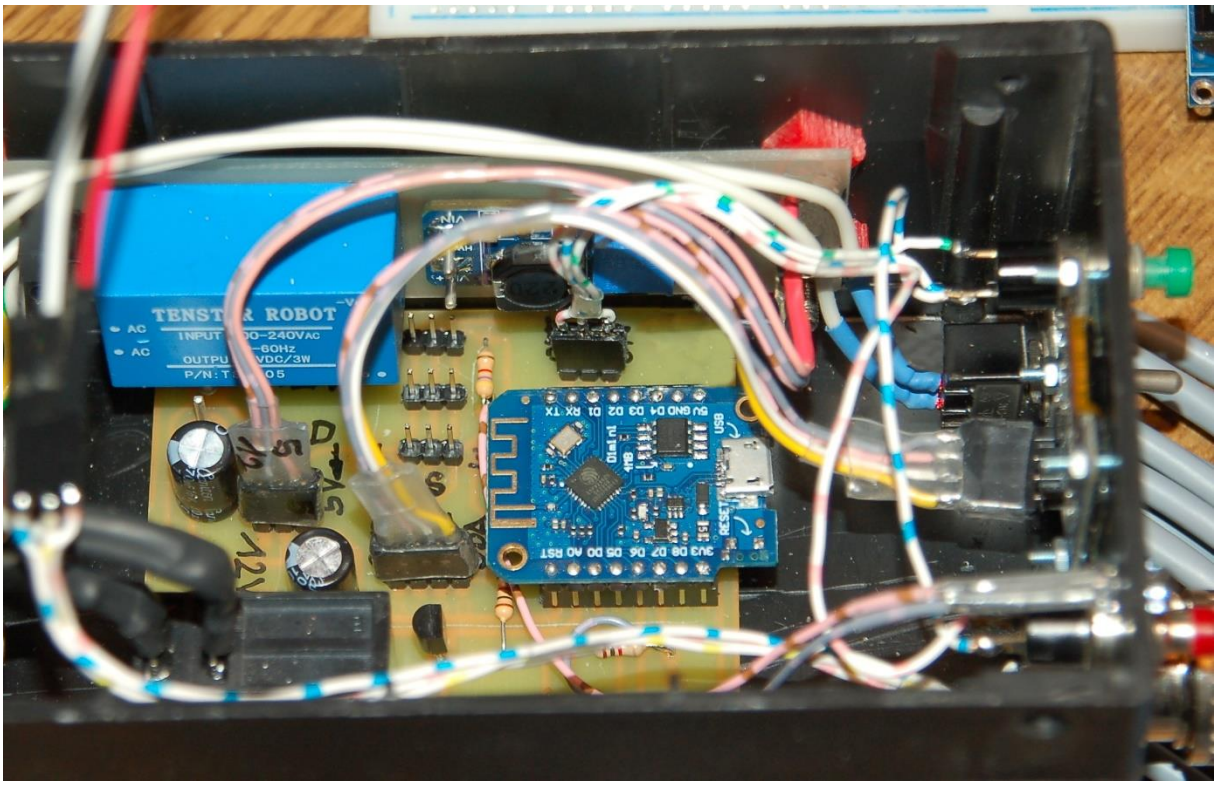


Abbildung 1: Innenleben

This article is also available as a PDF document in Deutsch and Englisch.

Do you also know sentences like: "Do you still check the temperature in the greenhouse?" or "Shouldn't we turn on the heating in the greenhouse after all? There's supposed to be frost tonight!" With us, these questions will be a thing of the past in the future. Because from now on mine wakes up

WLAN-Gärtner

about the temperatures in and around the greenhouse. Together with my Linux server and the smartphone, the triumvirate takes care of the registration and recording of the temperatures in the open air, in the greenhouse and in the heatable climate box with the particularly sensitive plants located in it. In addition, I can turn on the heating from anywhere or simply activate the automatic. How this is done, I'll tell you in this blog post, welcome!

Hardware

The basic version of the WLAN gardener consists of the following parts:

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, …

I programmed the gardener in MicroPython. I have put together below what is needed for this.

Software

For flashing and programming the ESP32:
Thonny oder
µPyCraft
packetsender for testing the ESP32 as a TCP/UDP server

Used Firmware:

MicropythonFirmware
Please choose a stable version

MicroPython-Programs:
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 - Language - Modules and Programs

You can find detailed instructions for installing Thonny here. There is also a description of how the Micropython firmware is burned onto the ESP chip.

MicroPython is an interpreter language. The main difference to the Arduino IDE, where you always and exclusively flash entire programs, is that you only have to flash the MicroPython firmware once at the beginning on the ESP32 before the

controller understands MicroPython instructions. You can use Thonny, µPyCraft or esptool.py for this. I have described the process for Thonny here.

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. You simply save an enormous amount of time if you can do simple tests of the syntax and hardware through to trying out and refining functions and entire program parts via the command line before you knit a program out of it. For this purpose I also like to create small test programs over and over again. As a kind of macro, they combine recurring commands. From such program fragments, entire applications can develop.

Autostart

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 ESP chip. The program starts automatically the next time it is reset or switched on.

Testing programs

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 ESP chip. The program starts automatically the next time it is reset or switched on.

In between, Arduino IDE again?

If you want to use the controller together with the Arduino IDE again later, simply flash the program in the usual way. However, the ESP32 / ESP8266 then forgot that it ever spoke MicroPython. Conversely, every Espressif chip that contains a compiled program from the Arduino IDE or the AT firmware or LUA or ... can easily be provided with the MicroPython firmware. The process is always as described here.

Die Schaltung

The circuit uses a relay to switch higher currents and higher voltages than the ESP8266 can handle. A relay contains one or more switching contacts. Only the switch is not closed or opened by hand, but operated by an electromagnet. Because the magnetic coil already requires currents of around 60mA, which a GPIO pin of the controller cannot deliver, a switching transistor must also be used. The relay named in the component list already contains everything necessary. The connection S can be connected directly to a GPIO pin.

With a relay, we always have at least two circuits, the control circuit, which supplies the magnet coil with low currents, and the load circuit, which is completely decoupled from it, which can switch high currents and voltages.

We build the test circuit on a breadboard. This makes it easier to exchange parts. The superstructure is supplied via the USB cable from the PC. The productive system is later supplied, if everything works as required, by a mini power supply unit with 5V output voltage directly from the 230V network. That is easily enough for the coil current of the load relay. At this point, however, there must be a warning.

- The relay specified in the parts list has a switching volume of around 300W. This means that the relay can only handle currents up to approx. 10A at voltages up to 30V.
- The connection for the switch contact SK of the load circuit is only a short distance (2mm) + to the components and conductor tracks (signal line S) of the control circuit.

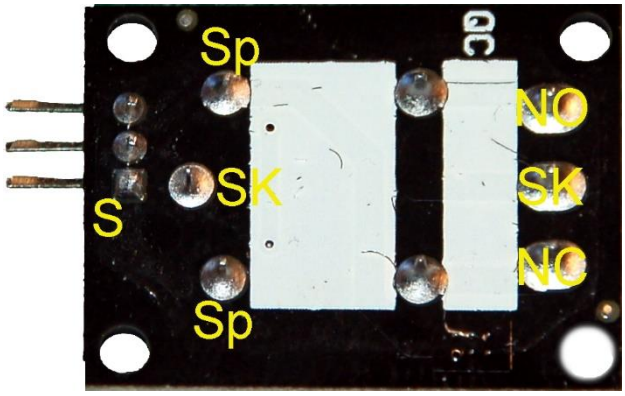


Abbildung 2: Relais von unten

Therefore it is not a good idea to switch voltages higher than 50V with this relay. A breakdown from the 230V network to the low-voltage side primarily endangers you and, on the other hand, your structure up to and including the PC. Touching 230V lines is life-threatening!

- • A suitable high-load relay must be used to switch higher outputs, for example heating devices. Such components can be found in disused washing machines or microwave ovens.
- **• Do not handle circuits carrying 230V if you do not know exactly what you are doing. If necessary, get help from a professional.**

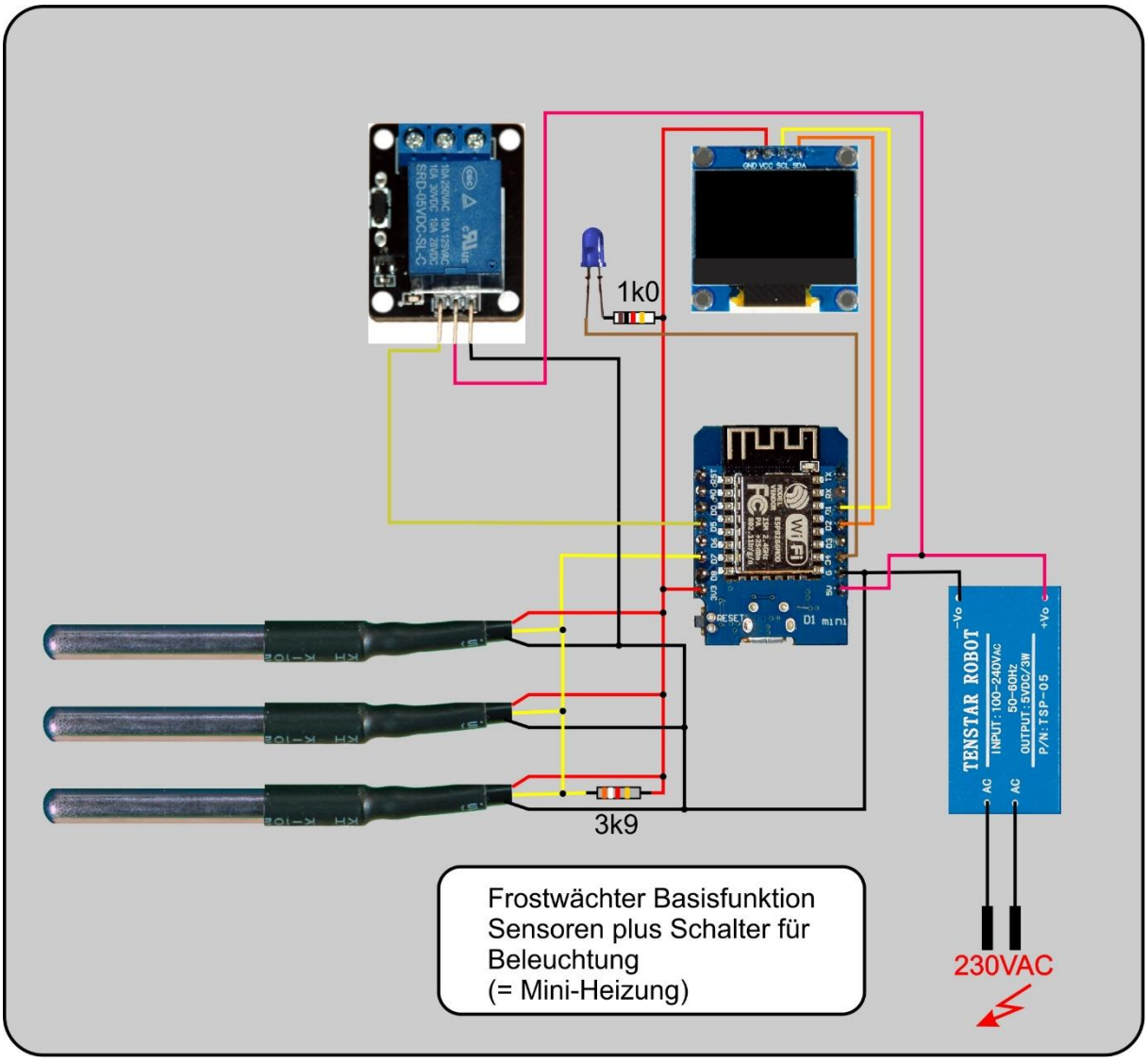


Abbildung 3: Basisschaltung

The relay in the circuit shown can switch low-voltage halogen lamps or LED lighting fixtures without any problems. In the first case, in addition to the lighting, there is also a heat flow that cannot be neglected, and which does justice to a mini heater. Otherwise, the sensors record the temperatures in the surrounding area and in the greenhouse / garden house (3m cable) and in the heatable plant box (1m cable). The ESP8266 keeps these values ready for querying from a client, smartphone or Linux server (also Raspi). The latter can thus also record a climate curve over the entire winter.

The operating status is shown on an OLED display. The blue LED is on port 2 and therefore parallel to the built-in LED. However, you will not notice the latter if the whole thing is built into a housing. The additional LED can then be attached so that it is clearly visible. It flashes every second and represents the server's heartbeat.

When using the power supply unit, we must ensure that parts that are connected to the mains voltage cannot be touched. For this purpose I made a carrier board that offers space for the boost converter in addition to the encapsulated power supply. I absolutely needed it because my relay has a 12V excitation coil. Alternatively, you

could use a separate 12V plug-in power supply and use a step-down converter to derive the 5V for the ESP8266.

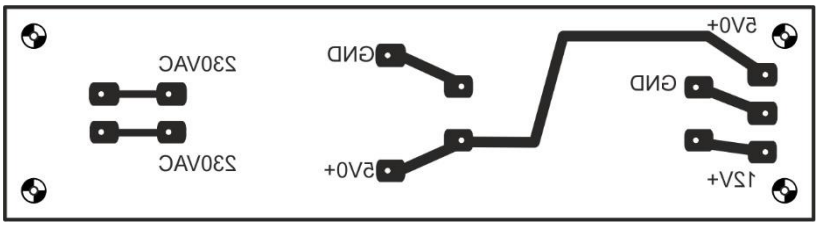


Abbildung 4: Trägerplatine

Abbildung 5: Netzteil

The circuit board layout is printed on absorbent paper with a laser printer and ironed onto the cleaned copper layer of the circuit board with an iron and then etched. How that works in detail I have descibed here.

Die Programmierung des Servers

The sensor data is transmitted to the client on the smartphone or the Linux box via UDP. This protocol works without a connection, which means that the transmission is not secured by handshaking. It is quick and easy to use for this. Different participants can request data from the server, even more so the server can also send this data without requests from clients. This makes it possible, for example, to send emergency calls to a cell phone, even if it has not asked.

There is also a minor drawback. The thing with the emergency calls only works in the send / receive area of the ESP8266 and mobile phone / Linux box. Nevertheless, you can contact the Linux machine from anywhere via HTTP, because a second TCP web server will run on it, which can be reached worldwide. So that the apple doesn't have to fall far from the pear tree, we will of course also write this web server in Python in the second part of the series. Linux and especially the Raspi offer Python as a native language.

But first of all, let's talk about the operating software for the ESP8266, which we can essentially group into two parts, processing sensor data and server services. In addition, a few issues run on the 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)
```

We import some modules, then the data for network access via WLAN router follow. The Windows machine in the development network can receive packetsender data on 10.0.1.110:9181 through the program

```
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)
```

After we have created an I2C object, we check whether the display reports on the bus. It has the device address 60 = 0x3C. The blink () function is used in various places to display the operating status. The optional parameter inverted must be set to True when called, if the LED is low-active, i.e., as in our case, lights up when the cathode is pulled to GND by the GPIO pin.

Then we define the LED connection and give the first signs of life, three short flashes. The display object is instantiated and the display is deleted.

```
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()
```

The next sequence operates the DS18B20 sensors on the one-wire bus. We create a bus object, determine the number of sensors and read out their ROM code. In roms we have a dictionary that assigns the ROM code to the location. This means that we can read out exactly the assigned sensor at any time. Its value is temporarily stored in a second dict. The assignment of the sensors to the locations is done with the temperatur.py program, which we will come to later.

If no sensors were found, the program cannot work properly and the start is aborted. Otherwise we get a message on the display and the output of the pairs Location: ROM code on a terminal.

```
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)
```

The relay is on GPIO pin 14. It is HIGH-active, a 1 at the output thus closes the NO (Normally Open) contact against switch contact SK.

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

We'll declare a few more functions. Only the function sendTemperature () deserves special mention. It takes the address of the network device to which the data should be sent. This can be the address of a requesting device or a device whose address we define ourselves, such as monitor above.

The function queries the sensors and compiles a response string from it, which is then sent. Sensors that cannot be read out do not cause the program to be terminated, they merely result in a false report. The try-except sections take care of that. The function returns the temperature value in the plant box so that it can be accessed more quickly.

```
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
```

The following boot sequence establishes the connection to the WLAN access point. A network interface object is created and activated. During the registration process we get flashing signals (long-short) from the blue LED until we have been assigned a temporary IP by the DHCP server in the access point. So that we don't just get the status message as a number, we have the plain text displayed in the terminal using the connectstatus dictionary.

Because we are programming a server, we are not satisfied with the prescribed IP, we assign the 10.0.1.180 ourselves. Of course, this IP must not be assigned otherwise in the LAN. We read in the connection data again and provide them for information and verification on the terminal and on the OLED display.

```
# ******************** 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)
```

Now we create a socket object and bind it to all of our network interfaces (we only have one) and the port number 9003. We set a short timeout of 0.1 seconds. This is enough to register incoming requests and ensures that the program does not get stuck in the receiving loop, but continues to run through the while loop after this time. In this way, other things can be done in addition to radio communication.

```
# ******************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 the service loop, we take care of upcoming tasks. In the first place is the query of the receive buffer of the UDP socket.

If a request is received, it is read and we note the IP and the port number of the client. But because in most cases there is nothing to read, the receive loop that we call with s.recvfrom () only waits 0.1 seconds and then throws an exception, which we catch with try - except. At the moment, no treatment is planned for this case, which is

why we only write one pass in the except branch. That is the instruction to just do nothing.

But if we had a request, then it has to be parsed. This is what the following if structures do. We use a trick to save storage space. We achieve the automatic heating control by assigning the variable rec, which always contains the current command, with "heat on" or "heat off". These commands are recognized and executed in the further course of the parsing chain. Here we benefit from the fact that the temperature of the plant box is returned as an individual value.

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)
```

A message is sent to the client for each job, from which it can evaluate whether the command was recognized. The blink command closes the loop. As long as the LED is flashing, we know that the server is still running.

Test

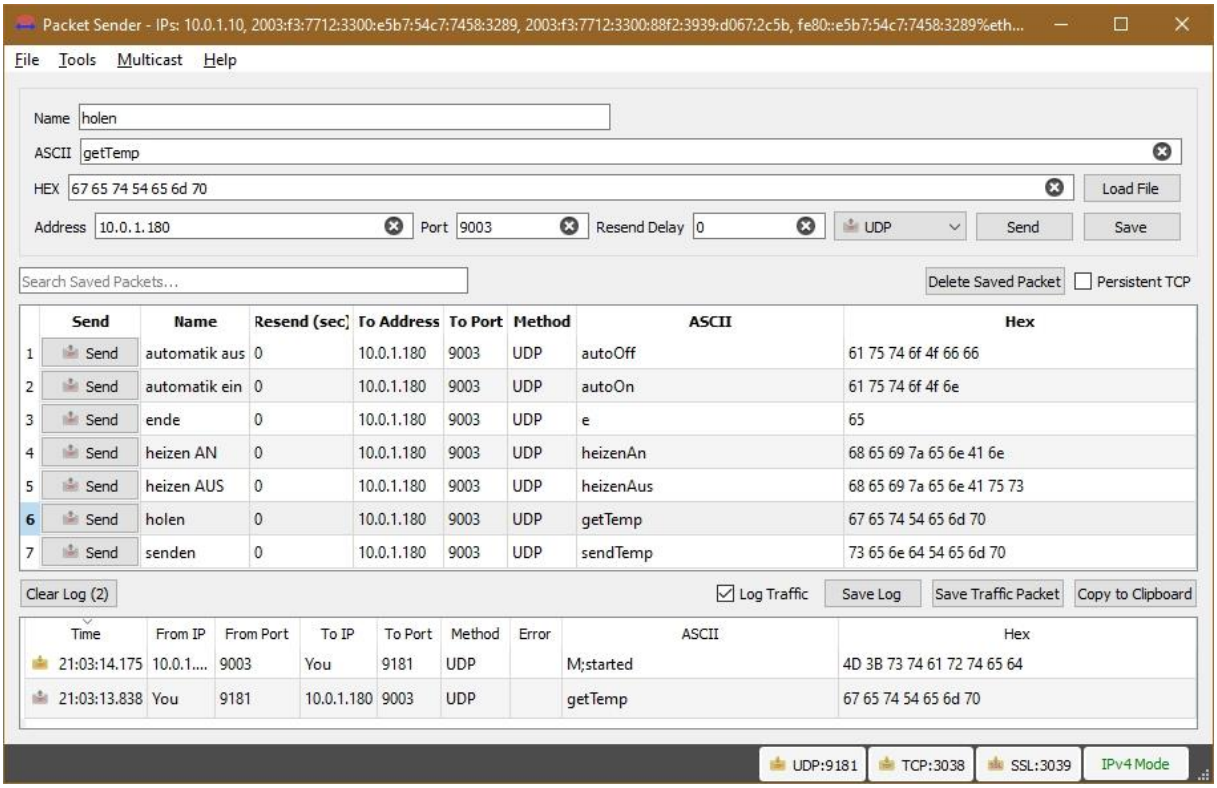
Before we can test our program, we have to assign the sensors to their location and register them. This is possible with the little brother of our server program temperatur.py. If we connect a sensor to us, we start the program. It should recognize the sensor and output its ROM code in the terminal. The assignment results from the future location at which you want to attach it. Enter the new ROM code in the roms list using copy & paste. Repeat the process until the third sensor has been recognized and entered. Now transfer the list elements to the xmitter1.py program and save it.

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(" ")
```

The packetsender program now allows us to carry out the first tests on our unit. We start the program xmitter1.py, which you can also download as a whole, on the ESP8266. If the correct connection data is entered in packesender, we enter one of the commands listed above and send it off. We can see from the response from the server whether the transmission was successful.



Ausblick

The next episode deals with the programming of a client on a Linux machine that sends a request to the ESP8266 every 10 minutes via time control via a cron job to query the temperatures. The client must manage the data in two files. The temperature data will be in the gh_daten file, and there will be a second message file in which everything is entered that is preceded by the M; arrives at the client. These can also be things that the server sends on its own.

At the end of the day, the temperature data should be stored in a daily file so that it is permanently available. For example, by calling up a browser from a mobile phone or tablet.

There are also two hardware buttons to be able to switch the heating manually on site and we are giving the server a soil moisture sensor so that it can notify us when the plants need water.

Until then, have fun tinkering and programming.