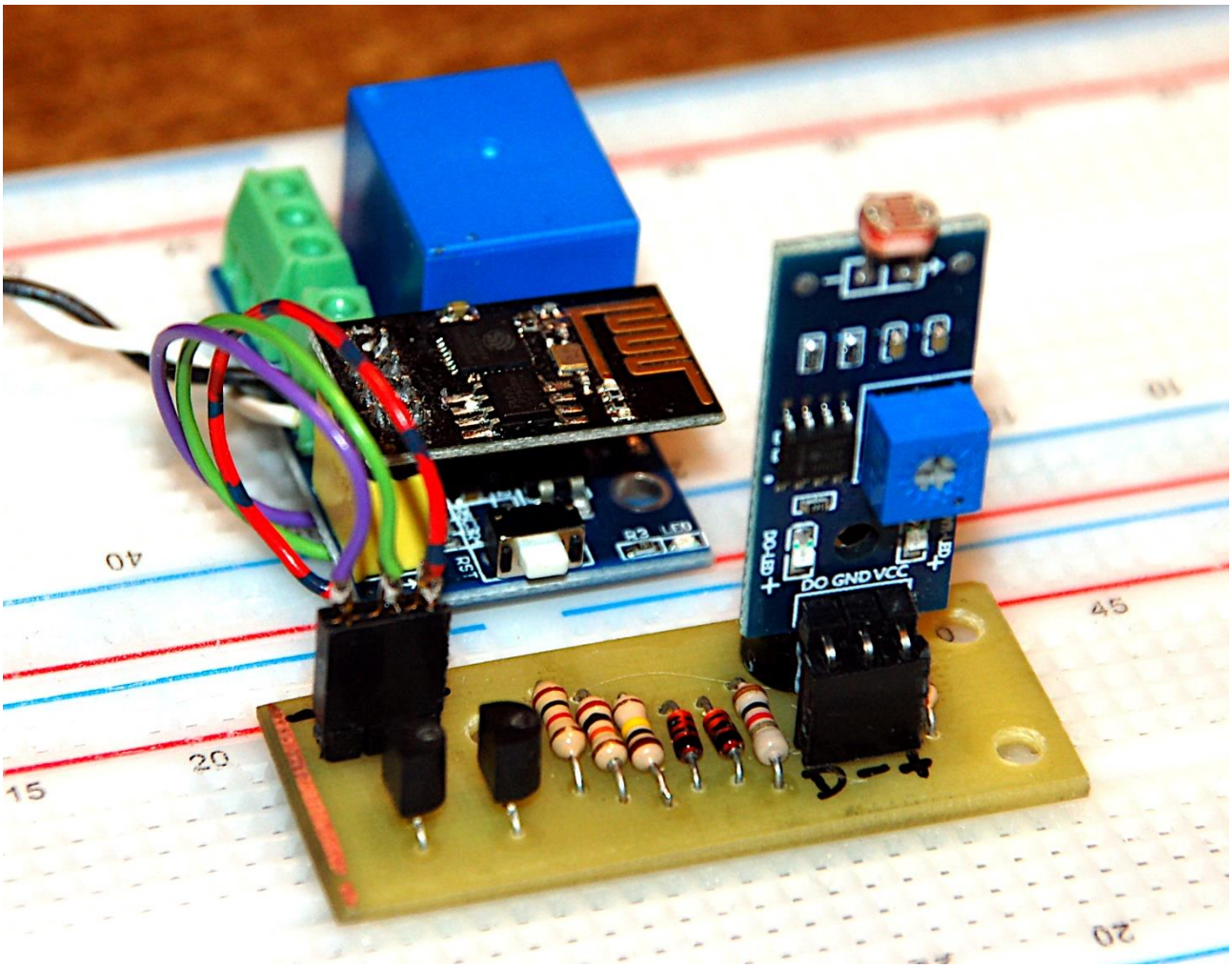


Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

In a previous post we had used the ESP32 and some of its ESP8266-12F brothers for sensor polling. For the radio connection I had a WLAN router in operation. A solution without a router seemed to me to involve too much effort and there was also a problem with the reuse of IP addresses by the controlling client. Perhaps you have had the "OSError: [Errno 98] Address already in use" error yourself. Well, that is exactly what I have now found a simple solution for, which I will not withhold from you. The project developed in this way provides up to 9 wireless switches that are controlled and queried directly by an ESP32 without a WLAN router as an intermediate link. The ESP32 receives its instructions via a SIM808 as SMS messages. In the same way, the client receives a message about the outcome of the operation. As a minimalist solution for the switching units, I have provided relay modules with an ESP8266-01. But before we get into the nitty-gritty, welcome to the project

Attendance Guy with ESP32, ESP8266-01 and SIM808

The LCD keypad from the earlier episodes is also used again, mainly the display. Of course, the RST button again fulfills the function of an emergency brake.

Using the emergency brake means terminating the current program precisely without restarting it immediately. All objects, variable contents and function definitions created up to that point are retained for manual access via REPL. In this way, functions can be tested, for example, without having to re-enter a whole series of imports, etc. each time. A decisive advantage of MicroPython is that such tests can be carried out easily via the REPL command line.

Hardware – etwas Zuwachs

Here is the list of ingredients for the attendance simulator (aka Attendance Guy). You probably already have the "heavy" assemblies if you have already been busy building in the previous episodes.

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	I2C IIC Adapter serielle Schnittstelle für LCD Display 1602 und 2004
4	Widerstand 10kΩ
1	SIM-Karte (beliebiger Anbieter)
3	ESP8266-01S ESP8266-01S Wlan WiFi Modul 5V mit Relais Adapter
1	USB-zu-ESP8266 01 Serial Wireless Wifi Module für ESP8266-01
3	Foto Widerstand Photoresistor Licht Sensor Modul LDR5528
1	USB-zu-ESP8266 01 Serial Wireless Wifi Module für ESP8266-01
3	Elektrolytkondensator 470µF, 6,3V oder höher
6	Transistoren BC548 (NPN Kleinleistung, 30V, 100mA)
6	Widerstand 1,0kΩ
3	Widerstand 10kΩ
3	Widerstand 15kΩ
3	Widerstand 100kΩ
6	Dioden 1N4148
4	Battery Expansion Shield 18650 V3 inkl. USB Kabel
4	Li-Akku Typ 18650 oder besser
4	5V-Steckernetzteil / 12V-Steckernetzteil, 1-2 A oder 220V zu 5V Mini-Netzteil
3	Optionale Netzteile 5VDC bis 30VDC für die Versorgung von Beleuchtungskörpern

The circuit for the Attendance Guy is largely taken over by episode 2. New are the WIFI modules with relay adapters, the light sensor modules LDR5528, the USB-to-ESP8266-01 adapter and the various small parts. Of course, components from previous episodes are also used again. The number of 9 possible relay stages should be sufficient in most cases, but can be increased if the program is adapted accordingly. The project is almost freely

scalable in this direction. The available IP addresses in the subnet used represent an absolute limit.

Please note the following for the power supply of the circuits:

- • A safe power supply can be provided with 5V plug-in power packs. This is more reliable than a battery supply for long-term operation. Alternatively, a 220VAC to 5VDC converter can also be used.
- • No electrical data can be seen on the relay housing. The data sheet states 5A, 30VDC / 50VAC. You should not use these modules to switch higher voltages, also or especially with regard to the conductor path on the relay board. The distances between the voltage to be switched and the 5V-V level of the control unit are in places just under a millimeter. Despite the optocoupler, there is no galvanic separation between the input stage and the relay driver because the signal ground is connected through. **Only switch voltages higher than 50V with the relay if you know exactly what you are doing!**
- • Depending on the "consumer" to be switched, another suitable power supply unit must be used. If necessary, the 5VDC supply can also be derived from this using a suitable controller.

The Software

Verwendete Software:
For flashing and programming ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:
Micropython Firmware
Pick a Stable-Version

MicroPython-Module und Programme
GPS-Modul für SIM808 und GPS6MV2(U-Blocks)
LCD-Standard-Modul
HD44780U-I2C-Erweiterung zum LCD-Modul
attendance_guy.py Hauptprogramm der ESP32-Steuereinheit
server.py Hauptprogramm der ESP8266-Relaismodule

Tricks and Infos 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, µPyCraft 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 and hardware to trying out and refining functions and entire program parts. For this purpose I also like to create small test programs over and over again. They form a kind of macro because they combine recurring commands. One such program evolved into the attendance_guy.py program, which we will discuss shortly. If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file under boot.py in the workspace and upload it to the ESP32 / ESP8266-01. The program starts automatically the next time it is reset or switched on.

Such programs are started manually 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.

If you want to use the controller again later together with the Arduino IDE, simply flash the program in the usual way. However, the ESP32 / ESP8266 then forgot that it ever spoke MicroPython.

The inner workings of Attendance Guy

Two separate programs are required for the Attendance Guy project, attendance_guy.py and server.py. The first belongs to the ESP32 which controls the ESP8266-01 satellites. Server.py must be installed on these relay units. Both require that the controllers have been flashed with the MicroPython firmware beforehand.

Die Hardware

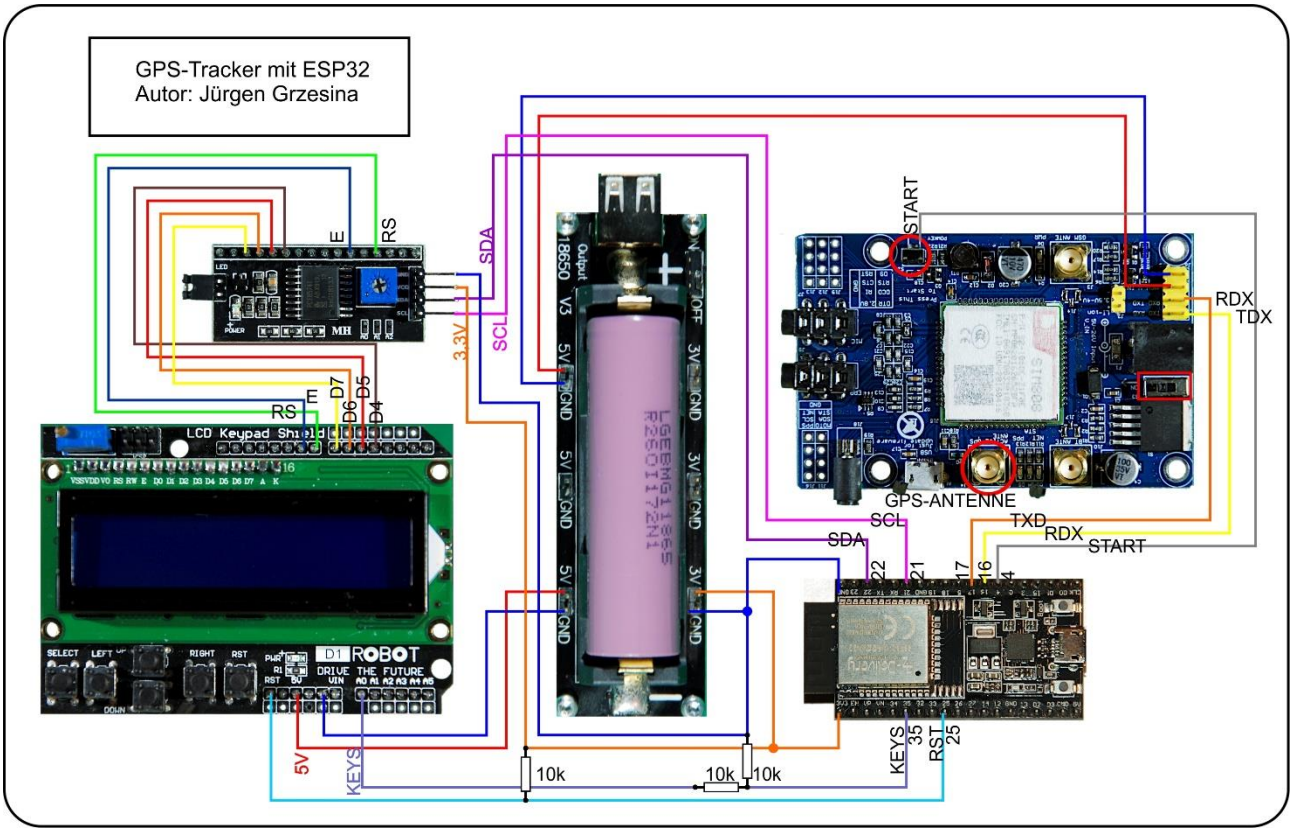
Two programs - two hardware types, the ESP32 takes over the communication to the SIM808, receives the SMS messages, checks them, forwards them to the ESP8266-01 units and receives the result messages from them, in order to finally send them to the client as SMS messages to send back.

A UDP server runs on the ESP8266-01 relay stages. He receives the orders from the ESP32, filters them again himself, executes the order and reports the result via UDP to the ESP32, which then decides whether the message should be forwarded via SMS.

The units are equipped with hardware according to the tasks. The SIM808 is attached to the ESP32 on UART port 2 and the LCD keypad display is attached to the I2C parallel adapter. The 4x4 keyboard matrix and BMP280 are not used in this blog, but can remain connected.

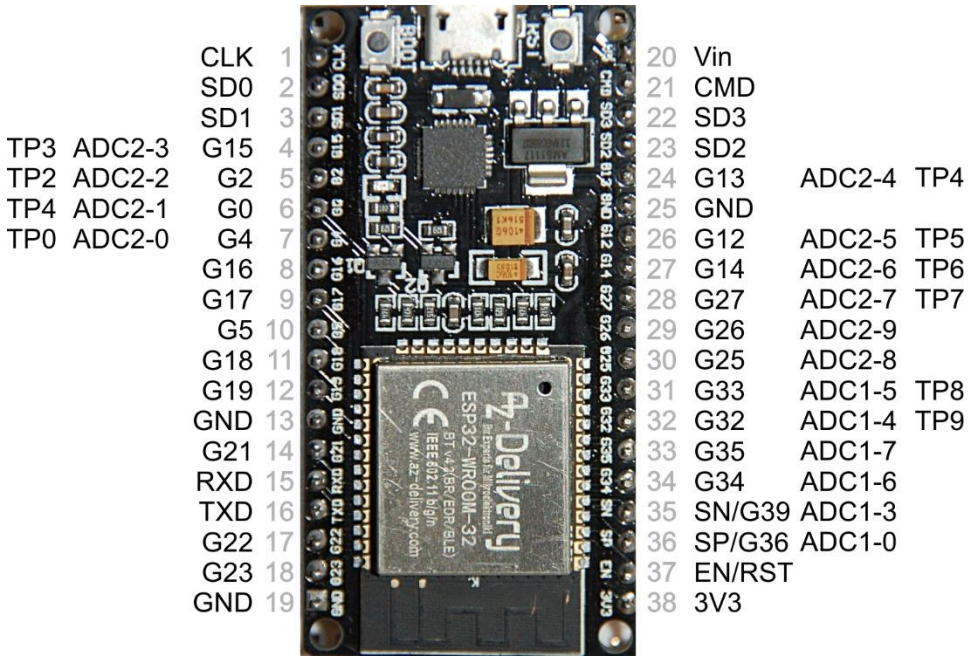
The control unit

Before programming, however, the circuit has to be set up. The circuit diagram of the ESP32 control unit shows the necessary connections. It makes sense to replace the battery adapter with a 5V / 3.3V power supply.



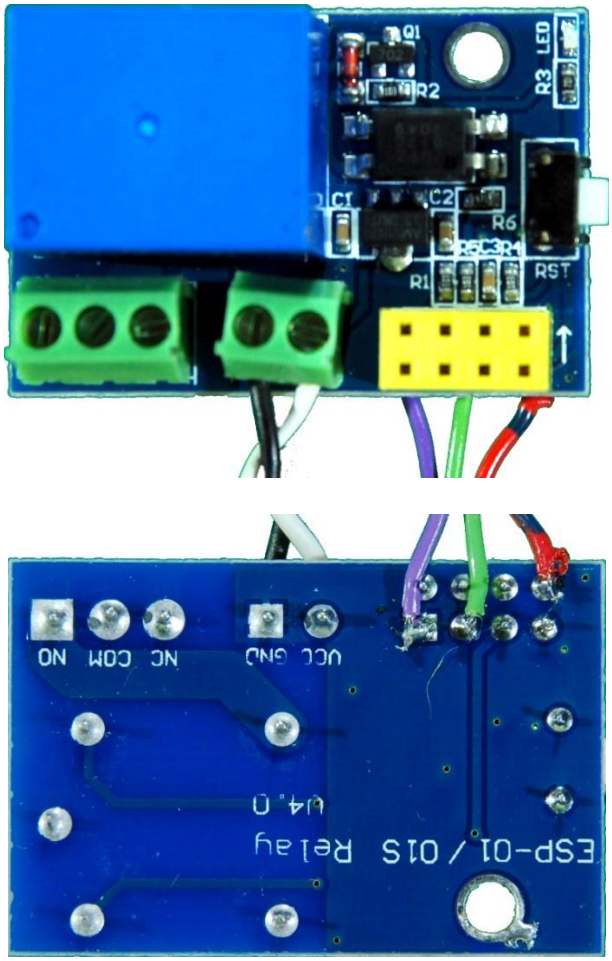
You will receive a more legible copy of the illustration in DIN A4 with the Download der PDF-Datei .

If you replace the battery holder with a power supply unit, you must make absolutely sure that the ESP32 and the serial-parallel adapter must be supplied with 3.3V as shown here. The display and SIM808 need a 5V supply voltage. The ESP8266-01 relay stages also need 5V. You can also connect 5V to the ESP32, but only to pin Vin (pin 20) or 5V, never to the 3.3V pin! A maximum of 3.3V applies to all connections of the ESP32!

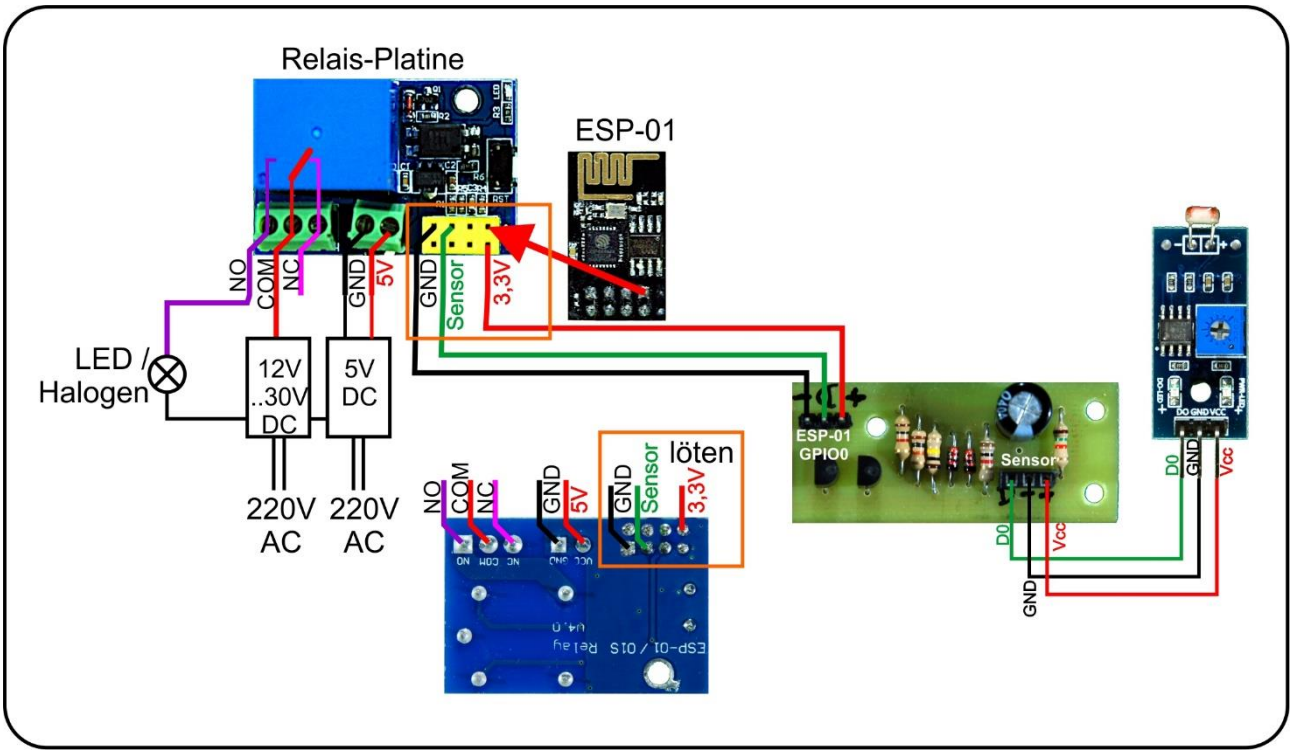


I am using the LCD keypad again because the display does a good job in the autonomous operation of the ESP32. Status reports, error messages, requests to press a button, etc. can be easily communicated via it. The control via the serial-parallel adapter easily adjusts the 3.3V outputs of the ESP32 to the 5V pull-up connections of the display.

ESP8266-Relais-Modul



By plugging the ESP8266 onto the relay board, almost all connections are already established. The circuit diagram also shows the 5V supply voltage and the connection of 12V lighting. The 5V can optionally be obtained from the 12V source using the 7805 controller. These parts are not included in the hardware list above. The relay module must not be operated with more than 5V, as this would damage the relay coil.



The control of the switching transistor via an optocoupler is regulated in such a way that a HIGH level at the GPIO0 of the ESP8266 de-energizes the relay coil and a LOW level causes the relay to pick up. A relay that behaves in this way is called "LOW level triggered". In the first case the contact between COM (common aka switching contact) and NC (normally closed aka break contact) is closed, in the second case COM is on NO (normally open aka working contact). Our program just has to ensure that the GPIO0 pin behaves in the same way and that a HIGH level is present at the connection when the ESP8266-01 unit is booted. The relay board takes care of the rest.

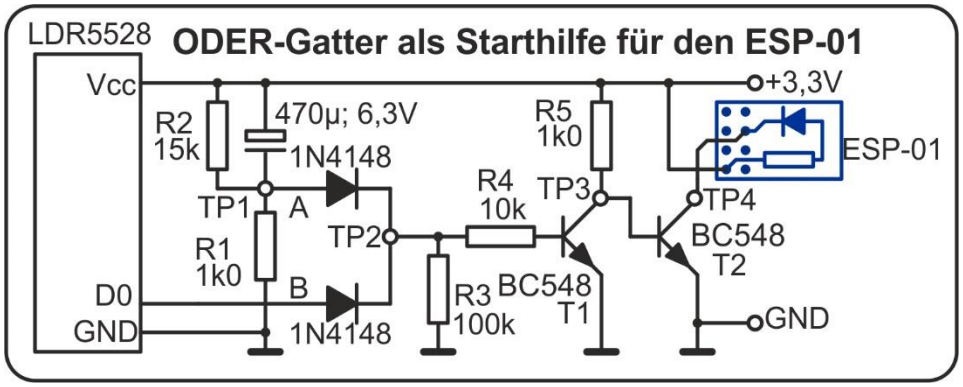
So much for the normal function intended by the manufacturer. But we want more! We want to see whether the switching process was successful, so we need feedback. When we switch on lamps, we can use an LDR (Light Dependent Resistor) to control it. It reduces its value when light falls on it, in the dark it has a high resistance. Unfortunately, the ESP8266-01 cannot measure voltages and certainly no resistance values.

Never mind, the light sensor module LDR5528 does it for us. A digital signal is available at its output, the state of which depends on the ambient brightness and the position of the potentiometer trimmer on the board. When it is bright, output D0 is LOW (0), when it is dark it is HIGH (1). The threshold for switching is set by the trimmer. Well problem solved! - think!

Now a circumstance comes into play that has to do with the boat characteristics of the ESP8266 in general and those of the ESP8266-01 in particular. The controller of the ESP8266-01 is just an ESP8266. With this controller, for a normal start, after switching on or after a reset, the lines GPIO0, GPIO1 and GPIO2 must be at HIGH potential. GPIO1 is the TDX line that is used for data transfer between PC (USB) and controller and is therefore excluded from the start for our purposes. GPIO0 supplies the relay control. The line is pulled to HIGH by the ESP8266-01 via a pull-up resistor during start-up, which means that the relay is always off after the start.

Unfortunately, we do not know what the ambient brightness is when the relay unit boots. If we connect the D0 output of the LDR5528 module to GPIO2, the ESP8266-01 will not start if it is light at this moment because the level is then LOW. So can we forget the ESP8266-01 module for our purpose?

It does not work! - Doesn't exist! This is the advertising slogan of a well-known hardware store. And so it is here too. What doesn't work will be done! The blue LED of the module is connected via a resistor between GPIO2 and + 3.3V and the two pull the connection to Vcc. We now only have to prevent the potential 0 level of the LDR5528 from reaching the GPIO2 during the short start phase. This is exactly what the following additional circuit does, which is basically an OR circuit with a subsequent open collector driver.



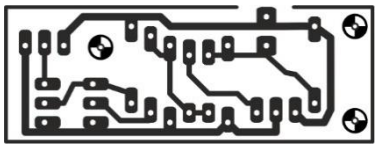
When switching on, the electrolytic capacitor is charged via the resistor R1. The voltage at point TP1 falls from 3.3V within a few tenths of a second to a value below 0.7V at test point T2. Within this time span, TP2 is HIGH, regardless of the level at input B from D0. If the electrolytic capacitor is charged to more than 1.9V, input A from TP1 appears as LOW, and the level at B alone determines the voltage at TP2. The transistor T1 buffers the high-resistance signal and unfortunately also negates it. Therefore we use a second BC548 or something similar. The collector of this transistor is connected to GPIO2, which is internally connected to Vcc via a resistor and the blue LED. If the basis of T2 is at 0V, T2 blocks and GPIO2 is at HIGH level. If the base is more than 0.7 V, the transistor switches through and GPIO2 is pulled against GND.

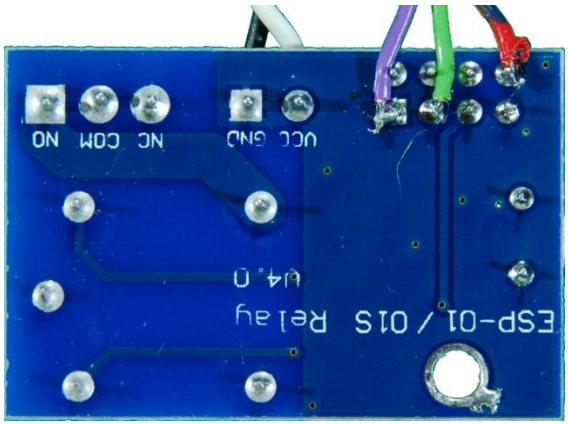
The table gives an overview of the levels at the various test points. A forward voltage of the diodes of typically 0.7V is assumed for the specified values.

LDR5528	TP1	TP2	TP3	TP4	Relais	
0	>2,6V	>1,9V	0	1	aus	
0	<1,3	<0,6	1	0	an	
1	d.c.	d.c.	0	1	aus	

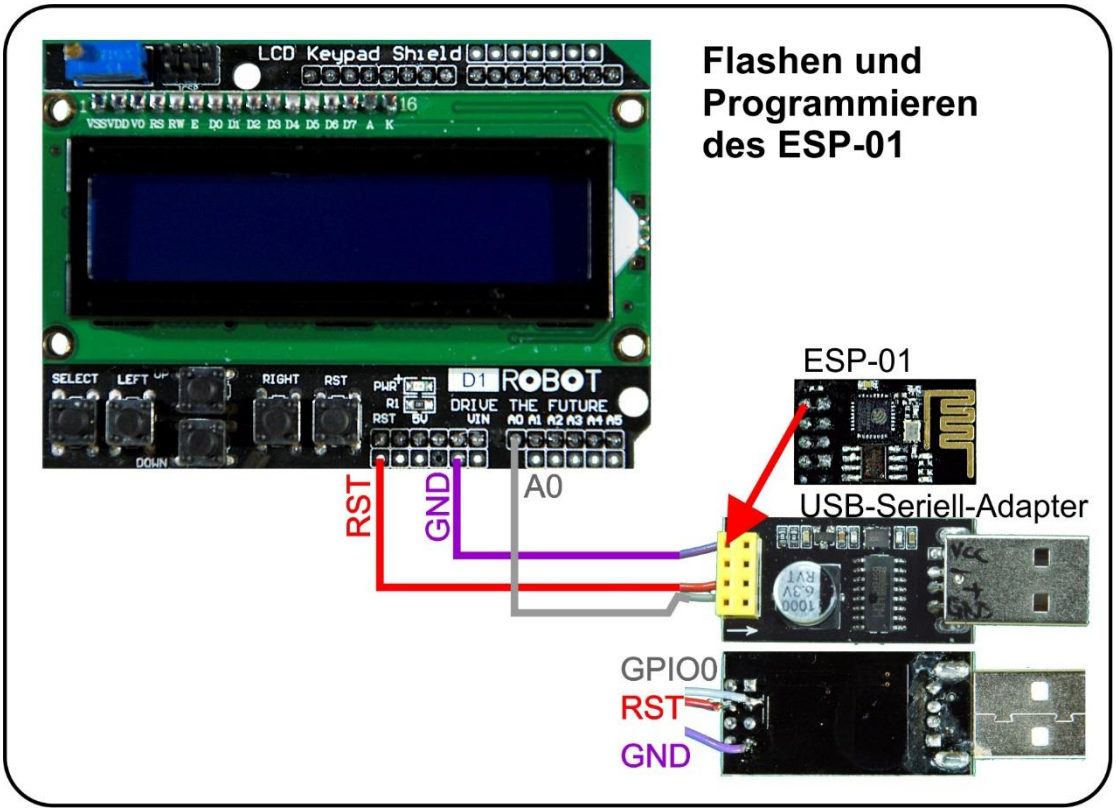
d.c. = don't care

The circuit can be built on a breadboard or a breadboard. For professionals there is a circuit board layout for download PDF-FIle. The connection is made on the underside of the relay board by soldering short wire connections (violet = GND, red = Vcc, green = TP4).



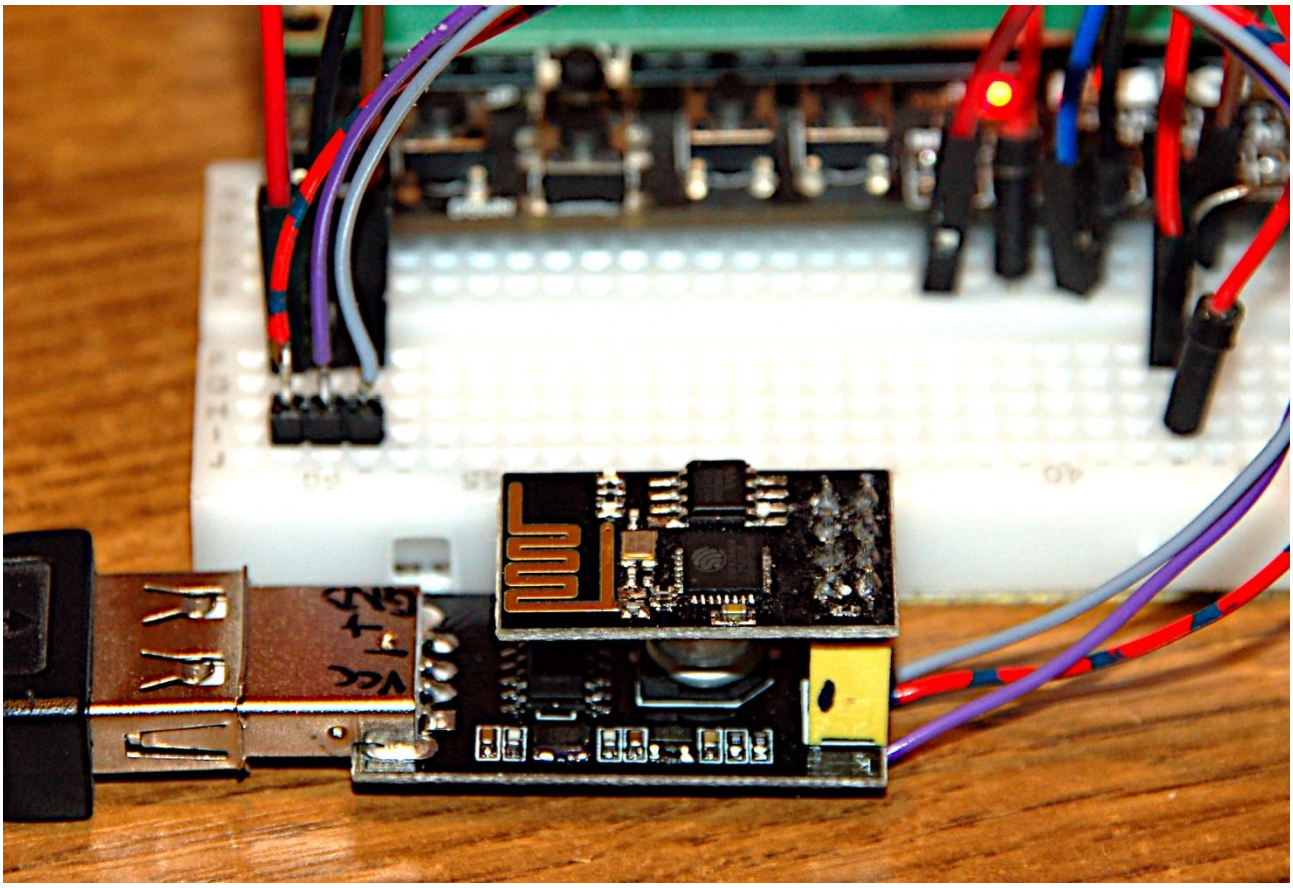


Finally, in preparation, we have to flash the ESP8266-01 modules with the MicroPython firmware. For this I use a USB-TTL adapter with a slot for the controller board of the ESP8266-01.



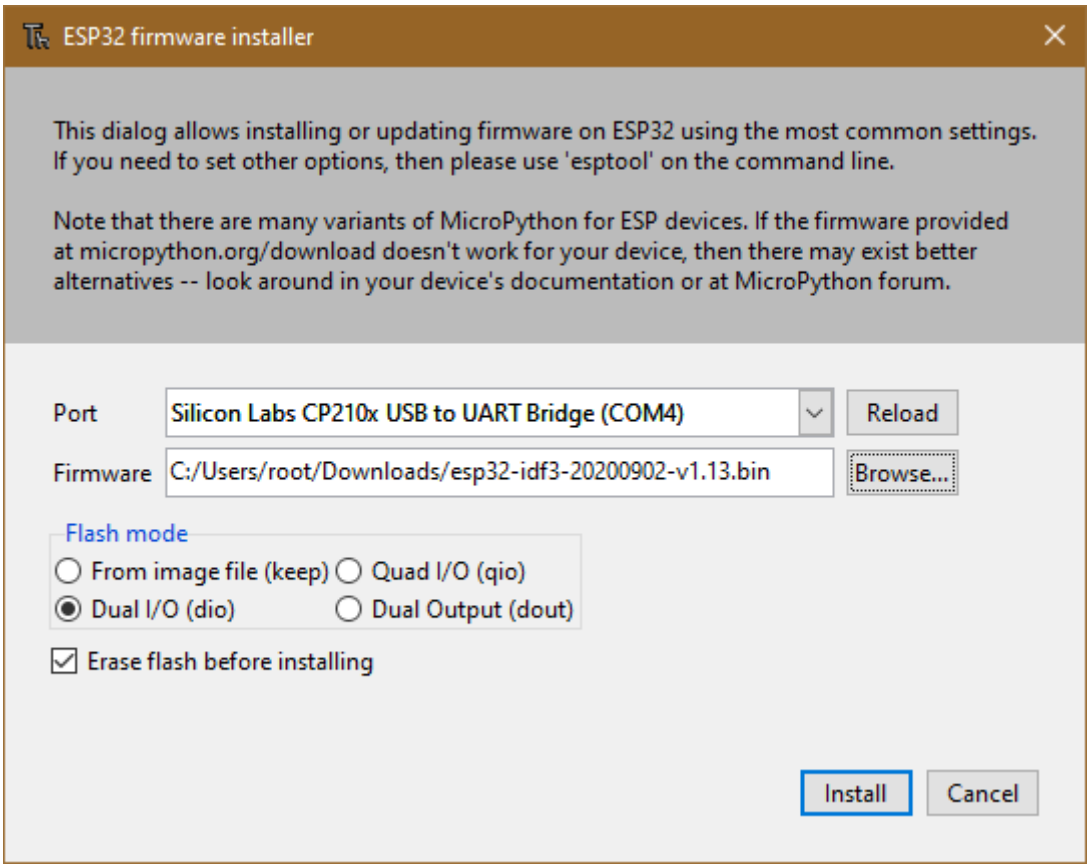
In the illustration you can see the three cables that I have also attached and that make the process much easier. I connect RST, GPIO0 and GND with a three-pin piece of pin header. Using jumper cables, I then connect the RST connection to the pin of the same name on the LCD keypad, the GPIO0 line goes to the A0 connection. The RIGHT key, as a flash key, thus provides the connection to GND. Of course, GND must be connected to

GND. Now I can manually enter the level sequence, which the automatic flash system presents on the ESP32 under program control, RST + FLASH hold, RST release + FLASH hold, FLASH release. Please remember that you have to perform the process twice, once to delete the flash content and a second time to flash the firmware.



The procedure for burning the firmware is otherwise identical for the ESP32 and ESP8266-01. To do this, download the two latest bin files:
esp8266-1m-20210418-v1.15.bin
esp32-idf4-20210202-v1.14.bin

In Thonny, open the Thonny options window via Run - Select interpreter. At the bottom right, a left click on Install or update firmware opens the firmware installer.



Set the port and select the appropriate firmware file. Now start the burn process and remember to press the buttons on the ESP8266-01.

The Software

Both software parts have to fulfill a number of other tasks in addition to operating the UDP connection. This is particularly true for the program on the ESP32, which has to manage the SMS traffic. Both parts are also busy parsing the messages and assembling and passing on the results.

So that both parts don't get stuck in the receiving loop, I built in a timeout of 2 seconds when creating the UDP socket. This enables the controller to carry out other tasks in the main loop after this time has elapsed.

The ESP8266-01-UDP-Server

Let's start with the simpler program for the ESP8266-01. Each of the ESP8266-01 units represents a UDP server in a free subnet of the 10.0.0.0 range. Because nobody needs 16.7 million switching points, I have shrunk the whole thing down to the 10.0.2.0 subnet. I use 9 of the remaining 252 addresses, from 10.0.2.101/24 to 10.0.2.109/24. The ESP32 as administrator also has the 10.0.2.199/24. The network mask of 255.255.255.0 is derived from the suffix / 24. That means 24 bits, i.e. the first 3 bytes of the IP, represent the network part of the IP address, the last byte the device address. The port number of the server is 9000. In a WLAN with a router, all server stations could therefore also be addressed via broadcast, for example to collectively query their properties.

But we are going a different way, which works without a wireless router. Therefore, each ESP8266-01 unit is operated as a separate access point and contacted by the central ESP32 if required.

Te program server.py

The program is quite clear. After importing various MicroPython internal modules, we define a few variables, structures and functions. An important point is the definition of the unit number and thus the IP address. The permanent part of the IP address is also required for this.

```
# ****************** set station number *************
unitNumber = 1 # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
subnet = "10.0.2.10"
use = "RELAIS"
# ********************************************** *********
```

The unitNumber is simply appended to the subnet string as a string. For another station, only a different value has to be specified for unitNumber. That's all. Copy the program text into an empty, newly created file and save it as boot.py in your working directory. From there, upload the file to one of the flashed ESP8266-01. The program starts automatically with the next reset.

The next important point in the program is setting up the access point. We'll go through this in more detail because this part appears in a similar form in the attendance_guy.py program.

```
#A We create a network interface object and
#B activate it
#C From the number of the unit we create the SSID for the access point and
#D the IP address
#E We configure the interface object with the generated address data. It is important to
        enter the IP mask, gateway and DNS address as tuples.
#F Only 0 is accepted as the authentication method, which means that the access point is
        accessed without password protection.
#G We retrieve the MAC address from the configuration, decode it as a string and output
        it.
#H We assign the name to the interface object, the password is an empty string.
#J We may wait for the interface setup to complete.
```

```
nic = network.WLAN(network.AP_IF)  # A
nic.active(True) # B
ssid="unit"+str(unitNumber) # C
passwd="uranium238"
IP=teilnetz+str(unitNumber) # D
print("Weise IP zu:",IP)

# Start als Accesspoint
nic.ifconfig((IP,"255.255.255.0",IP,IP)) # E

print(nic.ifconfig())
```

```
# Authentifizierungsmodi ausser 0 werden nicht unterstuetzt
nic.config(authmode=0) # F

MAC=nic.config("mac") # G
# 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) #H

while not nic.active(): # J
  print(".",end="")
  sleep(0.5)

print("Unit{} AP connected".format(unitNumber))
```

We continue with the establishment of a UDP socket.

```
portNum=9000 #K
#print("Fordere Server-Socket an")
# ---------------- Server starten -------------------------
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM) #L
s.bind(('', portNum)) # M
s.settimeout(2.0) # N
blink(2,0.5,True) # P
blink(0.3,0.8,True)
blink(0.3,0.8,True)
blink(0.3,1.5,True)
print("Socket established, waiting...")
print("Empfange Anfragen auf ",IP,":",portNum, sep='')
```

#K Definition of the port number for all units
#L We create an IPv4 UDP socket object, UDP is based on datagram packets.
#M We bind the socket to the IP address and port number generated above.
IP and port number are transferred as a tuple.
#N We set the timeout to 2 seconds. If there is no request during this time, the input loop is exited and further commands can be processed.
#P Because the server does not have a display, readiness to receive is indicated by a flashing signal. After the pin GPIO0 has been defined as the LED connection, the relay switches the connected lighting in the programmed pulse sequence.

We first look in the server loop to see whether there is a request. The recvfrom () function is exited if this is the case or if the 2 seconds of the timeout have expired. The return of the recvfrom () function is a tuple consisting of the request text and the address of the sender. When assigning, we break down the tuple to the two variables. In MicroPython this is called unpacking. Up to 160 characters can be read in one go.

```
        request, addr = s.recvfrom(160)
        response=""
        r=request.decode("utf8")
```

A response string is prepared. The request is delivered as a bytes object and must be decoded into string format for further processing.

The request is then parsed. I set the format as follows.

SET: value, value may be 0 or 1
GET: STATUS
BROADCAST

Of course, the BROADCAST command only returns the properties of the server itself as a response, because there are no other stations in its subarea. The command is primarily used to query the purpose of the ESP8266-01 unit. The information about this is entered at the beginning of the program.

```
# ****************** set station number *************
unitNumber = 2 # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
subnet = "10.0.2.10"
usage = "RELAIS"
# ********************************************** *********
```

The ESP32 as an intermediary

This part of the project is very interesting in terms of handling additional tasks in the server loop and, above all, the attendance_guy.py program answers the question of how to avoid the error message "OSError: [Errno 98] Address already in use". I didn't find anything while searching the web, so I had no choice but to; "Young people research". The solution is that simple.

The configuration data for the network and SMS operation are right at the beginning.

```
# ****************************************
#
phoneNumber = "0xxxxxxxxxxx"
SSID = "here goes your WLAN SSID"
PASSWD = "your password goes here"
#
# ****************************************
```

phoneNumber records the phone number of the cell phone from which control SMS are to be sent. SSID and PASSWD refer to a WLAN with a router as an access point. This is very useful for initial tests, but is no longer needed later. Similar to the ESP8266-01 units, there is a commented sequence in the program that manages the login to a router access point.

```
blinkLed=Pin(2,Pin.OUT)
ctrl=Pin(25,Pin.IN,Pin.PULL_UP)
request = bytearray(50)
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(21),sda=Pin(22))
```

```
d=LCD(i2c,adr=0x27,cols=16,lines=2)

g=GSM(switch=4,disp=d,key=ctrl)
g.simGPSInit()
g.simOn()
g.simStopGPSTransmitting()
```

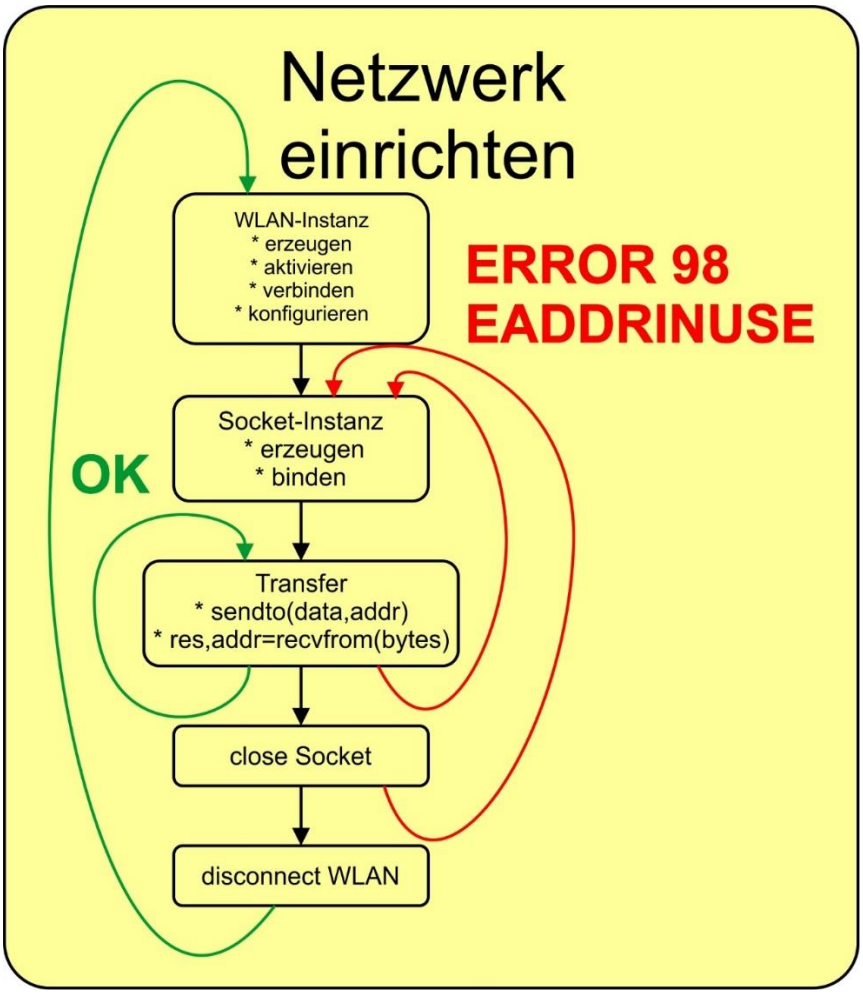
blinkLed replaces the usual LED on pin GPIO2, which is missing on the ESP32. ctrl is the connection of the emergency brake, which is implemented using the RST button on the LCD keypad. The receive buffer request is defined in advance for 50 characters.

The I2C object is used to define the display object d, which in turn flows into the GSM instance g, in addition to the switch output and the ctrl key.

We initialize the hardware, switch on the module and then immediately deactivate the forwarding of GPS data, which would only clutter the UART buffer.

A number of functions now follow. Together they form a chain of actions to establish a connection and to transmit messages. In contrast, the main program is very clear with its 30 lines.

Unlike in previous projects, the UDP client must first establish a network connection to each access point before data can be exchanged via UDP. That is the crux of this application. Because this is exactly what made it possible to use the same IP address for the client over and over again. Once again very clearly: It does not work to just close one socket with an existing WLAN connection and to reuse the same IP when opening it again, but you have to trample the entire network connection and re-establish it. The following graphic shows the options for the procedure.



A socket can be understood as a gateway to data exchange, like a file handle on a medium, i.e. external hard drive, USB stick, printer, etc. Two things are required for the connection, a line and the interface software. The situation is similar for a WLAN connection. You need a radio link, that is the connection between client and access point and an interface software, that is the UDP socket here. In order to be able to establish a new connection, the previous one must be crushed, and this includes closing the socket and then disconnecting the radio link. Only then did MicroPython forget which IP address was previously in use. Simple, isn't it?

Because the establishment of the radio link has to be carried out repeatedly, I set up this part as the connectToLocalAP (unit) function. The unit parameter is the number of the ESP8266-01 station that I want to address.

The definition of the connection data looks a little different for the ESP32 client.

```
# ********** Define station data **********
Subnet = "10.0.2."
Attempts = 20
Stations = [# [IP, tool, present, status]
    ["10", "PC", False, ""],
    ["101", "Relay", False, ""],
    ["102", "Relay", False, ""],
    ["103", "TVsim", False, ""],
```

]

Subnet really only contains the network part here. Attempts specifies how often the loop may be run through to establish a connection. Stations is a list of lists with information on the device address, the task and the availability information of the satellite unit. The procedure in the function is essentially the same as with the ESP8266-01 units.

connectToLocalAP returns a tuple that contains the WLAN object and, as a tuple, its own and the target address including port number.

(nic, (STA_IP, 9191), (targetIP, targetPort))

The target address is used as the input parameter of the openSocket () function. Worth mentioning because it is important is the socket.SO_REUSEADDR parameter when creating the socket instance. This parameter and the procedure described above for opening and closing a connection allow us to use the same IP address for the client over and over again. The selected function name should remind of the command to open a file. Basically, that's nothing else.

```
def openSocket(localAdr): # localAddr=(STA_IP,STA_Port)
    s=socket.socket(socket.AF_INET, socket.SOCK_DGRAM, \
                    socket.SO_REUSEADDR)
    s.settimeout(5.0)
    s.bind(localAdr) # localAddr has to be a tuple
    return s
```

openSocket () returns the socket object s that is still needed for the data transfer. This part is done by the xmitJob () function. It takes the socket object, the message to be sent and the destination address as positional parameters.

```
def xmitJob(s,mesg,targetAdr):
    # s= open socket; mesg = coded job
    # targetAdr has to be a tuple
    global request
    s.sendto(mesg,targetAdr) # targetAdr has to be a tuple
    sleep(1) # wait for response
    request,adr=s.recvfrom(50)
    r=request.decode("utf8")
    print(r,adr)
    return r
```

s.recvfrom () waits up to 5 seconds for a response after sending and decodes it if necessary. The (possibly empty) response string r is returned.

```
def closeSocket(s):
    s.close()
    s=None
    gc.collect()

def killConnection(nic):
    nic.disconnect()
```

closeSocket () and killConnection () ensure that the connection is properly closed.

Incoming SMS messages have to be parsed in order to tease out their command content. An SMS can contain several orders that must follow a certain syntax. Instead of a general description, I'll just give an example.

SMS text:
unit:1,get:status
Unit:2,set:1
UNIT:3,Set:0

Upper and lower case is not important, but the colons ":" and commas matter.

The answer on the cell phone should look something like this:

UNIT:1,GET:STATUS ->
UNIT1:LIGHT:ON
UNIT:2,SET:1 ->
UNIT2:RELAIS:ON
LIGHT:OFF
UNIT:3,SET:0 ->
UNIT3:RELAIS:OFF
LIGHT:OFF

When the message is parsed, the status "REC READ" and the permitted telephone number are checked. If the caller number is incorrect or if the SMS has already been processed, it is deleted immediately. Each of the three jobs listed above is packed as an entry in the job list, invalid or garbled information is ignored. Colons and commas serve as orientation for the parser, which works with the string methods split () and find (). Each entry in jobList is itself a list and contains the number of the unit to be addressed, the job as a string and the assigned value as a number. The job list is returned.

```
def readSMS(index,mode):
    data=g.gsmReadSMS(index,mode)
    print(data)
    status=data[0]
    if status == "REC READ" or not(phoneNumber[1:] in data[1]):
        print("obsolete SMS",g.gsmDeleteSMS(index))
        return []
    else:
        text=data[4].upper()
        print("Text der SMS:",text)
        zeilen=text.split("\n")
        jobListe=[]
        for i,text in enumerate(zeilen):
            p1=text.find("UNIT:")
            if p1!= -1: # mindestens 1 Colon gefunden
                unit,text=text[5:].split(",")
                unit=int(unit)
                try:
                    job,value=text.split(":")
                    if job in ["SET","GET"] and \
                            value in ["0","1","STATUS"]:
                        jobListe.append([unit,job,value])
                    else:
                        print("Invalid Job or value")
                except:
                    if text.upper()=="BROADCAST":
                        job="BROADCAST"
                        value=0
                        jobListe.append([unit,job,value])
                    else:
                        print("Invalid job\n")
            else: # unit: not found
                print("Invalid syntax\n")
        #g.gsmDeleteSMS(index)
        return jobListe
```

Knock, knock, who's there, sang Mary Hopkin in 1970. Our program doesn't sing, but it looks to see who is there, within reach. When going through the list of stations, the reaction of the satellite stations is taken as an occasion to change the entry for present from False to True. Only these stations will later be taken into account when sending commands.

```
def knockKnockWhosThere():
    global Stationen  # (IP,device,present,status)
    anzahl=len(Stationen)
    for i, l in enumerate(Stationen):
        if i>0:
            print(i,l)
            n=connectToLocalAP(i) # (nic,STA-ADR,TargetADR)
            print("response",n)
            if not(n is None):
                l[2]=True
                n[0].disconnect()
            else:
                l[2]=False
```

The function that processes the job list is doJobs (). It takes the jobList list from readSMS (), compiles the commands for the ESP8266-01 units from the fields of the entries and sends them after the connection to the access point has been established and the socket has been established. The responses from the individual jobs are compiled in the response string and returned as the overall result. Socket and WLAN connection are closed.

```
def doJobs(joblist):
    global Stationen
    response=""
    if joblist==[]:
        return response
    for job in joblist:
        print("Job:",job)
        # Jobs abarbeiten
        unit=job[0]
        if Stationen[unit][2]:
            print("\n---------------\nStation:",Stationen[unit])
            mesg=":".join(job[1:])
            n=connectToLocalAP(unit)
            if not(n is None):
                sock=openSocket(n[1])
                response=response+"UNIT:"+str(unit)+","+mesg+\
                         " -> "+xmitJob(sock,mesg,n[2])+"\n"
                closeSocket(sock)
                sock=None
                killConnection(n[0])
            else:
                response=response+("UNIT:"+str(unit)+" failed\n")
        print("***************** DONE ***********")
    return response
```

We are almost done with the discussion of the program, only the main loop is missing. When the program starts, we delete all old SMS messages and look around for destinations. We report any stations we have found, then we clean the UART buffer.

In the loop, we check for unread, i.e. fresh, SMS messages. Each of the messages is broken down, a job list is created from it and this is then processed line by line. Each job provides an answer list that is sent back to the client via SMS. Thanks to the functions that do the real work, the main program is very easy to read. The end of the loop is the emergency brake.

```
clearOldSMS()
knockKnockWhosThere()
for i,einheit in enumerate(Stationen):
    if einheit[2]:
        print("Gefunden:",i,einheit)
g.simFlushUART()
while 1:
    SMSListe=g.gsmFindSMS("REC UNREAD",1)
    print("SMS-Index:",SMSListe)
    if len(SMSListe) != 0:
        task=SMSListe[0]
        print("TASK:",task)
        jobListe=readSMS(task,0)
        print("Jobliste:",jobListe)
        ergebnis=doJobs(jobListe)
        print("Jobs fertig:",ergebnis)
        g.gsmSendSMS(phoneNumber,ergebnis)
        sleep(5)
        print(g.gsmDeleteSMS(task))
        print("Weitere SMS:",g.gsmFindSMS("ALL"))
    else:
        print("Keine SMS - nothing to do")
        pass
        sleep(5)
    g.simFlushUART()
    #
    if ctrl.value() == 0:
        print("Program stoped")
        d.clearAll()
        d.writeAt("PROGRAMM STOPED",0,0)
        sys.exit()
```

I have to sensitize you for a position in the listing. She teased me for a long time. the 5 second pause after the message has been sent determines whether the message that has just been processed is actually deleted by gsmDeleteSMS (). If the command comes too soon, the message is retained and is subsequently executed again and again. You certainly don't want that, especially since you are being bombarded with text messages from your attendance guy.

Are you now ready to build elegant housings for the units or to change or expand the programs according to your ideas? Then I hope you enjoy it at least as much as I had while developing the application. I have linked the modules used and the two complete programs at the beginning so that you can get started right away.

PDF in deutsch

PDF in english