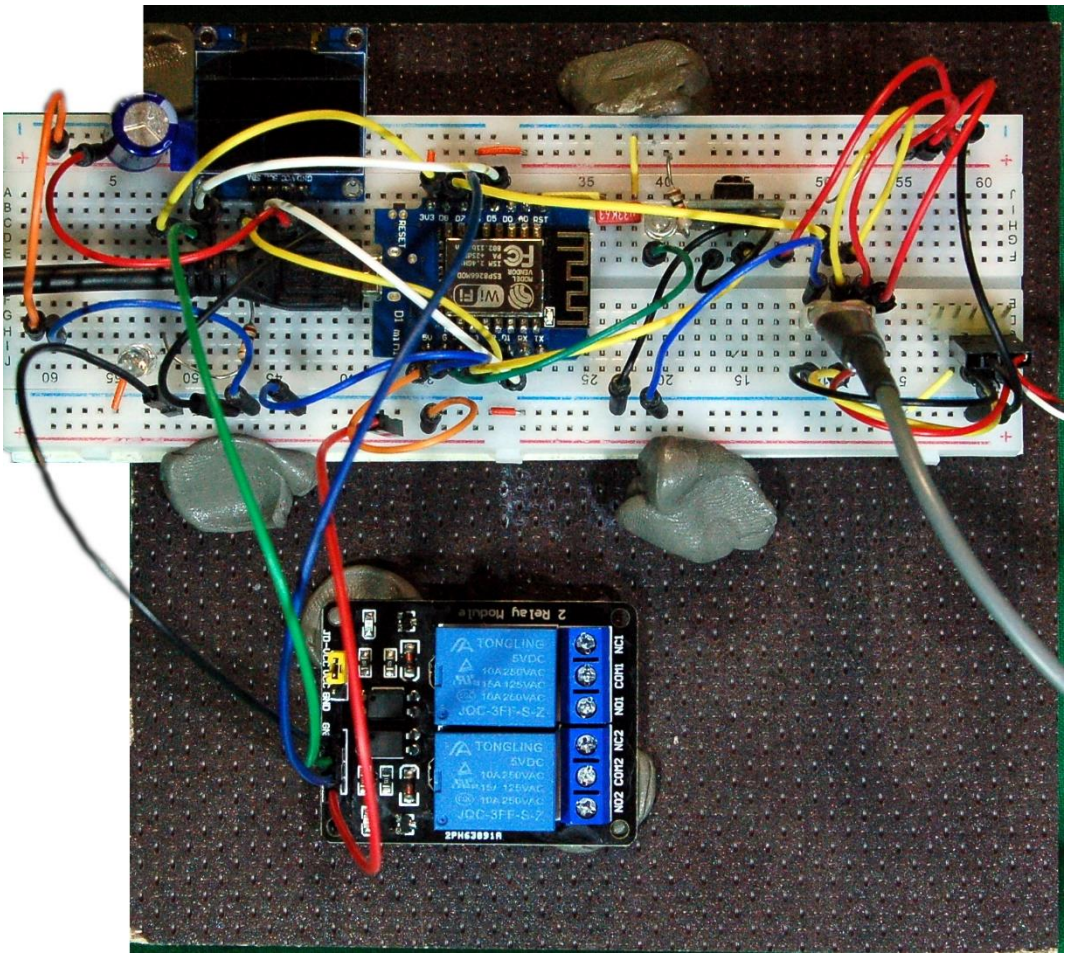

Abbildung 1: MQTT-Client - Heizung

Der MQTT-Client aus der zweiten Episode dieser Themenreihe wird heute Zuwachs bekommen. Ein ESP8266 D1 Mini soll die Vorlauf- und Rücklauftemperaturen der Heizungsanlage messen und über den Mosquitto-Server bereitstellen. Außerdem wird er auf den Eingang zweier Signale warten, durch welche über zwei Relais die Umwälzpumpe und der Brenner der Heizung geschaltet werden.

Als drittes Modul stelle ich einen MQTT-Monitor vor, dessen "Chef", ein ESP32, die Daten der gesamten Anlage auf einem LCD-Keypad ausgibt und über die eingebauten Tasten Schalt-Befehle an die beiden anderen Clients sendet. Ich freue mich, dass Sie wieder mit dabei sind. Willkommen bei der Reihe

Server und Clients unter MicroPython auf dem Raspi und der ESP-Familie

mit dem heutigen Titel

3. Erweiterung des MQTT-Home-Systems

In der zweiten Folge dieser Reihe hatte ich bereits angesprochen, dass durch den Einsatz von MQTT ein Steuerungs- und Überwachungssystem enorm skalierbar ist. Neue Einheiten ersetzen oder ergänzen bestehende.

Die beiden Module, die ich Ihnen heute vorstelle, tun genau das, wir ergänzen eine Messstelle und eine Kontrolleinheit. Letztere kann sowohl auf die Daten der neuen

Messstation als auch auf die der vorigen zurückgreifen. Am Mosquitto-Broker sind keine Eingriffe nötig. Wird eine Station entfernt, fehlen dem Server diese Informationen, und fragt ein Client danach, wird er einfach keine neuen werte erhalten. Es gibt aber auch keine Fehlermeldungen oder Systemabstürze. Ebenso werden Befehle an die entfernte Station von dieser nicht mehr abgerufen, anstehende Aktionen werden nur nicht mehr ausgeführt. Der Server als Vermittler hat damit kein Problem. Fazit daraus ist, dass Module dieser Reihe nicht alle verwirklicht werden müssen. Das System wird funktionieren, wenn der Raspi mit dem Broker läuft und wenigstens ein Modul als Publisher und/oder Subscriber vorhanden ist.

Hardware

Die folgenden Teile werden für die beiden Stationen, die wir heute bauen wollen benötigt.

1	ESP32 Dev Kit C unverlötet
1	LCD1602 Display Keypad Shield HD44780 1602 Modul
1	I2C IIC Adapter serielle Schnittstelle für LCD Display 1602 und 2004
2	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI Wifi Development Board mit CP2102
2	KY-016 FZ0455 3-Farben RGB LED Modul
2	Spannungsversorgung 5V, 500mA für die ESP32/ESP8266
1	2 x 1M Kabel DS18B20 digitaler Edelstahl Temperatursensor Temperaturfühler
1	2-Relais Modul 5V mit Optokoppler Low-Level-Trigger
1	3 x Mini Breadboard 400 Pin mit 4 Stromschienen
diverse	Jumperkabel
1	Raspberry Pi mit 16GB Class10 – SD-Karte
1	Spannungsversorgung 5V; 2,5A für den Raspi
1	Netzwerkkabel

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:

MicropythonFirmware
Bitte eine Stable-Version aussuchen

Die MicroPython-Programme zum Projekt:

dhtclient.py
heizung.py
monitor.py
umqttsimple.py
dashboard.json (Folge 4)

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, damit 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 kompilieren 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 Makro 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 kompiliertes 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.

Überwachung der Heizung – Hausaufgabenlösung

Natürlich gehört zur Überwachung und Steuerung einer Heizung mehr als nur der nachfolgend beschriebene Aufbau, der lediglich die Temperaturen von Vorlauf und Rücklauf erfasst und Brenner und Umwälzpumpe schalten kann. Da wären noch Außenfühler, Wohnungsthermostat, Zeitschaltung etc. zu nennen. Das alles kann in unser System nachträglich ja noch einfließen.

Die Schaltung

Beim Aufbau der Schaltung war gerade ein ESP8266-D1-Mini zur Hand. Ebenso gut eignet sich ein Amica oder ein ESP8266-Node-MCU. Die Energieversorgung geschieht am einfachsten, wie beim DHT-Client über die USB-Buchse, ein passendes USB-Kabel und ein 5V-Steckernetzteil. Der Programmtext muss sich als boot.py für einen autonomen Programmstart im Flash des ESP8266 befinden. Für das Programm dhtclient.py aus dem vorangegangenen Beitrag, laden wir es in Thonny und speichern es sogleich unter dem Namen boot.py in Workspace ab. Jetzt muss boot.py nur noch zum ESP8266 hochgeladen werden. Nach einem Reset startet der Controller selbständig durch. An den Blinkzeichen können wir den Fortgang beobachten.

Ein Hinweis zum ESP8266 Node-MCU V3. Dieses Board kann am Pin **Vin** zwar mit 5V von einer externen Quelle versorgt werden, stellt aber an diesem Anschluss die 5V vom USB-Anschluss nicht zur Verfügung. Um die Relais auf dem Duo-Modul zu schalten, muss dieses also mit einer eigenen 5V-Leitung versorgt werden.

Das eingesetzte Doppel-Relais-Modul kann auch für Schaltleistungen bis 300W an 230VAC verwendet werden. Die Mittelspannungskontakte der Relais sind durch einen Luftspalt von der Niederspannungsseite hinreichend abgetrennt. Module ohne diesen Luftspalt sollten nicht für Schaltspannungen größer als 50 V verwendet werden. Die Relais sind LOW-Level-getriggert. Das heißt, dass der NO-Kontakt (Normaly Open) geschlossen wird, wenn am Eingang INx (x=1 oder 2) GND-Pegel anliegt.

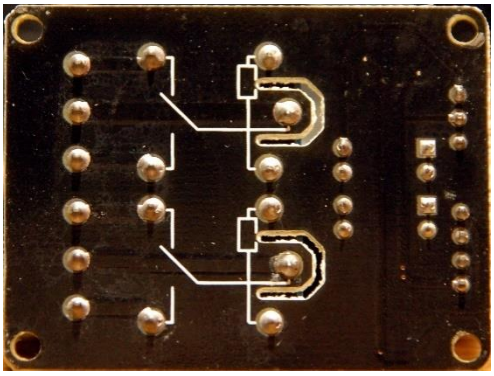

Abbildung 2: Relaismodul - Unterseite

Der Taster hat im Moment noch keine Funktion, kann aber selbst, oder durch einen digitalen Sensor ersetzt, als Eingabemedium zum Veröffentlichen von Nachrichten über den Broker Verwendung finden. Einzige Bedingung ist, dass zur Startzeit am Pin GPIO0 HIGH-Pegel anliegen muss. Andernfalls schaltet der ESP8266 in den Flash-Modus.

Die LED signalisiert mit den drei Farben folgende Zustände:

blau: lang – kurz blinkend; Verbindungsaufnahme zum WLAN-Router
grün: kurz – lang blinkend; Heartbeat des Hauptschleifendurchlaufs
rot: kurz – kurz – kurz blinkend; Fehleranzeige

Um etwa die gleiche Helligkeit der LEDs zu erreichen, sind verschiedene Vorwiderstände verwendet worden.

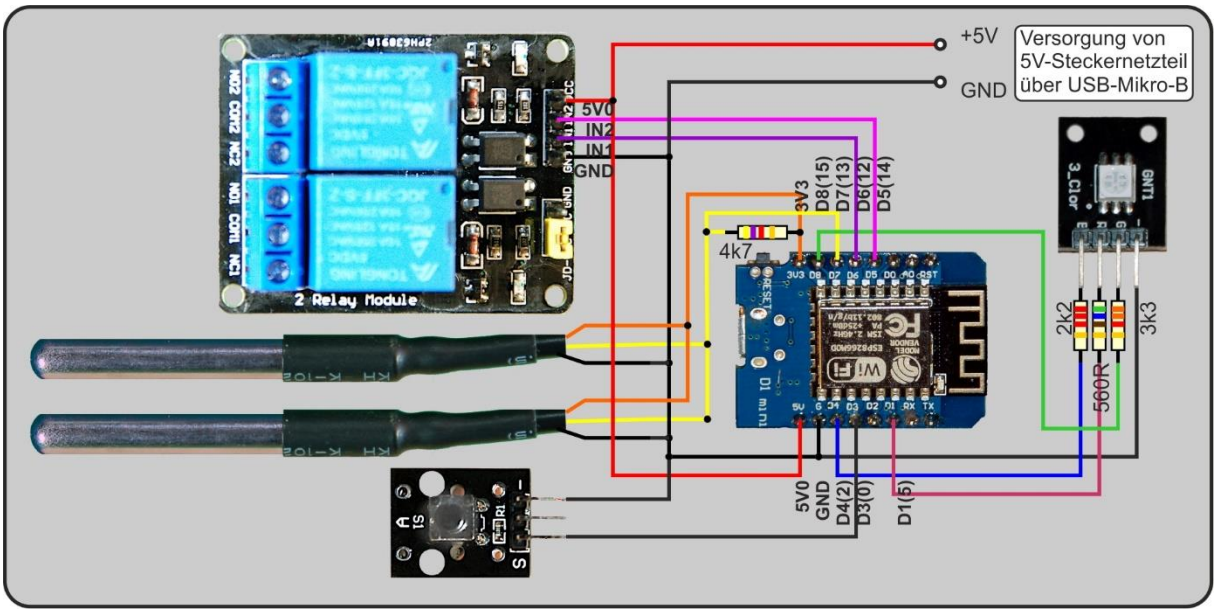

Abbildung 3 Schaltung Heizungs-Client

Das Programm für die Heizungssteuerung

Anhand des Heizungsmoduls will ich zeigen, dass die Umwidmung einer bestehenden Einheit auf andere Sensoren und einen anderen Controller ohne große Änderungen einer bereits bestehenden Programmumgebung machbar ist.

Die Umstellung von einem DHT22 auf zwei DS18B20 erfordert ein paar Änderungen im Vergleich zum Programm dhtclient.py aus dem zweiten Blogpost.

Anstelle des dht-Moduls importieren wir die Module onewire und ds18x20

```
from onewire import OneWire
from ds18x20 import DS18X20
```

Die Pinbelegung unterscheidet sich vom dhtclient, weil beide Schaltungen unabhängig voneinander entwickelt wurden. Natürlich kann man auch von vornherein

darauf achten, dass die Belegungen weitestgehend gleichbleiben, dann spart man sich das Umschreiben der Pinnummern. Pin 14 wird als onewire-Objekt an den Konstruktor des ds-Pins übergeben. Dann lesen wir die ROM-Codes der DS18B20-Chips ein, durch die wir die Sensoren einzeln ansprechen können. Zwei Pins für die Ansteuerung der Relais werden hinzugefügt.

```
ds_pin = Pin(14)     # D7@esp8266
ds = DS18X20(OneWire(ds_pin))
chips = ds.scan()
print("Chips",chips)
relHeating=Pin(13,Pin.OUT,value=1)
relPump=Pin(12,Pin.OUT,value=1)
```

Weitere Topics für das Publizieren und Abonnieren werden ergänzt, beziehungsweise ersetzen die vorherigen.

```
topicVorlauf="heizung/vorlauf"
topicRuecklauf="heizung/ruecklauf"
topicPump="heizung/pumpe"
topicHeating="heizung/maschine"
topicPumpDone="heizung/pumpe/done"
topicHeatingDone="heizung/maschine/done"
```

Die fettformatierten Zeilen in der Funktion readDS18B20() ersetzen einen Teil der Funktion readDHT(). Bei der Montage der Sensoren achten wir auf die korrekte Zuordnung von Chip0 zum Vorlauf und Chip1 zum Rücklauf. Die beiden Temperaturwerte bekommen wir in der Rückgabe wieder als Tuple.

```
def readDS18B20():
    try:
        ds.convert_temp()
        sleep(0.85)
        vorlauf=ds.read_temp(chips[0])
        ruecklauf=ds.read_temp(chips[1])
        print("Vorlauf, Ruecklauf",vorlauf,ruecklauf)
        return vorlauf, ruecklauf
    except OSError as e:
        print("Sensorfehler ---->",e)
        return None
```

In der Funktion **connect2Broker**() ändern wir die Subscription auf die neuen Topics. und passen gegebenenfalls die Ausgabebefehle an. Letztere können im Produktionssystem ohne Schaden weggelassen werden.

```
    client.subscribe(topicPump)
    client.subscribe(topicHeating)
```

In der Funktion **messageArrived**() ändern wir die vorhandene Abfrage und fügen eine zweite hinzu. Nach jeder Aktion publishen wir eine Rückmeldung für den Auftraggeber.

```
    if topic == topicPump:
        if msg=="an":
            relPump.value(0)
            client.publish(topicPumpDone, "AN")
        else:
            relPump.value(1)
            client.publish(topicPumpDone, "AUS")
    if topic == topicHeating:
        if msg=="an":
            relHeating.value(0)
            client.publish(topicHeatingDone, "AN")
        else:
            relHeating.value(1)
            client.publish(topicHeatingDone, "AUS")
```

Schließlich passen wir in der Hauptschleife die Passage an, in der die Sensoren abgefragt werden und veröffentlichen die neuen Daten. Alles andere wird unverändert aus dhtclient.py übernommen.

```
while True:
    try:
        client.check_msg()
        if sendNow():
            resp=readDS18B20()
            if resp is not None:
                vorlauf,ruecklauf = resp
                vorlauf = (b'{0:3.1f}'.format(vorlauf))
                ruecklauf =  (b'{0:3.1f}'.format(ruecklauf))
                client.publish(topicVorlauf, vorlauf)
                client.publish(topicRuecklauf, ruecklauf)
```

Hier finden Sie die gesamte Datei heizung.py als Download zum Vergleich. Nur ca. 18% der Programmzeilen mussten angepasst/ergänzt werden.

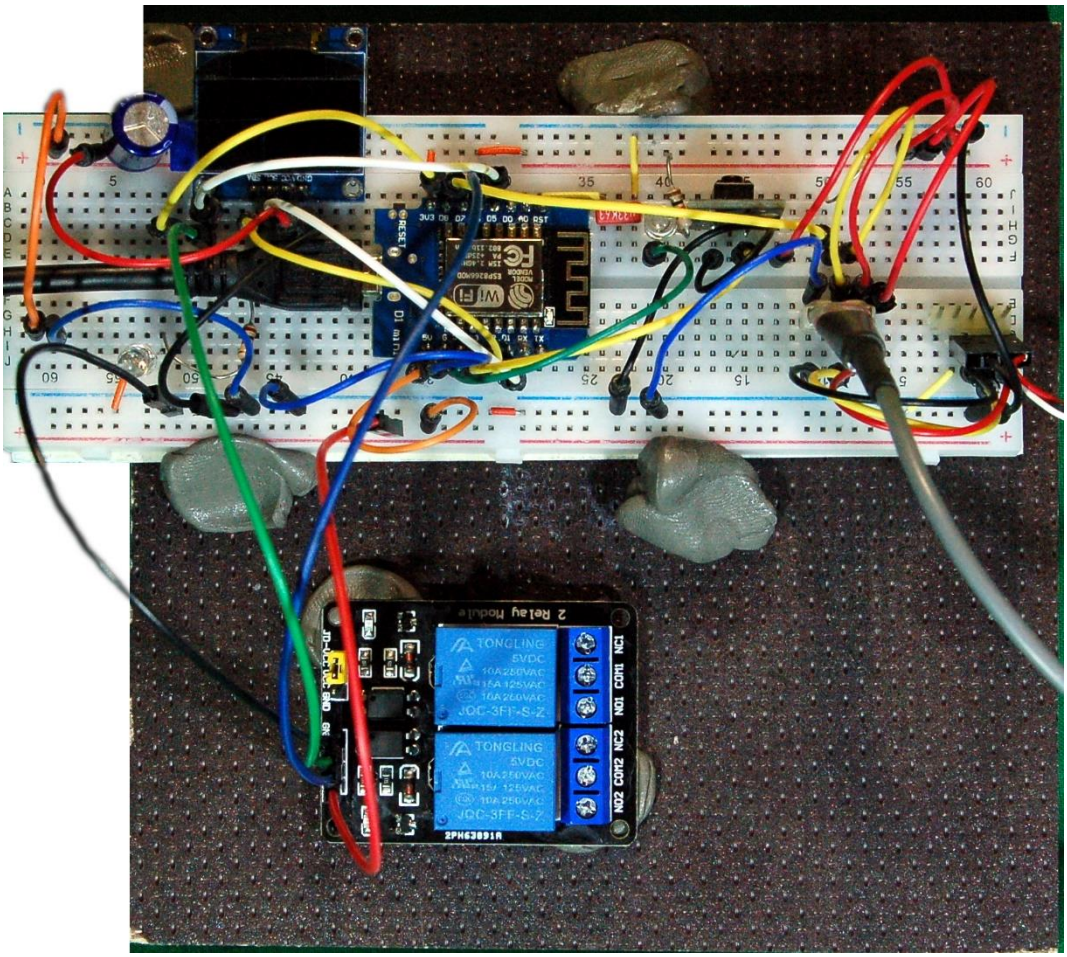

Abbildung 4: MQTT-Client - Heizung

Abbildung 4 verrät meinen Trick zur reversiblen Montage von Baugruppen auf einem Grundbrett. Die graue Knete ist Eternit-Dichtungsmasse. Sie klebt nicht an den Fingern hält aber die Teile gut fest und lässt sich rückstandslos vom Untergrund und den Bauteilen entfernen.

Test des Heizungsclients

Es ist wieder an der Zeit, einen Test des bisherigen Equipments durchzuführen. Läuft Mosquitto auf dem Raspi? Dann starten wir in Thonny das Programm heizung.py und warten die Verbindung zum WLAN-Router und dann zum Broker ab, danach müssten im 5-Sekunden-Abstand die Temperaturmeldungen vom ESP8266 im Terminalbereich erscheinen.

Dann öffnen wir ein Terminal zum Raspi. Das muss jetzt über Putty passieren, weil am Raspi keine Tastatur und kein Bildschirm mehr angeschlossen sind. Wir starten Putty, öffnen die Verbindung zum Raspi, die wir im ersten Teil der Reihe angelegt haben und melden uns am Raspi an.

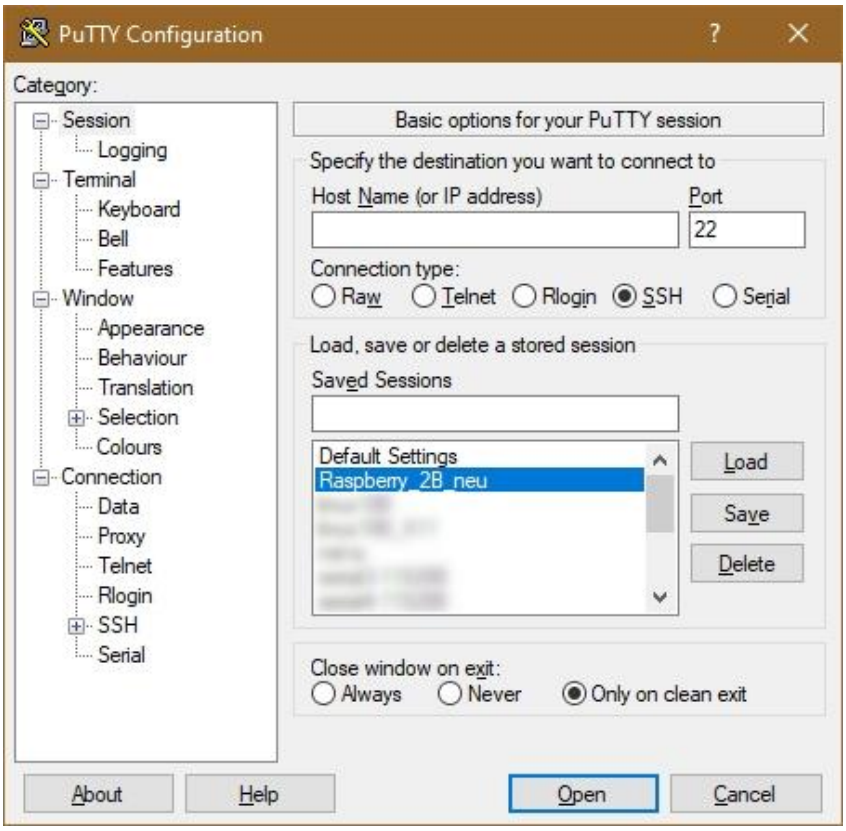

Abbildung 5: Putty öffnen

Wir abonnieren die Topics heizung/vorlauf und heizung/ruecklauf und bekommen mit jeder neuen Abtastung der Sensoren am Heizungs-Client sofort die neuesten Ergebnisse geliefert.

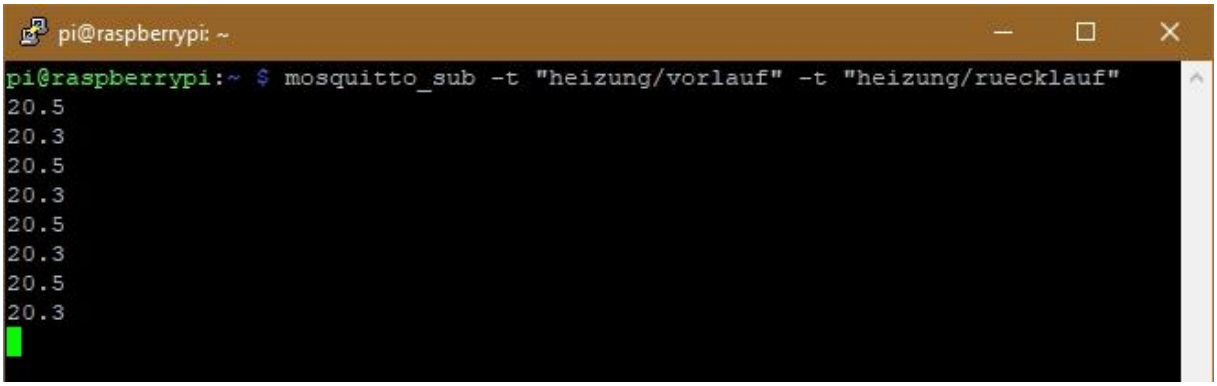

Abbildung 6: Meldungen des Heizungs-Clients

Brechen wir mit Strg+C ab, geben wir auf der Kommandozeile folgende Befehle ein und beobachten die LEDs auf der Relaisplatine. Wenn die LEDs jeweils an- und ausgehen, haben die Schaltung und unser Programm den Test bestanden.

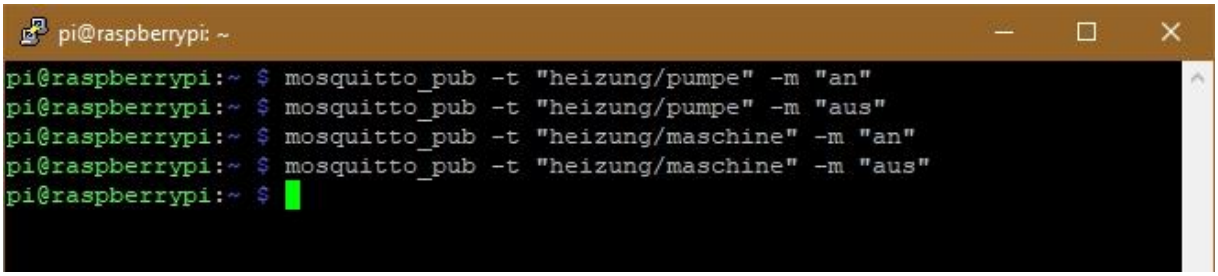

Abbildung 7: Publishing - Wir schalten die Heizung

Um die Rückmeldungen zu sehen, öffnen wir ein zweites Terminal zum Raspi und geben die folgende Subscription ein.

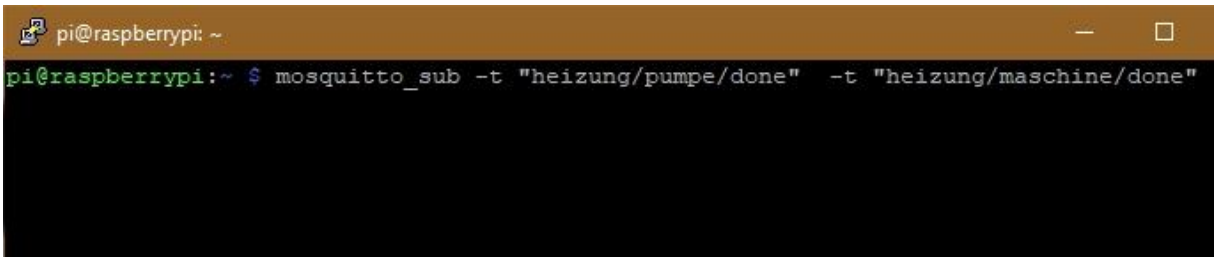

Abbildung 8: Subscription starten

Vom Pulisher-Fenster aus geben wir noch einmal die Schaltbefehle.

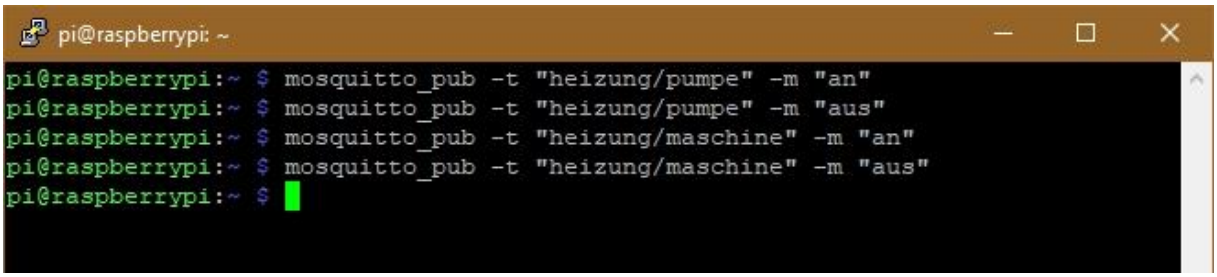

Abbildung 9: Publishing - Wir schalten die Heizung noch einmal zur Probe

Im Subscription-Fenster erscheinen die Nachrichten vom Heizungs-Client.

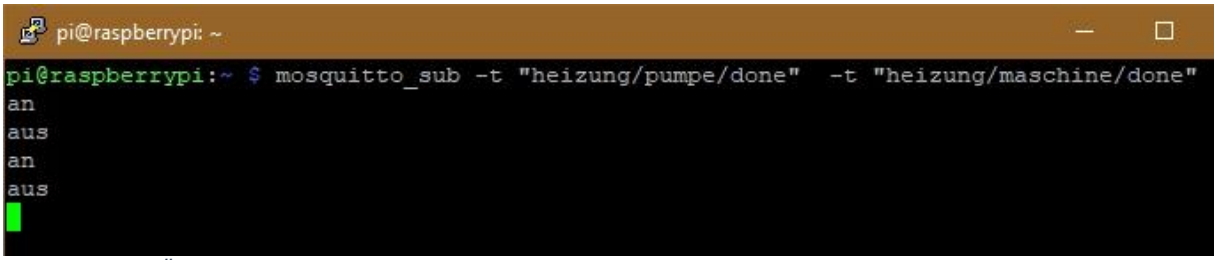

Abbildung 10: Überprüfung der Rückmeldungen

Hausaufgabe:
Mit einem Soundmodul (Mikrofon und Komparator) kann man die Zeiten erfassen, in denen der Brenner läuft. Das Programm heizung.py müsste zur Auswertung eine MQTT-Nachricht absenden, wenn der Brenner einschaltet und wenn er wieder ausschaltet. Ein MQTT-Client auf dem Raspi könnte die Meldungen abonnieren und daraus eine Statistik erstellen.

Der ESP32 als Schaltzentrale

Unsere beiden Clients verhalten sich in gewisser Weise passiv, weil sich an ihnen (bisher) keine Bedienungselemente befinden. Mit dem Monitor-Client soll sich das nun ändern.

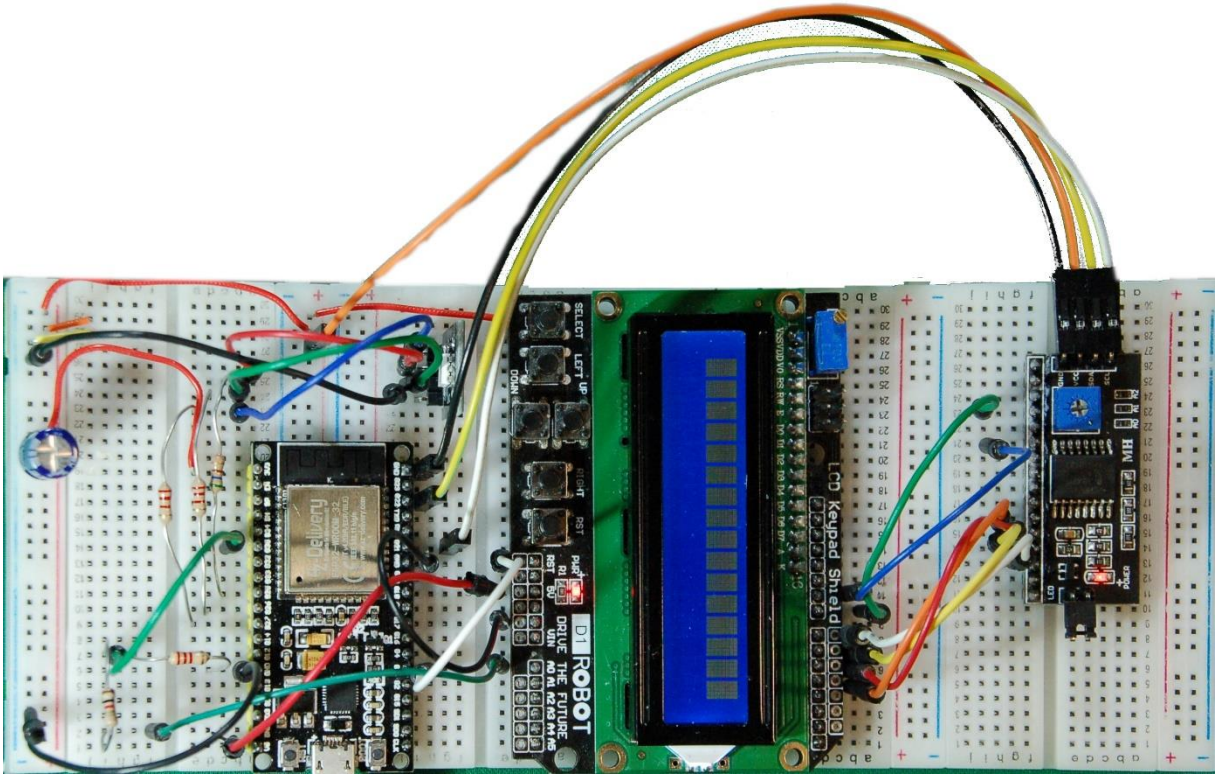

Abbildung 11: MQTT-Monitor

Die Schaltung des MQTT-Monitors

Ein ESP32 empfängt sämtliche, bislang verfügbaren MQTT-Nachrichten, erlaubt aber über die Tasten des LCD-Key-Pads auch das Versenden von Nachrichten an die beiden anderen Clients. Dort werden die entsprechenden Topics abonniert und in Schaltsignale zu den Relais umgesetzt. Als Antwort erhalten wir die Nachrichten, welche die Clients nach dem Schaltvorgang veröffentlichen.

Auch für den Monitor-Client wird das Programm dhtclient.py als Basis der Programmierung benutzt. Wir finden auch die RGB-LED zum Signalisieren der Programmzustände wieder. Leider besitzt das Display nur zwei Zeilen, aber ich wollte auf die Tasten nicht verzichten. Mit einem kleinen Trick kann man mit den 6 Tasten drei Relais ein- und ausschalten obwohl nur fünf Tasten über den analogen Eingang GPIO35 abgefragt werden können. Das Modul ist eigentlich für den Arduino UNO konzipiert und besitzt eine RST-Taste, die wir zweckentfremden und getrennt über GPIO0 abfragen.

Die Tasten liegen bis auf die RST-Taste an einer Widerstandskaskade. Die Spannung, die sich bei Betätigung einer Taste am Ausgang A0 des LCD-Key-Pads einstellt, wird zu deren Decodierung verwendet. Das geschieht im Modul keypad.py. Gegebenenfalls müssen die Grenzwerte der Bereiche im Konstruktor leicht verändert

werden. Die Grenzen dürfen sich aber nicht überlappen und die Tastenwerte sollten in der Bereichsmitte liegen.

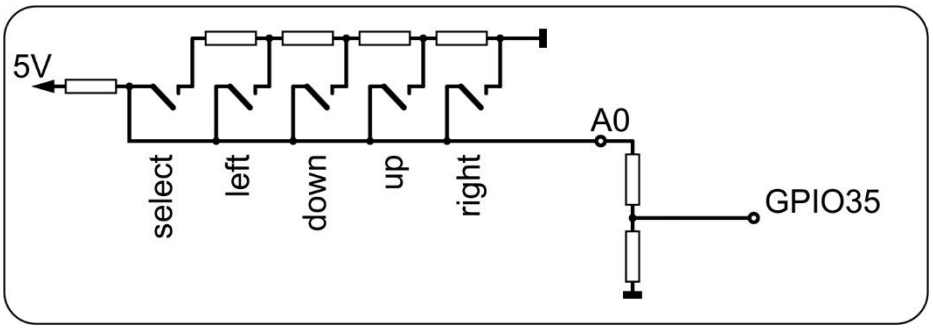

Abbildung 12: Switch_Kaskade

Nachdem im Leerlauf die 5V von der LCD-Keypad-Versorgung dort an A0 anliegen, muss die Spannung auf ein für den ESP32 erträgliches Maß reduziert werden. Das erledigt der Spannungsteiler mit 1kΩ und 2,2kΩ am Ausgang A0. Am Eingang GPIO35 bleiben die Spannungswerte dadurch unter 3,3V.

Das Display und damit das gesamte LCD-Keypad, muss an 5V betrieben werden. Der 5V-Anschluss am ESP32, Pin Vin, ist dafür völlig ausreichend. Wir versorgen das Board also auch wieder über das USB-Kabel und ein Steckernetzteil von 5V. Ein Teil der LCD-Anschlüsse wird auch auf 5V hochgezogen. Das bedeutet, dass das Display nicht direkt an den ESP32 angeschlossen werden kann. Der I2C-Parallel-Umsetzer mit dem PCF7485 übernimmt daher auch Funktion die eines Pegelwandlers. Dessen Ausgangsstufen können zwar nur bis maximal 1mA liefern aber bis zu 25mA aufnehmen. Genau das brauchen wir, um die Eingänge des Displays auf GND-Potenzial zu ziehen. Die SCL-Leitung wird durch den ESP32 auf 3,3V gehalten, und der PCF8574 kann auch die SDA-Leitung auf kein höheres Potenzial anheben, weil wir ihm nur 3,3V als Versorgungsspannung gönnen. Also alles in kuschligen Tüchern für den ESP32.

Die Beleuchtung des Displays lässt sich übrigens nicht ausschalten, weil sie an unzugänglichen Stellen fest verdrahtet ist.

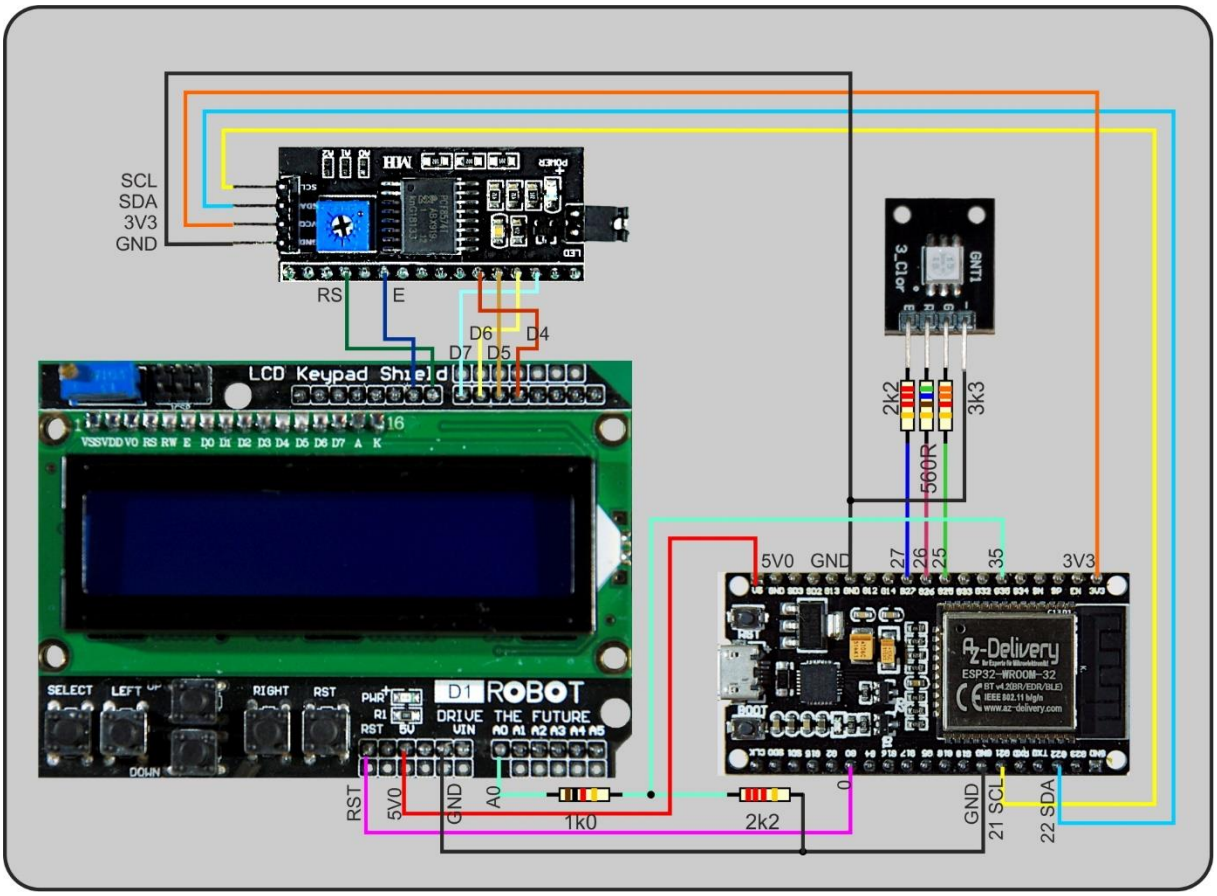

Abbildung 13: MQTT-Monitor Schaltung

Kommen wir zum Programm für den MQTT-Monitor. Ich hatte schon angedeutet, dass auch dafür das Programm des DHT-Clients Pate steht. Den Sensorteil können wir entfernen, dafür müssen wir den Subscription- und Publishingteil erweitern und uns um die Ansteuerung des LCDs kümmern.

Das Monitor-Programm

Nach dem Entfernen des dht-Imports fügen wir die folgenden Module hinzu. Die Klasse I2CBus macht die Bedienung der Schnittstelle etwas griffiger. KEYPAD_LCD ist die Schnittstelle zur Hardwarebehandlung der Tastenabfrage, während KEYPAD allgemein Routinen für die Eingabe und Tastenauswertung zur Verfügung stellt – warte auf eine Taste, hole ein ASCII-Zeichen, Inputfunktion. LCD bietet, so weit sinnvoll, dieselben Befehle wie die Klasse OLED. PCF8574U_I2C ist der Hardwaretreiber für den LCD-Prozessor und HD44780U liefert die Befehle, die in den Methoden von LCD genutzt werden.

```
from i2cbus import I2CBus
from keypad import KEYPAD,KEYPAD_LCD
from lcd import LCD
from hd44780u import HD44780U, PCF8574U_I2C
```

Dann erzeugen wir die Objekte für Display und Tastatur.

```
i2c=I2C(scl=Pin(21),sda=Pin(22))
ibus=I2CBus(i2c)

d=LCD(i2c,adr=0x27,cols=16,lines=2) # LCDPad am I2C-Bus
d.clearAll()

kpdrv=KEYPAD_LCD()
kp=KEYPAD(kpdrv,d)
```

Die Topics kennen wir bereits, die Variablen puffern den Status der entsprechenden Werte für die Anzeige-Funktion.

```
topic=[
        "heizung/vorlauf",
        "heizung/ruecklauf",
        "heizung/pumpe",
        "heizung/maschine",
        "keller/temperature",
        "keller/humidity",
        "keller/ventilator",
      ]
vorlauf=0
ruecklauf=0
pumpe="AUS"
heizung="AUS"
temperatur=0
feuchte=0
ventilator="AUS"
```

Die Darstellung auf dem Display muss auf drei Screens verteilt werden. Die Variable **screen** ist der entsprechende Pointer. **nextMsgDelay** gibt die Zeit für den Bildschirmwechsel in Millisekunden vor. Die Pins für die Status-LEDs werden neu verteilt und der übliche Tasten-Pin deklariert.

```
screen=0
nextMsgDelay=1500

statusLed=Pin(27,Pin.OUT,value=0) # blau=2
onairLed=Pin(26,Pin.OUT,value=0)  # gruen=1
errorLed=Pin(25,Pin.OUT,value=0)  # rot
led=[errorLed,onairLed,statusLed ]
red,green,blue=0,1,2

taste=Pin(0,Pin.IN,Pin.PULL_UP)
```

Vielleicht ist Ihnen aufgefallen, dass die sieben abonnierten Topics in Form einer Liste definiert wurden. Hier kommt der Grund dafür.

```
    for t in topic:
        client.subscribe(t)
        print(t)
```

In der weitgehend übernommenen Funktion **connect2Broker**() können wir jetzt die Subscriptions in einer einfachen for-Schleife erledigen, anstatt diese einzeln aufzuführen.

Auch die Funktion **messageArrived**() behält ihren Kopf, nur die Liste der Abfragen muss ausgebaut werden. Die eingegangenen Daten werden in den dafür vorgesehenen Variablen für die spätere Anzeige eingelagert. Damit geänderte Werte aus der Funktion zurück zum Hauptprogramm finden, müssen die Variablen als **global** deklariert werden.

```
    global vorlauf,ruecklauf,pumpe,heizung
    global temperatur,feuchte,ventilator
```

```
    if thema == topic[0]:
        vorlauf=float(msg)
    if thema == topic[1]:
        ruecklauf=float(msg)
    if thema == topic[2]:
        pumpe = msg
    if thema == topic[3]:
        heizung = msg
    if thema == topic[4]:
        temperatur = float(msg)
    if thema == topic[5]:
        feuchte = float(msg)
    if thema == topic[6]:
        ventilator = msg
```

Neu ist die Funktion **showScreen**(), welche den Datenbestand in drei Screens zur Anzeige bringt. Die Nummer des Screens wird als Parameter übergeben. Vor jeder Ausgabe wird das Display gelöscht.

```
def showScreen(nbr):
    d.clearAll()
    if nbr == 0:
        d.writeAt("K-Temp: {} *C".format(temperatur),0,0)
        d.writeAt("K-Hum:  {} %".format(feuchte),0,1)
    if nbr == 1:
        d.writeAt("V-Lauf: {} *C".format(vorlauf),0,0)
        d.writeAt("R-Lauf: {} *C".format(ruecklauf),0,1)
    if nbr == 2:
        d.writeAt("UP:{}; HZG:{}".format(pumpe.upper(),\
                                  heizung.upper()),0,0)
        d.writeAt("Ventilator:{}".format(ventilator.upper()),\
                                  0,1)
```

Die WLAN-Anmeldung bleibt wie gehabt. In der Hauptschleife fragen wir zuerst die Tasten ab. Die Wartezeit für das Abfragen der Tasten durch die Methode kp.waitForKey() wird auf 0.5 Sekunden festgelegt, das ist ausreichend und sorgt für einen flüssigen Schleifendurchlauf. Die Zuordnung der Tasten ist beliebig veränderbar. Sie ist in der Tabelle dargestellt.

Taste	**Wert**	**Funktion**
SELECT	4	Pumpe aus
LEFT	3	Pumpe an
UP	1	Brenner an
DOWN	2	Brenner aus
RIGHT	0	Lüfter aus
RST		Lüfter an

showNow() ist die Funktion, die durch einen Rückgabewert von True anzeigt, dass der Timer für die Anzeige des aktuellen Screens abgelaufen ist. **TimeOut**() gibt eigentlich eine Referenz auf die in ihrem Inneren definierte Closure **compare**() zurück, die der Variablen showNow zugewiesen wird.

Dort, wo bislang der Aufruf der Funktion zum Einlesen der Temperaturwerte stand, befindet sich jetzt die Tastaturabfrage. Je nach Tastenwert wird die entsprechende Nachricht publiziert.

```
showNow=TimeOut(nextMsgDelay)
while True:
    t=kp.waitForKey(0.5)
    if t == 4:
        client.publish("heizung/pumpe", "aus")
    elif t== 3:
        client.publish("heizung/pumpe", "an")
    elif t== 1:
        client.publish("heizung/maschine", "an")
    elif t== 2:
        client.publish("heizung/maschine", "aus")
    elif t == 0:
        client.publish("keller/ventilator", "aus")
    else:
        pass
    t=-1
    if taste.value() == 0:
        client.publish("keller/ventilator", "an")
    try:
        client.check_msg()
        if showNow():
            showScreen(screen)
            showNow=TimeOut(nextMsgDelay)
            screen=(screen + 1) % 3
            blink(0.03,0.02,green)
    except OSError as e:
        blink(0.2,0.3,red,inverted=True)
```

```
        blink(0.2,0.3,red,inverted=True)
        blink(0.2,0.3,red,inverted=True)
        restartClient()
    sleep(0.1)
```

Mit jedem Schleifendurchlauf prüfen wir jetzt mit **client.check_msg**() ob eine neue Nachricht vorliegt, die in **messageArrived**() decodiert wird. Ein Rückgabewert der Timerfunktion **showNow**() zeigt uns im Falle von **True** an, dass der nächste Screen angezeigt werden muss. Wir stellen dann auch den Timer neu ein und erhöhen die Screen-Nummer modulo 3. Der Teilerrest der Screennummer liefert damit immer im gleichen Zyklus die Werte 0, 1 und 2.

Auch dieses Programm monitor.py steht zum Download bereit. Ist Ihr Aufbau fertig? Laufen die beiden anderen Clients? Ist Mosquitto empfangsbereit? Dann starten wir den Test der Monitoreinheit.

Nach den üblichen LED-Signalen zum Verbindungsaufbau sollte die grüne LED im Zwei-Sekundenrhythmus aufblitzen. Der Display-Refresh mit dem Wert von **nextMsgDelay** alle 2000ms. Das Display sollte nun die folgenden Screens anzeigen.

Abbildung 14: Keller - Temperatur und Luftfeuchte

Abbildung 15: Heizung - Vorlauf - Rücklauf

Abbildung 16: Keller, Heizung - Schaltzustände

Beim Betätigen der Tasten müssen die entsprechenden Relais an den Clients ansprechen und die Schaltzustände im Display ausgewiesen werden. Wenn das alles wunschgemäß funktioniert, klopfen Sie sich bewundernd auf die Schulter. Die beiden Hardwareteile haben Sie damit bravourös gemeistert.

In der nächsten Folge werden wir uns um den Raspi, genauer um Node-RED kümmern. Wir erhalten damit eine Schaltzentrale, die im LAN über einen Browser erreichbar ist. Soll weltweit darauf zugegriffen werden, dann muss im Router eine Freigabe auf den Node-RED-Server auf dem Raspi eingerichtet werden. Außerdem werden wir uns einen Weg ansehen, der eine Zeitsteuerung der Heizung etabliert. Auch dazu hilft uns der Raspi.

In der Zwischenzeit könnten Sie sich ja auch schon einmal Gedanken machen, wie eventuell ein Temperaturfühler in der Wohnung ins System integriert werden könnte. Viel Vergnügen beim Basteln, Programmieren und Grübeln.

Zum ausdrucken und Abspeichern gibt es diese Folge natürlich wieder als PDF-Datei zum Download.