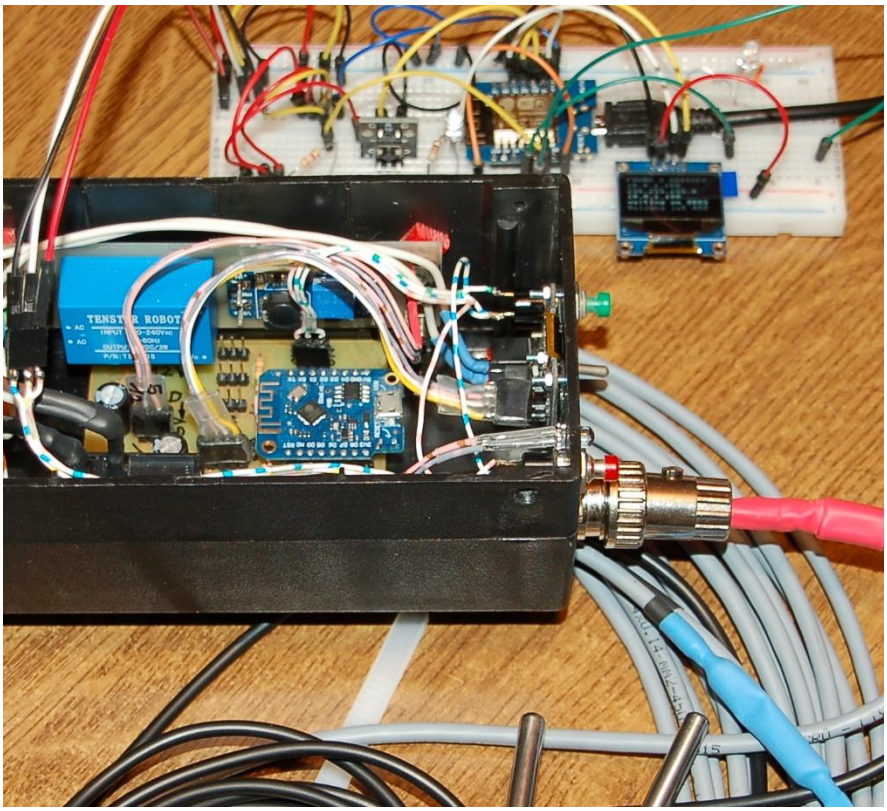


Abbildung 1: Produktivsystem und Testaufbau

This is what the inside of my finished device looks like. The test setup can be seen in the background. I'm currently assigning the three DS18B20 sensors (in the foreground) of the production version to their duty stations. The three individual cables to the bus are brought together in the connector, at the bottom right of the housing. The procedure is described in detail in the first part of the frost monitor.

Today we will add a few more modules to the device and awaken the subservient spirits on the Linux machine. Welcome to

Freeze Guardian - WLAN-Gärtner Teil2

Let's stay with the hardware extensions.

Hardware

In order to be able to switch the heating on site, there are two buttons whose actuation is registered by the ESP8266 and converted into a switching process of the relay. An external pull-up resistor is not required because the internal one is switched on by software.

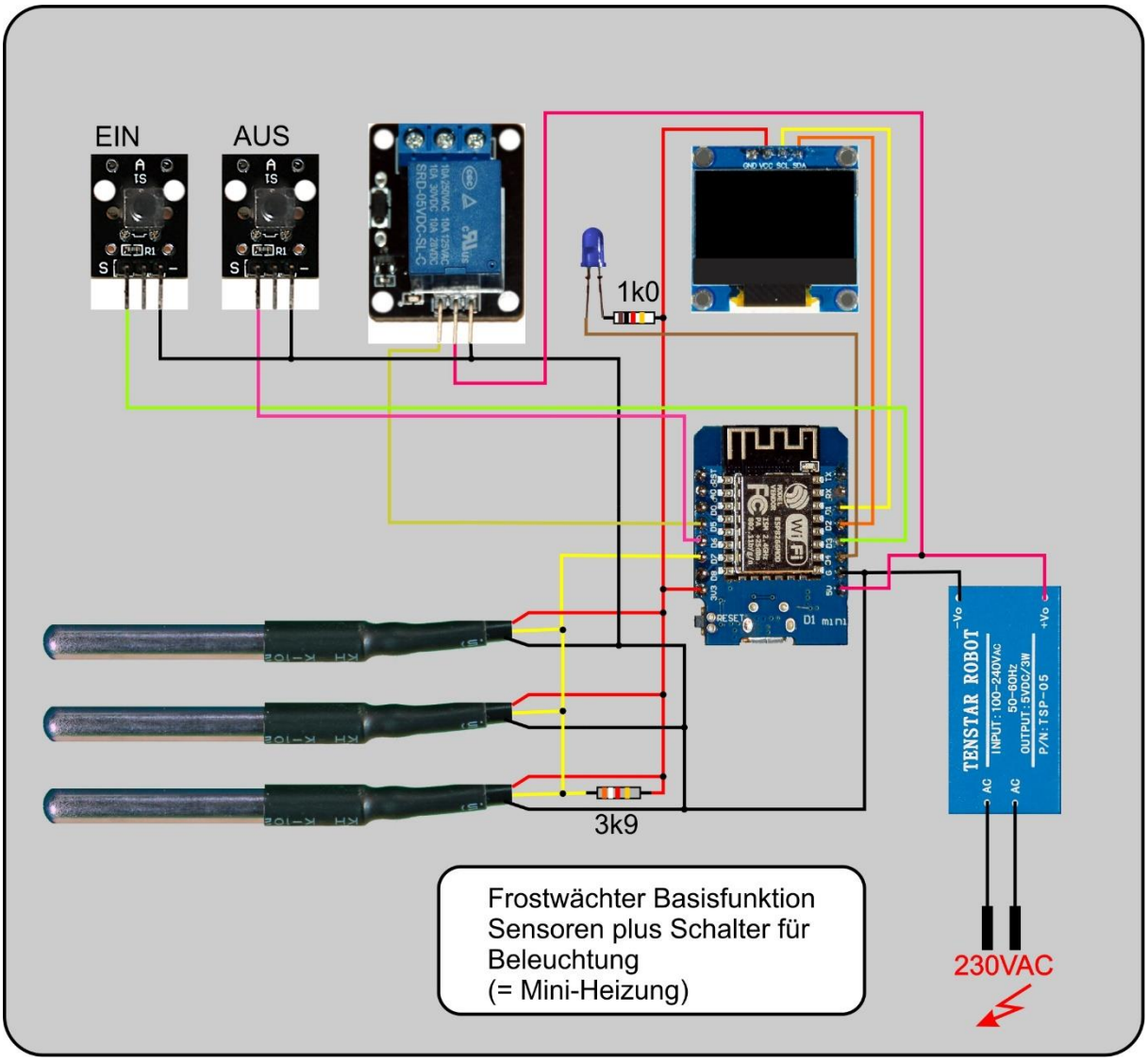


Abbildung 2: Basisschaltung plus Taster

With the simple relay stage, I could not operate the oil radiator, which is supposed to keep the inner room of the greenhouse at frost-free temperatures. So a more powerful relay was needed. I found that in a disused washing machine. It also controlled the heating circuit there, so it is ideal for this purpose! The coil voltage of 12V could not be supplied by the small power supply from TENSTAR, which is why I put a boost converter (aka step-up converter) into service. That seemed easier and cheaper to me than installing an additional damp-proof socket for a plug-in power supply in the greenhouse. The current consumption of the relay coil is 58mA. With 0.7W, the relay is the main consumer of energy, alongside the ESP8266 D1 mini with its 42mA at 5V supply voltage (= 0.2W). Together with less than 10mW, the sensors are insignificant. The power supply unit can easily handle the brief additional load during the transmission times of the module (a few milliseconds every 10 minutes).

A BC337 (NPN transistor) is used for control, which can ensure sufficient current through the relay coil in saturation. The base resistance reduces the base current strength to a minimum. The (freewheeling) diode tweaks voltage peaks that occur on the coil when the transistor is switched off and can endanger it.

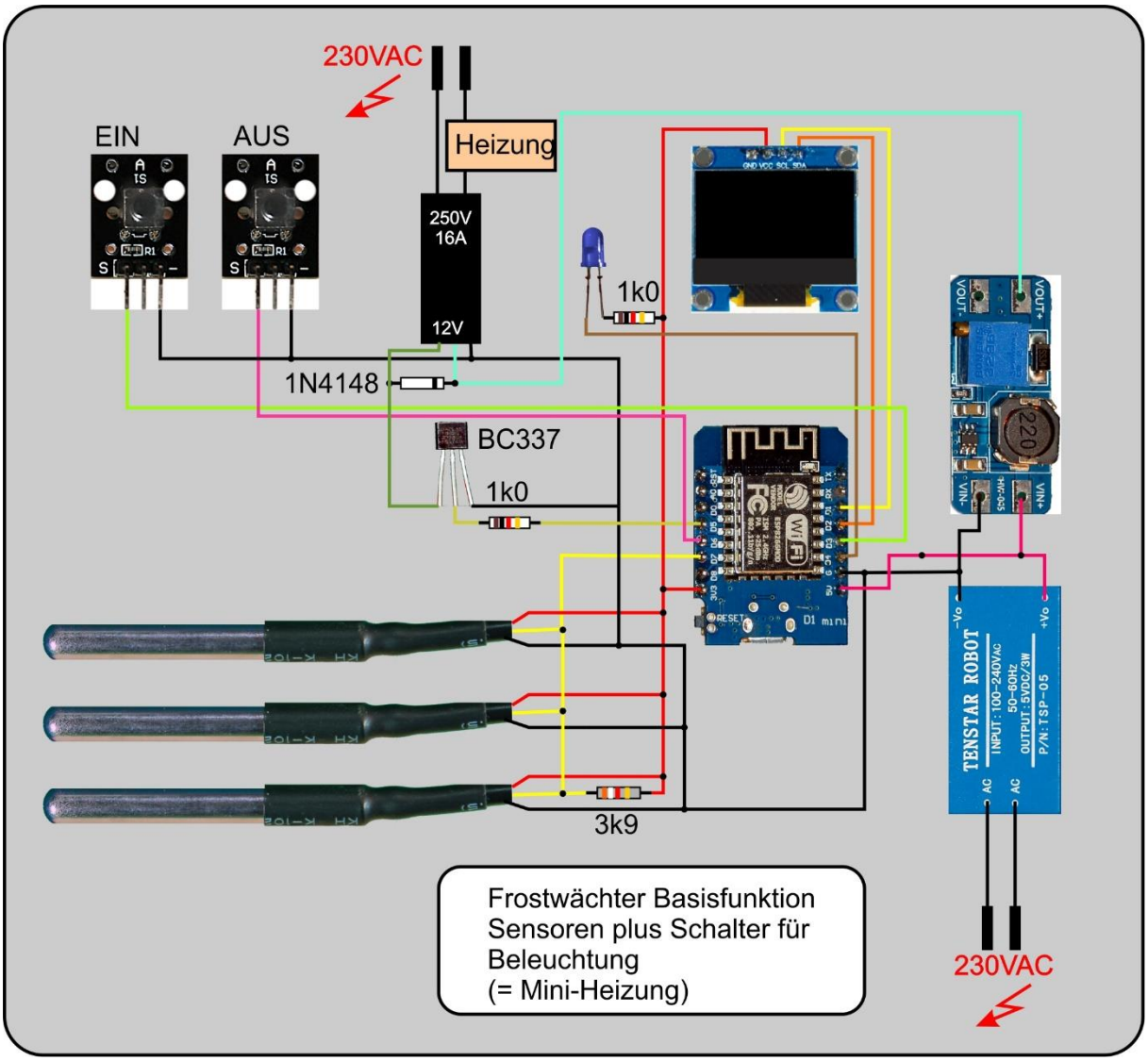


Abbildung 3: Basisschaltung plus Taster,Hochlastrelais

In the third expansion stage, I added a soil moisture sensor that is connected via the analog input A0. If more than 768 counts of 1023 are measured, then the substrate is too dry and the ESP8266 should sound the alarm bells. The threshold value can of course be adapted in the program to the requirements on site.

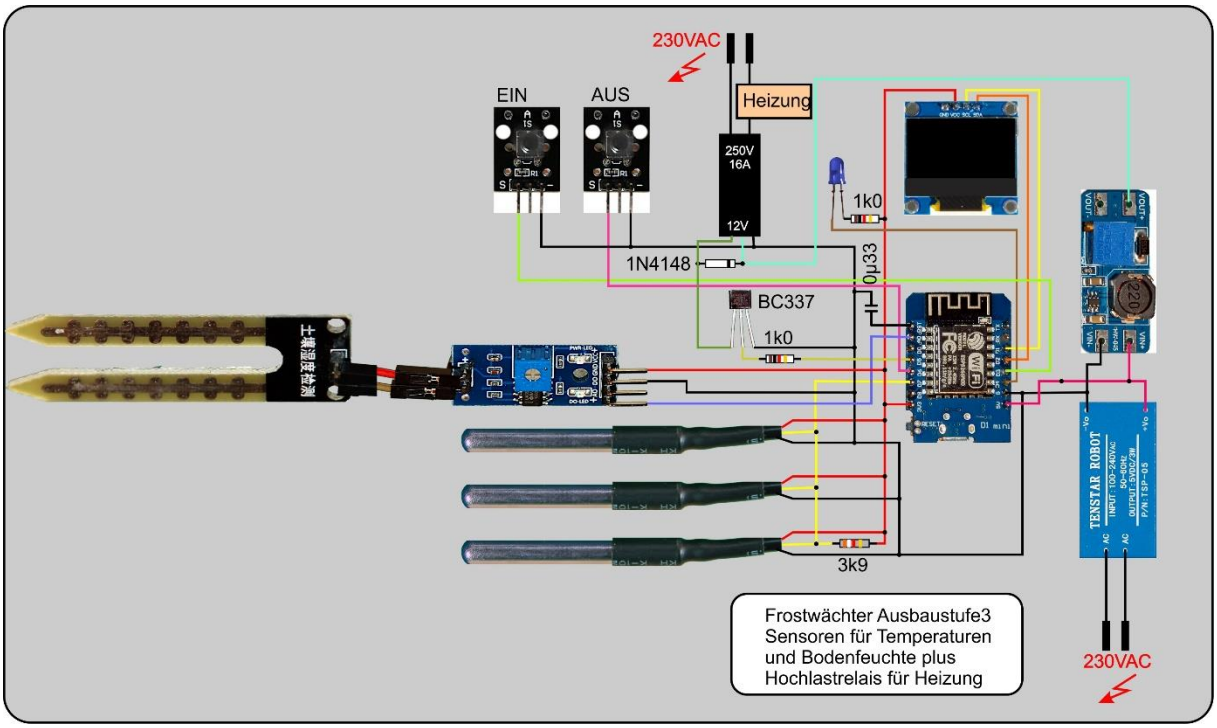


Abbildung 4: Schaltung komplett mit Feuchtefühler

I don't want to hide an important tip at this point. Every now and then the ESP8266 crashed for no apparent reason. That's not particularly bad because it automatically reboots afterwards, but it's annoying when something doesn't work as it should. For example, talking on the cordless phone tended to trigger reboots, even from a distance. Since the installation of a capacitor of 330nF between the RST pin and GND, the problem has been solved and the ESP8266 runs without interruption, like clockwork - for days ... In the circuit diagram in Figure 4, this capacitor is already drawn.

In the summary it now looks like this. Only the parts that have been added compared to the previous version are listed.

1	MT3608 DC-DC Netzteil Adapter Step up Modul
2	Taster
1	Bodenfeuchte-Sensor
1	Hochlastrelais
1	Transistor BC337
1	Diode 1n4148
1	Widerstand 1,0 kΩ
1	Kondensator 330nF

I have developed two carrier boards to make it easier to assemble the various modules. I already presented the first of these in the last post. It combines the power supply unit and the boost converter.

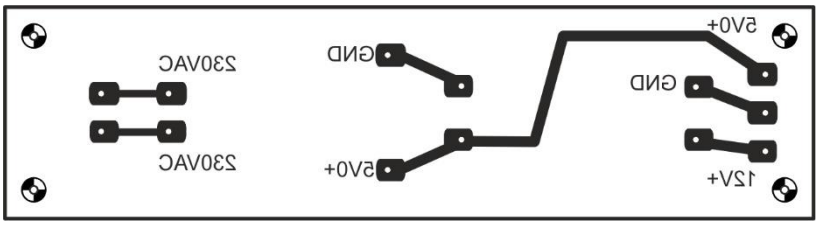
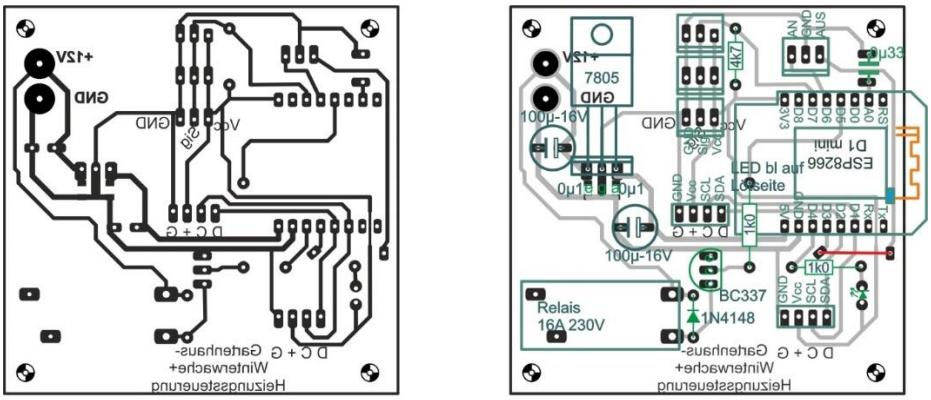


Abbildung 5: Trägerplatine

The second PCB accommodates the ESP8266 D1 mini, the load relay, the transistor driver stage and the plug-in contacts to the periphery. Both layouts can be downloaded as a PDF file.

Instead of the linear regulator LM7805, three post pins (Ø 1.3mm) are fitted and connected to the corresponding connections on the other carrier board in this implementation of the circuit. The order from the left is: 12V - GND - 5V. The charge controller would be intended in the event that a 12V source is on hand. It would then provide the 5V and make the use of Tenstar power supply and boost converter superfluous.



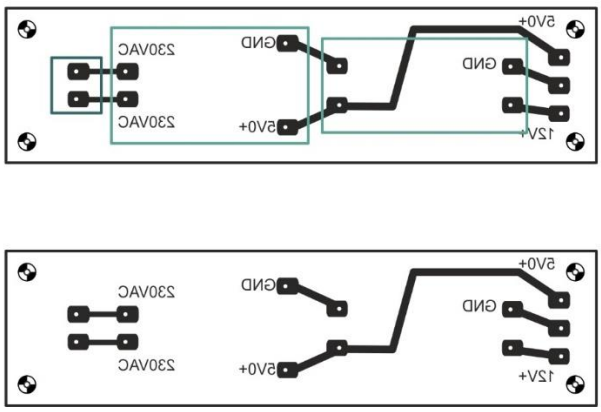


Abbildung 6: Trägerplatinen – verkleinerte Darstellung!

The hardware expansions have of course also made for additional software work. While the tools stayed the same, there is a new operating program for the "gardener".

The Software

For flashing and programming the ESP8266:
Thonny oder
µPyCraft
packetsender for testing the ESP32 / ESP8266 as a UDP server and client

Used Firmware:

MicropythonFirmware
Please choose a stable version
Note: With regard to the I2C interface, different initialization rites may occur**.**

Die MicroPython-Programme zum Projekt:

xmitter2.py Betriebsoftware des Temperaturwächters
client9001.py UDP-Client für Linux oder Raspi
archivate.py zum Archivieren der Tagesdateien
converttemp.py Auftraggeber zum Einlesen der Temperaturwerte

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.

Neue Programmteile

The humidity sensor delivers an analog signal that is read in via A0.

```
adc=machine.ADC(0)
fSchwelle=768
fCnt=-1
fState=0
```

The parameters for heating control have been added.

```
heater=machine.Pin(14,machine.Pin.OUT) # D5@esp8266
heater.value(0)
heaterState=0
heaterState==heater.value()
heizSchwelle=4
hstate=["aus","an "]
timeDelay=4
```

The boot block has also been expanded and adapted.

```
s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', 9003))
print("waiting on port 9003...")
s.settimeout(0.1)
d.writeAt("waiting on 9003",0,4)
readStatus()
heizen(heaterState,monitor)
```

The operation of the relay has been outsourced to a function in order to save memory space. The firmware of the ESP8266 called for this several times. Most of the comment lines also fell victim to this fact. The heating function takes the desired

status and the socket address of the recipient for the feedback and takes care of everything that is necessary.

So that the heaterState and auto operating parameters can be restored after a crash and restart (this happened from time to time), these states are written to the state.txt file after each change. From there they are read in when the program starts. At the very first start this file does not exist yet, so both parameters are set to 0. As soon as one of the parameter values is changed, the file is created.

```
def heizen(state,addr):
    global heaterState
    heater.value(state)
    d.writeAt("Heizung {}".format(hstate[state]),0,5)
    heaterState=heater.value()
    writeStatus()
    s.sendto("H;Heizung {}\n".format(hstate[state]),addr)

def writeStatus():
    with open("state.txt","w") as f:
        f.write("{}\n".format(heaterState))
        f.write("{}\n".format(auto))

def readStatus():
    global heaterState,auto
    try:
        with open("state.txt","r") as f:
            heaterState=int(f.readline())
            auto=int(f.readline())
        print(heaterState,auto)
    except:
        heaterstate=0
        auto=0

def feuchte(addr):
    global fCnt,fState
    f=adc.read()
    text=""
    if f >= fSchwelle:
        fState=1
        fCnt+=1
        if fCnt%86400==0:
            text="F;ZU TROCKEN!\n"
            fCnt=1
    elif fState==1:
        text="F;FEUCHTE OK!\n"
        fCnt=-1
        fState=0
    else:
        pass
    if text != "":
        s.sendto(text,addr)
        s.sendto(text,monitor)
```

The function for, or better against, dry soil has been added. Whenever the measured value f exceeds the set threshold, fState is set to 1 and a counter is increased. When the remainder of the division reaches the value 0, an alarm message is sent to the network device whose address was transferred in the addr parameter. fCnt is preset to -1 so that an alarm is given the first time the humidity limit is exceeded. So that this doesn't happen every second, the content of fCnt is counted up and the next alarm is only triggered after 86400 seconds, which corresponds to about a day. As soon as the value falls below the limit, fCnt is set back to the initial value -1. This is done by the elif branch, which reacts to fState = 1 and sends the all-clear. FState is reset to 0 so that the branch can only become active again after the next "too dry" message. If the initially empty string in text was assigned a message, this leads to the message being sent before the function is exited.

Before the boot sequence begins, you have the option of canceling the program with the button for "Heating off". The time duration for this is specified by the timeDelay variable, the value of which is set earlier in the program.

```
ledAn()
start = time()
end = start + timeDelay
currentTime=start
while currentTime < end:
    currentTime = time()
    if taste.value() == 0:
        sys.exit()
ledAus()
```

In the service loop, the sequences for switching the heating on and off have been replaced by the function calls. If one of the processes was initiated by radio, the "gardener" sends the response message back to the caller. However, if the control was carried out using the buttons on the device, the message about this is sent to the device whose address is stored in monitor. Each process generates a response, which is preceded by a key letter and a ";" begins. This makes it easier for the client to decode and assign the response.

Finally, the soil moisture is checked with each run and an alarm is triggered if necessary.

The "gardener" program can be ended by pressing both buttons. In this case, messages are sent and shown on the display, the heating is switched off because it is no longer monitored. The exit command terminates the program and reboot initiates a restart.

```
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("H;"+rec+"\n",adr)
        if rec=="autoOff":
            auto=0
            s.sendto("H;"+rec+"\n",adr)
        if rec=="getTemp":
            ds_sensor.convert_temp()
            s.sendto("M;started\n",adr)
        if rec=="sendTemp":
            print("Senden")
            boxtemp=sendTemperatur(adr)
            print(auto)
            if auto==1:
                sleep(1)
                try:
                    boxtemp=float(boxtemp)
                    print(boxtemp)
                    if boxtemp < heizSchwelle:
                        heizen(1,adr)
                    else:
                        heizen(0,adr)
                except:
                    print("failed")
        if rec=="heizenAn" :
            heizen(1,adr)
        if rec=="heizenAus":
            heizen(0,adr)
        if rec=="exit":
            heater.value(0)
            d.clearAll()
            d.writeAt("*** SHUT OFF ***",0,5)
            break
        if rec=="status":
            s.sendto("S;{}{}\n".format(heaterState,auto),adr)
        if rec=="reboot":
            s.sendto("M;REBOOTING\n",adr)
            machine.reset()
        rec=""
    except:
        if heaterOn.value()==0:
            heizen(1,monitor)
        if heaterOff.value()==0:
            heizen(0,monitor)
    feuchte(client)
    if taste.value()==0 and heaterOn.value()==0:
        d.writeAt("*** SHUT OFF ***",0,5)
        s.sendto("H;SHUT OFF",monitor)
        sleep(1)
        heater.value(0)
        sys.exit()
    blink(50,950,True)
```

Linux-Jobs

Basic Linux knowledge is required for the following procedure. Comprehensive descriptions and explanations of commands and their effects would go far beyond the intention of this blog post and would go beyond the scope of this article. Reference is made to corresponding documents from the UBUNTU community.

So I'm going to use the commands of the bash shell without describing their syntax and effect in any more detail or going into all conceivable switches and parameters. That is the task of a book introducing the Linux operating system.

I would like
1. Log in from the Windows machine to the Linux box via SSH and run, among other things, X11 window programs with a graphical user interface on the Windows machine
2. create a new user guardian
3. Create a UDP client for guardian, which can send commands to the "gardener", represented by the ESP8266, and receive data from it in order to write them into files
4. Write a CPython program that archives the gh_daten and messages files under guardian rights at the end of the day
5. Define a cron job for guardian that triggers the archiving of the daily data
6. Run a TCP web server that I want to write in Python and that can query the two files
7. In addition, the web server should be able to pass on commands to the UDP client, which sends them to the "gardener" and receives replies from him

Zu 1.

The connection to the Linux box / Raspi is created by the terminal program Putty, which, by the way, can also be used as a terminal for the connection to the ESP8266 / 32. Follow the Link and download the appropriate version for your system. If you log in directly to the Linux console, you can skip this and the next step.

If you want to dock with Linux from the Windows machine and want to use the graphical user interface there, XMing must be installed on the Windows machine.

First start XMing as administrator and then Putty. After starting putty, enter the IP of your Linux machine and then switch to Connection - SSH -X1. Check the box next to X11 forwarding. Go back to the Session category and save the settings under a "meaningful" name

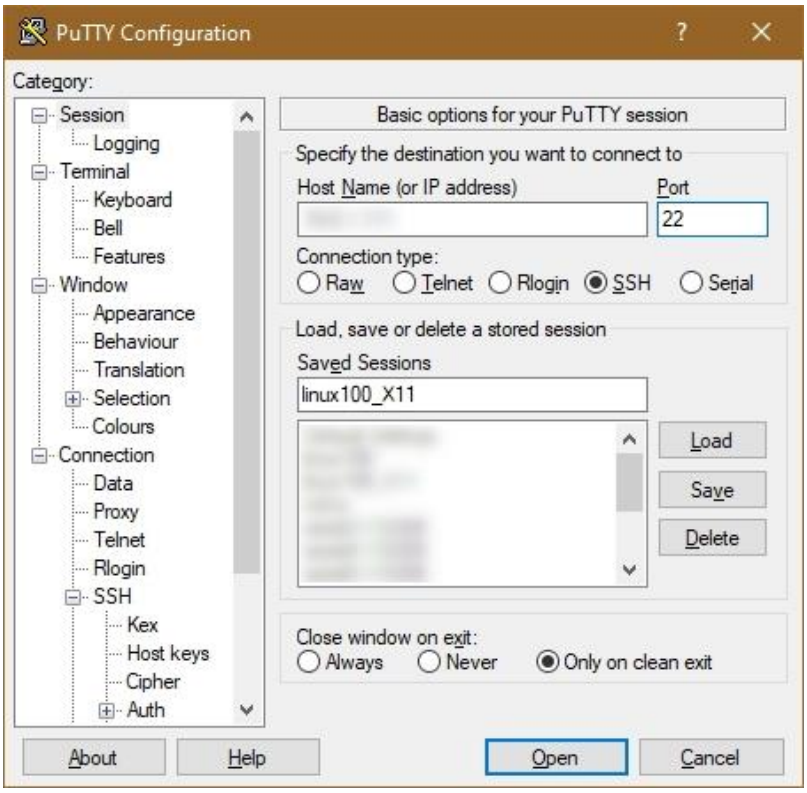


Abbildung 7: Putty_Session-Parameter

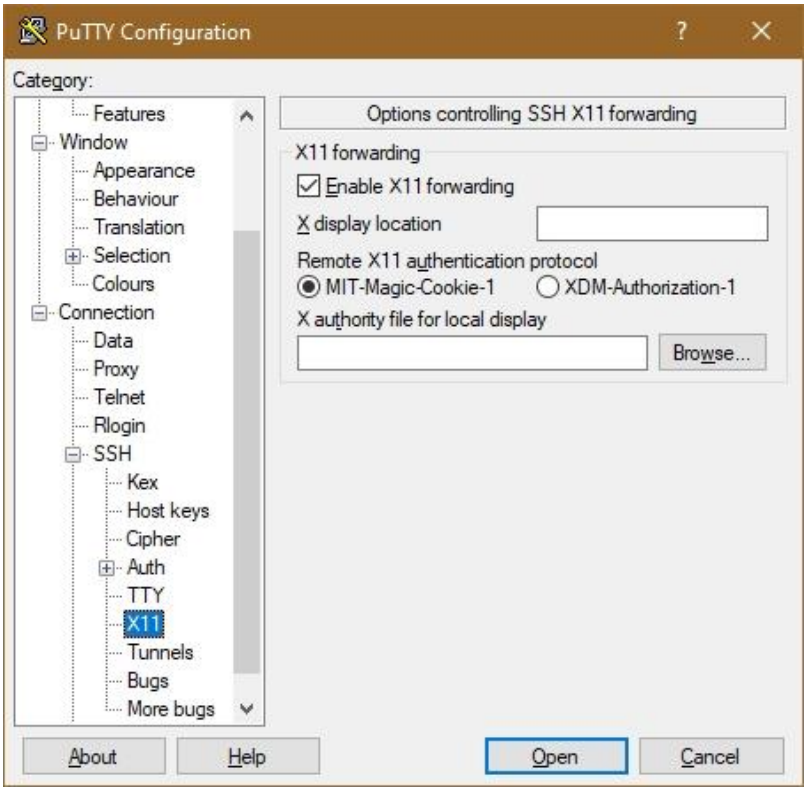


Abbildung 8: Putty_X11-Einstellungen

Now log in to the Linux machine via Putty (Open). For the further procedure you need administrator rights on the Linux machine!

Zu 2.

First of all, we create a new user whose account should run our applications. I have called him guardian, he is given a directory with the same name in the home folder and primarily belongs to the guardian group that is automatically set up. So that he has access to web services, we add him to the group www-Data.

sudo useradd -s / bin / bash -U -m -G www-data guardian

So that we can log in as guardian, the user needs a password, so we give him one.

sudo passwd guardian

Now it's your turn to come up with a password.

zu 3.

If this has not yet happened, we will now install Python on the Linux computer. Check that with the call

python3

Output: (on my 32-bit system, Python 3.5 is the last supported version)
Python 3.5.2 (default, Nov 12 2018, 1:43:14 PM)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>

If the answer does not look similar, it contains a hint as to what the call should be or how you can carry out the installation. Python 3.5 is the ultimate feel for a 32-bit system. Higher releases on 64-bit systems open up further functionalities that are not up for discussion in this context.

Next we need a decent development environment, for example idle, which can be operated in a similar way to Thonny. Call up the graphical user interface of idle under putty. Watch out! This is only possible from Windows if you have installed XMing as described under 1., started it as admin and trained Putty on it, or if you are working directly on the Linux console.

idle &

If the program does not start, it must also be installed first. In this case, too, your system will give you an indication of this. So that the putty window is free again after the call, add the "&" to the call; it sends the idle process to the background.

Now let's create a new program file using idle's Python shell.

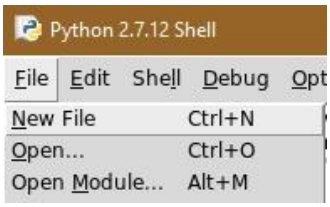


Abbildung 9: idle_NewFile

A new window opens and we can get started.

The first line contains the shebang, #! / Usr / bin / env python3.5. It indicates which interpreter is responsible for executing the following script and where it can be found. Even if Idle is running under Python 2.7, you can use it to edit programs of higher versions. When starting from the command line, the correct interpreter is called via the shebang. When it is called, the script is passed to it as a parameter. The execution flag must be set for the program file. We'll do that when we've created and saved the program.

By the way, we can generally use the same Python commands on the Linux computer as on the ESP8266. What's more, Linux runs CPython, which has a significantly larger range of languages. However, the reverse can also apply. The time module does not offer various methods under CPython that we are used to from MicroPython. There are also no hardware timers available.

What we usually do not need on the Linux computer is the network login, provided the Linux box is connected to the network via a cable interface, which I now assume. So - get to work!

So that our program can receive data asynchronously from the "gardener" or from the mobile phone via the web server, it has to run constantly in the background. It should query the temperatures at (somewhat) the same time intervals. We could do this using a time loop that replaces the hardware timer of the ESP8266. However, this involves considerable inaccuracies.

More precisely, this would be done by a cron job that would clock the query with the accuracy of the system time. However, if the program does not run permanently, the constant readiness to receive would no longer apply.

It would be like this if the cronjob client9001.py started directly. But there is also the network-based exchange of messages between programs via UDP / TCP, and that also works within the computer. So we write a short program that is called by the crontab and that tells the client9001.py that it should now start a query. Let's first look at the client program itself.

Download client9001.py

```
#! /usr/bin/python3.5
import socket
import sys,os
from time import sleep,strftime,time  # A
from subprocess import check_output

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
s.bind(('', 9001))  # B
target=("10.0.1.180",9003)
print("Socket established, waiting...")
s.settimeout(0.1) # timeout, damit 'while 1:' durchlaeuft # C
#intervall=10*60  # 10 Minuten = 600 Sekunden
getFlag=0
save=False
einlesen=True  # D

def TimeOut(t):  # E
    start=time()
    def compare():
        return int(time()-start) >= t
    return compare

while 1:
    z=None
    datei=""
    antwort=""  # F
    if einlesen:
        einlesen=False
        try:
            print("Scan Anfordern")
            s.sendto("getTemp\n".encode(),target)
            getFlag=1
            readTimeOut=TimeOut(2)
        except:
            print("sending getTemp timed out!")


    if getFlag==1 and readTimeOut():  # G
        getFlag=0
        try:
            print("Sendung anfordern")
            s.sendto("sendTemp\n".encode(),target)
            save=True
        except:
            print("sending sendTemp timed out!")

    try:  # H
        antwort,adr=s.recvfrom(250)
        antwort=(antwort.decode()).replace(".",",")
```

```
        print("Antwort1:",antwort[0],antwort[2:],adr)
        typ=antwort[0]
        z=strftime("%d.%m.%Y %H:%M;")+antwort[2:]
        if typ=="T":
            datei="daten-gh"
            save=True
        elif typ =="A":  # J
            z=""
            save=False
            einlesen=True
        elif typ =="H" or typ =="F" or typ =="S":
            print("Antwort2:",antwort[0],antwort[2:],adr)
            datei="messages"
            save=True
        elif typ =="B":  # K
            try:
                befehl=antwort[2:].strip("\r\n")+"\n"
                s.sendto(befehl.encode(),target)
                z=None
                sleep(1)
                try:
                    reply,ADR=s.recvfrom(250)
                    s.sendto(reply,adr)
                    datei="messages"
                    z=strftime("%d.%m.%Y
%H:%M;")+(reply.decode())[2:]
                    save=True
                except:
                    s.sendto("M;DONE\n".encode(),adr)
            except:
                print("Befehl {}
fehlgeschlagen:{}".format(befehl,e))
        elif typ == "R":  # L
            print("R-Befehl angekommen")
            temperatur=check_output(["tail", "-1",
"arch/daten-gh"]).decode().strip("\n")
            s.sendto(temperatur.encode(),adr)
            print("gesendet {} an {}:".format(temperatur,adr))
            z=None
        elif typ == "N":  # M
            temperatur=check_output(["tail", "-1",
"arch/daten-gh"]).decode().strip("\n")
            nachrichten=check_output(["tail", "-10",
"arch/messages"]).decode().strip("\n")
            print(temperatur)
            for mesg in nachrichten.strip("\n").split("\n"):
                print(mesg)
            kombi=temperatur+"_"+nachrichten
            s.sendto(kombi.encode(),adr)
            z=None
        else:
            print(z)
```

```
            z = None
        print("zeile:",z,datei,save)
    except:
        if antwort:
            print("Fehler beim Dekodieren",antwort)
        pass # Es liegt keine wichtigen Nachricht vor

    try:
        if (z is not None) and save:  # N
            print("schreiben",z)
            datei="/home/guardian/arch/"+datei
            print("Datei",datei)
            save=False
            with open(datei,"a") as f:
                f.write(z)
    except:
        print("data write error")
```

(A)
The strftime () method is interesting for import businesses, as it can create a string with the appropriate parameters from the system time.

(B)
We build a UDP socket and bind to port number 9001.

(C)
A short timeout ensures that the loop runs smoothly.

(D)
The control parameters for writing to the files and reading in the values from the ESP8266 are set.

(E)
TimeOut () creates a closure, compare (), which is a simple, flexible time delay. I have described exactly what a closure is and how to work with it in the document Closures and Decorators.pdf. Here, the TimeOut () function is given a time interval in seconds, the expiry of which is monitored by Closure compare (). Put simply, a closure is a function that can remember the value of its local variable between two calls. Here that is the time limit.

(F)
Further control parameters are reset for each loop pass. The first if query tests the read in Boolean variable to determine whether the ESP8266 should query the temperature values. If so, reading is reset and the order is sent. The timer for reading in is armed with 2 seconds. The sending process is secured by try and except.

(G)
After 2 seconds the data should be available. We send the command for transfer. and prepare to write to the file: save = True.

(H)
Any inquiries and answers will then be dealt with. You have the shape

X; command / response

to / from the ESP8266 and other network devices. UDP enables an open exchange of messages between all participating devices.

The first character X stands for the type of payload that follows from the 3rd character and indicates how to proceed with it.
A; Crontab has sent the order for reading in temperature values; read in is set to True (J)
H; A response from the ESP8266 to a heating-related command
M; an unspecific message from the ESP8266 of minor importance
F; an asynchronous message from the ESP8266 on soil moisture
T; received temperature message
B; Command prefix from external device or the web server (K)
R Send the temperatures to the corresponding client (L)
N Send the temperatures and the last 5 messages to the corresponding client (M)
To do this, we use the check_output () method from the subprocess module, which makes the output of bash commands usable in Python.

(N)
If z contains a message to be backed up and the backup has been arranged (save = True), the corresponding file is now written to, which is located in the / home / guardian / arch directory of the user guardian. The archived daily files also end up there.

We start the program client9001.py as user guardian from his home directory as follows, after we have set the execution flag for guardian and his group.

guardian @ ganymed: ~ $ ls -lias client *

2119841 4 -rw-rw-r-- 1 guardian guardian 1977 Oct 24 18:00 client9001.py

guardian @ ganymed: ~ $ chmod 774 client9001.py
guardian @ ganymed: ~ $ ls -lias client *

2119841 4 -rwxrwxr-- 1 guardian guardian 1977 Oct 24 18:00 client9001.py

Now we can start the file. We do this in such a way that the process continues even when we exit the terminal window.

guardian @ ganymed: ~ $ nohup ./client9001.py &

Output:
[1] 25646
guardian @ ganymed: ~ $ nohup: ignore input and append output to 'nohup.out'

Then we check whether the process is running and what PID (process ID) it has.

guardian @ ganymed: ~ $ ps aux | grep client9001

guardian 25646 0.1 0.3 13288 7416 pts / 1 S 11:00 0:00 /usr/bin/python3.5
./client9001.py
guardian 25649 0.0 0.0 6352 836 pts / 1 S + 11:01 0:00 grep --color = auto
client9001

The number in bold is the PID we need to end the process again. This is done by the following command on the console.

guardian @ ganymed: ~ $ kill 25646

If no error messages appear after starting client9001.py, we create the two day files daten_gh and messages.

guardian @ ganymed: ~ $ mkdir arch
guardian @ ganymed: ~ $ cd arch
guardian @ ganymed: ~ / arch $ touch data_gh
guardian @ ganymed: ~ / arch $ touch messages
guardian @ ganymed: ~ / arch $ ls -lisa

-rw-rw-r-- 1 guardian guardian 0 Oct 23 19:33 data-gh
-rw-rw-r-- 1 guardian guardian 0 Oct 23 19:33 messages

guardian @ ganymed: ~ / arch $ cd ..

The program that Crontab uses to issue the order to read in the values is still missing. Interprocess communication allows us to send messages from one program to another. After the program has been saved, the execution flag for user and group must be set as with client9001.py.

converttemp.py

```
#! /usr/bin/python3.5
import socket
import sys,os
from time import sleep,strftime,time
IPA="10.0.1.100"
portNumA=9005  #
print("Fordere Client-Socket an")
client = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
client.setsockopt(socket.SOL_SOCKET,socket.SO_REUSEADDR,1)
client.settimeout(2)
client.bind(('', portNumA))    # an lokale IP und Portnummer
binden
print("Sende Anfragen auf {}:{}".format(IPA,portNumA))
# ****************** Ziel: Klima-Agent client9001.py
targetPort=9001
target=("10.0.1.100",targetPort) # UDP-Client auf Linux100

print("Temperaturen anfordern")
client.sendto("A;einlesen\n".encode(),target)
sleep(1)
```

```
antwort,adr=client.recvfrom(256)
reply=antwort.decode().strip("\n")
print(reply)
client.close()
```

Zu 4.

The next program archivate.py must be started shortly before midnight on schedule. It copies the daily files to the daily archives and submits new, empty daily files after the old ones have been deleted. To do this, it uses bash commands that are called using the os.system () method. rm (remove) removes files, cp (copy) copies and tac lists a file from the last to the first line. When copying, the -a switch ensures that the file rights of the source are also transferred. After saving, we of course also set the execution flags for archivate.py.

Download archivate.py

```
#!/usr/bin/python3.5
#
# Dieses Script archiviert die Temperaturdaten eines Tages
# sowie die wichtigen Meldungen
# und legt für 00:00 neue leere Dateien vor.
#
import sys,os,time
# Datum ermitteln
nameTemp=time.strftime("ghaus-%Y-%m-%d")
nameMesg=time.strftime("mesg-%Y-%m-%d")
# Tageswerte archivieren
try:
    os.system ("rm /home/guardian/arch/{}".format(nameTemp))
except:
    pass
try:
    os.system ("rm /home/guardian/arch/{}".format(nameMesg))
except:
    pass
os.system ("tac /home/guardian/arch/daten-gh > \
           /home/guardian/arch/{}".format(nameTemp))
os.system ("rm /home/guardian/arch/daten-gh")
os.system ("tac /home/guardian/arch/messages > \
           /home/guardian/arch/{}".format(nameMesg))
os.system ("rm /home/guardian/arch/messages")
# leere Tagesdatei vorlegen
os.system ("cp -a messages_template \
            /home/guardian/arch/messages")
os.system ("cp -a daten_template \
            /home/guardian/arch/daten-gh")
```

Now is the time to test the entire system. If you have not yet done this, please send the program xmitter2.py as boot.py to the ESP8266 as described in the chapter "The software - Autostart", and then restart the controller (reset). Start client9001.py in the

background from the command line in Linux. We can emulate an external device with the help of packetsender.

Abbildung 10: Clienttest

Time	From IP	From Port	To IP	To Port	Method	Error	ASCII	Hex
13:48:21.519	10.0.1....	9001	You	9181	UDP		DONE\n	44 4F 4E 45 0A
13:48:20.368	You	9181	10.0....	9001	UDP		B;autoOff	42 3b 61 75 74 6f 4f 66 66

Abbildung 11: Clienttest_Antwort

In addition, the files daten_gh and messages should now have grown. What is expressed in the file size.

guardian @ ganymed: ~ $ ls -lisa arch
i
2122500 4 drwxrwxr-x 2 guardian guardian 4096 Oct 24 23:58.
2100799 4 drwxr-xr-x 17 guardian guardian 4096 Oct 25 07:47 ..
2122506 4 -rw-rw-r-- 1 guardian guardian 1323 Oct 25 13:57 data-gh
2119836 4 -rw-rw-r-- 1 guardian guardian 286 Oct 25 13:48 messages

We can also look at the content directly.

guardian @ ganymed: ~ $ cat arch / data-gh
10/24/2021 10:25; 19.3; 19.4; 19.1
10/24/2021 10:26 AM; 19.2; 19.2; 19.1
10/24/2021 10:27 AM; 19.3; 19.2; 19.1
10/24/2021 10:29 AM; 19.2; 19.3; 19.0
10/24/2021 10:30 AM; 19.3; 19.4; 19.1

We start the archiving process manually.
guardian @ ganymed: ~ $ ls -lisa arch
-rw-rw-r-- 1 guardian guardian 0 Oct 23 19:33 data-gh
-rw-rw-r-- 1 guardian guardian 2107 Oct 25 14:05 ghaus-2021-10-25
-rw-rw-r-- 1 guardian guardian 286 Oct 25 14:05 mesg-2021-10-25
-rw-rw-r-- 1 guardian guardian 0 Oct 23 19:33 messages

to 5.
Works as expected. So let's activate the cron job.

guardian @ ganymed: ~ $ crontab -e

The standard editor opens. If you have never called ctrontab -e before, you can now first choose one of the installed editors. Then the list of cron jobs appears. Archivate.py should be called at 11:58 p.m. every day, every month and every weekday. Each line stands for exactly one order for the cron-Deamon, the master of time. The penultimate line is the timer for requesting the temperature values, which should take place every 10 minutes.

```
... ..
# For more information see the manual pages of crontab (5) and cron (8)
#
# m h dom mon dow command
* / 10 * * * * /home/guardian/converttemp.py
58 23 * * * /home/guardian/archivate.py
```

After everything has been entered and checked, we save and exit the editor.

In the terminal window, the data-gh file should now receive a new entry every 10 minutes. We check this with the following command.

guardian @ ganymed: ~ $ cat arch / data-gh

Output:

```
11/3/2021 10:48 AM; 17.88; 18.75; 18.19
11/3/2021 10:58 AM; 17.88; 18.75; 18.19
... ..
11/3/2021 12:28 PM; 18.06; 19.00; 18.38
11/3/2021 12:38 PM; 18.06; 19.00; 18.38
11/3/2021 12:48 PM; 18.06; 19.00; 18.38
```

With every command to the ESP8266 and every change in humidity, the messages file also grows.

Wrapping up

What was new in this post? Let's summarize:

- Extension of the hardware with a soil moisture sensor
- Interference suppression of the ESP8266 through a capacitor at the RST input
- The carrier boards for the power supply units and the ESP8266 make assembly easier
- We recognize that features can help save storage space
- Cronjobs enable us to precisely time control
- We have implemented a software timer with a closure
- Sending messages via UDP allows us to communicate between separate processes on one computer

Outlook

The only thing missing now is the web server for the long distance traffic. Because of the non-trivial length of 170 lines, I put you off to the next but one episode of the "Gardener" series, because there is also a lot to explain. And as a delicacy, there is

also an app for the mobile phone that can directly access the ESP8266 and also receive and implement alarm messages asynchronously. To make it easier for you to tinker with the app, the next episode will first deal with the use of the MIT Appinventor2. This is a web-based tool with the help of which we can program apps for Android mobile phones or iPhones by layering "building blocks".