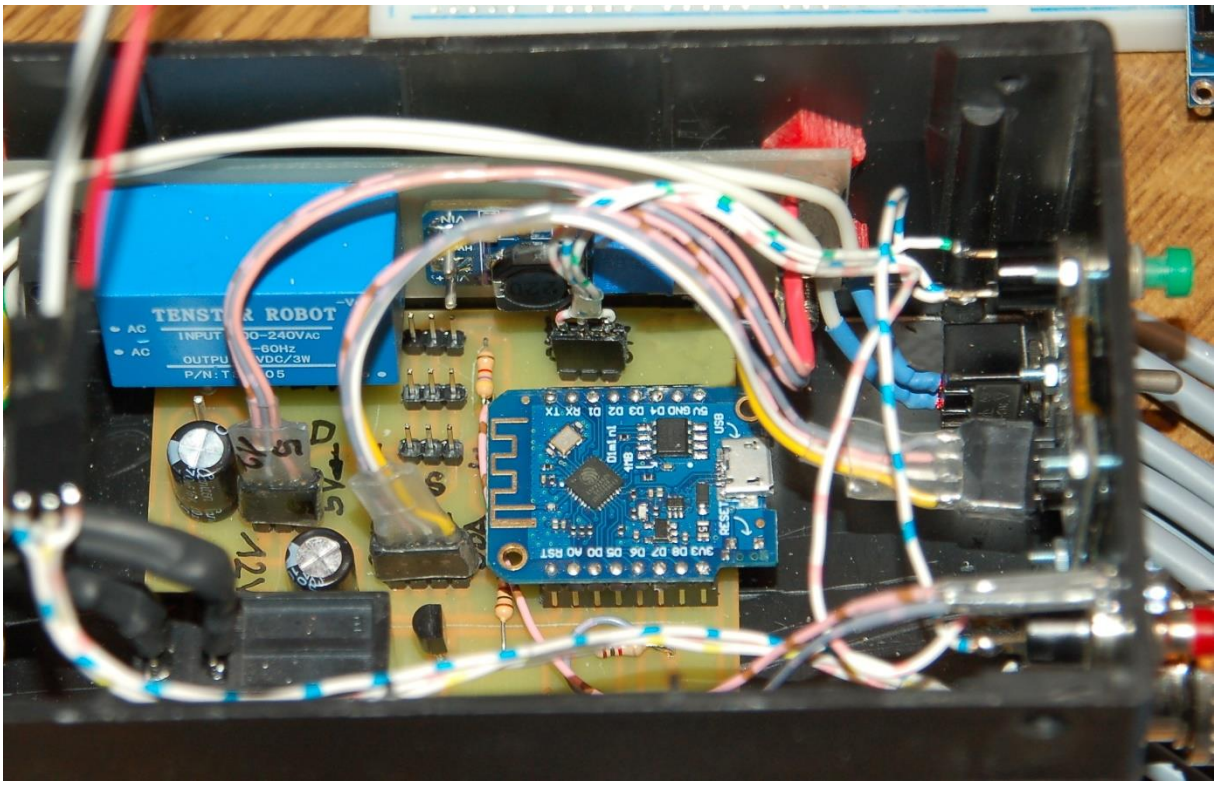

Abbildung 1: Innenleben

Diesen Beitrag gibt es auch als PDF-Dokument in Deutsch und Englisch.

Kennen Sie auch Sätze wie: "Schaust du noch im Treibhaus nach der Temperatur?" oder "Sollten wir nicht doch die Heizung im Treibhaus anmachen? Heute Nacht soll Frost kommen!". Bei uns gehören diese Fragen in Zukunft der Vergangenheit an. Denn ab sofort wacht mein

WLAN-Gärtner

über die Temperaturen im Treibhaus und drum herum. Zusammen mit meinem Linux-Server und dem Smartphone kümmert sich das Triumvirat um die Registrierung und Aufzeichnung der Temperaturen im Freien, im Treibhaus und in der darin beheimateten, heizbaren Klimabox mit den besonders empfindlichen Gewächsen. Außerdem kann ich von überall die Heizung zuschalten oder einfach die Automatik aktivieren. Wie das gemacht wird, das verrate ich Ihnen in diesem Blogbeitrag, herzlich willkommen!

Hardware

Die Basisversion des WLAN-Gärtners besteht aus folgenden Teilen:

1 ESP8266 D1 mini
2 DS18B20 mit 3m Kabel
1 DS18B20 mit 1m Kabel
1 Mininetzteil 230VAC zu 5VDC
1 OLED-Display
1 Relais High-Level-getriggert
1 Netzschalter 250V / 1A
1 Kunststoffgehäuse#
1 MB-102 Breadboard Steckbrett mit 830 Kontakten

Optional für höhere Schaltleistung
1 Stepup-Converter
1 Hochlastrelais (z. B. aus einer alten Waschmaschine oder Mikrowelle)
1 Einbausteckdose
1 1m Kabel 3x1,5 mit Schuko-Stecker
diverse Kleinteile wie Kabelschelle, Schrauben, …

Programmiert habe ich den Gärtner in MicroPython. Was dafür gebraucht wird, habe ich nachfolgend zusammengestellt.

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft
packetsender zum Testen des ESP8266 als UDP-Client und -Server

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

temperatur.py zum Ermitteln der Zuordnung der Sensoren
xmitter1.py die Firmware für den Gärtner
oled.py und
ssd1306.py Module zum Betrieb des Displays

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung, wie die MicropythonFirmware auf den ESP-Chip gebrannt wird.

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

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

Autostart

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

Programme testen

Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Zwischendurch doch mal wieder Arduino-IDE?

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

Die Schaltung

Die Schaltung verwendet ein Relais zum Schalten von höheren Stromstärken und höheren Spannungen, als sie der ESP8266 verträgt. Ein Relais beinhaltet einen oder mehrere Schaltkontakte. Nur wird der Schalter nicht von Hand geschlossen oder geöffnet, sondern durch einen Elektromagneten betätigt. Weil die Magnetspule auch schon Stromstärken von etwa 60mA benötigt, die ein GPIO-Pin des Controllers aber nicht liefern kann, muss zusätzlich ein Schalttransistor verwendet werden. Das in der Bauteillisteliste genannte Relais enthält bereits alles Nötige. Der Anschluss S kann direkt an einen GPIO-Pin gelegt werden.

Bei einem Relais haben wir also stets mindestens zwei Stromkreise, den Steuerkreis, der mit niedrigen Stromstärken die Magnetspule versorgt und den davon völlig abgekoppelten Lastkreis, der hohe Stromstärken und Spannungen schalten kann.

Die Testschaltung bauen wir auf einem Breadboard auf. Das erleichtert den Austausch von Teilen. Versorgt wird der Aufbau über das USB-Kabel vom PC. Das Produktivsystem wird später, wenn alles wunschgemäß funktioniert, durch ein Mininetzteil mit 5V Ausgangsspannung direkt aus dem 230V-Netz versorgt. Das reicht locker auch für den Spulenstrom des Lastrelais. An dieser Stelle muss aber unbedingt eine Warnung stehen.

- Das in der Teileliste angegebene Relais hat ein Schaltvolumen um die 300W. Das heißt, dass das Relais nur bei Spannungen bis 30V Stromstärken bis ca. 10A verkraftet.
- Der Anschluss für den Schaltkontakt SK des Lastkreises liegt in nur geringem Abstand (2mm) +zu den Bauteilen und Leiterbahnen (Signalleitung S) des Steuerkreises.

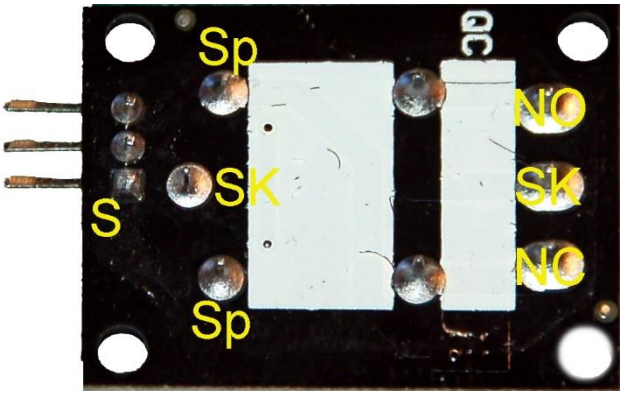

Abbildung 2: Relais von unten

Daher ist es keine gute Idee, Spannungen höher als 50V mit diesem Relais zu schalten. Ein Durchschlag vom 230V-Netz auf die Niederspannungsseite gefährdet in erster Linie Sie und andererseits Ihren Aufbau bis hin zum PC. 230V führende Leitungen zu berühren ist lebensgefährlich!

- **Befestigungsschrauben, die durch das Kunststoffgehäuse nach außen führen, müssen Kunststoffschrauben sein, um im Fall des Defekts einer 230V-Leitung Unfälle durch Stromschlag zu verhindern.**

- Zum Schalten höherer Leistungen, zum Beispiel von Heizgeräten, muss ein geeignetes Hochlastrelais verwendet werden. Man findet solche Bauteile in ausgedienten Waschmaschinen oder Mikrowellengeräten.
- **Hantieren Sie nicht mit 230V-führenden Schaltungen, wenn Sie nicht ganz genau wissen was Sie tun. Holen Sie sich gegebenenfalls Hilfe von einem Fachmann.**

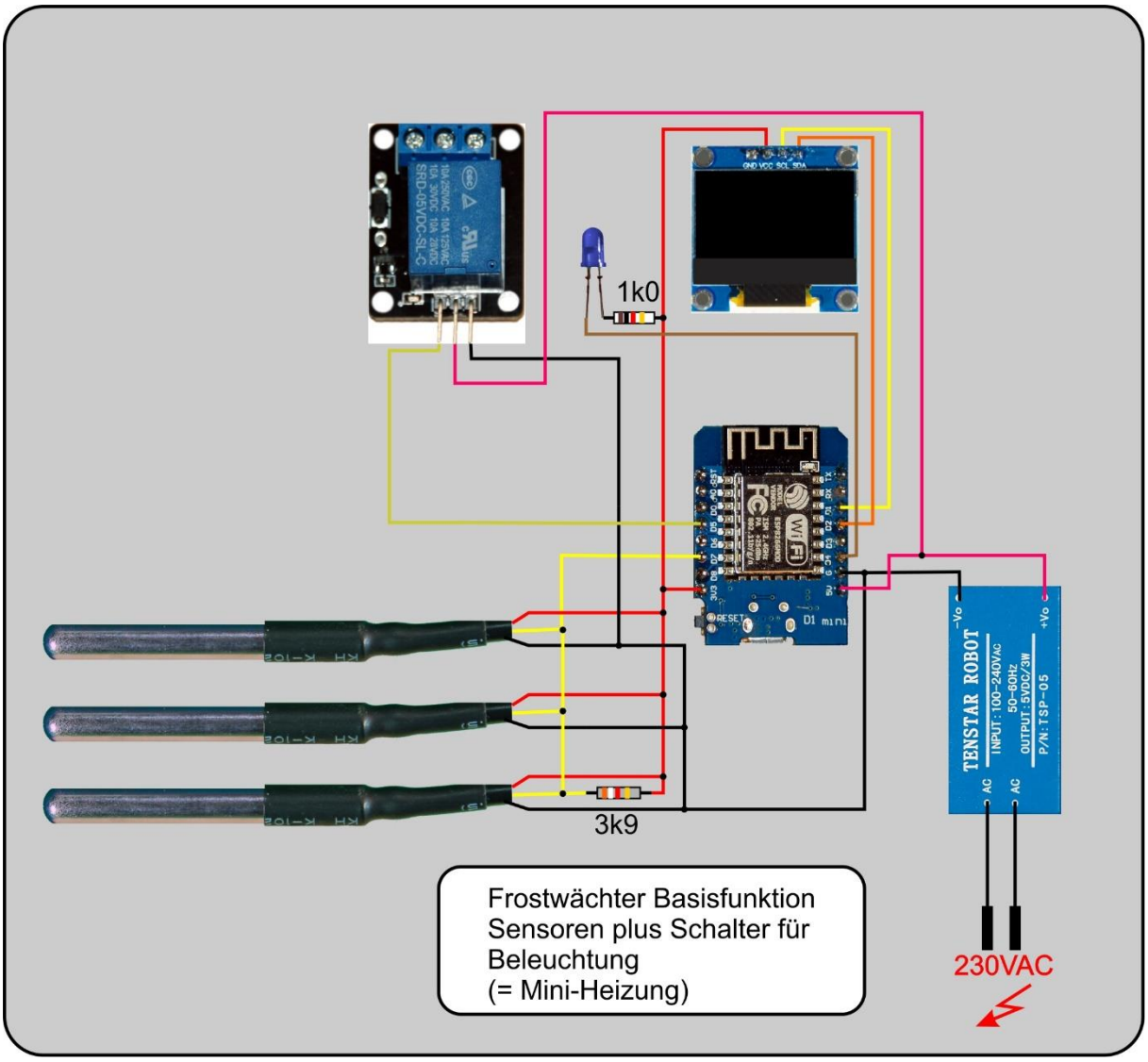

Abbildung 3: Basisschaltung

Die Stromaufnahme der Relaisspule liegt bei 58mA. Mit 0,7W ist das Relais, neben dem ESP8266 D1 mini mit seinen 42mA bei 5V Versorgungsspannung (=0,2W), der Hauptkonsument von Energie. Die Sensoren fallen zusammen mit weniger als 10mW nicht ins Gewicht. Auch die kurze Mehrbelastung während der Sendezeiten des Moduls (ein paar Millisekunden alle 10 Minuten) steckt das Netzteil locker weg.

Das Relais in der dargestellten Schaltung kann problemlos Niedervolt-Halogenlampen oder LED-Beleuchtungskörper schalten. Im ersteren Fall ist neben der Beleuchtung auch ein nicht zu vernachlässigender Wärmestrom zu verzeichnen, der durchaus einer Mini-Heizung gerecht wird. Ansonsten nehmen die Sensoren die Temperaturen in der Umgebung und im Treibhaus/Gartenhaus (3m-Kabel) sowie in der heizbaren Pflanzenbox (1m-Kabel) auf. Der ESP8266 hält diese Werte zur

Abfrage von einem Client, Smartphone oder Linuxserver (auch Raspi), bereit. Letzterer kann somit auch eine Klimakurve über den ganzen Winter hinweg aufzeichnen.

Der Betriebszustand wird auf einem OLED-Display dargestellt. Die blaue LED liegt an Port 2 und somit parallel zur eingebauten LED. Allerdings wird man von Letzterer nichts wahrnehmen, wenn das Ganze in ein Gehäuse eingebaut wird. Die Zusatz-LED kann dann gut sichtbar angebracht werden. Sie blinkt etwa im Sekundentakt und stellt das Lebenszeichen (Heartbeat) des Servers dar.

Bei der Verwendung des Netzteils müssen wir unbedingt darauf achten, dass Teile, die mit der Netzspannung verbunden sind, nicht berührt werden können. Ich habe mir für diesen Zweck eine Trägerplatine hergestellt, die neben dem vergossenen Netzteil auch Platz für den Boostconverter bietet. Den brauchte ich unbedingt, weil mein Relais eine 12V-Erregerspule besitzt. Alternativ könnte man ein separates 12V-Steckernetzteil verwenden und daraus durch einen Stepdown-Wandler die 5V für den ESP8266 ableiten.

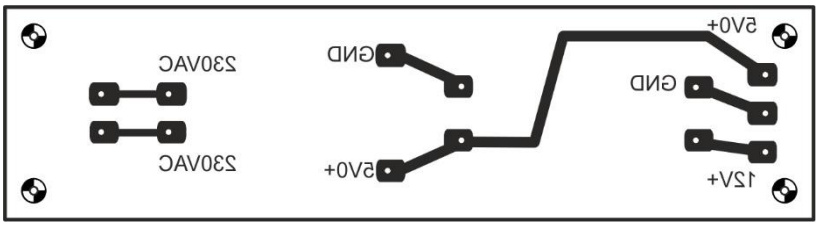

Abbildung 4: Trägerplatine

Abbildung 5: Netzteil

Das Platinenlayout wird mit einem Laserdrucker auf gut saugfähiges Papier gedruckt und mittels Bügeleisen auf die gereinigte Kupferschicht der Platine gebügelt und anschließend geätzt. Wie das im Einzelnen abläuft habe ich hier beschrieben.

Die Programmierung des Servers

Die Übertragung der Sensordaten zum Client auf dem Smartphone oder der Linuxkiste geschieht via UDP. Dieses Protokoll arbeitet verbindungslos, was bedeutet, dass die Übertragung nicht durch Handshaking abgesichert ist. Dafür ist es schnell und einfach zu nutzen. Verschiedene Teilnehmer können Daten vom Server abfragen, mehr noch der Server kann auch ohne Anfragen von Clients an diese Daten senden. Damit ist es zum Beispiel möglich, Notrufe auf ein Handy zu senden, auch wenn dieses nicht angefragt hat.

Es gibt auch einen kleinen Nachteil. Die Sache mit den Notrufen funktioniert nur im Sende-Empfangsbereich von ESP8266 und Handy/Linuxkiste. Dennoch kann man von überallher die Linuxmaschine über HTTP anfunken, denn da drauf wird ein zweiter TCP-Webserver laufen, den man weltweit erreichen kann. Damit der Apfel nicht weit vom Birnbaum fallen muss, werden wir im zweiten Teil der Reihe diesen Webserver natürlich auch in Python schreiben. Linux und vor allem der Raspi bieten Python quasi als Muttersprache an.

Jetzt erst aber einmal zur Betriebssoftware für den ESP8266, die wir im Wesentlichen in zwei Teile gruppieren können, Sensordaten aufbereiten und Serverdienste. Daneben laufen noch ein paar Ausgaben auf das OLED-Display.

```
from time import time,sleep,sleep_ms,ticks_ms
import network
import socket
import ubinascii
from machine import Pin, I2C, ADC
import os, sys
from oled import OLED
from ssd1306 import SSD1306
from onewire import OneWire
from ds18x20 import DS18X20
# *****************************************************
# Geben Sie hier Ihre eigenen Zugangsdaten an
mySid = 'here goes your ssid'; myPass = "here goes your pass"
monitor=("10.0.1.110",9181)
```

Wir importieren einige Module, dann folgen die Daten für den Netzwerkzugang via WLAN-Router. Die Windowsmaschine im Entwicklungsnetz kann durch das Programm **packetsender** Daten auf 10.0.1.10:9181 empfangen

```
i2c=I2C(-1,scl=Pin(5),sda=Pin(4))
print(i2c.scan())

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

Nachdem wir ein I2C-Objekt erzeugt haben, schauen wir nach, ob sich das Display auf dem Bus meldet. Es hat die Geräteadresse 60 = 0x3C. Die blink()-Funktion dient an verschiedenen Stellen dazu den Betriebszustand anzuzeigen. Der optionale Parameter **inverted** muss beim Aufruf auf **True** gesetzt werden, falls die LED low-aktiv ist, also wie in unserem Fall leuchtet, wenn die Kathode durch den GPIO-Pin auf GND gezogen wird.

Dann definieren wir den LED-Anschluss und geben die ersten Lebenszeichen, drei kurze Blitze. Das Anzeigeobjekt wird instanziiert und das Display gelöscht.

```
led=Pin(2, Pin.OUT) # D4@esp8266
blinkLed = led
# LED aus
led.value(1)
ledOn = const(0)
ledOff= const(1)

for i in range(4):
    blink(50, 200, True)

d=OLED(i2c)
d.clearAll()
```

Die nächste Sequenz bedient die DS18B20-Sensoren am One-Wire-Bus. Wir erzeugen ein Busobjekt, stellen die Anzahl der Sensoren fest und lesen deren ROM-Code aus. In roms halten wir ein Dictionary vor, das dem Standort den ROM-Code zuordnet. Dadurch können wir jederzeit genau den zugeordneten Sensor auslesen. Dessen Wert wird in einem zweiten Dict zwischengelagert. Die Zuordnung der Sensoren zu den Standorten geschieht mit dem Programm **temperatur**.py, zu dem wir später kommen.

Wurden keine Sensoren gefunden, dann kann das Programm nicht zweckdienlich arbeiten, es bricht den Start ab. Andernfalls bekommen wir eine Meldung auf dem Display und auf einem Terminal die Ausgabe der Paare Standort:ROM-Code.

```
ds_pin = Pin(13)     # D7@esp8266
ds_sensor = DS18X20(OneWire(ds_pin))
chips = ds_sensor.scan()
numberOfChips = len(chips)
roms = {
        "Gartenhaus": bytearray(b'(\x1dt$\n\x00\x00i'),
        "Pflanzenbox":bytearray(b'(p&\x12c \x01\x03'),
        "Umgebung":   bytearray(b'(\x02\xd3\x8e1!\x03>'),
       }
temperature= {
              "Gartenhaus": 0,
              "Pflanzenbox":0,
              "Umgebung"   :0,
             }
if numberOfChips==0:
    warnung = "NO SENSOR DEVICE"
    d.writeAt(warnung,0,0)
    print(warnung)
    sys.exit()
else:
    chipsFound="FOUND {} DEVICES".format(numberOfChips)
    d.writeAt(chipsFound,0,0)
    print('Found DS devices: ')
    for chip in roms.items():
        print(chip)
```

Das Relais liegt am GPIO-Pin 14. Es ist HIGH-aktiv, eine 1 am Ausgang schließt also den Kontakt NO (Normally Open) gegen Schaltkontakt SK.

```
heater=Pin(14,Pin.OUT) # D5@esp8266
heater.value(0)
heaterState=0
heaterState==heater.value()
timeDelay=2
```

Wir declarieren ein paar weitere Funktionen. Besondere Erwähnung verdient nur die Funktion sendTemperatur(). Sie nimmt die Adresse des Netzwerkgeräts, an welches die Daten gesendet werden sollen. Das kann die Adresse eines anfragenden Geräts sein oder eben auch ein Gerät, dessen Adresse wir selbst festlegen, wie **monitor** weiter oben.

Die Funktion fragt die Sensoren ab und setzt einen Antwortstring daraus zusammen, der schließlich gesendet wird. Nicht auslesbare Sensoren verursachen keinen Programmabbruch sondern lediglich eine Fehlanzeige. Dafür sorgen die try-except-Abschnitte. Damit schneller darauf zugegriffen werden kann, gibt die Funktion den Temperaturwert in der Pflanzenbox zurück.

```
def sendTemperatur(adr):
    global temperature
    answer="T;"
    try:

answer="{0:6.2f};".format(ds_sensor.read_temp(roms["Umgebung"]
))
    except:
        answer="Umg. fehlt;"
    try:

h="{0:6.2f};".format(ds_sensor.read_temp(roms["Gartenhaus"]))
        answer=answer+h
    except:
        answer=answer+"Haus fehlt;"
    try:

h="{0:6.2f}\n".format(ds_sensor.read_temp(roms["Pflanzenbox"])
)
        answer=answer+h
    except:
        answer=answer+"Box fehlt\n"
    print(answer)
    s.sendto(answer,adr)
    return h
```

Die folgende Bootsequenz stellt die Verbindung zum WLAN-Accesspoint her. Ein Netzwerk-Interface-Objekt wird erzeugt und aktiviert. Während des Anmeldevorgangs bekommen wir so lange Blinkzeichen (lang-kurz) von der blauen LED, bis uns vom DHCP-Server im Accesspoint eine vorläufige IP zugewiesen wurde. Damit wir die Statusmeldung nicht nur als Zahl bekommen, lassen wir uns den Klartext mit Hilfe des Dictionarys **connectstatus** im Terminal anzeigen.

Weil wir einen Server programmieren, geben wir uns mit der verordneten IP aber nicht zufrieden, sondern vergeben selbst die 10.0.1.180. Natürlich darf diese IP im LAN nicht anderweitig vergeben sein. Wir lesen die Verbindungsdaten wieder ein und geben Sie zur Information und Überprüfung am Terminal sowie am OLED-Display aus.

```
# ******************** Bootsequenz ************************
nic = network.WLAN(network.STA_IF)  # erzeugt WiFi-Objekt nic
nic.active(True)                    # nic einschalten
#
# Verbindung mit AP im lokalen Netzwerk aufnehmen
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("Status: ", nic.isconnected())
  while nic.status() != network.STAT_GOT_IP:
    print(".",end='')
    blink(800,200,inverted=True)
# Wenn verbunden, zeige Status und Config-Daten
print("\nStatus: ",connectStatus[nic.status()])
STAconf =
nic.ifconfig(("10.0.1.180","255.255.255.0","10.0.1.180", \
                  "10.0.1.180"))
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",STAconf[1],\
                    "\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
d.writeAt(STAconf[0],0,1)
d.writeAt(STAconf[1],0,2)
d.writeAt(STAconf[2],0,3)
```

Nun erzeugen wir ein Socket-Objekt und binden es an alle unsere Netzwerkschnittstellen (wir haben nur eine) und die Portnummer 9003. Wir setzen einen kurzen Timeout von 0,1 Sekunden. Das reicht, um ankommende Anfragen zu registrieren und sorgt dafür, dass sich das Programm nicht in der Empfangsschleife festhängt, sondern nach dieser Zeit die while-Schleife weiter durchläuft. So können neben dem Funkverkehr noch andere Sachen erledigt werden.

```
# ******************Socket for UDP-Server*****************
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind(('', 9003))
print("waiting on port 9003...")
s.settimeout(0.1) # timeout, damit 'while True:' durchlaeuft
d.writeAt("waiting on 9003",0,4)
if heaterState==1:
    d.writeAt("Heizung ist AN ",0,5)
else:
    d.writeAt("Heizung ist AUS",0,5)
auto=0
```

In der Dienstschleife kümmern wir uns um anstehende Aufgaben. An erster Stelle steht die Abfrage des Empfangspuffers des UDP-Sockets.

Ist eine Anfrage eingetroffen, wird sie gelesen, und wir merken uns die IP und die Portnummer des Clients. Weil es aber in den meisten Fällen nichts zu lesen gibt, wartet die Empfangsschleife, die wir mit **s.recvfrom**() aufrufen nur 0,1 Sekunden und

wirft dann eine Exception, die wir mit try – except abfangen. Im Moment ist für diesen Fall keine Behandlung vorgesehen, weshalb wir in den except-Zweig nur ein **pass** schreiben. Das ist die Anweisung, einfach nichts zu tun.

Hatten wir aber eine Anfrage, dann gilt es diese zu parsen. Das tun die nachfolgenden if-Strukturen. Durch einen Trick sparen wir Speicherplatz. Die automatische Heizungssteuerung erreichen wir durch Belegen der Variablen rec, in der sich immer der aktuelle Befehl befindet, mit "heizenAn" oder "heizenAus". Diese Befehle werden im weiteren Verlauf der Parserkette erkannt und ausgeführt. Hier kommt uns weiterhin zugute, dass die Temperatur der Pflanzenbox als Einzelwert zurückgegeben wird.

getTemp	startet eine Konvertierung in den DS18B20-Chips
sendTemp	schickt die Werte an die anfragende Station und überprüft ob die Heizung ein- oder ausgeschaltet werden muss.
autoOn	aktiviert die automatische Heizungsregelung
autoOff	deaktiviert die Automatik
e	beendet das Programm
heizenAn	schaltet das Relais im Heizkreis ein
heizenAus	schaltet das Relais im Heizkreis aus

```
# ********************* Dienstschleife *********************
while 1:
    gc.collect()
    try:
        rec,adr=s.recvfrom(150)
        rec=rec.decode().strip("\r\n")
        print(rec)
        if rec=="autoOn":
            auto=1
            s.sendto("M;"+rec,adr)
        if rec=="autoOff":
            auto=0
            s.sendto("M;"+rec,adr)
        if rec=="getTemp":
            ds_sensor.convert_temp()
            s.sendto("M;started",adr)
        if rec=="sendTemp":
            print("Senden")
            boxtemp=sendTemperatur(adr)
            if auto==1:
                try:
                    boxtemp=float(boxtemp)
                    if boxtemp < 3:
                        rec="heizenAn"
                    else:
                        rec="heizenAus"
                except:
                    pass
```

```
        if rec=="heizenAn" :
            heater.value(1)
            d.writeAt("Heizung ist AN ",0,5)
            heaterstate=heater.value()
            s.sendto("M;Heizung ist AN",adr)
        if rec=="heizenAus":
            heater.value(0)
            d.writeAt("Heizung ist AUS",0,5)
            heaterstate==heater.value()
            s.sendto("M;Heizung ist AUS",adr)
        if rec=="e":
            heater.value(0)
            d.clearAll()
            d.writeAt("*** SHUT OFF ***",0,5)
            break
        rec=""
    except:
        pass

    blink(50,950,True)
```

Zu jedem Auftrag wird eine Meldung an den Client gesandt, aus welcher er evaluieren kann, ob der Befehl erkannt wurde. Der blink-Befehl schließt die Schleife ab. Solange die LED blinkt, wissen wir, dass der Server noch läuft.

Test

Bevor wir unser Programm testen können, müssen wir noch die Sensoren ihrem Standort zuordnen und sie registrieren. Das gelingt mit dem kleinen Bruder unseres Serverprogramms **temperatur**.py. Schließen wir **einen** Sensor an uns starten wir das Programm. Es sollte den Sensor erkennen und seinen ROM-code im Terminal ausgeben. Die Zuordnung ergibt sich aus dem zukünftigen Standort, an dem Sie ihn anbringen möchten. Den neuen ROM-Code tragen Sie durch Copy&Paste in die Liste roms ein. Wiederholen Sie den Vorgang, bis der dritte Sensor erkannt und eingetragen ist. Übertragen Sie die Listenelemente jetzt in das Programm xmitter1.py und speichern Sie dieses ab.

temperatur.py

```
from time import sleep, sleep_ms, time, ticks_ms
from machine import Pin, reset
from onewire import OneWire
from ds18x20 import DS18X20

ds_pin = Pin(13)
ds_sensor = DS18X20(OneWire(ds_pin))

chips = ds_sensor.scan()
numberOfChips = len(chips)
roms =  [
        bytearray(b'(\x1dt$\n\x00\x00i'), #GHaus
        bytearray(b'(\x02\xd3\x8e1!\x03>'), # ambiant
        bytearray(b'(p&\x12c \x01\x03'), # box
        ]
print('Found DS devices: ')
for chip in chips:
  print(chip)

led = Pin(2, Pin.OUT)
# LED aus
led.value(1)
ledOn = 0
ledOff= 1
taste = Pin(0, Pin.IN)
**********************************************************

def blink(dauer):
  start = ticks_ms()
  current = start
  end = start+dauer
  led.value(ledOn)
  while current <= end:
    current=ticks_ms()
  led.value(ledOff)
```

```
def ds1820Error():
  while True:
    blink(200)
    sleep_ms(200)
    blink(200)
    sleep_ms(200)
    blink(1000)
    sleep_ms(1000)

#
************************************************************
if len(roms) < 1:
    #ds1820Error()
    pass

ds_sensor.convert_temp()
sleep_ms(750)
for rom in chips:
    print(rom)
    print(ds_sensor.read_temp(rom))
print(" ")
```

Das Programm **packetsender** erlaubt uns jetzt die ersten Tests unserer Einheit. Wir starten das Programm xmitter1.py, das Sie auch als Ganzes herunterladen können, auf dem ESP8266. Wenn in **packesender** die korrekten Verbindungsdaten eingetragen sind, geben wir einen der oben aufgeführten Befehle ein und senden ihn ab. An der Antwort vom Server sehen wir, ob die Übermittlung geklappt hat.

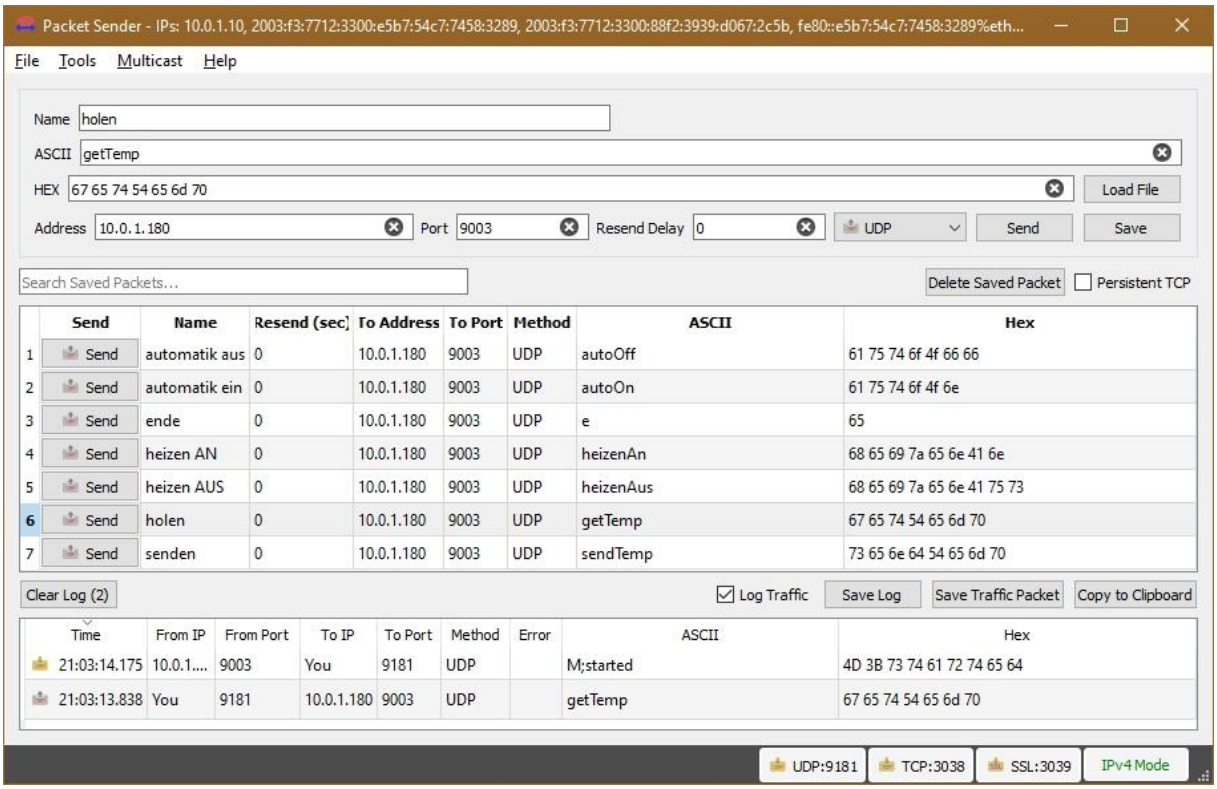

Abbildung 6: Packetsender

Ausblick

In der nächsten Folge geht es um die Programmierung eines Clients auf einer Linuxmaschine, die per Zeitsteuerung über einen Cronjob alle 10 Minuten eine Anfrage an der ESP8266 richtet, um die Temperaturen abzufragen. Die Daten muss der Client in zwei Dateien verwalten. Die Temperaturdaten werden in der Datei **gh_daten** stehen, und es wird eine zweite Datei **meldungen** geben, in die alles eingetragen wird, was mit dem Prefix M; beim Client ankommt. Das können auch Dinge sein, die der Server von sich aus verschickt.

Am Ende des Tages sollen die Temperaturdaten in einer Tagesdatei abgelegt werden, um permanent verfügbar zu sein. Zum Beispiel mittels Browseraufruf von Handy oder Tablet.

An Hardware kommen noch zwei Taster dazu, um die Heizung auch vor Ort händisch schalten zu können und wir spendieren dem Server noch einen Bodenfeuchte-Sensor, damit er uns melden kann, wenn die Pflanzen Wasser brauchen.

Bis dahin viel Freude beim Basteln und Programmieren.