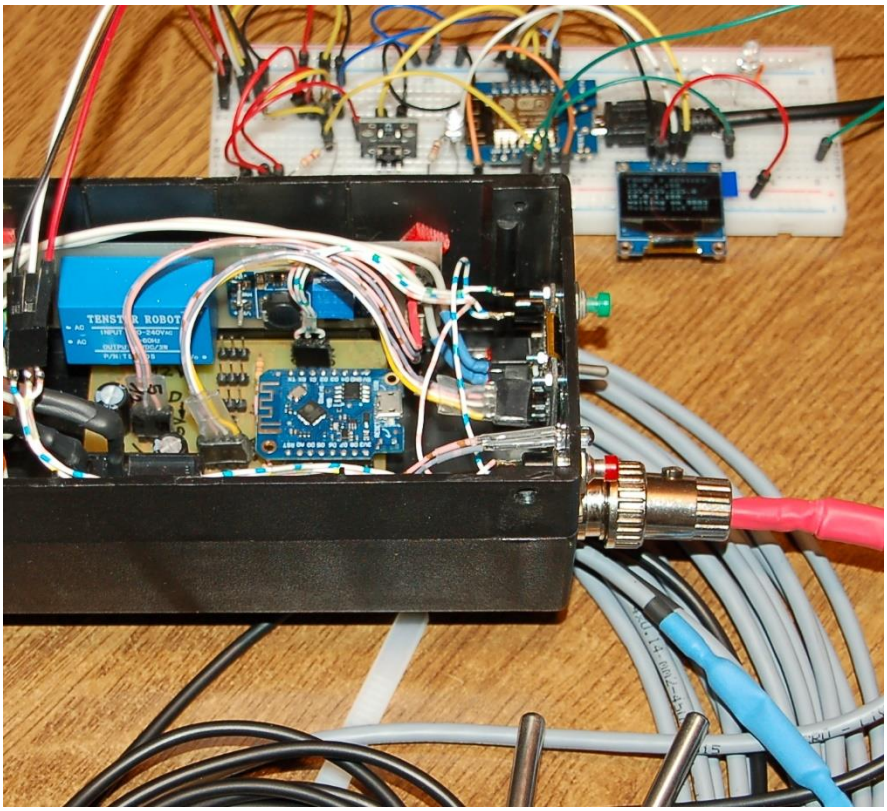


Abbildung 1: Produktivsystem und Testaufbau

So sieht das Innere meines Fertiggeräts aus. Im Hintergrund ist der Testaufbau zu sehen. Ich bin gerade beim Zuordnen der drei DS18B20-Sensoren (im Vordergrund) der Produktionsversion zu ihren Dienstorten. Im Stecker, rechts unten am Gehäuse, werden die drei einzelnen Kabel zum Bus zusammengeführt. Im ersten Teil des Frostwächters ist das Vorgehen genau beschrieben.

Heute werden wir dem Gerät noch einige Baugruppen hinzufügen und die dienstbaren Geister auf der Linuxmaschine wecken. Willkommen zu

Freeze Guardian - WLAN-Gärtner Teil2

Bleiben wir gleich bei den Erweiterungen der Hardware.

Hardware

Um die Heizung vor Ort schalten zu können, gibt es zwei Taster, deren Betätigung vom ESP8266 registriert und in einen Schaltvorgang des Relais umgesetzt werden. Ein externer Pullup-Widerstand ist nicht erforderlich, weil der interne softwaremäßig eingeschaltet wird.

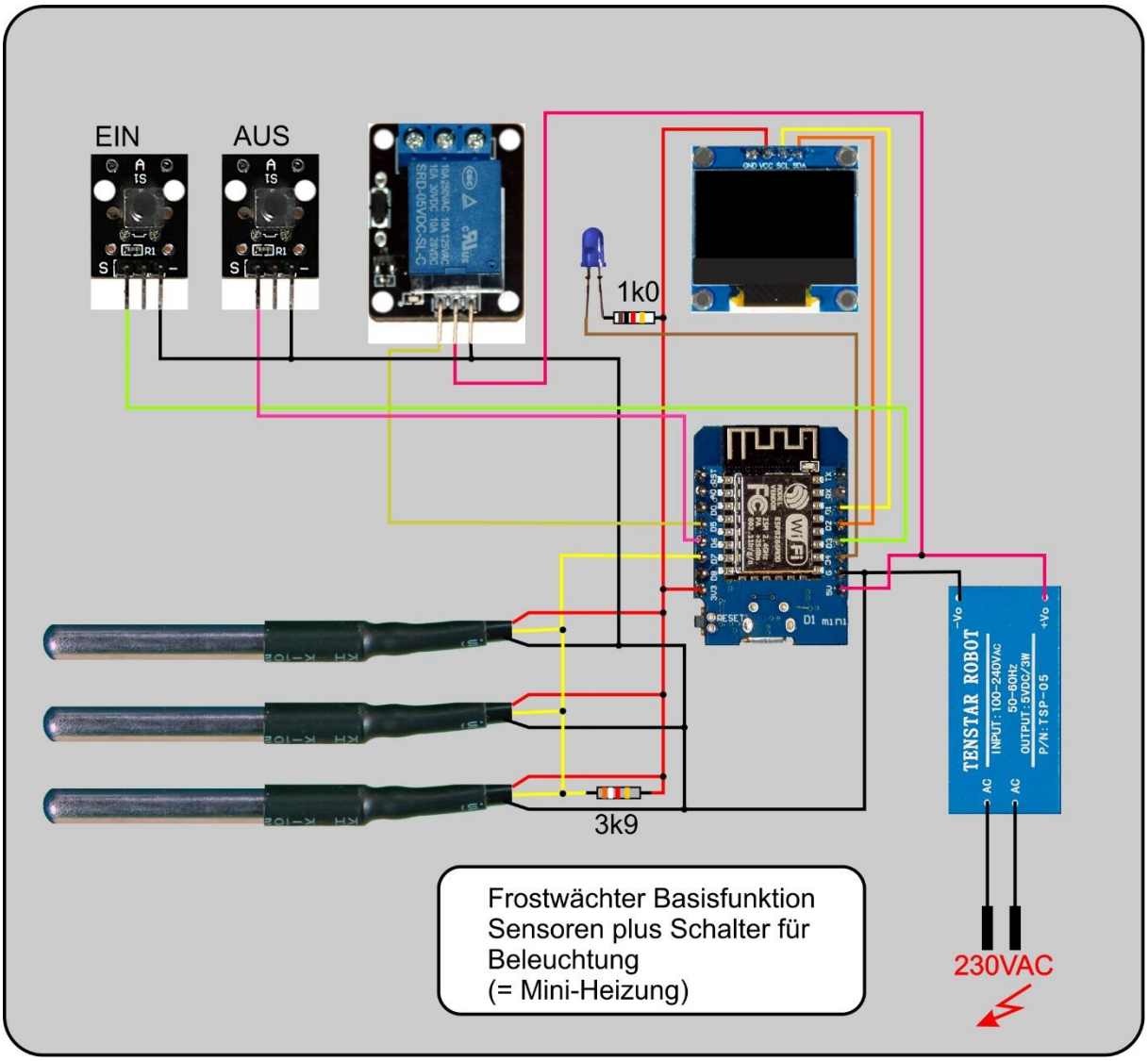


Abbildung 2: Basisschaltung plus Taster

Mit der einfachen Relaisstufe konnte ich den Ölradiator, der das innere Gemach des Treibhauses auf frostfreien Temperaturen halten soll, nicht bedienen. Also musste ein leistungsfähigeres Relais her. Das habe ich in einer ausgedienten Waschmaschine gefunden. Dort hat es auch den Heizkreis gesteuert, also ideal für diesen Zweck! Die Spulenspannung 12V konnte das kleine Netzteil von TENSTAR nicht liefern, weshalb ich einen Boost-Converter, (aka Step-Up-Wandler) in Dienst stellte. Das erschien mir einfacher und billiger, als im Treibhaus eine zusätzliche Feuchtraum-Steckdose für ein Steckernetzteil zu installieren. 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.

Für die Ansteuerung dient ein BC337 (NPN-Transistor), der in der Sättigung für genügend Stromstärke durch die Relaisspule sorgen kann. Der Basiswiderstand verringert die Baisstromstärke auf ein Minimum. Die (Freilauf-)Diode zwickt

Spannungsspitzen, die beim Ausschalten des Transistors an der Spule auftreten und diesen gefährden können.

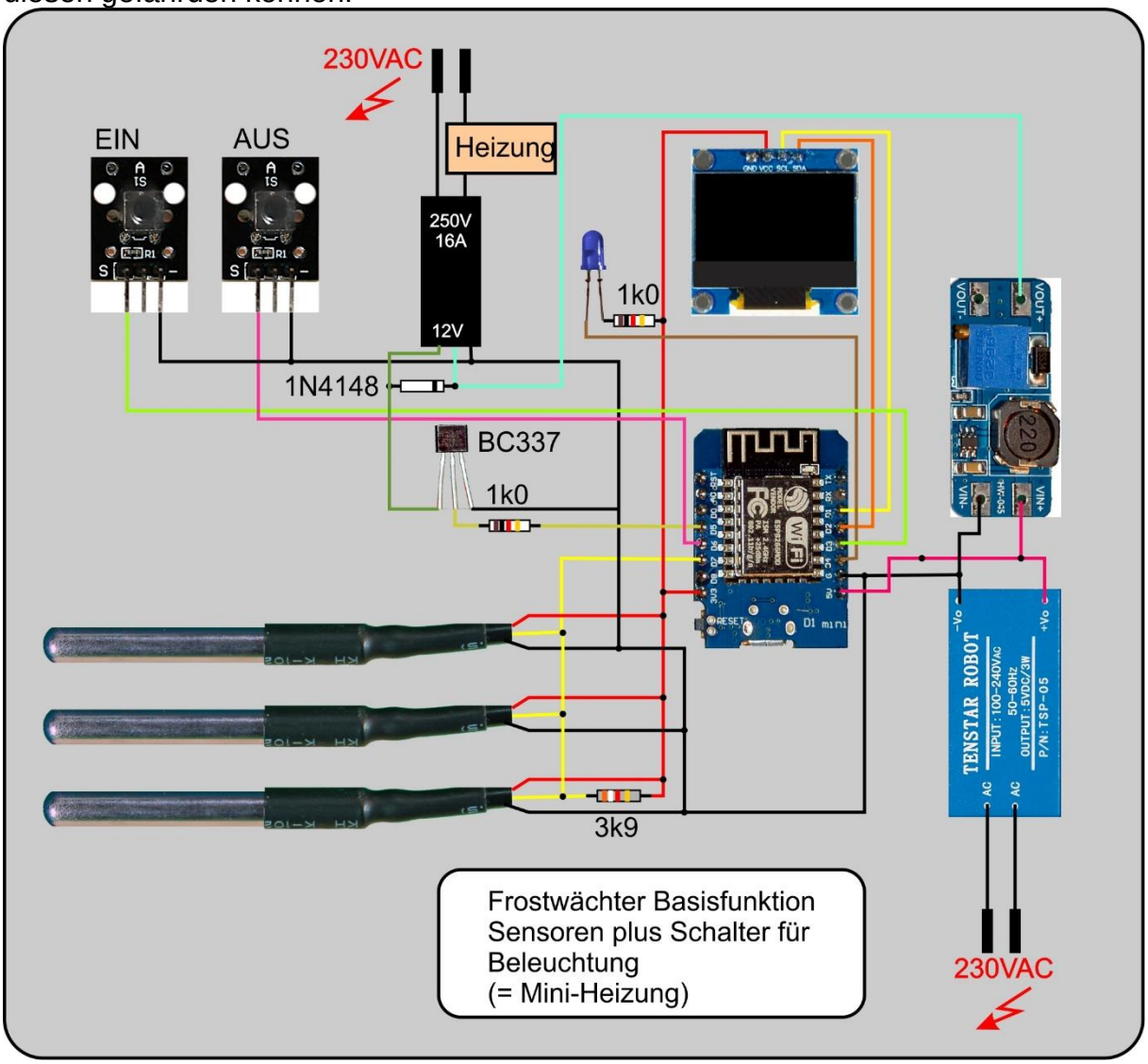


Abbildung 3: Basisschaltung plus Taster,Hochlastrelais

In der dritten Ausbaustufe habe ich noch einen Bodenfeuchtesensor spendiert, der über den analogen Eingang A0 angeschlossen wird. Werden mehr als 768 counts von 1023 gemessen, dann ist das Substrat zu trocken und der ESP8266 soll die Alarmglocken schrillen lassen. Der Schwellenwert kann natürlich im Programm an die Erfordernisse vor Ort angepasst werden.

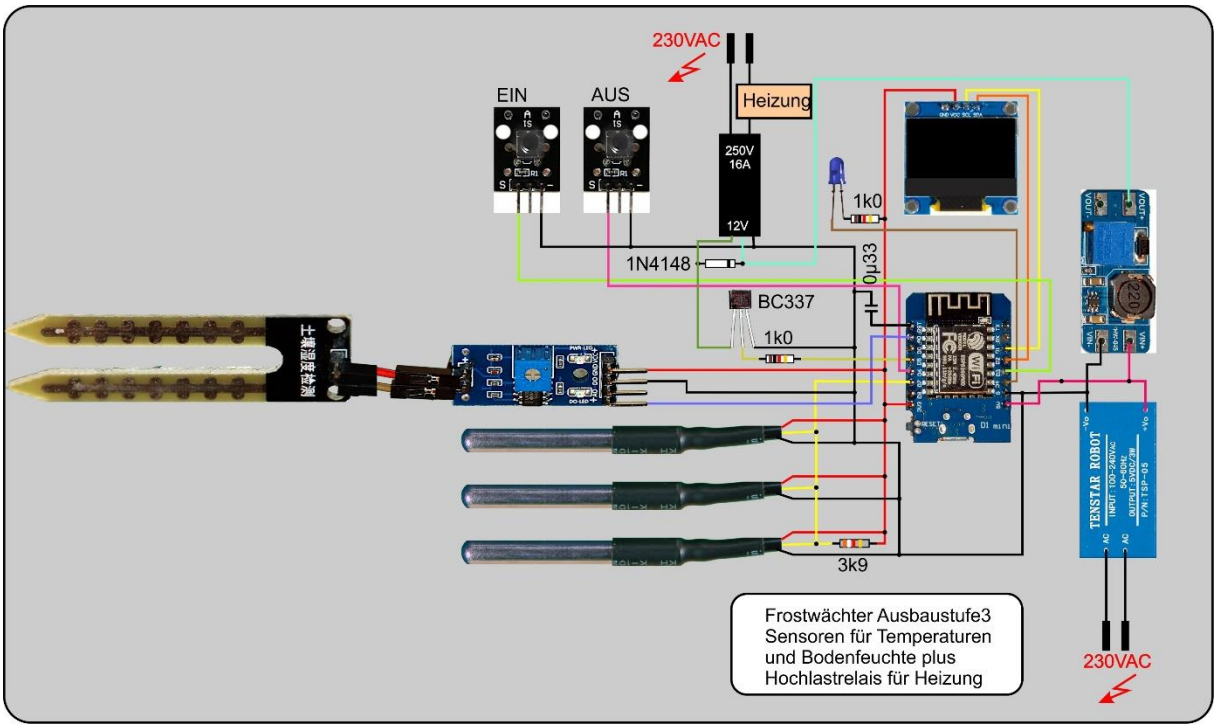


Abbildung 4: Schaltung komplett mit Feuchtefühler

Einen wichtigen Tipp will ich an dieser Stelle nicht verschweigen. Hin und wieder stürzte der ESP8266 aus unersichtlichen Gründen ab. Das ist nicht besonders schlimm, weil danach automatisch der Neustart durchläuft, aber es ist nervig, wenn etwas nicht so funktioniert wie es soll. Zum Beispiel löste das Telefonieren mit dem schnurlosen Telefon gerne Neustarts aus, auch auf einige Entfernung. Seit dem Einbau eines Kondensators von 330nF zwischen RST-Pin und GND ist das Problem gelöst und der ESP8266 läuft ohne Unterbrechung durch, wie ein Uhrwerk – tagelang … Im Schaltbild der Abbildung 4 ist dieser Kondensator bereit eingezeichnet.

In der Zusammenfassung sieht das nun so aus. Aufgeführt sind nur die Teile, die im Vergleich zur Vorgängerversion dazugekommen sind.

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

Zur bequemeren Montage der verschiedenen Module habe ich zwei Trägerplatinen entwickelt. Die erste davon habe ich bereits im letzten Beitrag vorgestellt. Sie fasst Netzteil und auch schon den Boost-Converter zusammen.

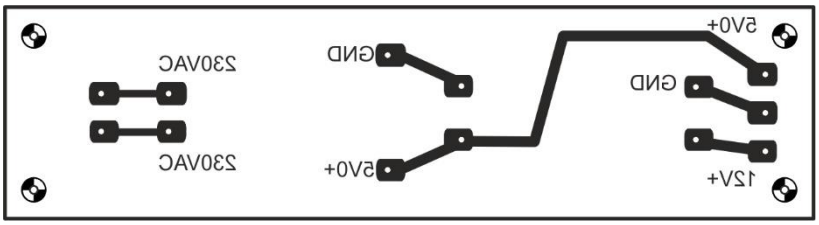
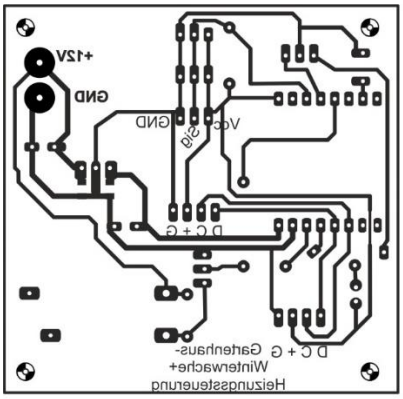
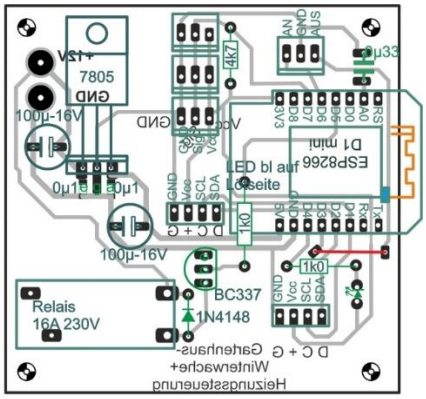


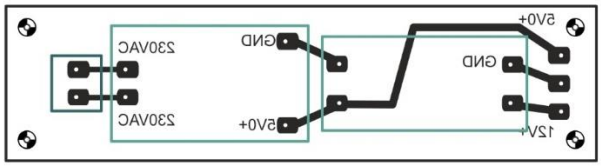
Abbildung 5: Trägerplatine

Das zweite PCB nimmt den ESP8266 D1 mini, das Lastrelais, die Transistor-Treiberstufe und die Steckkontakte zur Peripherie auf. Beide Layouts können als PDF-Datei heruntergeladen werden.

Anstatt des Linearreglers LM7805 werden in dieser Realisierung der Schaltung drei Pfostenstifte (Ø 1,3mm) bestückt und mit den entsprechenden Anschlüssen der anderen Trägerplatine verbunden. Die Reihenfolge von links ist: 12V – GND – 5V. Der Laderegler wäre für den Fall gedacht, dass eine 12V-Quelle zur Hand ist. Er würde dann die 5V zur Verfügung stellen und den Einsatz von Tenstar-Netzteil und Boost-Converter erübrigen.







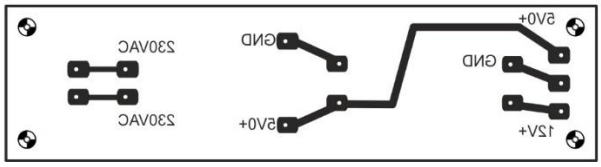


Abbildung 6: Trägerplatinen – verkleinerte Darstellung!

Die Hardware-Erweiterungen haben natürlich auch für zusätzlichen Softwareaufwand gesorgt. Während die Werkzeuge dieselben blieben, gibt es ein neues Betriebsprogramm für den "Gärtner".

Die Software

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

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
Hinweis: Hinsichtlich der I2C-Schnittstelle können abweichende Initialisierungs-Riten auftreten.

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 - 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.

Neue Programmteile

Der Feuchtesensor liefert ein analoges Signal, das über A0 eingelesen wird.

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

Die Parameter für die Heizungssteuerung wurden ergänzt.

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

Auch der Boot-Block wurde erweitert und angepasst.

```
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)
```

Die Bedienung des Relais wurde in eine Funktion ausgelagert um Speicherplatz zu sparen. Die Firmware des ESP8266 rief mehrmals dazu auf. Diesem Umstand fielen auch die meisten Kommentarzeilen zum Opfer. Die Funktion **heizen** nimmt den gewünschten Status und die Socketadresse des Empfängers für die Rückmeldung und erledigt alles Notwendige.

Damit nach einem Absturz und Neustart (das kam hin und wieder vor) die Betriebsparameter **heaterState** und **auto** restauriert werden können, werden diese Zustände nach jeder Änderung in die Datei **state.txt** geschrieben. Von dort werden sie beim Programmstart eingelesen. Beim allerersten Start ist diese Datei noch nicht vorhanden, deshalb werden beide Parameter auf 0 gesetzt. Sobald einer der Parameterwerte geändert wird, wird die Datei angelegt.

```
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)
```

Dazugekommen ist die Funktion für, oder besser gegen, trockenen Boden. Immer, wenn der Messwert **f** die eingestellte Schwelle übersteigt, wird **fState** auf 1 gesetzt und ein Zähler erhöht. Wenn der Teilungsrest den Wert 0 erreicht, wird eine Alarmmeldung an das Netzwerkgerät gesandt, dessen Adresse im Parameter **addr** übergeben wurde. **fCnt** ist mit -1 vorbelegt, sodass bereits beim ersten Überschreiten des Feuchtegrenzwerts Alarm gegeben wird. Damit das nun nicht jede Sekunde wieder passiert, wird der Inhalt von **fCnt** hochgezählt und erst nach 86400 Sekunden, das entspricht etwa einem Tag, der nächste Alarm ausgelöst. Sobald der Grenzwert unterschritten wird, wird **fCnt** wieder auf den Ausgangswert -1 gesetzt. Das macht der **elif**-Zweig, der auf **fState**=1 reagiert und Entwarnung sendet. Damit der Zweig erst nach der nächsten "**zu trocken**"-Meldung wieder aktiv werden kann, wird **fState** auf 0 zurückgesetzt. Wurde der anfangs leere String in **text** mit einer Meldung belegt, führt das vor dem Verlassen der Funktion zum Versenden der Nachricht.

Bevor die Bootsequenz beginnt hat man die Möglichkeit, das Programm mit der Taste für" **Heizung aus**" abzubrechen. Die Zeitdauer dafür gibt die Variable **timeDelay** vor, deren Wert weiter oben im Programm gesetzt wird.

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

In der Dienstschleife wurden die Sequenzen zum Ein- und Ausschalten der Heizung durch die Funktionsaufrufe ersetzt. Wurde einer der Vorgänge über Funk initiiert, dann sendet der "Gärtner" die Reaktionsmeldung an den Aufrufer zurück. Erfolgte die Steuerung aber über die Tasten am Gerät, dann geht die Meldung darüber an das Gerät, dessen Adresse in **monitor** abgelegt ist. Jeder Vorgang erzeugt eine Rückmeldung, die mit einem Schlüssel-Buchstaben und einem folgenden ";" beginnt. Das erleichtert dem Auftraggeber die Dekodierung und Zuordnung der Antwort.

Schließlich wird mit jedem Durchlauf die Bodenfeuchte überprüft und gegebenenfalls ein Alarm abgesetzt.

Das Programm des "Gärtners" kann durch Drücken beider Tasten beendet werden. In diesem Fall werden Meldungen versandt und am Display ausgegeben, die Heizung wird ausgeschaltet, weil sie nicht mehr überwacht wird. Durch den Befehl **exit** wird das Programm beendet und mit **reboot** ein Neustart eingeleitet.

```
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

Für das folgende Vorgehen werden grundlegende Linuxkenntnisse vorausgesetzt. Umfassende Beschreibungen und Erklärungen von Befehlen und deren Auswirkungen würden weit über die Intention dieses Blogbeitrags hinausgehen und den Rahmen sprengen. Auf entsprechende Dokumente der UBUNTU-Community wird verwiesen.

Ich werde also die Befehle der bash-Shell benutzen, ohne deren Syntax und Wirkung näher zu beschreiben oder auf alle denkbaren Schalter und Parameter einzugehen. Das ist Aufgabe eines Buches zur Einführung in das Betriebssystem Linux.

Ich möchte

1. von der Windowsmaschine aus auf der Linuxkiste über SSH einloggen und dort unter anderem X11-Window-Programme mit grafischer Oberfläche auf der Windows-Maschine ausführen
2. einen neuen User guardian anlegen
3. einen UDP-Client für guardian erstellen, der Befehle an den "Gärtner", vertreten durch den ESP8266, senden und Daten von diesem empfangen kann, um sie in Dateien zu schreiben
4. ein CPython-Programm schreiben, das unter guardian-Rechten am Ende des Tages die Dateien gh_daten und messages archiviert
5. für guardian einen Cronjob definieren, der die Archivierung der Tagesdaten auslöst
6. einen TCP-Webserver laufen lassen, den ich in Python schreiben will und der die zwei Dateien abfragen kann
7. außerdem soll der Webserver Befehle an den UDP-Client weitergeben können, der diese an den "Gärtner" sendet und Antworten von diesem erhält

Zu 1.

Die Verbindung zur Linuxkiste/Raspi schafft das Terminalprogramm Putty, das man übrigens auch als Terminal für die Verbindung zum ESP8266/32 verwenden kann. Folgen Sie dem Link und laden Sie die entsprechende Version für Ihr System herunter. Wenn Sie direkt an der Linuxconsole einloggen, können Sie diesen und den nächsten Schritt übergehen.

Wenn Sie von der Winowsmaschine aus bei Linux andocken möchten, und dort die grafische Oberfläche nutzen wollen muss auf der Windowsmaschine XMing installiert sein.

Starten Sie zuerst **als Administrator XMing** und danach **Putty**. Geben Sie nach dem Start von putty die IP Ihrer Linuxmaschine ein und wechseln Sie dann nach

Connection – SSH -X1. Setzen Sie dort den Haken bei **X11-Forwarding**. Gehen Sie zurück zur Kategorie **Session** und speichern Sie die Einstellungen unter einem "sprechenden" Namen ab.

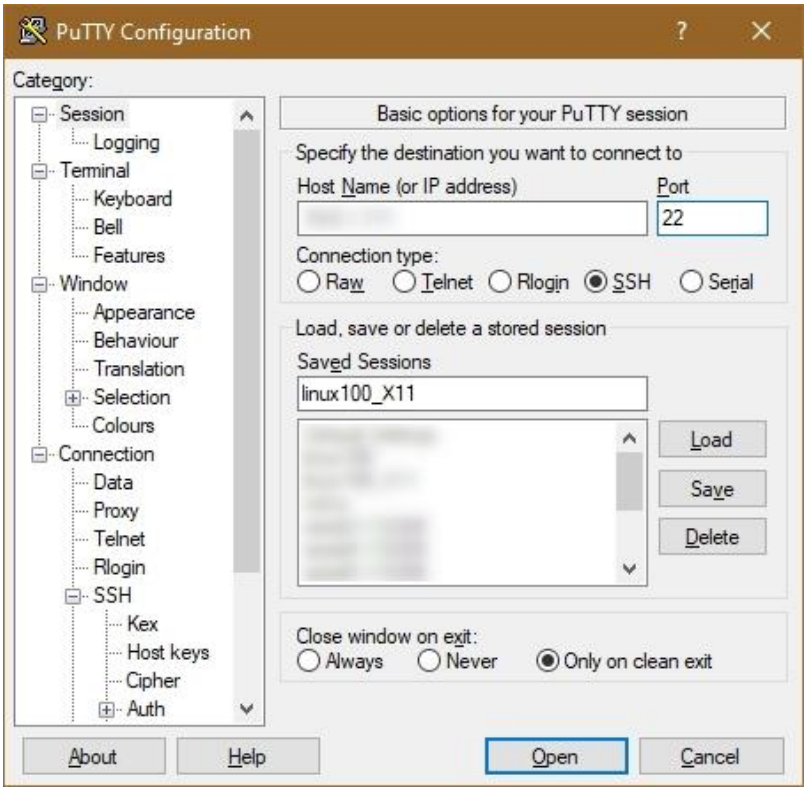


Abbildung 7: Putty_Session-Parameter

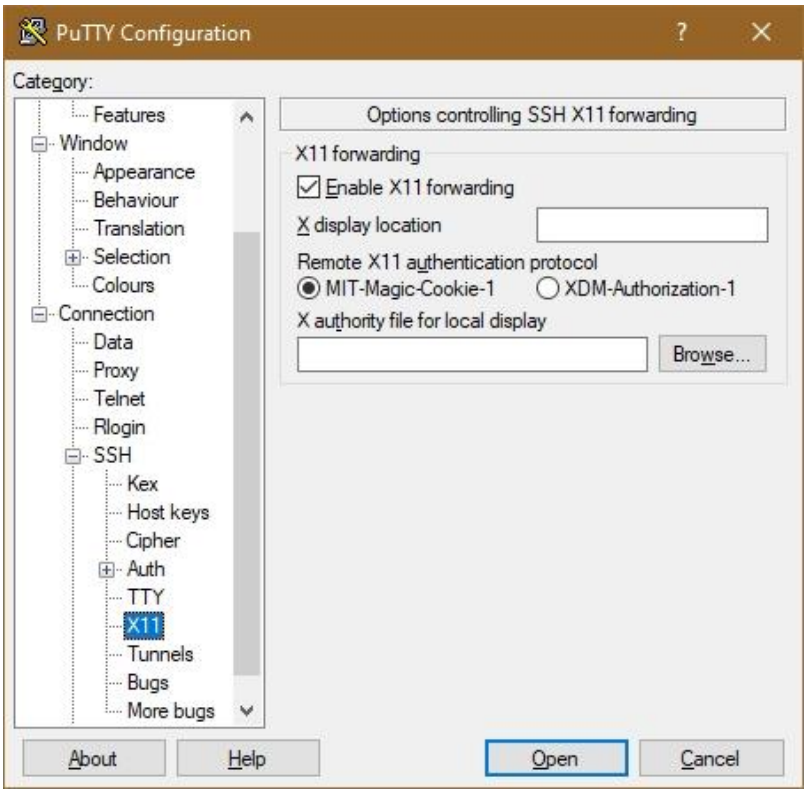


Abbildung 8: Putty_X11-Einstellungen

Loggen Sie sich jetzt via Putty (Open) auf der Linuxmaschine ein. Für das weitere Vorgehen brauchen Sie Administrator-Rechte auf der Linuxmaschine!

Zu 2.

Als Erstes legen wir einen neuen User an, unter dessen Konto unsere Anwendungen laufen sollen. Ich habe ihn **guardian** genannt, er bekommt ein Verzeichnis unter gleichem Namen im Ordner **home** und gehört primär zur automatisch mit eingerichteten Gruppe **guardian**. Damit er Zugriff auf Webdienste hat, fügen wir ihn der Gruppe www-Data hinzu.

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

Damit wir als guardian einloggen können, braucht der User ein Passwort, also geben wir ihm eines.

```
sudo passwd guardian
```

Jetzt sind Sie dran, sich ein Passwort auszudenken.

zu 3.

Falls das noch nicht geschehen ist, installieren wir nun Python auf dem Linuxrechner. Überprüfen Sie das mit dem Aufruf

```
python3
```

Ausgabe: (auf meinem 32-Bit-system ist Python 3.5 die letzte unterstützte Version)
Python 3.5.2 (default, Nov 12 2018, 13:43:14)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>>

Sieht die Antwort nicht so ähnlich aus, dann enthält sie einen Hinweis, wie der Aufruf lauten sollte oder wie Sie die Installation vornehmen können. Python 3.5 ist das höchste Gefühl für ein 32-Bit-System. Höhere Releases auf 64-Bit-Systemen eröffnen weitere Funktionalitäten, die in diesem Rahmen nicht zur Debatte stehen.

Als Nächstes brauchen wir eine vernünftige Entwicklungsumgebung, zum Beispiel **idle**, das sich ähnlich wie Thonny bedienen lässt. Rufen Sie unter putty die grafische Oberfläche von idle auf. Obacht! Das geht von Windows aus nur, wenn Sie, wie unter 1. beschrieben, XMing installiert, als Admin gestartet und Putty darauf abgerichtet haben oder direkt an der Linux-Konsole arbeiten.

```
idle &
```

Startet das Programm nicht, dann muss es ebenfalls zuerst installiert werden. Ihr System gibt Ihnen auch in diesem Fall einen Hinweis dazu. Damit das Putty-Fenster nach dem Aufruf wieder frei wird, fügen Sie dem Aufruf das "&" hinzu, es schickt den idle-Prozess in den Hintergrund.

Legen wir nun ein neues Programmfile über die Python-Shell von idle an.

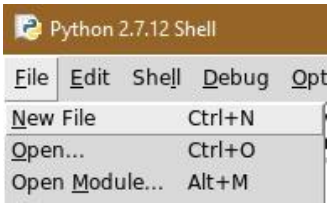


Abbildung 9: idle_NewFile

Ein neues Fenster tut sich auf, und wir können loslegen.

Die erste Zeile enthält das Shebang, **#!/usr/bin/env python3.5**. Es gibt an, welcher Interpreter für die Ausführung des nachfolgenden Scripts zuständig, und wo dieser zu finden ist. Auch wenn Idle unter Python 2.7 läuft können Sie damit Programme höherer Versionen editieren. Beim Start über die Kommandozeile wird dann über das Shebang der richtige Interpreter aufgerufen. Beim Aufruf wird ihm das Skript als Parameter übergeben. Bei dem Programmfile muss das Ausführungsflag gesetzt sein. Wir holen das nach, wenn wir das Programm erstellt und abgespeichert haben.

Wir können übrigens auf dem Linuxrechner im Allgemeinen dieselben Python-Befehle verwenden, wie auf dem ESP8266. Mehr noch, unter Linux läuft CPython, das über einen bedeutend größeren Sprachumfang verfügt. Allerdings kann auch das Umgekehrte gelten. Das Modul time bietet unter CPython verschiedene Methoden nicht an, die wir von MicroPython gewohnt sind. Es stehen auch keine Hardwaretimer zur Verfügung.

Was wir in der Regel auf dem Linuxrechner nicht brauchen, das ist die Netzwerkanmeldung, sofern die Linuxkiste über ein Kabelinterface am Netzwerk angeschlossen ist, wovon ich jetzt ausgehe. Also – ans Werk!

Damit unser Programm asynchron Daten vom "Gärtner" oder vom Handy via Webserver empfangen kann, muss es ständig im Hintergrund laufen. Es soll in (einigermaßen) gleichen Zeitabständen die Temperaturen abfragen. Das könnten wir mittels einer Zeitschleife machen, die den Hardwaretimer des ESP8266 ersetzt. Damit sind aber erheblich Ungenauigkeiten verbunden.

Genauer würde das ein Cronjob erledigen, der die Abfrage mit der Genauigkeit der Systemzeit takten würde. Wenn das Programm aber nicht permanent läuft, fiele die ständige Empfangsbereitschaft weg.

So wäre es, wenn der Cronjob **client9001**.py direkt starten würde. Aber es gibt da auch noch den netzwerkbasierten Austausch von Nachrichten zwischen Programmen via UDP/TCP, und das klappt auch rechnerintern. Also schreiben wir ein kurzes Prograämmchen, das vom Crontab aufgerufen wird und das dem client9001.py sagt, dass er jetzt gefälligst eine Abfrage starten soll. Betrachten wir zuerst das Client-Programm selbst.

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)
Bei den Importgeschäften ist die Methode strftime() interessant, die mit entsprechenden Parametern aus der Systemzeit einen String basteln kann.

(B)
Wir bauen einen UDP-Socket und binden an die Portnummer 9001.

(C)
Ein kurzer Timeout sorgt für flüssigen Schleifendurchlauf.

(D)
Die Steuerparameter für das Schreiben in die Dateien und das Einlesen der Werte vom ESP8266 werden gesetzt.

(E)
TimeOut() erzeugt eine Closure, **compare**(), die eine einfache, flexibel zu handhabende Zeitverzögerung darstellt. Was eine Closure ist und wie man damit arbeitet habe ich in dem Dokument Closures und Decorators.pdf genau beschrieben. Hier wird der Funktion **TimeOut**() ein Zeitintervall in Sekunden übergeben, dessen Ablauf die Closure **compare**() überwacht. Vereinfacht gesagt, eine Closure ist eine Funktion, die sich den Wert ihrer lokalen Variablen zwischen zwei Aufrufen merken kann. Hier ist das das Zeitlimit.

(F)
Für jeden Schleifendurchlauf werden weitere Steuerparameter rückgesetzt. Die erste if-Abfrage testet die boolsche Variable **einlesen**, ob eine Abfrage der Temperaturwerte vom ESP8266 erfolgen soll. Wenn ja, wird **einlesen** zurückgesetzt und der Auftrag gesendet. Der Timer für das Einlesen wird mit 2 Sekunden scharf gemacht. Der Sendeprozess ist durch **try** und **except** abgesichert.

(G)
Nach 2 Sekunden sollten die Daten vorliegen, Wir senden den Befehl zum Transfer. und bereiten das Schreiben in die Datei vor: save=True.

(H)
Es folgt die Behandlung eventueller Anfragen und Antworten. Sie haben die Form

X;Befehl/Antwort

zum/vom ESP8266 und anderen Netzwerkgeräten. UDP ermöglicht einen offenen Nachrichtenaustausch zwischen allen beteiligten Geräten.

Das erste Zeichen X steht für die Art der Payload, die ab dem 3.Zeichen folgt und gibt an, wie damit zu verfahren ist.

A; Crontab hat den Auftrag für das Einlesen von Temperaturwerten gesandt; einlesen wird auf True gesetzt (J)
H; Eine Antwort des ESP8266 auf einen heizungsbezogenen Befehl
M; eine unspezifische Mitteilung des ESP8266 von untergeordneter Wichtigkeit
F; eine asynchrone Mitteilung des ESP8266 zur Bodenfeuchte
T; eingegangene Temperaturnachricht
B; Befehlsprefix von externem Gerät oder dem Webserver (K)
R Sende die Temperaturen an den entsprechenden Client (L)
N Sende die Temperaturen und die letzten 5 Nachrichten an den entsprechenden Client (M)
Dazu nutzen wir die Methode **check_output**() aus dem Modul **subprocess**, das die Ausgabe von bash-Befehlen in Python nutzbar macht.

(N)
Enthält z eine zu sichernde Nachricht und wurde die Sicherung angeordnet (save=True), dann wird jetzt in die entsprechende Datei geschrieben, die sich im Verzeichnis /home/guardian/**arch** des Users guardian befindet. Dort landen auch die archivierten Tagesdateien.

Das Programm **client9001**.py starten wir als User **guardian** wie folgt aus seinem Homeverzeichnis, nachdem wir für guardian und seine Gruppe das Ausführungsflag gesetzt haben.

guardian@ganymed:~$ *ls -lias client**

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

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

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

Nun können wir die Datei starten. Wir tun das so, dass der Prozess auch dann weiterläuft, wenn wir das Terminalfenster verlassen.

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

Ausgabe:
[1] 25646
guardian@ganymed:~$ nohup: ignoriere Eingabe und hänge Ausgabe an 'nohup.out' an

Dann überprüfen wir, ob der Prozess läuft und welche PID (Prozess-ID) er hat.

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

Die fett dargestellte Zahl ist die PID, die wir benötigen, um den Prozess wieder zu beenden. Das erledigt folgender Befehl an der Konsole.

guardian@ganymed:~$ *kill **25646***

Wenn nach dem Start von client9001.py keine Fehlermeldungen erscheinen, legen wir die beiden Tagesdateien daten_gh und messages an.

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

-rw-rw-r-- 1 guardian guardian 0 Okt 23 19:33 daten-gh
-rw-rw-r-- 1 guardian guardian 0 Okt 23 19:33 messages

guardian@ganymed:~/arch$ *cd ..*

Fehlt noch das Programm, das Crontab nutzt, um den Auftrag zum Einlesen der Werte zu erteilen. Die Interprozesskommunikation erlaubt es uns, Nachrichten von einem Programm an ein anderes zu senden. Nachdem das Programm gespeichert ist, muss das Ausführungsflag für User und Gruppe gesetzt werden wie bei 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.

Das nächste Programm **archivate**.py muss termingerecht einmal kurz vor Mitternacht gestartet werden. Es kopiert die Tagesdateien auf die Tagesarchive und legt neue, leere Tagesdateien vor, nachdem die alten gelöscht sind. Dazu nutzt es bash-Befehle, die mittels der Methode **os.system**() aufgerufen werden. **rm** (remove) entfernt Dateien, **cp** (copy) kopiert und **tac** listet eine Datei von der letzten zur ersten Zeile. Der Schalter -a sorgt beim Kopieren dafür, dass auch die Dateirechte der Quelle mit übertragen werden. Nach dem Speichern setzen wir natürlich auch für archivate.py die Ausführunsflags.

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")
```

Jetzt ist es an der Zeit, das gesamte System zu testen. Falls Sie das noch nicht getan haben, schicken Sie jetzt bitte das Programm **xmitter2**.py als **boot**.py zum ESP8266 wie es im Kapitel "Die Software – Autostart" beschrieben ist, und starten dann den Controller neu (reset). Starten Sie **client9001**.py in den Hintergrund von der Kommandozeile in Linux. Ein externes Gerät können wir mit Hilfe von **packetsender** emulieren.

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

Außerdem sollten jetzt die Dateien **daten_gh** und **messages** Zuwachs bekommen haben. Was sich in der Dateigröße äußert.

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

Wir können uns den Inhalt aber auch direkt anschauen.

guardian@ganymed:~$ *cat arch/daten-gh*
24.10.2021 10:25;19,3;19,4;19,1
24.10.2021 10:26; 19,2;19,2;19,1
24.10.2021 10:27; 19,3;19,2;19,1
24.10.2021 10:29; 19,2;19,3;19,0
24.10.2021 10:30; 19,3;19,4;19,1

Die Archivierung rufen wir erst einmal von Hand auf.
guardian@ganymed:~$ *ls -lisa arch*
-rw-rw-r-- 1 guardian guardian   0 Okt 23 19:33 daten-gh
-rw-rw-r-- 1 guardian guardian 2107 Okt 25 14:05 ghaus-2021-10-25
-rw-rw-r-- 1 guardian guardian   286 Okt 25 14:05 mesg-2021-10-25
-rw-rw-r-- 1 guardian guardian   0 Okt 23 19:33 messages

zu 5.

Funktioniert wie erwartet. Also machen wir den Cronjob scharf.

guardian@ganymed:~$ *crontab -e*

Es öffnet sich der Standardeditor. Wenn Sie ctrontab -e noch nie vorher aufgerufen haben, dürfen Sie sich jetzt erst einmal einen der installierten Editoren aussuchen. Dann erscheint die Liste der Cronjobs. An jedem Tag, in jedem Monat und an jedem Wochentag soll archivate.py um 23:58 Uhr aufgerufen werden. Jede Zeile steht für genau einen Auftrag an den cron-Deamon, den Herrn der Zeit. Die vorletzte Zeile ist der Timer für das Anfordern der Temperaturwerte, was alle 10 Minuten erfolgen soll.

```
…..
# 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
```

Nachdem alles eingetragen und kontrolliert ist, Speichern wir ab und beenden den Editor.

Im Terminalfenster sollte jetzt alle 10 Minuten die datei **daten-gh** einen neuen Eintrag erhalten. Wir überprüfen das mit folgendem Befehl.

guardian@ganymed:~$ *cat arch/daten-gh*

Ausgabe:
03.11.2021 **10:48**; 17,88; 18,75; 18,19
03.11.2021 **10:58**; 17,88; 18,75; 18,19
…..
03.11.2021 **12:28**; 18,06; 19,00; 18,38
03.11.2021 **12:38**; 18,06; 19,00; 18,38
03.11.2021 **12:48**; 18,06; 19,00; 18,38

Mit jedem Befehl an den ESP8266 und jedem Feuchtewechsel erhält auch die Datei **messages** Zuwachs.

Wrapping up

Was gab es an Neuem in diesem Beitrag? Fassen wir zusammen:

- Erweiterung der Hardware durch einen Bodenfeuchte-Sensor
- Entstörung des ESP8266 durch einen Kondensator am RST-Eingang
- Die Trägerplatinen für die Netzteile und den ESP8266 erleichtern den Aufbau
- Wir haben erkannt, dass Funktionen dazu helfen können, Speicherplatz zu sparen
- Cronjobs ermöglichen uns eine präzise Zeitsteuerung
- Mit einer Closure haben wir einen Softwaretimer realisiert
- Das Versenden von Nachrichten via UDP erlaubt uns die Kommunikation zwischen getrennten Prozessen auf einem Rechner

Ausblick

Es fehlt jetzt bloß noch der Webserver für den Long Distance Trafic. Wegen des nicht trivialen Umfangs von 170 Zeilen vertröste ich sie aber damit auf die übernächste Episode der "Gärtner"-Reihe, denn es gibt auch einiges dazu zu erklären. Und als Schmankerl gibt es dann auch noch eine App fürs Handy, die direkt auf den ESP8266 zugreifen und auch Alarmmeldungen asynchron empfangen und umsetzen kann. Damit Sie sich beim Basteln der App leichter tun, behandelt die nächste Episode erst einmal den Umgang mit dem MIT-Appinventor2. Das ist ein webgebundenes Tool, mit dessen Hilfe wir Apps fürs Android-Handy oder i-Phone durch Schichten von "Bauklötzen" programmieren können.