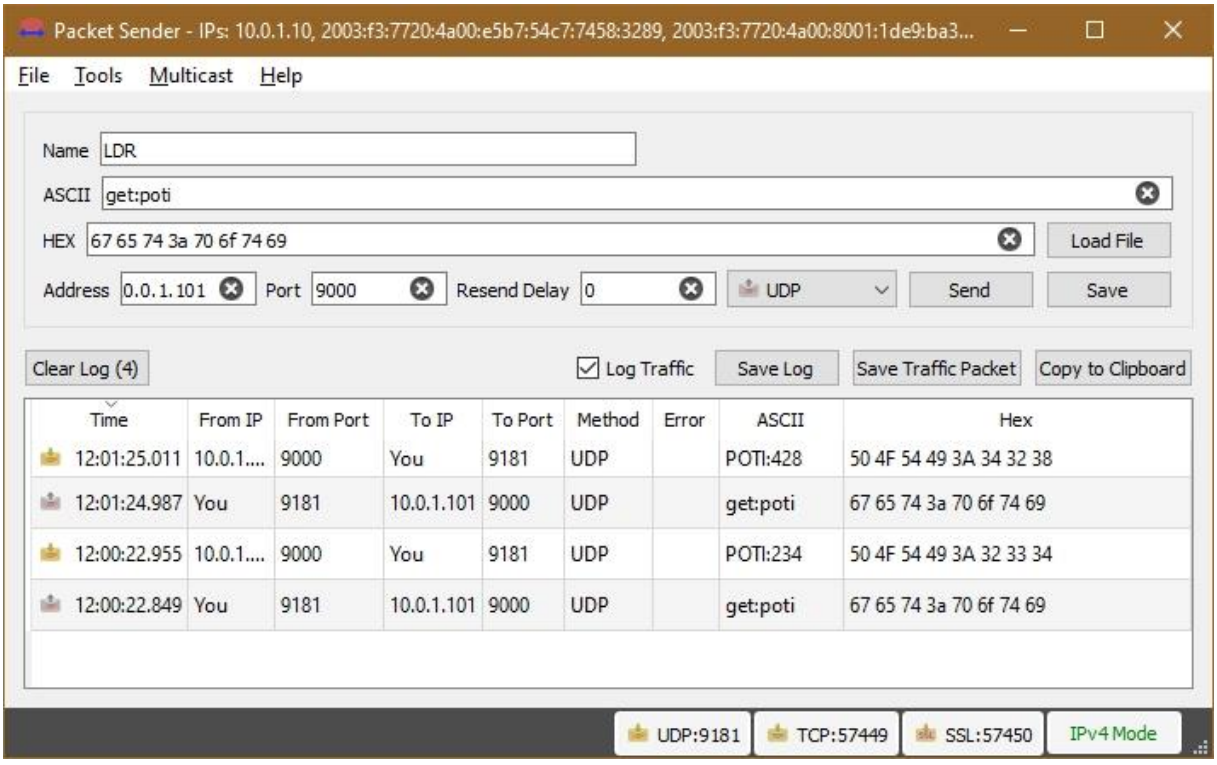
Wenn Sie nachträglich versuchen, mit Thonny wieder auf REPL zuzugreifen, werden Sie feststellen, dass der ESP8266 nach einem Reset sofort wieder bootet und Sie nicht ins System lässt. Klar, wir wollten das ja genauso haben. Da hilft nun leider kein Strg+C und

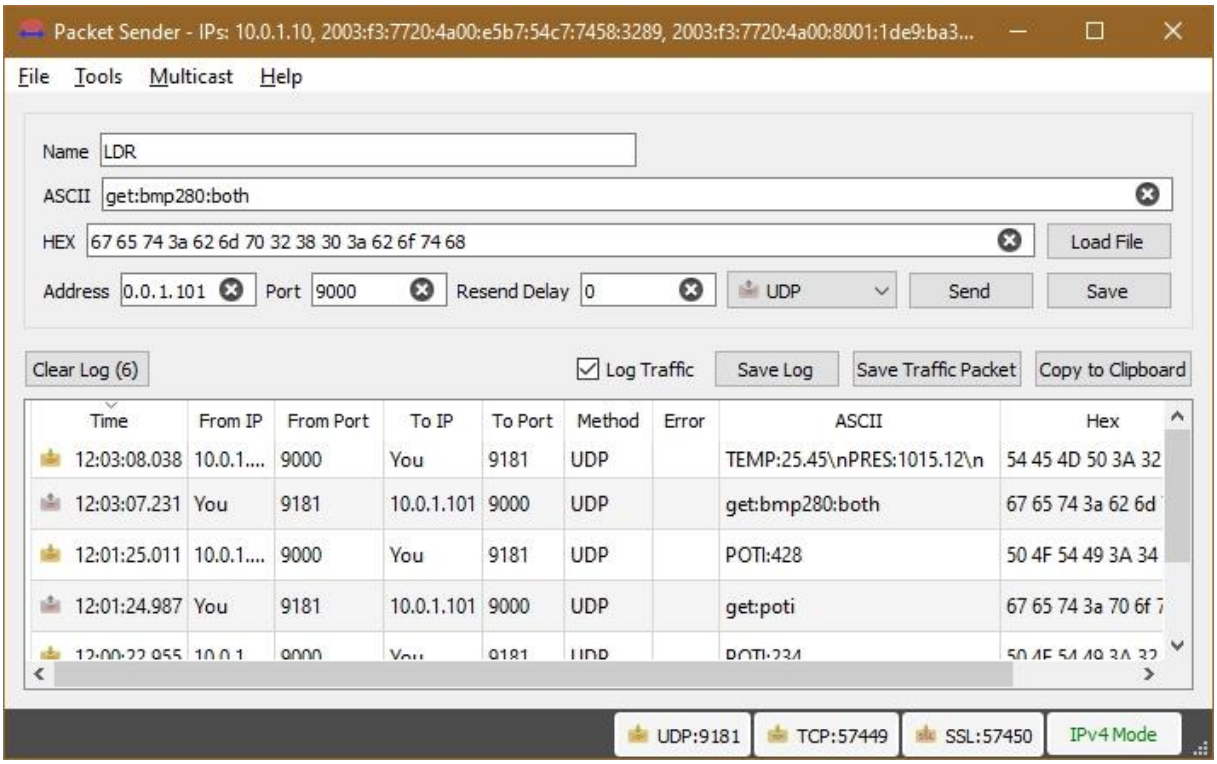
kein Klick auf das Stoppschild, Zetern und Wehklagen auch nicht, um den Controller zum Anhalten zu bewegen.

Ich weiß, dass das passiert und deshalb hat auch der ESP8266 eine eingebaute Notbremse, die Flash-Taste. Die drücken Sie so lange, bis der REPL-Prompt im Terminalfenster erscheint. To continue or not to continue…, das ist die Frage, deren Beantwortung in der fünftletzten Zeile des Programms eingeleitet wird.

Für den Test des ESP8266 können Sie das gleich ausprobieren. Ist der Controller mit MicroPython geflasht? Sind die Module auch im Flash, zusammen mit dem Server-Programmtext in boot.py? Dann, ESP8266 Kaltstart - RST! Kommt der Heartbeat? Bestens!

Was Sie jetzt brauchen, ist ein flexibler UDP-Client, mit dem Sie Empfang und Rücksendung der ESP8266-Einheit testen können. Zum Test sehr gut geeignet ist die Freeware packetsender. Man kann hier Steuerbefehle von Hand eingeben, via UDP an den Server auf dem ESP8266 schicken und so die Reaktion des Programms testen. Danach übernimmt der ESP32 das Kommando, wenn alles perfekt funktioniert. Installieren Sie also jetzt die Software und stellen Sie ganz unten den gewünschten UDP-Port Ihres Rechners ein, hier UDP:9181.





Jetzt geben Sie die Anfragen, welche sonst der ESP32 senden würde, einfach von Hand bei **ASCII** ein. **Send** schickt die Daten zum ESP8266. Kommen die richtigen Antworten, dann funktioniert auch dieser Teil des Projekts. Andernfalls haben Sie immer noch die Notbremse zum Nachbessern des Programms.

Der finale Test

Dafür weisen Sie der Variablen Trigger in Zeile 72 einen der flag-Werte zu und starten das Programm.

```
timeFlag    =0b00000001 # Intervallsteuerung
distanceFlag=0b00000010 # Streckenalarm
tempFlag    =0b00000100 # Temperaturalarm
orderFlag   =0b00001000 # SMS Anforderung per SMS
queryFlag   =0b00010000 # Abfragen von externen Sensoren per UDP
Trigger=0 # Hier die Flags der Dienste eintragen
```

Nachdem die Verbindung steht, sollte sich Ihr Handy mit einer neuen SMS-Nachricht melden. Testen Sie die anderen Optionen auch der Reihe nach durch. Bei SMS on demand senden Sie eine Nachricht mit dem Inhalt 'wetter' an den ESP32. Als Quittung erhalten Sie die Temperatur- und Luftdruckwerte vom BMP280 am ESP8266.

Für den autonomen Start des ESP32 muss jetzt nur noch der Text des Programms relais.py in die Datei boot.py kopiert werden. Der Upload zum Controller schließt das Projekt ab.

So, nun sind Sie mit ausreichend Know-how ausgestattet, um das Projekt weiterzuentwickeln und an Ihre Bedürfnisse anzupassen. Dazu wünsche ich Ihnen viel Spaß und viel Erfolg.

Ach ja – da war ja noch was! Unser SIM808 hat doch noch einen Knüller zu bieten, Sie können damit natürlich auch telefonieren. Wie das geht, und wie Sie eine Wähltastatur anschließen, das verrate ich Ihnen in der nächsten Folge. Bis dann!

Weitere Downloadlinks:

PDF in deutsch
PDF in english