

Diesen Beitrag gibt es auch als:

[PDF in deutsch](#)

This episode is also available as:

[PDF in english](#)

Most recently, it was about sending text messages over the cellular network via SMS. In this episode too, ESP32 and SIM808 come into play again. They form a kind of server or relay station to which text messages can be sent for control.

However, the controller is also able to send time-controlled or event-controlled messages itself. The cell phone or the PC, the Raspi, etc. is then the switching and receiving center, SMS functionality is required. Usually it will be the mobile phone or tablet and because the distance between the SIM808 and the mobile phone does not matter, there is no distance limit, provided that there is sufficient network coverage. The LCD keypad is actually superfluous because direct control of the remote ESP32 unit using the buttons is out of the question anyway. Nevertheless, a display does a good job when starting the relay unit and for other status messages. And - one button is essential, I use it as an emergency brake! Welcome to the fourth part of

## **GSM and GPS with MicroPython on the ESP32**

---

### **Hardware - some growth**

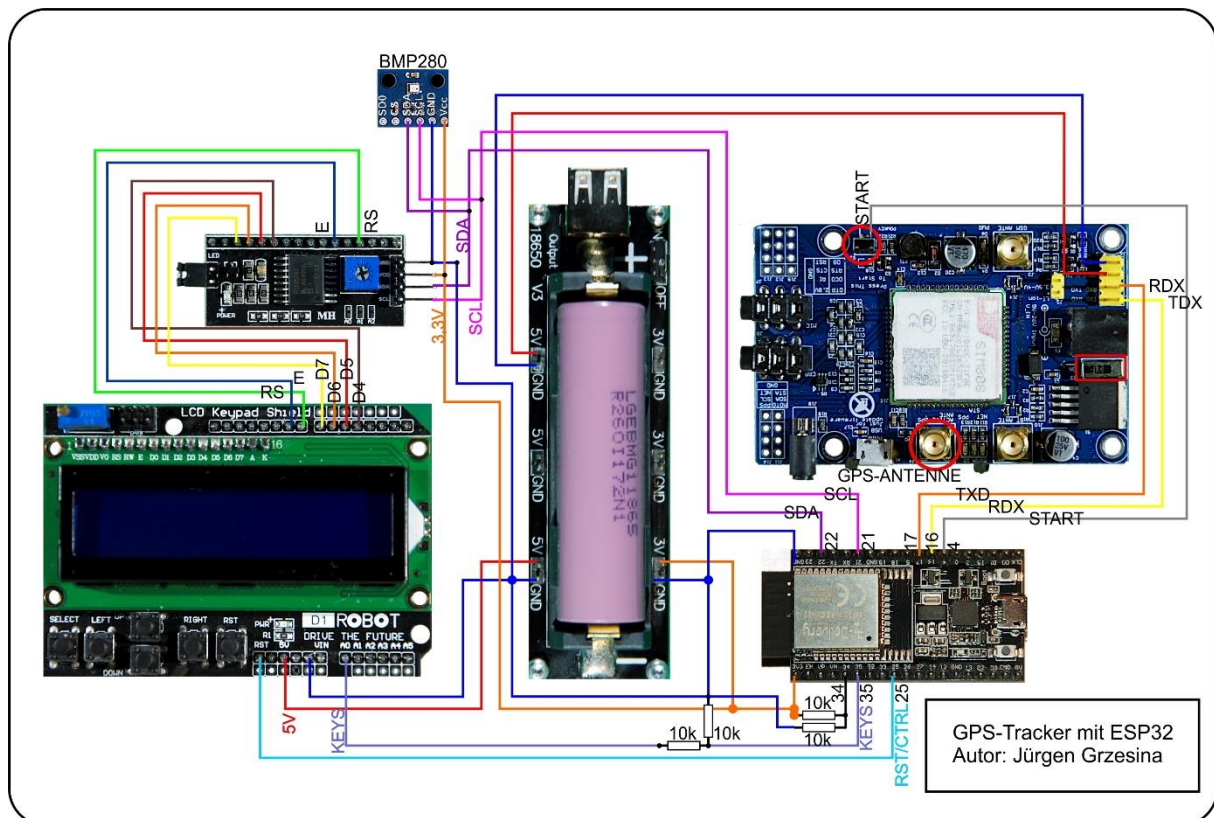
There is a little more hardware compared to Part 3. After all, we want to integrate at least one external radio unit into the project. Well, in the list below you can find all of the parts for the current project. Almost all of this has already been used in [Part 1](#), [Part 2](#) and [Part 3](#)

and of course also described in detail there. Of course, we reuse these components. In the third part of the series a SIM card was added, because without it you will not be able to send or receive SMS messages. And now we will add another option to the functionality of the relay station. For this purpose I have provided an ESP8266 together with an LDR resistor and another BMP280 as a radio sensor as an example. Instead of the LDR, any other sensor that can deliver analog signals can of course be used. In this example, an analog voltage at the A0 pin of the ESP8266 is sampled on the external radio unit. Other sensors such as DS18B20 (temperature), DHT22 AM2302 (humidity and temperature), GY-302 BH1750 (light sensor), etc. can of course be used just as well thanks to the various MicroPython modules that are available for this. We already used the module for the BMP280 in episode 3. As a rule, the modules can be easily adapted to your own needs. If necessary, a module from a library of the Arduino IDE is reworked for MicroPython. Most of the time, the result is a readable source text and the class methods do exactly what you want by adapting them.

Here is the list of ingredients for the new cooking session. After all, there are two main courses this time

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

The circuit for the relay station is first taken over 1: 1 from part 2. Later you decide for yourself which parts you want to leave out, replace or add new ones. The project is scalable in every direction. The technical basics for the implementation can be found in this article.



You will receive a more legible copy of the illustration in DIN A4 with the [Download der PDF-Datei](#) .

## Die Software

### Used Software:

For flashing and programming the ESP:

[Thonny](#) oder

[uPyCraft](#)

[ncat](#) as UDP-Server for Windows

[paketsender](#) for testing the ESP8266 as UDP-server

### Used 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 and information on MicroPython

The interpreter language MicroPython is used in this project. The main difference to the Arduino IDE is that you have to flash the MicroPython firmware on the ESP32 before the controller understands MicroPython instructions. You can use Thonny,  $\mu$ PyCraft or esptool.py for this. For Thonny, I described the process in the first part of the blog on this topic.

As soon as the firmware is flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without first having to compile and transfer an entire program. This is exactly what bothers me about the Arduino IDE. When developing the software for this blog, I made ample use of the direct dialog with the ESP32. The spectrum ranges from simple tests of the syntax to trying out and refining functions and entire program parts. For this purpose, I also like to create small test programs, as in the previous episodes. They form a kind of macro because they combine recurring commands.

Such programs are started from the current editor window in the Thonny IDE using the F5 key. This is faster than clicking the start button or using the Run menu. I also described the installation of Thonny in detail in the first part.

Let's reach into MicroPython's bag of tricks for a while. How does it work with the UDP transfer of information?

When you talk to your ESP32 over the USB line, you are using the message transfer principle. Entries via the keyboard are sent from the PC to the ESP32. REPL, MicroPython's command line interpreter, receives the message on the ESP32 and decodes it. Then the interpreter of the ESP32 looks what to do. The information is parsed, i.e. tapped for the meaningful content. The desired action is then triggered, the result of which is returned to the PC by the ESP32, implicitly as a print command. The receives this message and displays it in the terminal window.

UDP transfers are basically similar. But there are two differences. The transmission path is the radio link instead of the cable, and the transmission is not secured with UDP. This means that the UDP protocol neither ensures that the message has arrived, nor that the message has been transmitted unadulterated or completely. Furthermore, there is no guarantee that the information packets (aka datagrams) will arrive in the order in which they were sent. In the case of information with little content, the order does not matter because the content is transmitted in a packet. That is the case with our project. When transmitting measured values, it usually does not matter if a measured value is incorrectly, incompletely or not at all transmitted during rapid scanning.

There is one decisive advantage for this: UDP is quick and easy to use. And the protocol, just like TCP, can also transfer data in both directions between client and server.

If we assume that the WLAN connections already exist, the programming of the UDP client and server is limited to a few lines.

### **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:**

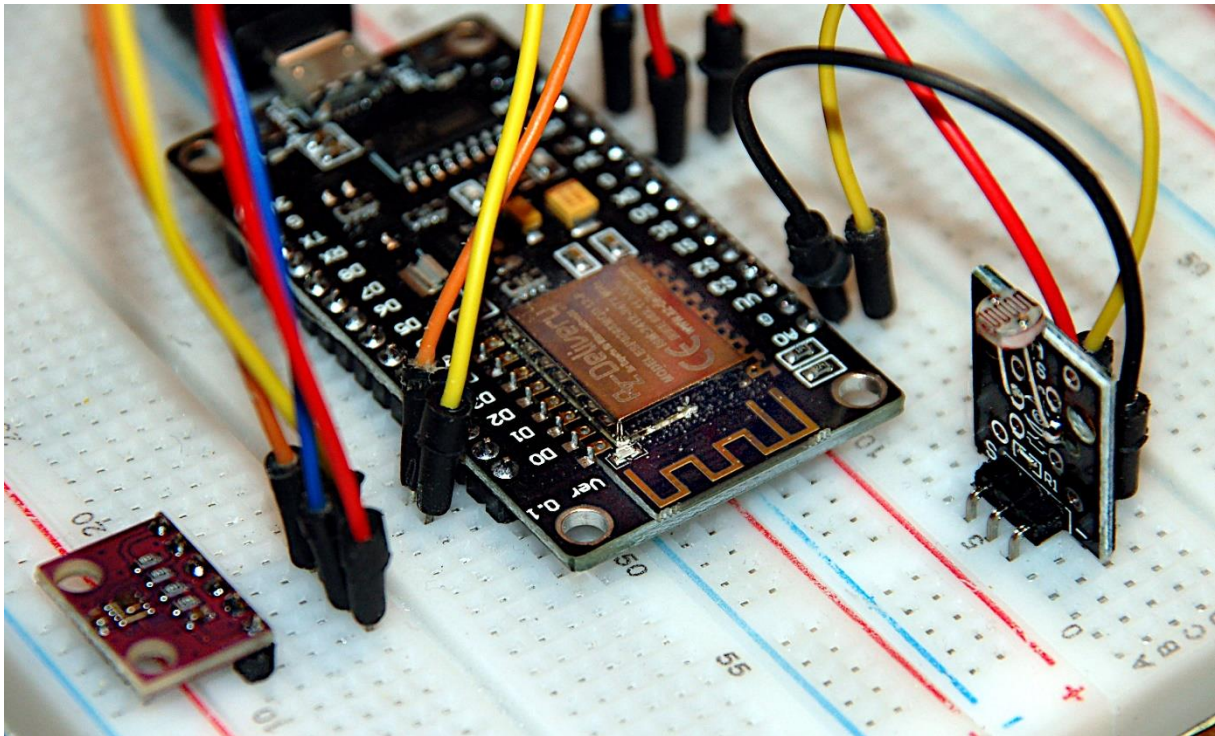
```
# serverexample.py
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 8888)) # IP comes from boot-section
s.settimeout(5.0) # Blockadebrecher
while 1:
    request, addr = s.recvfrom(256)
    print("Auftrag von {} empfangen:{}".format(addr, request))
    # Auftrag decodieren,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

sendto () is the replacement for print () and the input command is replaced by receivefrom ().

The information arriving at the server in the request variable is encoded as a byte object and should first be decoded into a string before further processing, which is easier to parse. The sender is in the variable addr. The server parses the request, takes care of the requested action and sends the result back to the sender.

As shown here, the data transmission between the external sensor unit (ESP8266 + LDR) and ESP32 in the SMS relay also works. The scope of the actual program parts is greater, because the parsing requires a large amount of programming effort. I have therefore put this part at the beginning because of the clearer representation of the core of the UDP transmission. We'll take a closer look at the corresponding parts of the program later. Here is a photo of a server unit in advance. Lower left the BMP280 and right the LDR. On the ESP8266 there are still 5 pins free for additional sensors or actuators.





## Addressing external sensors via SMS

The installation of Thonny, the programming environment, was dealt with in Part 1 of the series, as was the flashing of the MicroPython firmware. The corresponding firmware must of course also be flashed on the ESP8266. Today we have two separate units to program and test. This has to happen one after the other. First we prepare the ESP32.

The main purpose of the SIM808 in this blog episode is the relay function. That's why I don't go into the topic of GPS anymore, that was the main topic in Part 1 and Part 2. We still want to briefly take a look at the function of the GSM board, because we need it now because of its SMS messaging capability to send.

Because the UART0 interface is reserved for REPL, a second interface must be available for the conversation with the SIM808. The ESP32 provides such as UART2. The connections for RXD (reception) and TXD (transmission) can even be freely selected. For full duplex operation (send and receive simultaneously) the RXD and TXD connections from the ESP32 to the SIM808 must be crossed. You can understand this on the circuit diagram. The default values on the ESP32 are RXD = 16 and TXD = 17. The connection is organized by the `gps.py` module.

This begins when the SIM808 is switched on. If you followed my recommendation and soldered a cable to the power button, you can now switch on the SIM808 with the following command, provided that this cable is connected to pin 4 of the ESP32 and the instance `g` of the class `gps.GSM` has already been created.

```
>>> 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()
>>>
```

Commands to the SIM808 are transmitted in AT format. There are a huge number of commands that can be looked up in a PDF file. But don't worry, a few of these commands are enough for our project. Two of them are combined in the methods `init808 ()` and `deinit808 ()`.

```
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 switches on the power supply to the GPS module and AT + CGNSTST = 1 activates the transmission of the NMEA sentences to the ESP32 via the serial interface UART2. The controller receives the information from the SIM808 and provides it in the manner described above via the terminal and LCD.

The listing now follows to study the gps module in more detail. The three included classes GPS, SIM808 (GPS) and GSM (SIM808) build up a uniform namespace through inheritance, which we imported with `from gps import *`. Therefore all methods are available in the object `g` of the class GSM. If GSM is not needed (no SMS transfer), you can also enter via the SIM808 class, as was done in Part 2.

The SIM808 class takes care of hardware control and data transfer to and from the ESP32. The GPS class contains methods for decoding the NMEA sentences from the SIM808, for displaying them on the display and for calculating the course. Finally, the GSM class provides the methods for SMS transfer and management. This is exactly what we will now deal with and then take a closer look at the UDP data traffic.

```
"""
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 Datensätze 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 + MinuteFraktionen
        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):

```





```

        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 of 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)

```

In addition to the AT commands for switching on the GPS unit and opening the UART connection on the SIM808, there are a few more commands for SMS operation, which we will now examine.

AT commands must always end with a `\r\n` (carriage return and line feed = `\x0D` and `\x0A`) so that they are recognized as such by the SIM808. These characters are also

generated, for example, when you press the Enter key. In the case of the AT commands, however, the characters must be specified separately, as in the examples below.

The SIM808 also sends back answers or results with `\r\n`. Except for user data, such as message texts, the only important thing for us is whether the response corresponds to the expected character string. There is therefore a method that does exactly this check, `simSendCmdChecked()`. It returns `True` as the result if the check was successful, `False` otherwise. The method takes the AT command string, the expected response, the command type (command or data) and two optional timeout values as parameters. The first of these limits the time for the entire response from the SIM808, the second defines the time limit per character. Both prevent the method from seizing up and blocking the whole program. The class attribute `CMD` (value = 1) identifies the AT command as a command via the type parameter. When responding to a command, the rest of the UART buffer on the SIM808 is flushed, i.e. emptied, after the predefined response has been recognized, which is of course not desired for a text response. The `simWaitForResponse()` method is used to read in and check the response from the SIM808.

Here is an example. In order for SMS operation to be possible, the 7-bit text mode must be switched on. If the action was successful, the answer contains an 'OK', the existence of which we have to check.

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

`AT + CMGF = 1`: switches to 7-bit text mode, thus enabling plain text SMS mode. With `AT + CMGF = 0`, the SIM808 works in PDU mode (from Physical Data Unit). In this mode, the 7-bit characters are embedded in an 8-bit data stream, which does not result in plain text, but only hieroglyphs in the output.

The status of SMS messages allows them to be roughly selected, for example with a list command. The `stat` variable contains one of the following strings in addition to other possible strings: `"ALL"`, `"REC UNREAD"` or `"REC READ"`. This value is built into the command by the format instruction instead of the curly braces and is enclosed by the two double quotation marks. The entire AT command is enclosed in single quotation marks.

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

The '1' ensures that the status of the listed messages does not change when they are listed. Please note that the status string must be in quotation marks. The above command sends the following character string to the SIM808:

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

When sending SMS messages, the command must first be given the number of the connection. The SIM808 replies with a `">"`.

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

If `">"` was recognized, the message can be sent.

```
simSendCommand (mesg)
```

The end of the message is announced to the other station by sending an end-of-text character (chr (26) = \ x1A = Ctrl + Z).

In order to read an SMS message from the SIM808's memory, its index must be known. After sending the read command, the UART buffer of the SIM808 can be read out. It contains the text of the message. We wait until at least one character is available and then read in number of characters bytes. The value of this variable should be adapted to the length of the message. Please note, however, that in addition to the text, other information such as date, time and status contribute to the total length. The '1' ensures that the status does not change when listing.

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

Messages can also be deleted by specifying the index.

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

SMS operation is only possible after a SIM card has been recognized. The command

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

checks that.

The methods of the GSM class use the commands from the SIM808 class to execute. The following list compiles the SMS methods.

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

Some of the GSM methods are used for internal processing and control. The main program `relais.py` shows how the class is used.

The application program `relais.py` has become considerably more extensive compared to the pure GPS application from Part 3. This is mainly due to the integration of the WLAN functionality, but also to the added example for querying the wireless sensor via UDP. The program has a lot to offer. In detail it demonstrates:

- Time-controlled SMS (every x hours, minutes ...)
- event-controlled SMS (temperature outside of a range)
- SMS on demand (reply to an SMS)
- GPS tracker (distance exceeded or waypoint transmission)
- Transmit measured value
- Query radio sensor via UDP

The last point is the core topic of this post, everything else was already discussed in episode 3. That's why we're now looking at the relevant program snippets. Then we shed light on the server program on the ESP8266.

The network connection can be established either via the access point of the WLAN router or directly via the access point of the ESP32. Of course, not only one wireless sensor can be addressed. Only one thing has to be considered: The ESPs must all be in the same WLAN subnet. This is specified by the WLAN router if one is to be used. This is not absolutely necessary because the ESP8266 can provide its own access point. Both approaches have their advantages and disadvantages.

With WLAN router as access point:

Both the ESP32, with the SIM808 attached, and the ESP8266 start in station mode. Several ESP8266 can easily be addressed by the ESP32 simply by specifying the IP address. Different port numbers offer a further differentiation. Addressing via broadcast is possible.

With ESP8266's own access point:

The ESP32 must connect to a single access point when it starts. Only the server on this station can therefore be addressed. So that further ESP8266 can be addressed, the ESP32 would have to terminate the first WLAN connection in order to then establish another. But that is much more time-consuming.

Because it also makes the test environment easier, I implemented the first solution. Nevertheless, you can switch to the second. On the ESP32 this is done by pressing the RST button on the LCD keypad at startup. The display informs about the time. If the button is not pressed, the second solution starts and the ESP32 tries to find the access point of the ESP8266 specified in the program.

The first solution is commented out on the ESP8266. By uncommenting this area and commenting out the router solution, the operating mode can also be changed here. Of course, you can also establish a control with a button similar to that on the ESP32.

You can find the following pieces of code in the `relais.py` file. They cannot be run individually, but only represent the basis for your discussion. For a more detailed study, I suggest downloading the program and going through it parallel to the meeting.

These import lines, right at the beginning, prepare network access.

```
#
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 ergänzen: antwortet 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("+49xxxxxxxxxxxx",antwort)
            errorCnt=0
        else:
            print("Antwort nicht gueltig")
            errorCnt+=1
    except UnicodeError as e:
        print("Decodierfehler")
        errorCnt+=1
    if errorCnt>=10:
        g.gsmSendSMS("+49xxxxxxxxxxxx","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()
```

As soon as the WLAN connection is established, the UDP client is started. We create a socket for the IP-V4 family (AF\_INET) and define the data exchange via datagrams, which means that we agree on UDP as the transmission protocol. Then we bind the socket to the IP address specified above and the port number 9181. Both entries must be passed as tuples, hence the two opening and closing brackets. The inner pair marks the 2's tuple, the outer pair the parameter list of the function.

It is important to set a timeout, otherwise the program will get stuck in the s.recvfrom () command in the queryJob () function. The specified time in seconds is also important and must be selected in such a way that a response from the radio sensor can actually arrive during this period. The destination address in receiver must also be specified as a tuple. The display informs about the connection data and then we are already in the main loop.

```
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
```

An if sequence for the control is added to the main loop. The position is not relevant, I put the code at the end of the while loop. The query is called when this is requested by the queryFlag and when the waiting time for it has expired. Without the second condition, messages would be sent non-stop, which is certainly not what you want.

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

You probably want to test the program's UDP client right now. Of course you do, so let's get started! Have you registered the ESP32 with its MAC on the WLAN router? If not, please do so now, because if the router does not know the ESP32, it is very likely that it will not let it into the network. At least a well-behaved router should behave like this - for security reasons!

If the modules linked at the beginning bmp280.py, button.py, gps.py, hd44780u.py, i2cbus.py, keypad.py and lcd.py are in the bottle of the ESP32, the UDP client can be tested. In relais.py the variable Trigger in line 72 should be set to 0. Start the relais.py program with F5 in the editor window and let the start process run through while holding down the RST key on the LCD keypad. After various status messages, the LCD shows 'TARGET IS AT' with the IP of the ESP8266 station and 'Socket established, waiting ...' is output in the terminal window. Then the REPL prompt '>>>' appears.

Now you need a UDP server that the ESP32 can contact. The nmap package provides one with the ncat application. The network cat turns your PC into a UDP server. Download and install the freeware. The ncat.exe file can be found in the nmap installation directory. Open a Powershell, change to the installation directory and call up the ncat file as follows. The IP 10.0.1.107 is that of my Windows box, please replace it with the IP of your workstation.

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

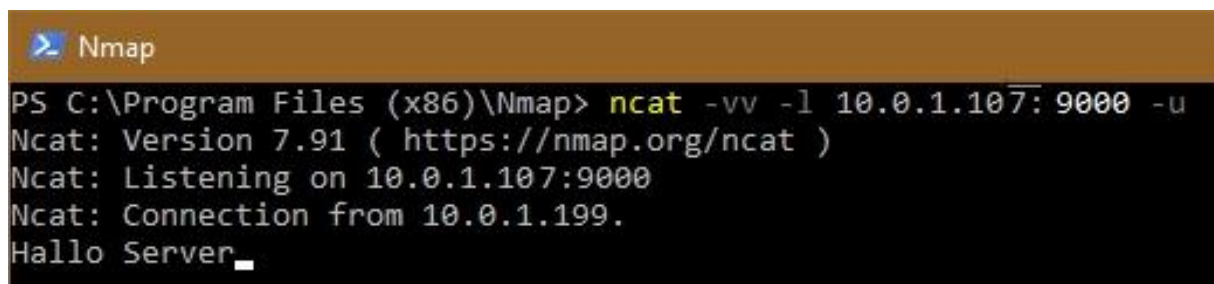
Tip:

To quickly open a Powershell in the installation directory, look for the directory in Explorer with a few clicks. Now right-click the directory entry while holding down the Shift key and select "Open Powershell window here" from the context menu.

If ncat is running, switch to Thonny's terminal window and enter the following instruction:

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

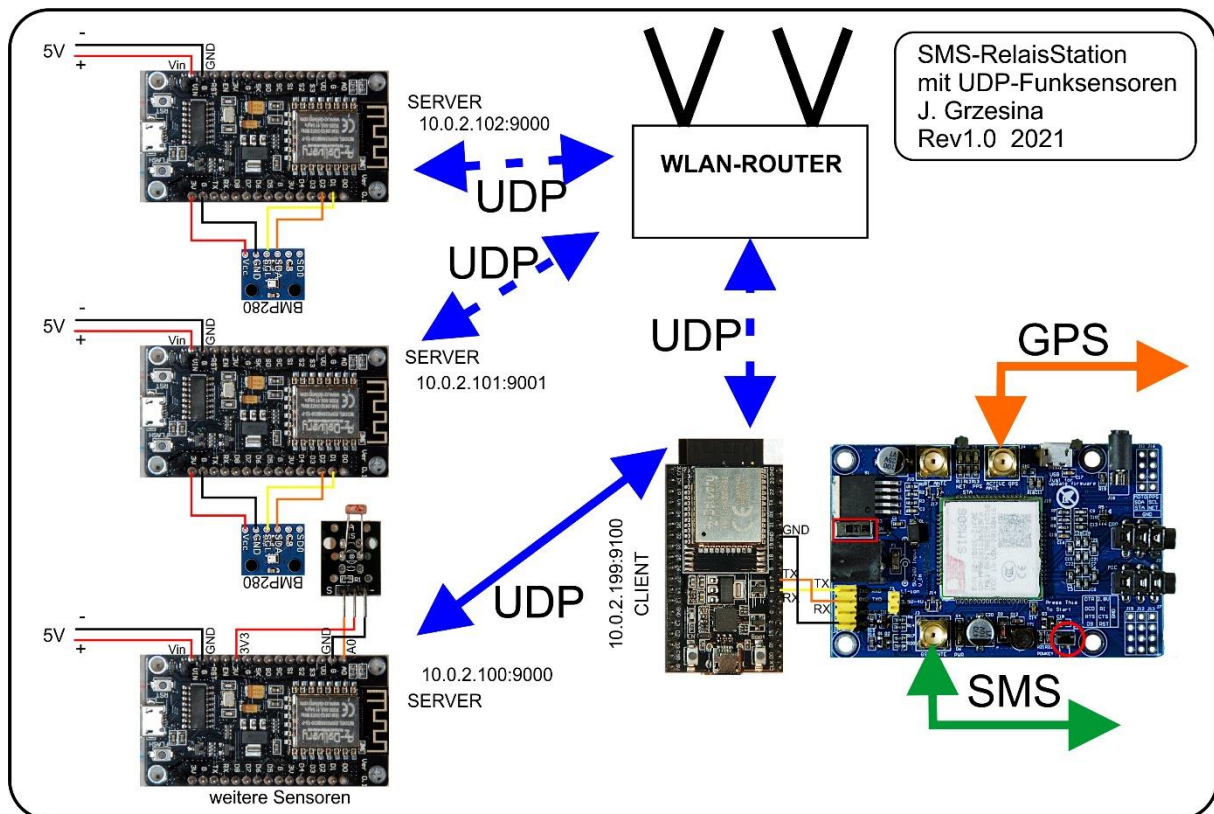
REPL tells you that 12 characters were sent and if 'Hello Server' appears in the ncat window, you have won.



```
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 as measurement servants

Now you are surely curious how the program on the ESP8266 controls the wireless sensor. I have already shown at the beginning how this works with UDP transfer. The usual dedicated modules are used to query sensors, which is nothing new. The graphic illustrates the overall situation. Dashed arrows indicate the route via a router, the solid arrow stands for the direct connection.



The radio transmission is based on a simple, self-made protocol that you can of course modify as you wish and, for example, expand with a checksum.

Commands to the ESP8266 have the format `action: device [: magnitude]`. The answer looks like this: `magnitude: value`. So you have the option of installing several sensors or actuators in a radio unit and accessing them via device and magnitude. You will see that the WLAN parts of the program are structured similarly to those of the control program on the ESP32. But because the ESP8266 does not have a display, status messages are given by a flashing light. For this I use the low-active, blue LED on pin 2, which controls the `blink()` function with different times and invertible levels. A `dict()` and another function ensure that the connection is again readable in plain text in the terminal window.

The structure of an access point on the ESP8266 between the lines `Setup accesspoint` and `Setup accesspoint end` is commented out because the example should run via the access point of the WLAN router. If you do not want this, please remove the comment characters in this area and add it to the area between `Setup Router connection` and `Setup Router connection end`. This can be done very quickly if you mark the area and remove the comment marks with `Alt + 4` or set them with `Alt + 3`.

Don't forget to enter your SSID and password, otherwise the connection won't work. You may also have to register the ESP8266 like its brother as a valid device on the access point so that it is accepted. Of course, the IP address, network mask, gateway and DNS address must also match your network. Until the connection to the router is established, the LED flashes every second.

The server start is the next station in the program. The completion of the preparations is indicated by a flashing signal with long - short - short - short.



An action is first searched for in the while loop. With get or set comes the next level where a valid device is searched for. This allows different sensors to be taken into account.

If only one value can be queried, this is determined, as in the Poti example. If the device can come up with several values, like the BMP280, then the name of the measured variable to be determined must follow. The situation is similar in the set branch, which is not discussed further here. Of course it is useful if, for example after switching a relay, the switching status is sent back. To query a BMP280, the modules bmp280.py and i2cbus.py must have been loaded into its flash memory. If everything goes to your satisfaction, copy the text of the server.py program into the boot.py file. After uploading to the ESP8266, like the ESP32, it starts autonomously. You can see from the flashing signals during the start phase and the heartbeat whether the program is running correctly.

After each measurement job, either the result or, if it fails, an error message is sent to the ESP32. He now has to decide whether to forward the message as an SMS message or keep it to himself and dispose of it. Each control option on the ESP32 can of course also assign a radio job itself. It just depends on what you program and enable through the flags, you are the boss!

```
#server.py
# ***** Importgeschaefte *****
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","10.0.2.101"))
#
# 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)

```

If you subsequently try to access REPL again with Thonny, you will find that the ESP8266 boots immediately after a reset and does not let you into the system. Sure, we wanted that too. Unfortunately, Ctrl + C and clicking on the stop sign, screaming and wailing, do not help to get the controller to stop either.

I know that this happens and that's why the ESP8266 has a built-in emergency brake, the flash button. Press and hold until the REPL prompt appears in the terminal window. To continue or not to continue ... that is the question, the answer to which is introduced in the fifth from last line of the program.

You can try it out for the test of the ESP8266. Is the controller flashed with MicroPython? Are the modules also in Flash, together with the server program text in boot.py? Then, ESP8266 cold start - RST! Is the heartbeat coming? Great!

What you now need is a flexible UDP client that you can use to test the reception and return of the ESP8266 unit. The freeware packetsender is very suitable for testing. You can enter control commands here by hand, send them to the server on the ESP8266 via UDP and thus test the response of the program. Then the ESP32 takes over the command if everything works perfectly. So now install the software and set the desired UDP port of your computer at the very bottom, here UDP: 9181.

Packet Sender - IPs: 10.0.1.10, 2003:f3:7720:4a00:e5b7:54c7:7458:3289, 2003:f3:7720:4a00:8001:1de9:ba3...

File Tools Multicast Help

Name: LDR

ASCII: get:poti

HEX: 67 65 74 3a 70 6f 74 69

Address: 0.0.1.101 Port: 9000 Resend Delay: 0 Method: UDP Send Save

Clear Log (4) ☒ Log Traffic Save Log Save Traffic Packet Copy to Clipboard

Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex
12:01:25.011	10.0.1....	9000	You	9181	UDP		POTI:428	50 4F 54 49 3A 34 32 38
12:01:24.987	You	9181	10.0.1.101	9000	UDP		get:poti	67 65 74 3a 70 6f 74 69
12:00:22.955	10.0.1....	9000	You	9181	UDP		POTI:234	50 4F 54 49 3A 32 33 34
12:00:22.849	You	9181	10.0.1.101	9000	UDP		get:poti	67 65 74 3a 70 6f 74 69

UDP:9181 TCP:57449 SSL:57450 IPv4 Mode

Packet Sender - IPs: 10.0.1.10, 2003:f3:7720:4a00:e5b7:54c7:7458:3289, 2003:f3:7720:4a00:8001:1de9:ba3...

File Tools Multicast Help

Name: LDR

ASCII: get:bmp280:both

HEX: 67 65 74 3a 62 6d 70 32 38 30 3a 62 6f 74 68

Address: 0.0.1.101 Port: 9000 Resend Delay: 0 Method: UDP Send Save

Clear Log (6) ☒ Log Traffic Save Log Save Traffic Packet Copy to Clipboard

Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex
12:03:08.038	10.0.1....	9000	You	9181	UDP		TEMP:25.45\nPRES:1015.12\n	54 45 4D 50 3A 32
12:03:07.231	You	9181	10.0.1.101	9000	UDP		get:bmp280:both	67 65 74 3a 62 6d
12:01:25.011	10.0.1....	9000	You	9181	UDP		POTI:428	50 4F 54 49 3A 34
12:01:24.987	You	9181	10.0.1.101	9000	UDP		get:poti	67 65 74 3a 70 6f 74 69
12:00:22.955	10.0.1....	9000	You	9181	UDP		POTI:234	50 4F 54 49 3A 32 33 34

UDP:9181 TCP:57449 SSL:57450 IPv4 Mode

Now simply enter the requests that the ESP32 would otherwise send by hand in ASCII. Send sends the data to the ESP8266. If you get the right answers, then this part of the project will also work. Otherwise you still have the emergency brake to touch up the program.

## The final test

To do this, assign one of the flag values to the trigger variable in line 72 and start the program.

```
timeFlag = 0b00000001 # interval control
distanceFlag = 0b00000010 # distance alarm
tempFlag = 0b00000100 # temperature alarm
orderFlag = 0b00001000 # SMS request via SMS
queryFlag = 0b00010000 # Queries from external sensors via UDP
Trigger = 0 # Enter the flags of the services here
```

After the connection is established, your cell phone should reply with a new SMS message. Also test the other options one after the other. With SMS on demand you send a message with the content 'weather' to the ESP32. As a receipt, you will receive the temperature and air pressure values from the BMP280 on the ESP8266.

For the autonomous start of the ESP32 only the text of the program relais.py has to be copied into the file boot.py. The upload to the controller completes the project.

So now you are equipped with sufficient know-how to further develop the project and adapt it to your needs. I wish you a lot of fun and success.

Oh yes - there was something else! Our SIM808 still has a hit, you can of course also make calls with it. In the next episode, I'll tell you how to do this and how to connect a keypad. See you!

More download links:

[PDF in deutsch](#)

[PDF in english](#)