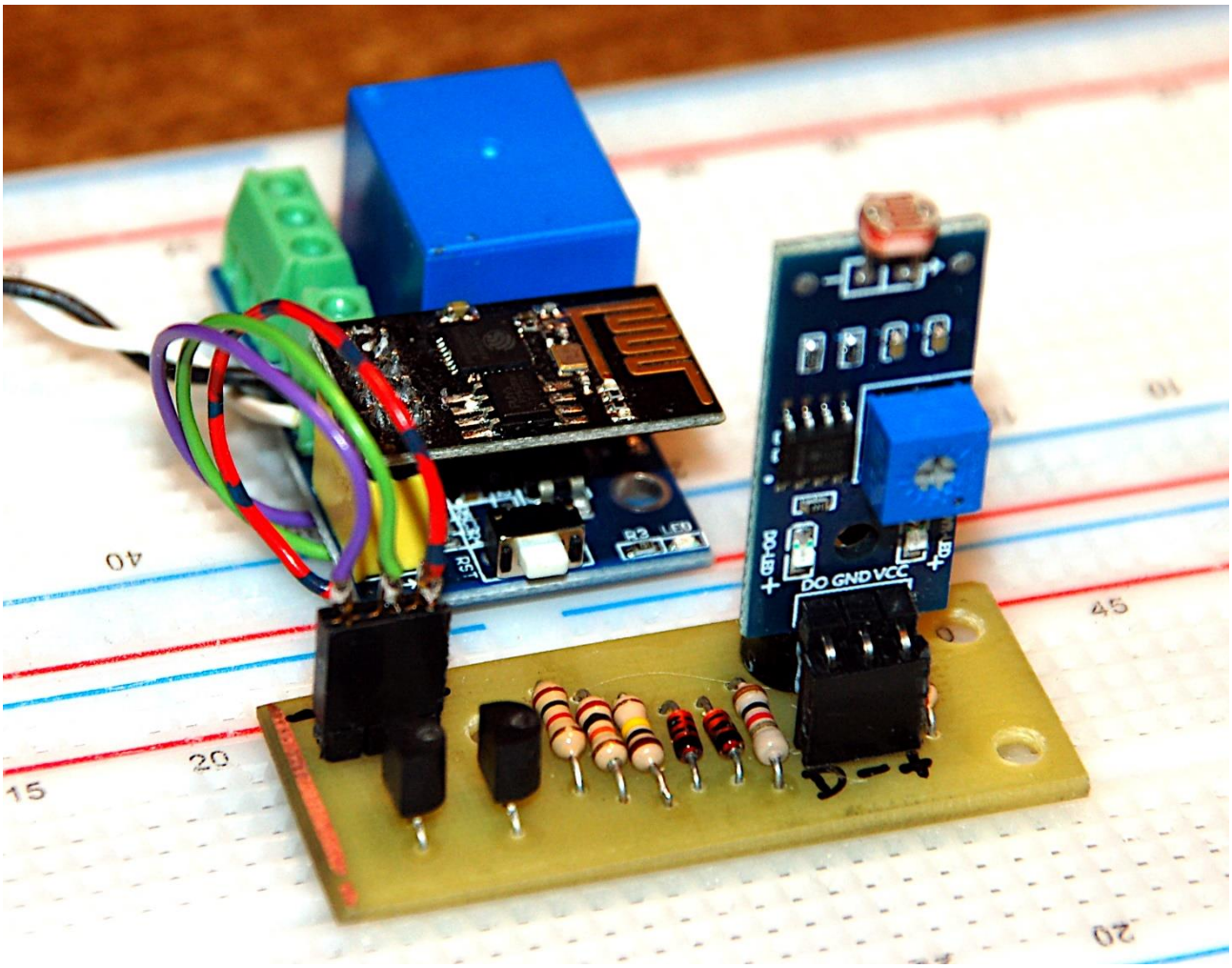


Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

In einem früheren Beitrag hatten wir den ESP32 und einige seiner ESP8266-12F-Brüder für die Sensorabfrage verwendet. Für die Funkverbindung hatte ich einen WLAN-Router in Betrieb. Eine Lösung ohne Router erschien mir mit zu viel Aufwand verbunden und außerdem gab es ein Problem bei der Wiederverwendung von IP-Adressen durch den steuernden Client. Vielleicht hatten Sie den Fehler " OSError: [Errno 98] Address already in use" ja selbst auch schon einmal. Nun, genau dafür habe ich inzwischen eine einfache Lösung gefunden, die ich Ihnen nicht vorenthalten will. Das damit entwickelte Projekt stellt **bis zu 9 Funkschalter** zur Verfügung, die, **ohne einen WLAN-Router als Zwischenglied**, direkt von einem ESP32 angesteuert und abgefragt werden. Der ESP32 bekommt seine Anweisungen über ein **SIM808 als SMS-Nachrichten**. Auf dem gleichen Weg erhält der Auftraggeber eine Nachricht über den Ausgang der Operation. Als minimalistische Lösung für die Schalteinheiten habe ich Relaismodule mit einem ESP8266-01 vorgesehen. Aber bevor wir ins Eingemachte gehen, erst einmal herzlich willkommen zum Projekt.

Attendance Guy mit ESP32, ESP8266-01 und SIM808

Auch das LCD-Keypad aus den früheren Folgen kommt wieder zum Einsatz, hauptsächlich das Display. Natürlich erfüllt die RST-Taste erneut die Funktion einer Notbremse.

Notbremse benutzen heißt, punktgenau das laufende Programm beenden, ohne sofortigen Neustart. Alle, bis dahin erstellten Objekte, Variableninhalte und Funktionsdefinitionen bleiben für den manuellen Zugriff über REPL erhalten. Auf diese Weise lassen sich zum Beispiel Funktionen testen, ohne vorher einen ganzen Rattenschwanz an Imports etc. jedes Mal neu eingeben zu müssen. Dass über die REPL-Kommandozeile solche Tests einfach durchgeführt werden können, ist ein entscheidender Vorteil von MicroPython.

Hardware – etwas Zuwachs

Hier ist die Liste der Zutaten für den Anwesenheits-Simulator (aka Attendance Guy). Vermutlich haben Sie bereits die "schweren" Baugruppen, wenn Sie in den vorangegangenen Folgen schon fleißig mitgebaut haben.

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

Die Schaltung für den Attendance Guy wird zum großen Teil von Folge 2 übernommen. Neu sind die WIFI-Module mit Relais-Adapter, die Lichtsensormodule LDR5528, der USB-zu-ESP8266-01-Adapter und die diversen Kleinteile. Natürlich werden auch Bauteile aus vorangegangenen Folgen wieder mit verwendet. Die Anzahl 9 möglicher Relaisstufen

sollte in den meisten Fällen ausreichen, kann aber erhöht werden, wenn das Programm entsprechend angepasst wird. Das Projekt ist in dieser Richtung fast frei skalierbar. Eine absolute Begrenzung stellen die verfügbaren IP-Adressen im verwendeten Teilnetz dar.

Zur Energieversorgung der Schaltungen sollten Sie folgendes beachten:

- Eine sichere Spannungsversorgung kann mit 5V-Steckernetzteilen erfolgen. Das ist für Langzeitbetrieb zuverlässiger als eine Batterieversorgung. Alternativ kann auch ein 220VAC zu 5VDC Converter verwendet werden.
- Auf dem Relaisgehäuse sind keine elektrischen Daten ersichtlich. Das Datenblatt gibt 5A, 30VDC/50VAC an. Sie sollten diese Module grundsätzlich nicht zum Schalten höherer Spannungen verwenden, auch oder gerade im Hinblick auf die Leiterbahnführung auf der Relaisplatine. Die Abstände zwischen der zu schaltenden Spannung und dem 5V-VPegel des Steuerteils betragen stellenweise nur knapp einen Millimeter. Es besteht trotz Optokoppler keine galvanische Trennung zwischen der Eingangsstufe und dem Relais-Treiber, weil die Signalmasse durchverbunden ist. **Schalten Sie höhere Spannungen als 50V nur dann mit dem Relais, wenn Sie ganz genau wissen, was Sie tun**!
- Je nach dem zu schaltenden "Verbraucher" ist ein weiteres geeignetes Netzteil zu verwenden. Gegebenenfalls kann daraus auch die 5VDC-Versorgung durch geeignete Regler hergeleitet werden.

Die Software

Verwendete Software:
Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:
Micropython Firmware
Bitte eine Stable-Version aussuchen

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 und Infos zu MicroPython

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

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Bei der Entwicklung der Software für diesen Blog habe ich von dem direkten Dialog zum ESP32 wieder reichlich Gebrauch gemacht. Das Spektrum reicht von einfachen Tests der Syntax und der Hardware bis zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Sie bilden eine Art Macro, weil sie wiederkehrende Befehle zusammenfassen. Aus einem solchen Programm hat sich das Programm **attendance_guy.py** entwickelt, das wir in Kürze besprechen werden. Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP32/ESP8266-01 hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

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

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat.

Das Innenleben von Attendance Guy

Für das Projekt Attendance Guy sind zwei getrennte Programme vonnöten, attendance_guy.py und server.py. Das erste gehört auf den ESP32, der die ESP8266-01-Satelliten steuert. Auf diesen Relaiseinheiten muss server.py installiert werden. Beides setzt voraus, dass die Controller vorher mit der MicroPython-Firmware geflasht wurden.

Die Hardware

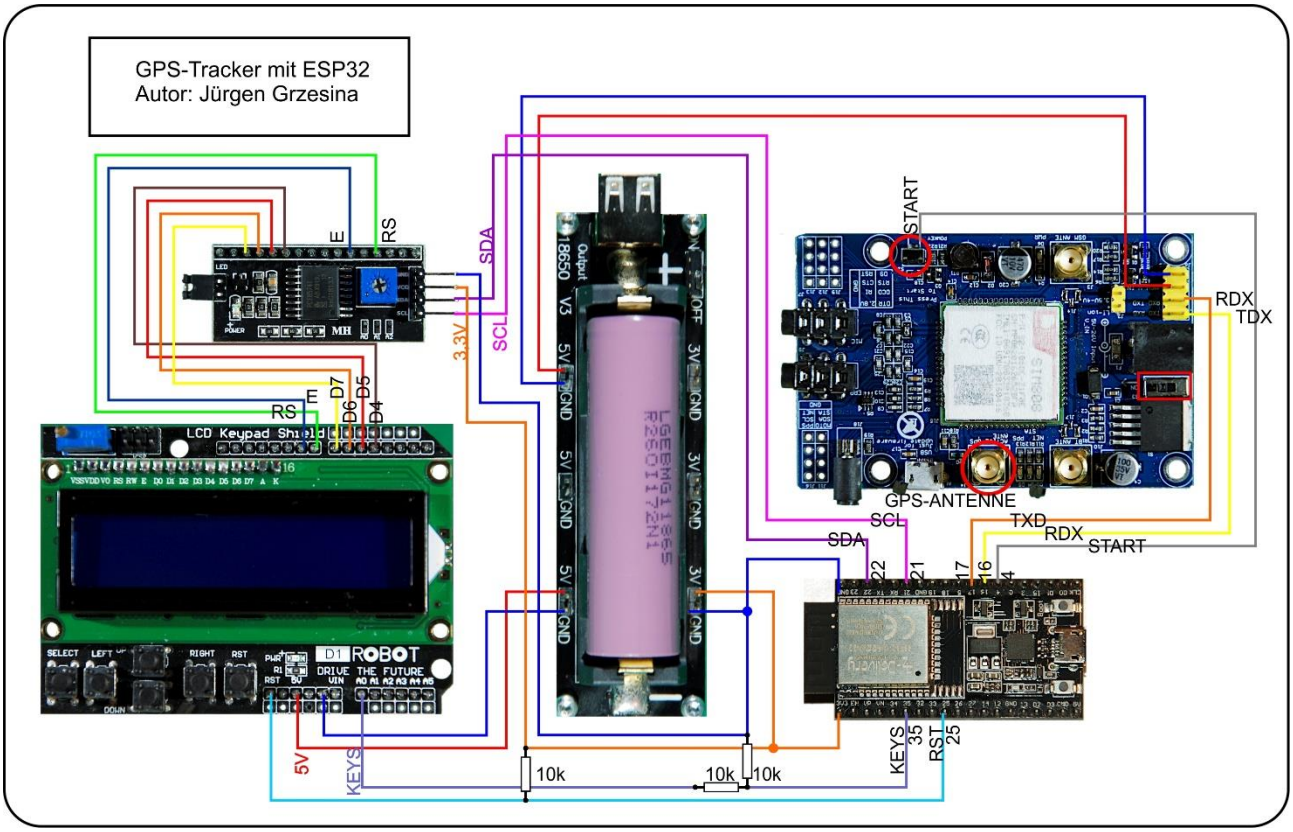
Zwei Programme – zwei Hardwaretypen, der ESP32 übernimmt die Kommunikation zum SIM808, empfängt die SMS-Nachrichten, prüft diese, gibt sie an die ESP8266-01-Einheiten weiter und empfängt von diesen die Ergebnismeldungen, um sie schließlich an den Auftraggeber als SMS-Nachricht zurückzusenden.

Auf den ESP8266-01-Relaisstufen läuft ein UDP-Server. Er nimmt die Aufträge vom ESP32 entgegen, filtert diese selbst noch einmal, führt den Auftrag aus und meldet das Ergebnis via UDP an den ESP32, der dann entscheidet, ob die Nachricht via SMS weitergegeben werden soll.

Den Aufgaben entsprechend sind die Einheiten mit Hardware ausgestattet. Am ESP32 hängt am UART-Port2 das SIM808 und über der I2C-Paralleladapter das Display des LCD-Keypads. Die 4x4-Tastaturmatrix und BMP280 werden in diesem Blog nicht gebraucht, können aber angeschlossen bleiben.

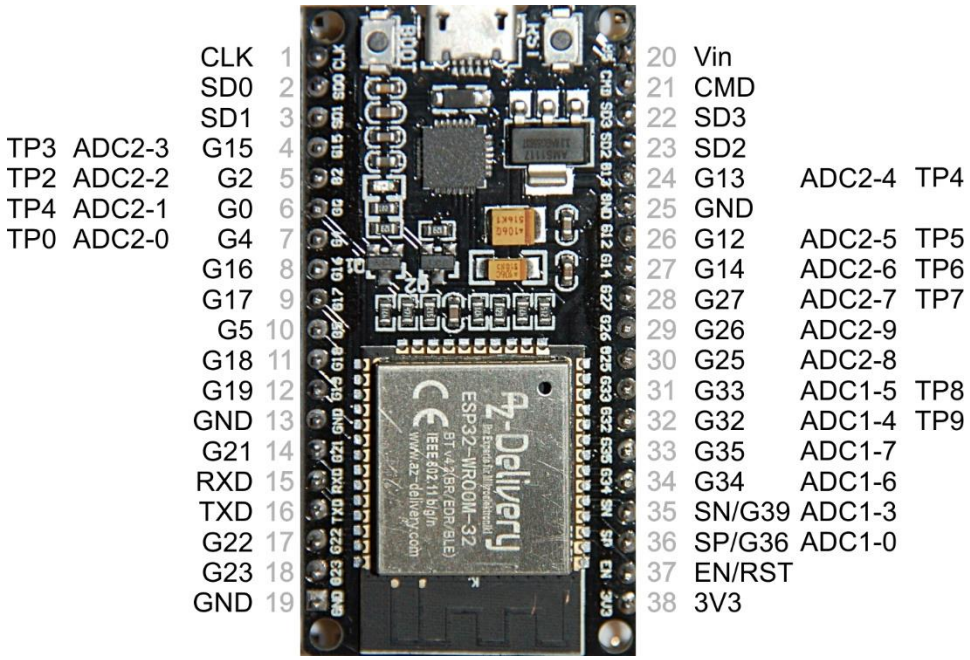
Die Steuereinheit

Vor der Programmierung steht aber erst einmal der Aufbau der Schaltung an.
Der Schaltplan der ESP32-Steuereinheit zeigt die notwendigen Verbindungen. Der Batterieadapter wird sinnvollerweise durch eine 5V / 3,3V Spannungsversorgung ersetzt.



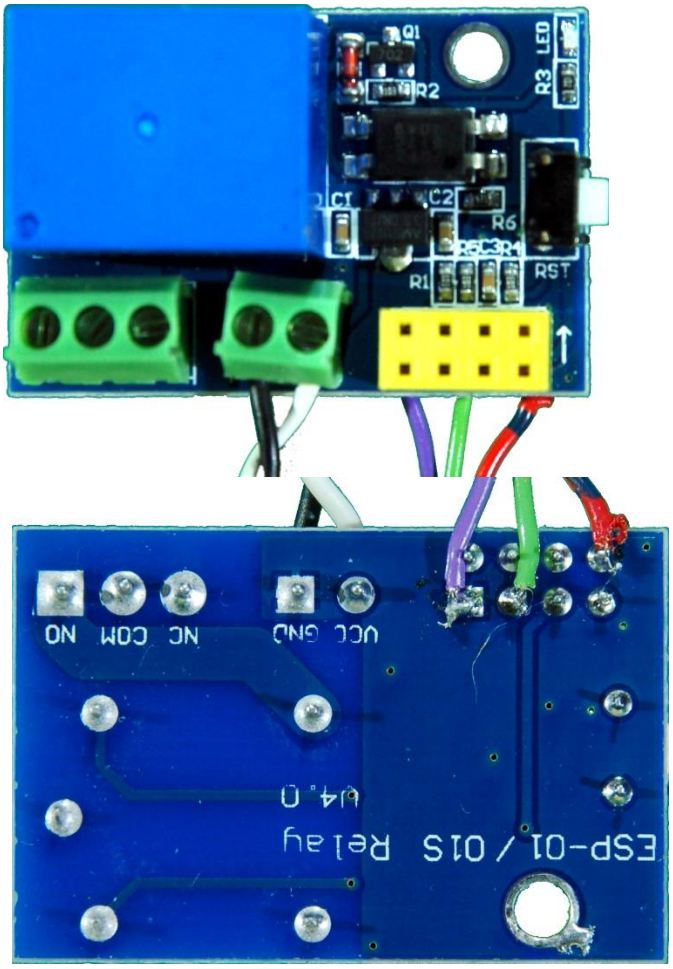
Ein besser lesbares Exemplar der Darstellung in DIN A4 bekommen Sie mit dem Download der PDF-Datei .

Wenn Sie den Batteriehalter durch ein Netzteil ersetzen, müssen Sie unbedingt darauf achten, dass der ESP32 und der Seriell-Parallel-Adapter wie hier gezeigt mit 3,3V versorgt werden müssen. Display und SIM808 brauchen 5V-Versorgungsspannung. Auch die ESP8266-01-Relaisstufen brauchen 5V. 5V können Sie auch am ESP32 anschließen, aber nur am Pin Vin (Pin 20) oder 5V, niemals am 3,3V-Pin! Für alle Anschlüsse des ESP32 gilt 3,3V maximal!

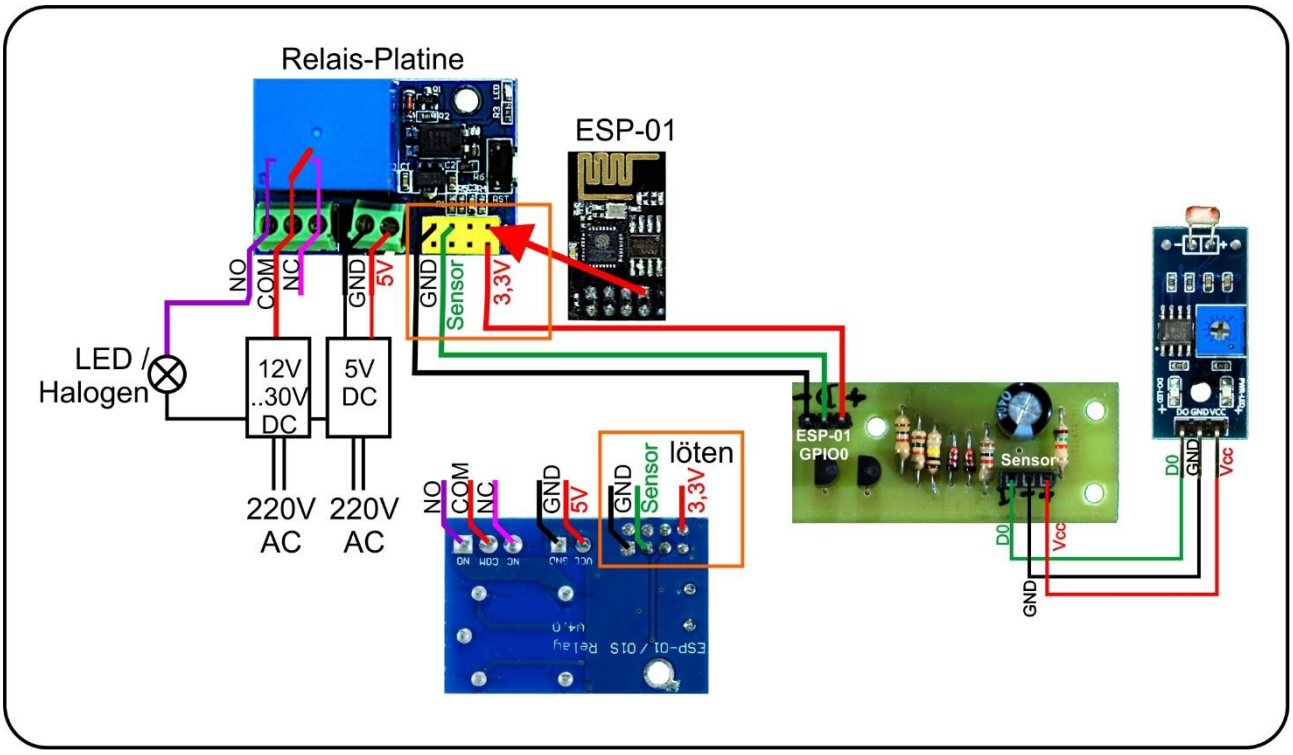


Das LCD-Keypad habe ich wieder im Einsatz, weil das Display gerade im autonomen Betrieb des ESP32 gute Dienste leistet. Statusberichte, Fehlermeldungen, Aufforderung zum Tastendruck, etc. lassen sich gut darüber kommunizieren. Die Ansteuerung über den Seriell-Parallel-Adapter erledigt auf einfache Weise die Anpassung der 3,3V-Ausgänge des ESP32 an die 5V-Pullup--Anschlüsse des Displays.

ESP8266-Relais-Modul



Indem der ESP8266 auf die Relaisplatine gesteckt wird, sind damit bereits fast alle Verbindungen hergestellt. Der Schaltplan zeigt ferner die Zuführung der 5V Versorgungsspannung und den Anschluss einer 12V-Beleuchtung. Die 5V können optional aus der 12V-Quelle durch den Regler 7805 gewonnen werden. Diese Teile sind nicht in der obigen Hardwareliste enthalten. Das Relaismodul darf nicht mit mehr als 5V betrieben werden, weil dadurch die Relaisspule Schaden leiden würde.



Die Ansteuerung des Schalttransistors über einen Optokoppler ist so geregelt, dass ein HIGH-Pegel am GPIO0 des ESP8266 die Relaisspule stromlos macht und ein LOW-Pegel dazu führt, dass das Relais anzieht. Ein Relais, das sich so verhält nennt man "LOW-Level getriggert". Im ersten Fall ist der Kontakt zwischen COM (common aka Schaltkontakt) und NC (normally closed aka Ruhekontakt) geschlossen, im zweiten Fall liegt com an NO (normally open aka Arbeitskontakt). Unser Programm muss nur dafür sorgen, dass sich der GPIO0-Pin genauso verhält und beim Booten der ESP8266-01-Einheit ein HIGH-Pegel an dem Anschluss anliegt. Den Rest übernimmt dann die Relaisplatine.

So viel zur normalen, vom Hersteller vorgesehenen Funktion. Wir wollen aber mehr! Wir wollen sehen, ob der Schaltvorgang erfolgreich war, dazu brauchen wir eine Rückmeldung. Wenn wir Lampen einschalten, dann können wir zur Kontrolle einen LDR (**L**ight **D**ependent **R**esistor aka lichtabhängiger Widerstand) einsetzen. Der verringert seinen Wert, wenn Licht darauf fällt, im Dunklen ist er hochohmig. Nun kann der ESP8266-01 leider weder Spannungen und schon gar keine Widerstandswerte messen.

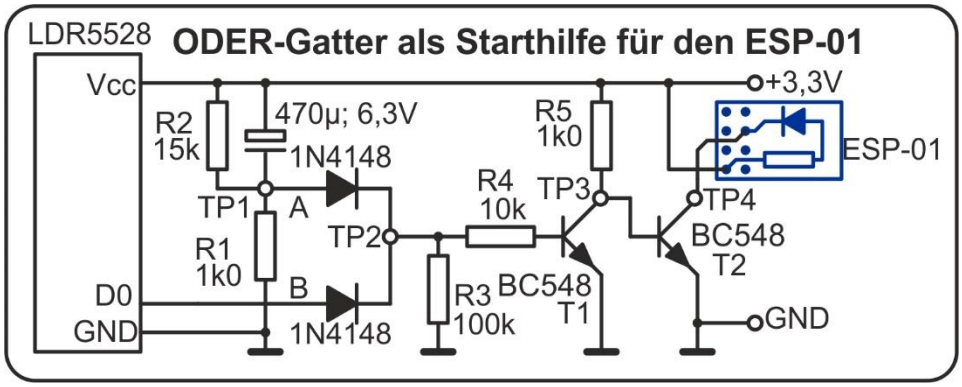
Macht nichts, das Licht-Sensor-Modul-LDR5528 erledigt das für uns. An dessen Ausgang steht ein digitales Signal bereit, dessen Zustand von der Umgebungshelligkeit und der Stellung des Potentiometer-Trimmers auf dem Board abhängt. Bei Helligkeit liegt der Ausgang D0 auf LOW (0), bei Dunkelheit auf HIGH (1). Die Schwelle für das Umschalten wird durch den Trimmer eingestellt. Gut Problem gelöst! - Denkste!

Jetzt kommt nämlich ein Umstand zum Tragen, der mit den Booteigenschaften des ESP8266 allgemein und denen des ESP8266-01 im Besonderen zu tun hat. Der Controller des ESP8266-01 ist eben ein ESP8266. Bei diesem Controller müssen für einen normalen Start, nach dem Einschalten oder nach einem Reset, die Leitungen GPIO0, GPIO1 und GPIO2 auf HIGH-Potenzial liegen. GPIO1 ist die TDX-Leitung, die zum Datentransfer zwischen PC (USB) und Controller gebraucht wird und scheidet für unsere Zwecke somit von vorneherein aus. GPIO0 versorgt die Relaissteuerung. Die Leitung wird vom

ESP8266-01 über einen Pullupwiderstand beim Start auf HIGH gezogen, das Relais ist dadurch nach dem Start stets aus.

Nur wissen wir leider nicht, welche Umgebungshelligkeit gerade herrscht, wenn die Relaiseinheit bootet. Schließen wir nämlich den D0-Ausgang des LDR5528-Moduls an GPIO2 an, dann wird der ESP8266-01 nicht starten, wenn es in diesem Moment hell ist, weil der Pegel dann auf LOW liegt. Also können wir das ESP8266-01-Modul für unseren Zweck vergessen?

Geht nicht! – Gibt's nicht! So lautet der Werbeslogan eines bekannten Baumarkts. Und so ist es auch hier. Was nicht geht, wird gehend gemacht! Zwischen GPIO2 und +3,3V ist über einen Widerstand die blaue LED des Moduls angeschlossen und die beiden ziehen den Anschluss nach Vcc. Wir müssen jetzt nur während der kurzen Startphase verhindern, dass der potenzielle 0-Pegel des LDR5528 an den GPIO2 gelangt. Genau das macht folgende Zusatzschaltung, die im Grunde eine ODER-Schaltung mit nachfolgendem Open-Kollektor-Treiber darstellt.



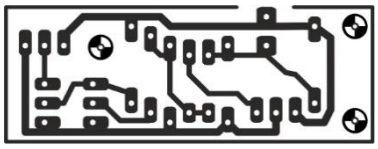
Beim Einschalten wird der Elektrolytkondensator über den Widerstand R1 aufgeladen. Die Spannung am Punkt TP1 fällt von 3,3V innerhalb von einigen Zehntelsekunden auf einen Wert unterhalb von 0,7V am Testpunkt T2. Innerhalb dieser Zeitspanne liegt TP2 also auf HIGH, egal welcher Pegel am Eingang B von D0 her anliegt. Ist der Elko auf mehr als 1,9V aufgeladen, erscheint der Eingang A von TP1 her als LOW, und der Pegel an B entscheidet allein über die Spannung an TP2. Der Transistor T1 puffert das hochohmige Signal und negiert es leider auch. Deshalb setzen wir einen zweiten BC548 o. ä. ein. Der Kollektor dieses Transistors liegt an GPIO2, der intern über einen Widerstand und die blaue LED an Vcc liegt. Ist die Basis von T2 auf 0V, sperrt T2 und GPIO2 liegt auf HIGH-Pegel. Liegen an der Basis mehr als 0,7 V, schaltet der Transistor durch und GPIO2 wird gegen GND gezogen.

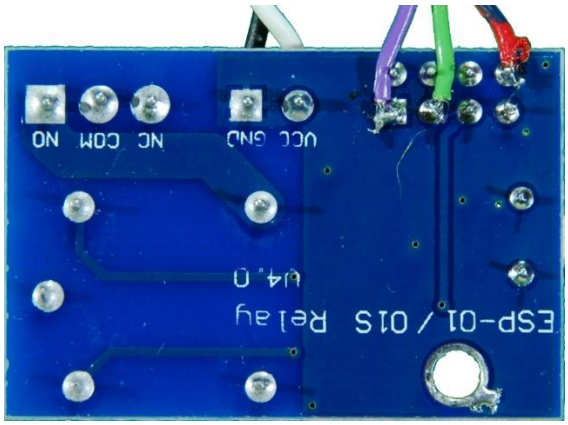
Die Tabelle gibt einen Überblick über die Pegel an den verschiedenen Testpunkten. Für die angegebenen Werte wird eine Vorwärtsspannung der Dioden von typisch 0,7V vorausgesetzt.

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	

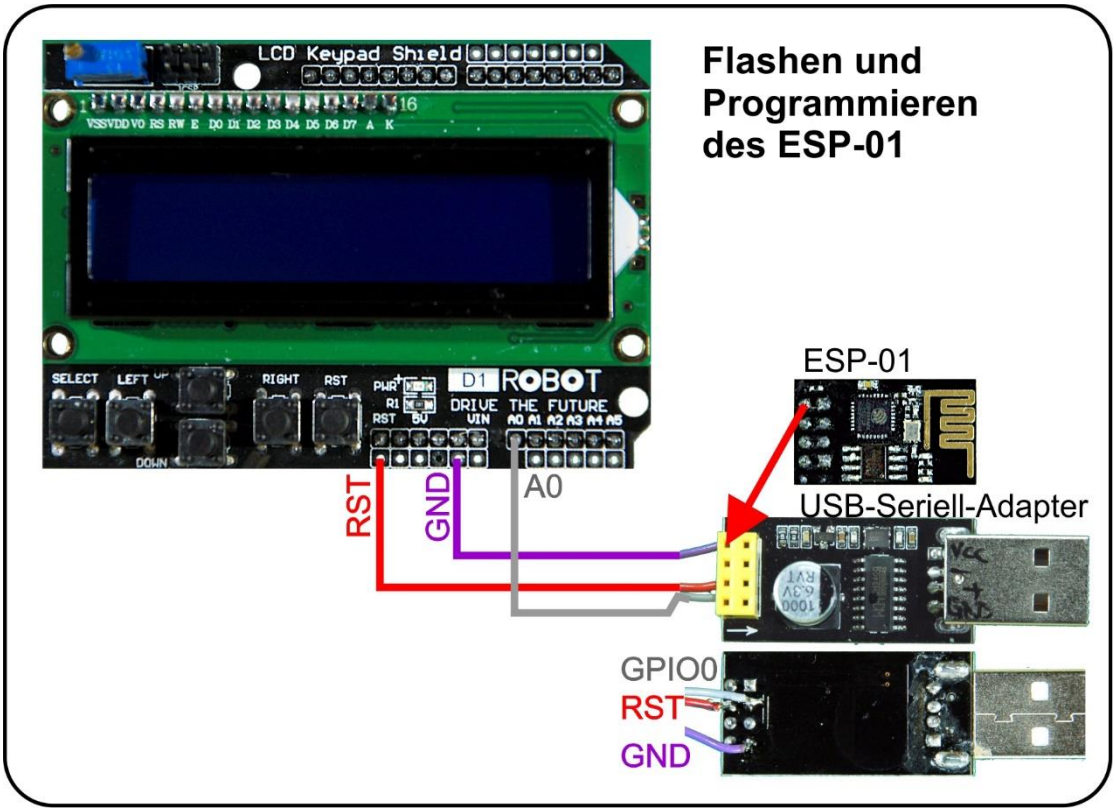
d.c. = don't care

Die Schaltung kann auf einem Breadboard oder einer Lochrasterplatine aufgebaut werden. Für Profis gibt es ein Platinenlayout zum Download als PDF-Datei. Der Anschluss erfolgt auf der Unterseite der Relaisplatine durch Anlöten von kurzen Drahtverbindungen (violett = GND, rot=Vcc, grün=TP4).

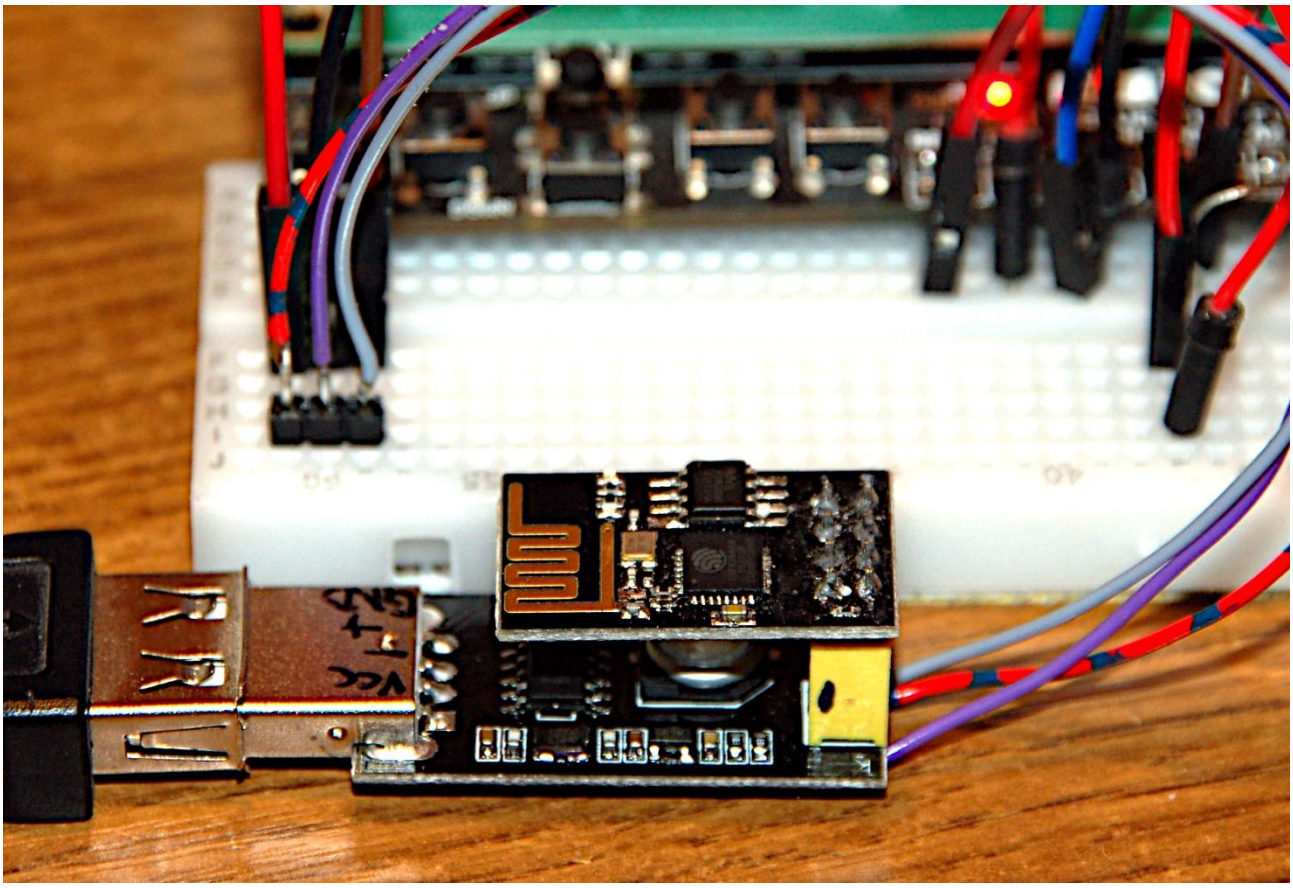




Abschließend zur Vorbereitung müssen wir die ESP8266-01-Module noch mit der MicroPython-Firmware flashen. Hierfür verwende ich einen USB-TTL-Adapter mit einem Steckplatz für die Controllerplatine des ESP8266-01.

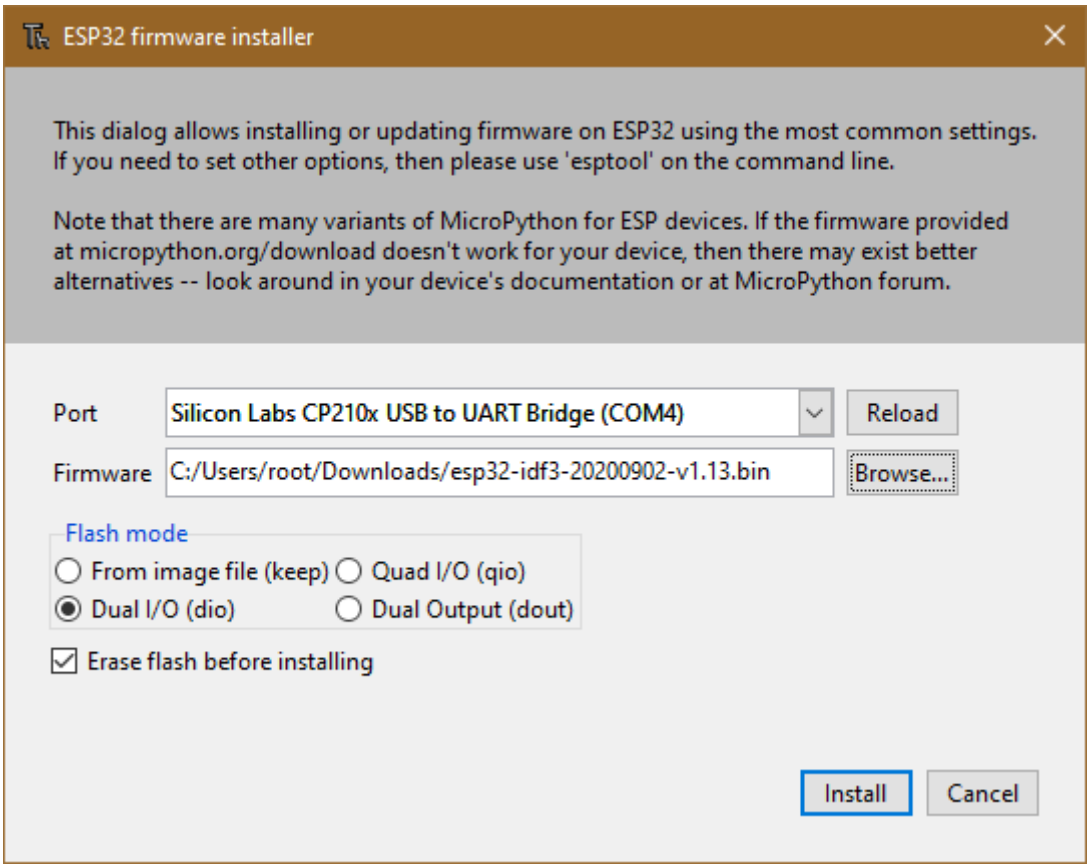


In der Abbildung sehen Sie die drei Leitungen, die ich zusätzlich angebracht habe und die das Vorgehen deutlich erleichtern. Ich verbinde RST, GPIO0 und GND mit einem dreipoligen Stück Stiftleiste. Mittels Jumperkabeln lege ich dann den RST-Anschluss an den gleichnamigen Pin auf dem LCD-Keypad, die GPIO0-Leitung kommt an den A0-Anschluss. Die Taste RIGHT liefert somit als Flash-Taste den Schluss auf GND. Natürlich muss GND mit GND verbunden werden. Nun kann ich die Pegelfolge, welche die Flashautomatik auf dem ESP32 programmgesteuert vorlegt, manuell eingeben, RST + FLASH hold, RST release + FLASH hold, FLASH release. Denken Sie bitte daran, dass Sie den Vorgang zweimal durchführen müssen, einmal zum Löschen des Flashinhalts und ein zweites Mal zum Flashen der Firmware.



Das Procedere zum Brennen der Firmware ist ansonsten beim ESP32 und ESP8266-01 identisch. Laden Sie dazu die beiden neuesten bin-Dateien herunter:
esp8266-1m-20210418-v1.15.bin
esp32-idf4-20210202-v1.14.bin

In Thonny öffnen Sie über **Run - Select interpreter** das Fenster **Thonny options**. Rechts unten öffnet ein Linksklick auf **Install or update firmware** den **Firmware installer**.



Stellen Sie den Port ein und wählen Sie die entsprechende Firmwaredatei. Starten Sie nun den Brennprozess und denken Sie daran, beim ESP8266-01 die Tasten zu drücken.

Die Software

Beide Softwareteile müssen neben der Bedienung der UDP-Verbindung eine Reihe anderer Aufgaben erfüllen. In besonderer Weise trifft das für das Programm auf dem ESP32 zu, der den SMS Traffic zu managen hat. Beide Teile sind des Weiteren mit dem Parsen der Nachrichten und dem Zusammensetzen und Weitergeben der Ergebnisse beschäftigt.

Damit beide Teile nicht in der Empfangsschleife kleben bleiben, habe ich bei der Erstellung des UDP-Sockets ein Timeout von 2 Sekunden eingebaut. Das ermöglicht dem Controller, nach Ablauf dieser Zeit andere Aufgaben in der Hauptschleife zu erledigen.

Der ESP8266-01-UDP-Server

Beginnen wir mit dem einfacheren Programm für den ESP8266-01. Jede der ESP8266-01-Einheiten stellt einen UDP-Server in einem freien Teilnetz des 10.0.0.0-er Bereichs dar. Weil niemand 16,7Millionen Schaltstellen braucht, habe ich das Ganze auf das Teilnetz 10.0.2.0 zusammengeschrumpft. Von den verbleibenden 252 Adressen nutze ich ganze 9, von 10.0.2.101/24 bis 10.0.2.109/24. Der ESP32 als Verwalter hat zusätzlich die 10.0.2.199/24. Die Netzwerkmaske von 255.255.255.0 leitet sich aus dem Suffix /24 her. Das heißt 24 Bit, also die ersten 3 Bytes der IP, stellen den Netzwerkanteil der IP-Adresse dar, das letzte Byte die Geräteadresse. Die Portnummer der Server ist 9000. In einem

WLAN mit Router könnte man alle Serverstationen daher auch via Broadcast ansprechen, um zum Beispiel deren Eigenschaften gebündelt abzufragen.

Wir gehen aber einen anderen Weg, der ohne einen WLAN-Router auskommt. Deshalb wird jede ESP8266-01-Einheit als eigener Accesspoint betrieben und von dem zentralen ESP32 bei Bedarf kontaktiert.

Das Programm server.py

Das Programm ist recht übersichtlich. Nach dem Import verschiedener MicroPython-interner Module legen wir ein paar Variablen, Strukturen und Funktionen fest. Eine wichtige Stelle ist die Definition der Nummer der Einheit und damit der IP-Adresse. Dazu ist auch der festbleibende Teil der IP-Adresse nötig.

```
# ****************** stationsnummer festlegen *************
unitNumber=1 # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
teilnetz="10.0.2.10"
verwendung="RELAIS"
# ********************************************************
```

Die **unitNumber** wird einfach als String an den Teilnetzstring angehängt. Für eine andere Station muss also nur ein anderer Wert für unitNumber angegeben werden. Das ist schon alles. Kopieren Sie den Programmtext in eine leere, neu angelegte Datei und speichern Sie diese als boot.py in Ihrem Arbeitsverzeichnis ab. Von dort laden Sie die Datei auf einen der geflashten ESP8266-01 hoch. Beim nächsten Reset startet das Programm automatisch.

Der nächste wichtige Punkt im Programm ist die Einrichtung des Accesspoints. Wir gehen das genauer durch, weil dieser Teil in ähnlicher Form auch im Programm attendance_guy.py vorkommt.

#A Wir erzeugen ein Netzwerkinterface-Objekt und
#B aktivieren es
#C Aus der Nummer der Einheit stellen wir die SSID für den Accesspoint her und
#D die IP-Adresse
#E Wir konfigurieren das Interface-Objekt mit den erzeugten Adressdaten. Es ist wichtig, IP-Maske, Gateway- und DNS-Adresse als **Tupel** anzugeben.
#F Als Authentifizierungsmethode wird nur 0 akzeptiert, das bedeutet, der Zugriff auf den Accesspoint geschieht ohne Passwortsicherung.
#G Aus der Konfiguration rufen wir die MAC-Adresse ab, decodieren sie als String und geben sie aus.
#H An das Interface-Objekt weisen wir den Namen zu, das Passwort ist ein leerer String.
#J Gegebenenfalls warten wir auf den Abschluss der Interfaceeinrichtung.

```
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))
```

Mit der Einrichtung eines UDP-Sockets geht es weiter.

```
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 Festlegung der Portnummer für alle Einheiten
#L Wir erzeugen ein IPv4 UDP-Socket -Objekt, UDP basiert auf Datagrammpaketen.
#M Wir binden den Socket an die oben erzeugte IP-Adresse und die Portnummer. IP und Portnummer werden als Tuple übergeben.
#N Das Timeout legen wir auf 2 Sekunden fest. Kommt in dieser Zeit keine Anfrage, dann wird die Eingabeschleife verlassen und es können weitere Befehle abgearbeitet werden.
#P Weil der Server kein Display besitzt, wird durch ein Blinkzeichen die Empfangsbereitschaft signalisiert. Nachdem als LED-Anschluss der Pin GPIO0 definiert wurde, schaltet das Relais die angeschlossene Beleuchtung in der programmierten Pulsfolge.

In der Serverschleife schauen wir zunächst nach, ob eine Anfrage vorliegt. Die Funktion **recvfrom**() wird verlassen, wenn das der Fall ist oder wenn die 2 Sekunden des Timeout abgelaufen sind. Die Rückgabe der Funktion **recvfrom**() ist ein Tuple aus dem Anfragetext

und der Adresse des Absenders. Bei der Zuweisung dröseln wir das Tuple gleich auf die beiden Variablen auf. In MicroPython nennt sich das Entpacken. Bis zu 160 Zeichen werden in einem Schwupp gelesen.

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

Es wird ein Antwortstring vorbereitet. Die Anfrage wird als Bytes-Objekt geliefert und muss zur weiteren Verarbeitung ins Stringformat decodiert werden.

Es folgt das Parsen der Anfrage. Das Format habe ich wie folgt festgelegt.

SET:Wert , Wert darf 0 oder 1 sein
GET:STATUS
BROADCAST

Der Befehl BROADCAST liefert natürlich nur die Eigenschaft des Servers selbst als Antwort ab, weil sich in seinem Teilbereich keine weiteren Stationen befinden. Der Befehl dient in erster Linie dazu, den Einsatzzweck der ESP8266-01-Einheit abzufragen. Die Information dazu wird am Beginn des Programms eingetragen.

# ****************** stationsnummer festlegen *************
unitNumber=2 # <<<<<<<<<<<<<<<<<<<<<<<<<<<<<<<
teilnetz="10.0.2.10"
verwendung="RELAIS"
***

Der ESP32 als Vermittler

Dieser Teil des Projekts ist sehr interessant, was das Handling zusätzlicher Aufgaben in der Serverschleife angeht und vor allem beantwortet das Programm attendance_guy.py die Frage, wie man die Fehlermeldung "OSError: [Errno 98] Address already in use" vermeidet. Bei der Suche im Web bin ich nicht fündig geworden, also blieb mir nichts anderes übrig als; "Jugend forscht". Dabei ist die Lösung so einfach.

Die Konfigurationsdaten für das Netz und den SMS-Betrieb stehen gleich am Beginn.

**
#
phoneNumber="0xxxxxxxxxx"
SSID="here goes your WLAN-SSID"
PASSWD="your password goes here"
#
**
phoneNumber nimmt die Telefonnummer des Handys auf, von dem Steuer-SMS gesendet werden sollen. SSID und PASSWD beziehen sich auf ein WLAN mit Router als Accesspoint. Das ist zu ersten Tests ganz nützlich, wird aber später nicht mehr benötigt. Ähnlich wie bei den ESP8266-01-Einheiten gibt es im Programm eine auskommentierte Sequenz, die die Anmeldung an einem Router-Accesspoint managt.

```
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()
```

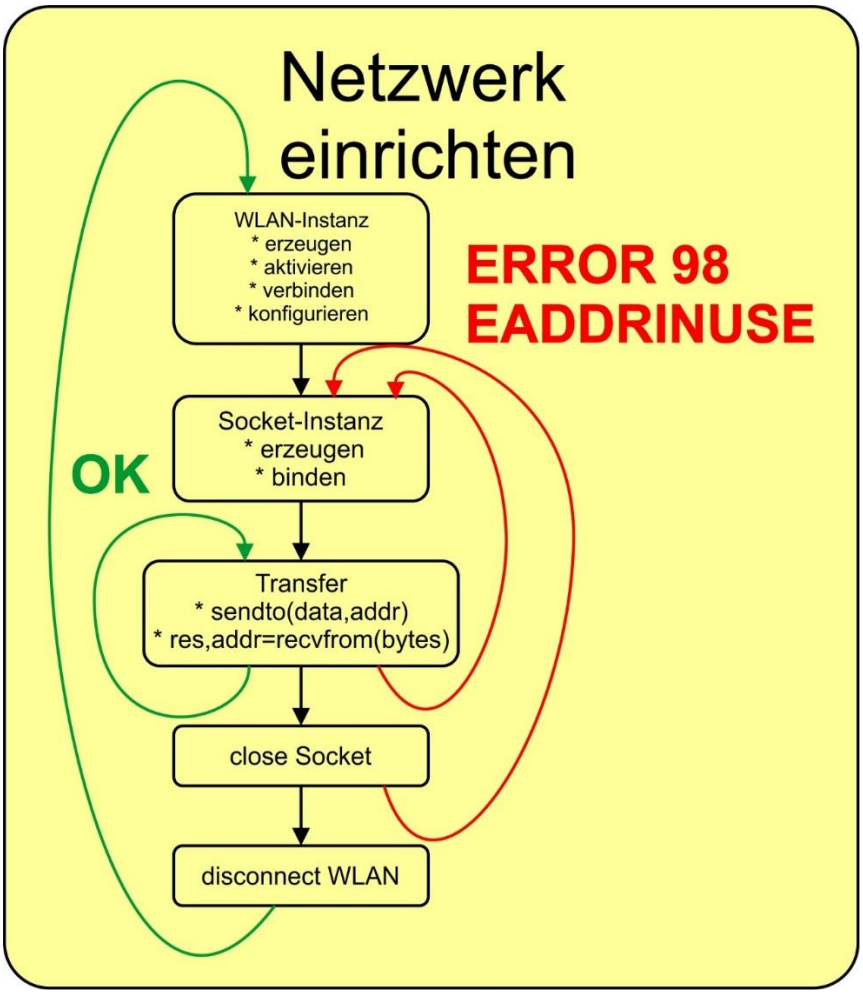
blinkLed ersetzt die sonst übliche LED am Pin GPIO2, die beim ESP32 fehlt. **ctrl** ist der Anschluss der Notbremse, die durch die RST-Taste am LCD-Keypad realisiert wird. Der Empfangspuffer **request** wird für 50 Zeichen schon mal vorab definiert.

Das I2C-Objekt wird für die Definition des Display-Objekts d benutzt, das wiederum in die GSM-Instanz g einfließt, neben dem Ausgang switch und der ctrl-Taste.

Wir initialisieren die Hardware, schalten das Modul ein und deaktivieren danach sofort die Weiterleitung von GPS-Daten, die uns nur den UART-Puffer vollkleistern würden.

Jetzt folgt eine Reihe von Funktionen. Sie bilden zusammen eine Kette von Aktionen, zum Aufbau einer Verbindung und zum Übertragen von Nachrichten. Das Hauptprogramm ist mit seinen 30 Zeilen demgegenüber sehr übersichtlich.

Der UDP-Client muss, anders als in den bisherigen Projekten, zu jedem Accesspoint zuerst eine Netzwerkverbindung herstellen, bevor per UDP Daten ausgetauscht werden können. Das ist der Kernpunkt dieser Anwendung. Denn genau dadurch wurde es möglich, dieselbe IP-Adresse für den Client immer wieder zu verwenden. Noch einmal ganz deutlich: Es funktioniert nicht, bei bestehender WLAN-Verbindung nur einen Socket zu schließen und beim erneuten Öffnen dieselbe IP wiederzuverwenden, sondern man muss die komplette Netzwerkverbindung einstampfen und neu herstellen. Die folgende Grafik zeigt die Möglichkeiten für das Vorgehen auf.



Ein Socket kann als Tor zum Datenaustausch verstanden werden, wie ein Dateihandle auf ein Medium, also externe Festplatte, USB-Stick, Drucker etc. Für die Verbindung sind zwei Dinge nötig, eine Leitung und die Schnittstellen Software. Für eine WLAN-Verbindung verhält es sich ähnlich. Sie brauchen eine Funkstrecke, das ist die Verbindung Client - Accesspoint und eine Schnittstellensoftware, das ist hier der UDP-Socket. Um eine neue Verbindung aufbauen zu können, muss die vorherige eingestampft werden, und dazu gehören das Schließen des Sockets und dann das Abbauen der Funkstrecke. Erst dann hat MicroPython nämlich vergessen, welche IP-Adresse zuvor in Benutzung war. Ganz einfach, oder?

Weil der Aufbau der Funkstrecke wiederholt ausgeführt werden muss, habe ich diesen Teil als Funktion **connectToLocalAP**(unit) eingerichtet. Der Parameter unit ist die Nummer der ESP8266-01-Station, die ich ansprechen möchte.

Die Festlegung der Verbindungsdaten sieht beim ESP32-Client etwas anders aus.

```
# ********** Stationsdaten festlegen **********
Teilnetz="10.0.2."
Versuche=20
Stationen=[ # [IP, tool, present, status]
   ["10","PC",False,""],
   ["101","Relais",False,""],
   ["102","Relais",False,""],
   ["103","TVsim",False,""],
   ]
```

Teilnetz enthält hier wirklich nur den Netzwerkanteil. Versuche gibt an, wie oft die Schleife für den Verbindungsaufbau durchlaufen werden darf. Stationen ist eine Liste von Listen, mit Angaben zur Geräte Adresse, der Aufgabe und der Verfügbarkeitsinformation der Satelliteneinheit. Die Vorgehensweise in der Funktion ist im Wesentlichen die gleiche wie bei den ESP8266-01-Einheiten.

connectToLocalAP gibt ein Tupel zurück, welches das WLAN-Objekt und als Tupel die eigene sowie die Target-Adresse samt Portnummer enthält.

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

Die Zieladresse wird als Eingabeparameter der Funktion **openSocket**() verwendet. Erwähnenswert, weil wichtig, ist der Parameter **socket.SO_REUSEADDR** bei der Erzeugung der Socketinstanz. Dieser Parameter **und** die oben beschriebene Vorgehensweise zum Öffnen und **Schließen** einer Verbindung ermöglichen es uns, die gleiche IP-Adresse für den Client immer wieder zu verwenden. Der gewählte Funktionsname soll an den Befehl zum Öffnen einer Datei erinnern. Im Grunde ist das auch nichts anderes.

```
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() gibt das Socketobjekt s zurück, das für den Datentransfer noch gebraucht wird. Diesen Teil erledigt die Funktion **xmitJob**(). Sie nimmt das Socketobjekt, die zu sendende Nachricht und die Zieladresse als Positionsparameter.

```
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**() wartet nach dem Senden bis zu 5 Sekunden auf eine Antwort und decodiert diese gegebenenfalls. Der (evtl. leere) Antwortstring r wird zurückgegeben.

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

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

closeSocket() und **killConnection**() sorgen für einen ordentlichen Abbau der Verbindung.

Angekommene SMS-Nachrichten müssen geparst werden, um deren Befehlsinhalt heraus zu kitzeln. Eine SMS kann mehrere Aufträge enthalten, die einer bestimmten Syntax folgen müssen. Statt einer allgemeinen Beschreibung gebe ich einfach ein Beispiel an.

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

Groß-Kleinschreibung spielt keine Rolle, wohl aber die Doppelpunkte ":" und Kommata.

Die Antwort am Handy sollte dann etwa so aussehen:

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

Beim Parsen der Nachricht wird auf den Status "REC READ" und auf die zulässige Telefonnummer geprüft. Stimmt die Anrufernummer nicht oder ist die SMS bereits bearbeitet, wird sie sofort gelöscht. Jeder der drei oben angeführten Jobs wird als Eintrag in die jobListe gepackt, ungültige oder verstümmelte Angaben werden übergangen. Doppelpunkte und Kommata dienen als Orientierung für den Parser, der mit den Stringmethoden **split**() und **find**() arbeitet. Jeder Eintrag in jobListe ist selbst eine Liste und enthält die Nummer der anzusprechenden Einheit, den Job als String und den zugewiesenen Wert als Zahl. Die jobListe wird zurückgegeben.

```
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 1970 Mary Hopkin. Unser Programm singt nicht, aber es schaut nach, wer denn so alles da ist, in Reichweite. Beim Durchgang durch die Liste **Stationen** wird beim gegenwärtigen Ausbaustand die Reaktion der Satellitenstationen als Anlass genommen, den Eintrag für present von False auf True zu ändern. Nur diese Stationen werden später bei der Übermittlung von Befehlen berücksichtigt.

```
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
```

Die Funktion, die die Jobliste abarbeitet ist **doJobs**(). Sie nimmt die Liste **jobListe** von **readSMS**(), setzt aus den Feldern der Einträge die Befehle für die ESP8266-01-Einheiten zusammen und schickt sie ab, nachdem die Verbindung zum Accesspoint steht und der Socket etabliert ist. Die Antworten der einzelnen Jobs werden im String **response** zusammengetragen und als Gesamtergebnis zurückgegeben. Socket und WLAN-Verbindung werden geschlossen.

```
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
```

Damit sind wir auch schon fast mit der Besprechung des Programms fertig, fehlt nur noch die Hauptschleife. Beim Programmstart löschen wir alle alten SMS-Nachrichten und schauen uns nach Zielen um. Gefundene Stationen lassen wir uns melden, dann putzen wir den UART-Buffer.

In der Schleife prüfen wir auf ungelesene, also frische SMS-Nachrichten. Jede der Meldungen wird zerlegt, daraus eine Jobliste erstellt und diese dann Zeile für Zeile verarbeitet. Jeder Job liefert eine Antwortliste, die per SMS zurück an den Auftraggeber geht. Dank der Funktionen, die die eigentliche Arbeit erledigen, liest sich das Hauptprogramm sehr leicht. Den Schluss der Schleife bildet die Notbremse.

```
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()
```

Für eine Stelle im Listing muss ich Sie noch sensibilisieren. Die hat mich lange gefoppt. die 5 Sekunden Pause nach dem Versenden der Nachricht entscheiden darüber, ob die eben bearbeitete Nachricht durch gsmDeleteSMS() auch wirklich gelöscht wird. Kommt der Befehl zu bald, dann bleibt die Nachricht erhalten und wird in Folge immer wieder ausgeführt. Das wollen Sie sicher nicht, vor allem deshalb, weil Sie mit SMS-Nachrichten von Ihrem Attendance Guy bombardiert werden.

Sind Sie jetzt bereit, formschöne Gehäuse für die Einheiten zu bauen oder die Programme gemäß Ihren Vorstellungen zu verändern oder zu erweitern? Dann wünsche ich Ihnen mindestens so viel Spaß dabei, wie ich bei der Entwicklung der Anwendung hatte. Die verwendeten Module und die beiden vollständigen Programme habe ich eingangs verlinkt, damit Sie gleich loslegen können.

PDF in deutsch

PDF in english