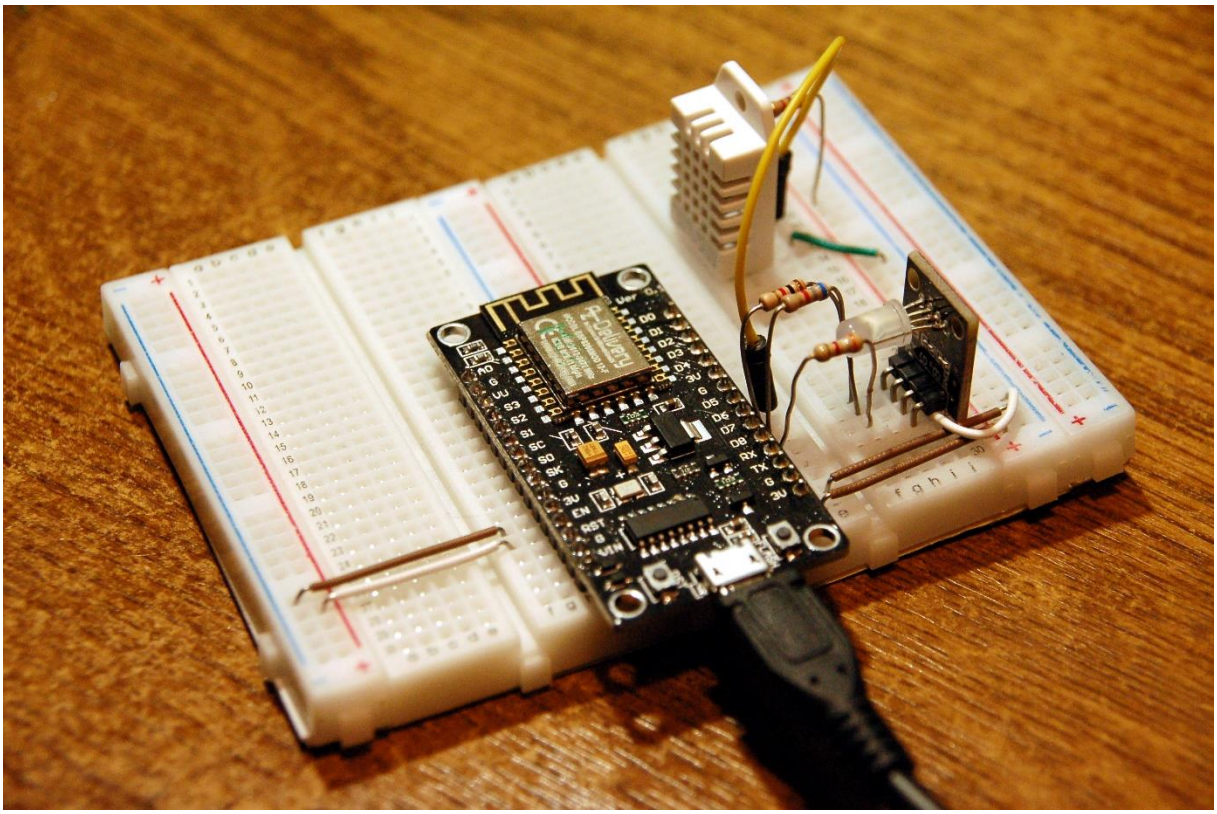

Abbildung 1: Der DHT22-Client mit einem ESP8266 Node-MCU C3

In der vorhergehenden Folge haben wir mit der Installation des Raspberry Pi, des Mosquitt-Brokers und von Node-RED die Infrastruktur für das MicroPython-MQTT-Projekt geschaffen. Der Mosquitto-Broker und Node-RED laufen auf dem Raspberry Pi. Auf einem ESP32 installieren wir nächstes Mal eine Anzeige und Steuereinheit.

Einen der beiden ESP8266 bringen wir heute dazu, einen DHT22- Sensor, auszulesen und eine LED-Beleuchtung oder ein Relais zu schalten. Ein weiterer ESP8266 soll später die Vorlauf- und Rücklauftemperatur der Heizungsanlage im Keller mit zwei DS18B20 messen und die Fernschaltung der Anlage über Relais erlauben.

Damit wieder willkommen zur Reihe

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

mit dem heutigen Titel

2. Ein ESP8266 als MQTT-Client

Der Vorteil von MQTT (Message Queuing Telemetry Transport) ist eindeutig der, dass Mess-, Steuerungs- und Überwachungsaufgaben dynamisch zu einem Automationssystem hinzugefügt werden können. Als Hardware benötigen wir für diese Episode folgende Teile, mit denen wir ein kleines, leicht erweiterbares Hausautomationssystem aufbauen werden. Die einzelnen Stationen binden wir dann nach und nach in das Gesamtsystem ein und erläutern daran detailliert die Funktionsweise von MQTT.

Hinweis:
Das MQTT-System ist, nicht nur in dieser Blogreihe, leicht zu modularisieren. Das heißt es müssen nicht alle Hardwaregruppen vorhanden sein. Grundvoraussetzung ist natürlich der Mosquitto-Server auf dem Raspi, denn ohne Server geht gar nichts. Ob Sie dann aber lieber einen Client nach dem Muster **dhtclient.py** oder den Client mit den **DS18B20**-Sensoren oder beide und den **Monitorclient** verwirklichen wollen, bleibt Ihnen überlassen. Einen Client brauchen Sie aber, denn sonst hat der Server niemand, mit dem er chatten kann. Es ist auch ganz einfach, weitere Einheiten hinzuzufügen.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F WIFI Wifi Development Board mit CP2102
1	KY-016 FZ0455 3-Farben RGB LED Modul
1	Spannungsversorgung 5V, 500mA für den ESP8266
1	DHT22 AM2302 Temperatursensor und Luftfeuchtigkeitssensor
1	Widerstand 4,7kΩ
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 verschiedene Entwicklungsstufen im Text
heizung.py (Folge 3)
monitor.py (Folge 3)
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.

Der ESP8266 als MQTT-DHT-Client

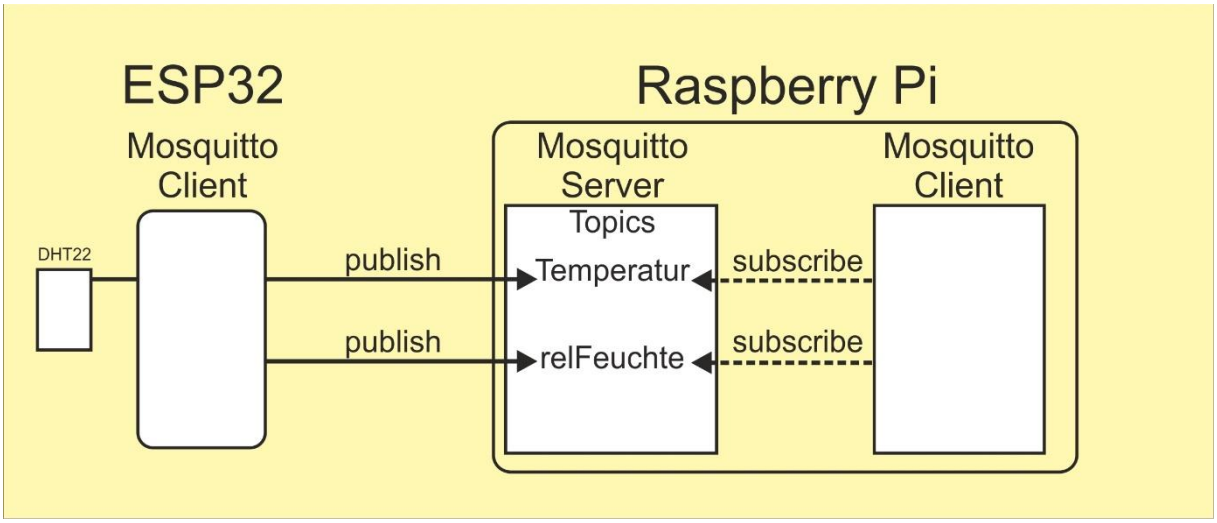

Abbildung 2: ESP8266_mqtt-dhtclient

Zur Wiederholung:
Der MQTT-Server heißt **Broker**. Er sammelt die Meldungen der Clients zu den verschiedenen Themenbereichen, die **Topics** genannt werden. Ein Client kann Nachrichten zu einem bestimmten Thema an den Broker senden, er ist dann ein **Publisher**. Bezieht ein Client Informationen vom Broker dann heißt er **Subscriber** oder Abonnent. Es ist durchaus möglich und Usus, dass beide Vorgänge auf einem Client ablaufen.

Um rasch zu einem realen Beispiel zu kommen, setzen wir zuerst einmal einen ESP8266 als Messwertaufnehmer ein, der seine Ergebnisse an den Mosquitto-Server auf dem Raspberry Pi sendet, er ist also ein Publisher.

Der Anschluss der Datenleitung des DHT22 erfolgt an GPIO14 = D5. Dort muss auch ein Pullup-Widerstand von 4,7kΩ gegen Vcc = 3,3V angeschlossen werden. Die 3,3V nehmen wir vom Pin 3V des ESP8266. Die Schaltskizze verdeutlicht den Zusammenhang. Den DHT22 gibt es auch als Modul mit dreipoliger Stiftleiste. Dort ist der Widerstand bereits mit verbaut. Auf die RGB-LED in der Schaltung gehe ich weiter unten ein.

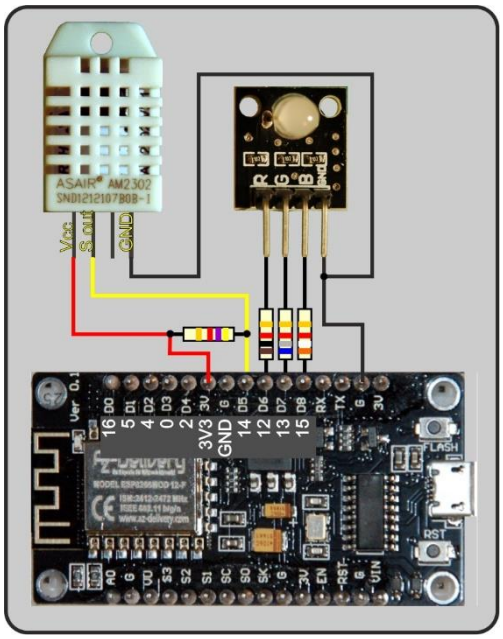

Abbildung 3: DHT22-Client

Die Abfrage des DHT22 ist eine Standardaufgabe für den ESP8266, denn wir brauchen dafür nicht einmal ein Modul hochzuladen, weil dieses bereits in den Kern der Firmware eingebaut ist. Genau fünf Befehle sind für dessen Nutzung nötig. Starten wir Thonny und geben die folgenden Befehle über die REPL-Kommandozeile ein.

Als Vorbereitung müssen wir die Pin-Klasse importieren, die wohnt im Modul **machine**. Dann importieren wir das Modul **dht** und erzeugen eine Instanz **ht** der Klasse **DHT22**, die in **dht** zu Hause ist. **ht** ist unser Sensorobjekt.

```
>>> from machine import Pin
>>> import dht
>>> ht=dht.DHT22(Pin(14))

>>>
>>> ht.measure()
>>> ht.temperature()
22.1
>>> ht.humidity()
38.8
>>>
```

ht besitzt die drei Methoden **measure**(), **temperature**() und **humidity**(). Zuerst muss eine Messung in Auftrag gegeben werden, das macht **measure**(). Dann brauchen wir nur noch die Temperatur und die rel. Luftfeuchte abzuholen. Für eine neue Messung wird der Zyklus einfach wiederholt. Nach dem Start der Messung braucht der DHT22 ein wenig Zeit, die er automatisch erhält, wenn wir die Befehle von Hand eingeben. In einem Programm sollten wir ihm daher 100ms gönnen. Lassen Sie uns eine Funktion **readDHT**() basteln, welche die bisherigen Erkenntisse vereint und außerdem den Ablauf gegen eventuelle Fehler absichert.

```
def readDHT():
    try:
        ht.measure()
        sleep(0.1)
        temp = ht.temperature()
        hum = ht.humidity()
        print("T,H",temp,hum)
        return temp, hum
    except OSError as e:
        print("Sensorfehler ---->",e)
        return None
```

Wenn kein Fehler passiert, läuft der **try**-Teil durch und wir erhalten als Rückgabe ein Tuple mit den Messwerten. Den print-Befehl können wir später weglassen, wenn wir uns von der korrekten Arbeitsweise der Funktion überzeugt haben. Sollte ein Problem auftreten, das einen Absturz des Programms verursachen würde, dann fangen wird das mit dem except-Teil ab und erhalten den Wert None als Rückmeldung. Sie können den Programmtext mit Copy and Paste direkt in das

Eingabefenster vom Thonny transportieren und mit "Enter" übernehmen. Ein nachfolgender Aufruf der neuen Funktion

```
>>> readDHT()
```

liefert dann Ausgabe folgender Form.

```
  T,H 16.3 42.7
(16.3, 42.7)
```

Die erste Zeile stammt vom print-Befehl, die zweite ist der Rückgabewert der Funktion. Tuples werden in MicroPython in runden Klammern notiert.

Dann kommen die Status-LEDs dran. Die blaue zeigt uns den Verbindungsverlauf zum WLAN-Router an, die grüne ist der Heartbeat, das Lebenszeichen des Clientprogramms und die rote informiert über eine Fehlfunktion. Zum Themenkreis In/Out gehört auch die Funktion **blink**.

Zusätzlich machen wir die Flash-Taste am Controllerboard gleich noch für uns nutzbar. Wir schalten den Pin 0 als Eingang und aktivieren den Pullup-Widerstand. Ein Tastendruck legt den Eingang an GND-Potenzial. Hier verrate ich auch die Zuordnung der D-Bezeichnungen der Arduino-IDE auf dem Board zu den GPIO-Pin-Nummern im MicroPython-Programm.

```
statusLed=Pin(15,Pin.OUT,value=0) # blau=2
onairLed=Pin(13,Pin.OUT,value=0)  # gruen=1
errorLed=Pin(12,Pin.OUT,value=0)  # rot=0
led=[errorLed,onairLed,statusLed ]
red,green,blue=0,1,2

taste=Pin(0,Pin.IN,Pin.PULL_UP)
# Pintranslator fuer ESP8266-Boards
# LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
# Arduino-Pins D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                 SC SD

def blink(pulse,wait,col,inverted=False):
    if inverted:
        led[col].off()
        sleep(pulse)
        led[col].on()
        sleep(wait)
    else:
        led[col].on()
        sleep(pulse)
        led[col].off()
        sleep(wait)
```

Diese vorbereitenden Maßnahmen bauen wir am besten fortfolgend schon einmal in ein Programm ein, damit wir sie mit einem Tastendruck auf F5 bequem aufrufen können, es sind doch schon reichlich viele Zeilen, zu viele, um sie jedes Mal erneut einzugeben.

In Thonny öffnen wir das Files-Menü und klicken auf New. Im Editorbereich bekommen wir einen neuen Reiter und darunter eine freie Arbeitsfläche.

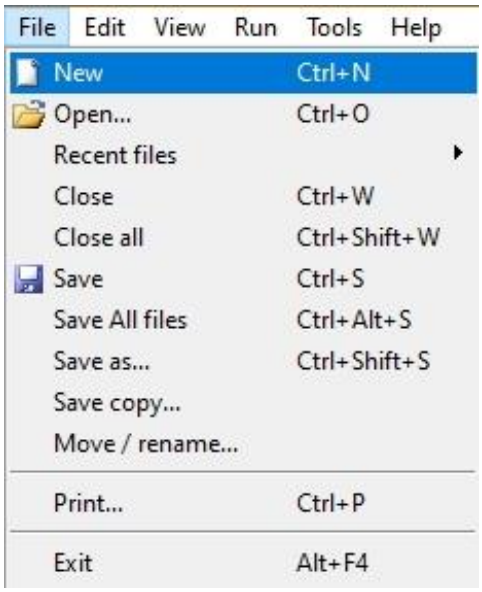

Abbildung 4: Neue Datei anlegen

Dort hinein schreiben wir die folgenden Zeilen und die beiden Funktionsblöcke samt den Definitionen der LED-Anschlüsse. Über **Save as** aus dem File-Menü speichern wir die Datei im Arbeitsverzeichnis (aka workspace) unter dem Namen **dhttest.py** ab. Den Ordner workspace legen wir natürlich zuvor im Explorer in einem Verzeichnis unserer Wahl an.

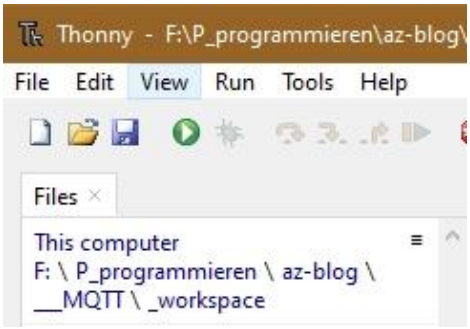

Abbildung 5: Pfad zum Arbeitsverzeichnis

Mit F5 können wir das Programm jetzt starten, danach sind wir nach einem Klick in den Terminalbereich wieder auf der Kommandozeile von REPL (Read Eval Print Loop = MicroPython-Shell), wo wir die Funktionen zum Test aufrufen können. Sobald wir die LED angeschlossen haben, tun wir das zuerst mit der Funktion **blink**(). Sie nimmt drei Positionsparameter, der vierte, **inverted**, ist optional und mit **False** als Defaultwert vorbelegt. In diesem Fall geht die mit **col** ausgewählte LED an, wenn der GPIO-Ausgang auf 1, also 3,3V liegt. Die Kathode der LED muss dann an GND-Potenzial liegen.

col ist die Nummer des Pin-Objekts in der Liste **led**. Listen werden in MicroPython durch eckige Klammern festgelegt. Damit die LED auch durch ihre Farben angesprochen werden können, wurden die Variablen red, green und blue mit den Nummern der Listenplätze belegt. Der Aufruf **statusLed.value(1)** hat also die gleiche Wirkung wie **led[blue].value(1)**. Objekte in der Liste werden durch den Index, die Platznummer, angesprochen, und die steht auch in eckigen Klammern.

Damit etwas blinken kann, muss selbstverständlich die 3-Farben-LED zuerst eingebaut werden. Die Anschlüsse zeigt die Abbildung 3. Die Zuordnung geht auch aus den Zeilen hervor, welche die LED-Ausgänge definieren.

```
statusLed=Pin(15,Pin.OUT,value=0) # blau=2
onairLed=Pin(13,Pin.OUT,value=0)  # gruen=1
errorLed=Pin(12,Pin.OUT,value=0)  # rot=0
```

So, wenn alles angeschlossen ist und das Programm komplett editiert und abgespeichert ist, kann's losgehen. Zur Kontrolle Ihres Programms können Sie den Link zu meiner Vorlage nutzen.

F5

>>> blink(2,3,blue)

Das lässt die blaue LED für 2 Sekunden aufblinken, um sie danach für 3 Sekunden auszuschalten. Von den 3 Sekunden merken Sie außer der Programmverzögerung nichts. Der REPL-Prompt ">>> " erscheint nach dem Ausgehen der LED aber eben erst nach drei Sekunden. Während der Leucht- und Dunkelphase passiert nichts, es kann kein Befehl ausgeführt werden, wenn der ESP8266 schläft, was wir mit dem Befehl **sleep**() erreichen. In unkritischen Situationen können derartige Verzögerungen ohne Nebenwirkung gut ins Programm eingebaut werden.

Der Programmtext bis hier her steht unter dhttest.py zum Download bereit.

Wenn aber während der Wartezeit andere Dinge erledigt werden müssen, dann ist eine andere Vorgehensweise angesagt. Damit sind wir auch schon bei der nächsten Funktion, **TimeOut**().

```
def TimeOut(t):
    start=ticks_ms()
    def compare():
        return int(ticks_ms()-start) >= t
    return compare
```

TimeOut() nimmt eine Zeitdauer in Millisekunden und gibt die Referenz auf eine **Funktion** zurück, die nach Ablauf der Frist den Wert **True** zurückgibt. Dieses trickreiche Konstrukt hat den Namen **Closure**. Über den Umgang damit, können sie sich in diesem PDF-Dokument genauer informieren. Fügen wir jetzt den Funktionstext unserem Programm hinzu. Außerdem müssen wir die Funktion **ticks_ms**() aus dem Modul **time** importieren. Dazu erweitern wir die entsprechende Importzeile um diesen Funktionsnamen. Dann gehen wir zum nächsten Test über.

from time import sleep, ticks_ms

Funktionstext eingeben oder kopieren
Speichern,

F5 und dann

```
>>> abgelaufen=TimeOut(10000)
>>> abgelaufen()
False
>>> abgelaufen()
False
>>> abgelaufen()
True
```

Der REPL-Prompt kommt jetzt sofort nach dem Aufruf wieder, und wir können die nachfolgenden Funktionsaufrufe abschicken. Wie funktioniert das?

Dem Namen **abgelaufen** wird durch Aufruf von **TimeOut**(10000) die Referenz auf die Funktion **compare**() übergeben, die innerhalb von **TimeOut**() definiert wurde. Diese Funktion **compare** greift auf den Parameter **t** zurück, den wir beim Aufruf von **TimeOut**() übergeben haben. Der besonderen Struktur dieses Konstrukts ist es zu verdanken, dass **compare**() auf den Inhalt von **t** auch dann noch zugreifen kann, wenn die Funktion **TimeOut** bereits verlassen wurde. Normalerweise werden ja alle Objekte, die innerhalb einer Funktion, also lokal, definiert wurden, beim Verlassen der Funktion eingestampft.

Damit sind wir in der Lage, Verzögerungen zu programmieren, während deren Ablauf beliebige andere Dinge ausgeführt werden können. Das ist wichtig für unser MQTT-Projekt. Mit der einen Funktion **TimeOut**() können beliebig viele verschiedene Timeouts programmiert werden, die an beliebiger Stelle im Programm abfragbar sind, sofern sie im jeweiligen Namensraum liegen.

Die Netzwerkverbindung für MQTT

Für MQTT brauchen wir einen Netzwerkzugang für unseren ESP8266. Die baut der nächste Abschnitt des Programms auf, nachdem wir einen üblen Dschinni in die Flasche beordert haben. Die Firmware des ESP8266 setzt ohne unser Zutun und Wissen einen Flaschengeist frei, der eine tagelange Fehlersuche hervorrufen kann. Sobald nämlich eine WLAN-Verbindung zu einem Router aufgebaut wird, wacht der Geist auf und bringt einen zur Verzweiflung, weil der ESP8266 ständig Neustarts durchführt. Das liegt an zwei Dingen.

Das erste Übel ist **webrepl**, die Funkkommandozeile, die über einen Browser bedient werden kann, ähnlich wie REPL über das USB-Kabel. Mit dem Flashen der Firmware ist das Teil in Wartestellung. Das Ausschalten geht so. Wir importieren **webrepl_setup** und geben am Prompt ein D ein. Danach starten wir den ESP8266 neu (RST). Diese Operation muss nur einmal nach jedem erneuten Flashen des ESP8266 mit der **Firmware** durchgeführt werden.

```
>>> import webrepl_setup

  > D

Reset durchführen
```

Das zweite Übel ist, dass das AP-Interface automatisch aktiviert ist. Ist zwar Schwachsinn, weil es schließlich Sache des Programmierers ist, welches Interface er benutzen möchte. Dazu kommt, dass der Fehler im Web nirgends dokumentiert ist. Man sucht sich also einen Wolf. An einem Amica bin ich auf diesen Umstand als Fehlerursache gestoßen. Das Board ist eine größere Plaudertasche als der ESP8266-NodeMCU. Das AP-Interface muss also definitiv ausgeschaltet werden, dann läuft alles ganz zahm, wie gewünscht. **Folgende Sequenz muss ins Programm, bevor unser eigentliches STATION-Interface initiiert wird**. MicroPython unterstützt den Parallelbetrieb der beiden Interfaces nicht, das ist der Hintergrund des Problems.

```
nac=network.WLAN(network.AP_IF)
nac.active(False)
nac=None
```

Diese Zeilen müssen mit **jedem Neustart des Controllers abgearbeitet** werden und gehören deshalb in ein Programm, welches das Station-Interface nutzt.

Jetzt aber zum Aufbau der WLAN-Verbindung. Als Erstes muss bei den Importen eine Zeile ergänzt werden.

import network

Das Modul versorgt uns mit diversen Konstanten und mit der Methode **WLAN**(), mit der wir das Netzwerk-Interface-Objekt **nic** im Station-Mode erzeugen.

Wenn sie den Text in Ihr Programm eintippen oder dorthin kopieren, denken Sie bitte daran, Ihre eigenen Zugangsdaten für den WLAN-Router einzusetzen. Die Funktion **hexMAC**() erzeugt aus dem Bytes-Objekt **byteMac** einen vernünftig lesbaren String der MAC-Adresse des aktiven Interfaces. **byteMac**[i] liefert das i-te Zeichen, hex() macht daraus einen String der Form 0xAB und davon interessieren uns die Zeichen ab der 2. Position. Trennzeichen ist ein "-".

```
def hexMac(byteMac):
    """
    Die Funktion hexMAC nimmt die MAC-Adresse im Bytecode
    entgegen und bildet daraus einen String fuer die Rueckgabe
    """
    macString =""
    for i in range(0,len(byteMac)):
        macString += hex(byteMac[i])[2:]
        if i <len(byteMac)-1 :
            macString +="-"
    return macString

# ***************** Connect to WLAN **********************
connectStatus = {
    0: "STAT_IDLE",
    1: "STAT_CONNECTING",
    5: "STAT_GOT_IP",
    2:  "STAT_WRONG_PASSWORD",
    3:  "NO AP FOUND",
```

```
    4:  "STAT_CONNECT_FAIL",
    }

myMQTTserver = "10.0.1.99"
#mySSID = 'Here_goes_your_SSID'
#myPass = 'Here_goes_your_Password'

# Unbedingt das AP-Interface ausschalten
nac=network.WLAN(network.AP_IF)
nac.active(False)
nac=None

# Wir erzeugen eine Netzwerk-STATION-Instanz
nic = network.WLAN(network.STA_IF)
nic.active(False)

# Abfrage der MAC-Adresse zum Eintragen im Router,
# damit die Freigabe des Zugangs erfolgen kann
MAC = nic.config('mac')
myID=hexMac(MAC)
print("Client-ID",myID)
```

Das Dict **connectStatus** liefert die Klartextmeldung, deren Nummer der Befehl **nic.status**() zurückgibt.

Es folgt das Ausschalten des AP-Modus, wie oben beschrieben. Auch das STATION-Interface wird nach dem Instanziieren zunächst deaktiviert. Dann lassen wir uns die MAC-Adresse des Interfaces anzeigen. Bei einstelligen Werten wird eine führende 0 ergänzt.

Die MAC-Adresse (Media Access Control) ist eine eindeutige Kennung jedes Netzwerkinterfaces. Die Kombination aus sechs zweistelligen Hexadezimalwerten muss in der Regel beim Router hinterlegt werden, damit dieser dem Netzwerkgerät den Zugang erlaubt. Man kann den Router zwar auch so einstellen, dass alle neuen Geräte zugelassen werden. Das ist aber sicherheitstechnisches Harakiri und nicht zu empfehlen. Die letzten drei Zeilen liefern uns die übliche Schreibweise der MAC-Adresse des Station-Interfaces.

Wir lassen das Programm bis hierher durchlaufen und tragen dann die angezeigte MAC am Router ein. Wo und wie das zu geschehen hat, hängt vom Gerät ab. Befragen Sie dazu bitte Ihr Router-Manual.

Den gesamten Text des bisherigen Programms können Sie hier herunterladen.

Die nächsten Schritte sind die Aktivierung des Station-Interfaces, die Zuweisung einer IP-Adresse und die Anmeldung am WLAN-Accesspoint. Durch das Angeben einer statischen IP-Adresse wird der DHCP-Client des ESP8266 ausgeschaltet, der versuchen würde, eine IP von einem DHCP-Server zu beziehen. Der läuft in der Regel auf dem WLAN-Router oder auf einem anderen Server im Haus. Die Reihenfolge der Dotted-Quads ist: IP-Adresse, Netzwerkmaske, Gateway=Router-IP,DNS-Server.

```
# Wir aktivieren das Netzwerk-Interface
nic.active(True)

# Aufbau der Verbindung
# Wir setzen eine statische IP-Adresse
nic.ifconfig(("10.0.1.98","255.255.255.0","10.0.1.25",\
              "10.0.1.25"))

# Anmelden am WLAN-Router
nic.connect(mySSID, myPass)
```

Der ESP8266 wartet jetzt auf die Freigabe vom WLAN-Router. In der while-Schleife wird der Verbindungsstatus abgerufen. Solange **isconnected**() nicht **True** zurückgibt, bleiben wir in der Schleife. ein Punkt gefolgt vom Statuscode wird ausgegeben. Weil beim Einsatz aber kein PC angeschlossen ist, das System soll ja autonom laufen, zeigt uns die blaue LED durch langes Leuchten und kurze Pause, dass die Verbindung noch nicht steht. Hat alles geklappt, dann bekommen wir die entsprechenden Mitteilungen im Terminalbereich. Diese letzten print-Zeilen kann man auch ohne Schaden im Produktionssystem löschen.

```
if not nic.isconnected():
    # warten bis die Verbindung zum Accesspoint steht
    while not nic.isconnected():
        print("{}.".format(nic.status()),end='')
        blink(0.8,0.2,blue) # blink blue LED lang-kurz

# Wenn verbunden, zeige Verbindungsstatus & Config-Daten
print("\nconnected: ",nic.isconnected())
print("\nVerbindungsstatus: ",connectStatus[nic.status()])
# War die Konfiguration erfolgreich? Kontrolle
STAconf = nic.ifconfig()
print("STA-IP:\t\t",STAconf[0],"\nSTA-NETMASK:\t",\
      STAconf[1],"\nSTA-GATEWAY:\t",STAconf[2] ,sep='')
```

Zum Vergleich gibt es hier wieder einen Download des Programms bis zu dieser Stelle. Ein Start mit F5 sollte folgendes Ergebnis liefern.

```
>>> %Run -c $EDITOR_CONTENT
Client-ID 10-52-1c-2-50-24
#7 ets_task(4020ee60, 28, 3fff92d0, 10)
1.1.1.
connected:  True

Verbindungsstatus:  STAT_GOT_IP
STA-IP:             10.0.1.98
STA-NETMASK:    255.255.255.0
STA-GATEWAY:    10.0.1.25
>>>
```

Ein Ping vom Raspi oder einem anderen Rechner im selben Teilnetz sollte jetzt auch positiv verlaufen.

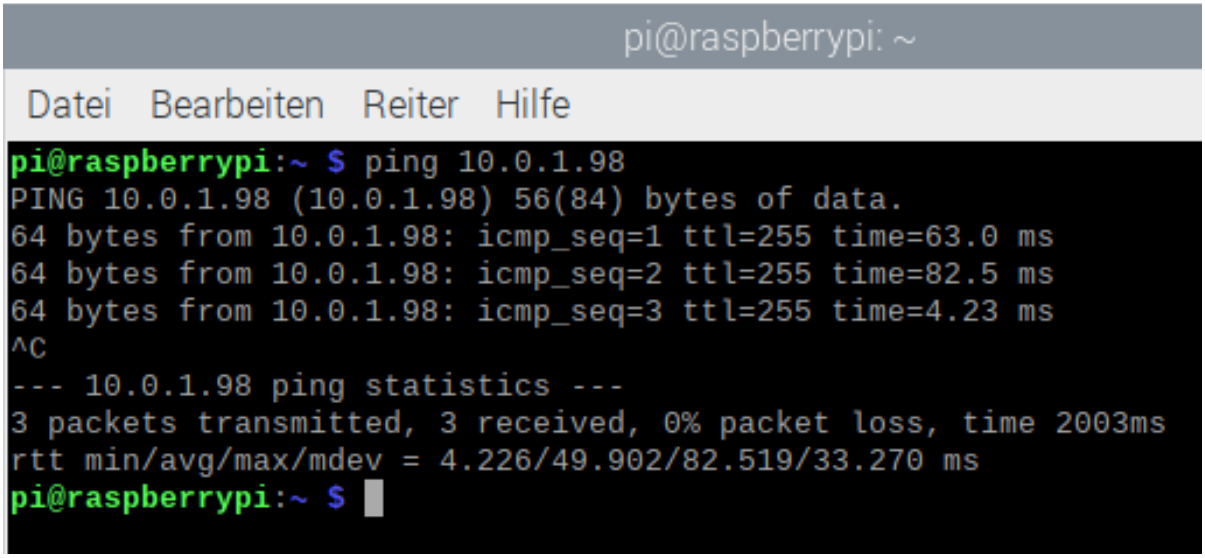

Abbildung 6: Ping-antwort vom DHT22-Client

Zu guter Letzt – die MQTT-Abteilung

Die Verbindung zum MQTT-Server wird durch eine Funktion aufgebaut, die auf das Modul **umqttsimple**.py zurückgreift. Folglich muss dieses Modul zum ESP8266 hochgeladen werden, damit wir die Klasse **MQTTClient** daraus importieren können. Wir fügen diese Zeile und noch eine Kleinigkeit in der Importabteilung hinzu. Die sieht jetzt so aus.

```
from machine import Pin, reset
from time import sleep, ticks_ms
import dht
import network
from umqttsimple import MQTTClient
import esp
esp.osdebug(None)
import gc
gc.collect()
```

Zunächst beschäftigen wir uns mit der Aufgabe, Messwerte an den Broker zu senden. Dieser Teil des Programms arbeitet also als Publisher. Dazu benötigen wir zwei Funktionen und ein paar Variablen.

Als Ergänzung zur Funktionsweise eines Brokers werfen wir einen Blick auf die Hierarchie des Namensraums der Topics. Ähnlich wie bei Verzeichnissen am PC gibt es verschiedene Ebenen, die durch einen Slash "/" voneinander getrennt werden. Neben Basisthemen wie dachboden, garten und wohnung wollen wir hier das Topic **keller** ansprechen. Dort gibt es den Vorratskeller, von dem uns die Temperatur und die rel. Luftfeuchte interessieren. Außerdem wollen wir dort einen Lüftermotor ansteuern. In der Heizung wird uns später die Vor- und Rücklauftemperatur interessieren. Außerdem möchten wir dann die Umwälzpumpe und den Brenner der Heizung fernschalten können. Bleiben wir aber vorerst bei den Topics, die wir durch die folgenden Variablen definieren.

```
nextMsgDelay=5000 # Messintervall in ms
topicTemp="keller/temperature"
topicHum="keller/humidity"
topicAirflow="keller/ventilator"
topicDone="keller/ventilator/done"
```

Beim Schalten eines Relais müssen wir beim ESP8266 NodeMCU V3, der hier im Einsatz ist, berücksichtigen, dass am Pin **Vin** eine externe 5V-Spannungsquelle angeschlossen werden kann, dass aber andererseits dort **nicht** die 5V des USB-Anschlusses zur Verfügung stehen. Für den Betrieb eines Relaismoduls ist daher eine externe 5V-Quelle auch während der Entwicklung notwendig. Später brauchen wir sie für den autonomen Betrieb sowieso. An den Steuerausgang D1 = GPIO5 schließen wir während der Entwicklungsphase als einfachere Lösung eine beliebige LED mit Vorwiderstand gegen Vcc=3,3V an. Die LED wird dann, ebenso wie das LOW-Level-getriggerte Relais durch 0V am Ausgang D1 eingeschaltet.

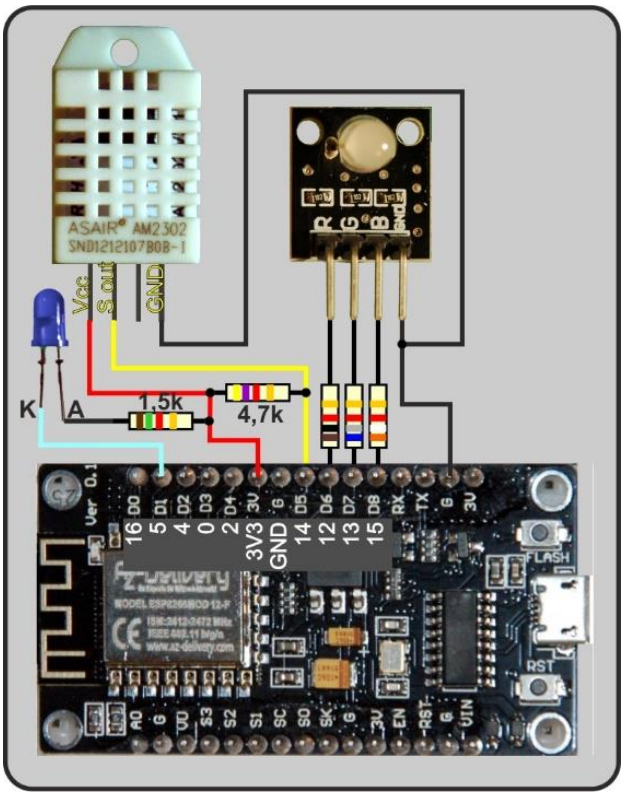

Abbildung 7: DHT22_client_mit LED am Schaltausgang

Die Funktion **connect2Broker**() versucht, die Verbindung zum Mosquitto-Server herzustellen. Gelingt das nicht, dann wird eine Exception geworfen, die wir im Hauptprogramm abfangen müssen. Dazu später mehr.

```
def connect2Broker():
    global myID, myMQTTserver
    # Client-Instanz erzeugen
    client = MQTTClient(myID, myMQTTserver)
    client.connect()
    print("Connected to "+myMQTTserver)
    return client
```

Der Aufruf des Konstruktors der Klasse MQTTClient liefert eine Client-Instanz, welche die Verbindung zum Broker aufbaut. Eine Referenz auf dieses Objekt wird an das aufrufende Programm zurückgegeben.

Kann die Verbindung nicht hergestellt werden, dann wird nach Ablauf der Zeit in Sekunden, die mit sleep() festgelegt wird, der ESP8266 neu gestartet.

```
def restartClient():
    print("connection to {} failed - Rebooting".\
           format(myMQTTserver))
    sleep(10)
    reset()
```

Bevor es in die Dienstschleife geht, versuchen wir die Verbindung zu mosquitto herzustellen. Scheitert das Vorhaben, dann sorgt der except-Teil für einen Neustart des Systems.

```
try:
    client = connect2Broker()
except OSError as e:
    restartClient()

sendNow=TimeOut(nextMsgDelay)
```

sendNow() ist die Timerfunktion, auf die uns die Funktion **TimeOut**() eine Referenz zurückgibt. Nach Ablauf des Intervalls, dessen Dauer wir im Parameter **nextMsgDelay** übergeben, liefert **sendNow**() den Wert **True** zurück. Jetzt geht es in die Mainloop.

```
while True:
    try:
        if sendNow():
            resp=readDHT()
            if resp is not None:
                temp, hum = resp
                temp = (b'{0:3.1f}'.format(temp))
                hum =  (b'{0:3.1f}'.format(hum))
                client.publish(topicTemp, temp)
                client.publish(topicHum, hum)
                blink(0.1,0.05,green)
                sendNow=TimeOut(nextMsgDelay)
            else:
                print("Sensorfehler!")
                blink(0.2,0.3,red)
                blink(0.2,0.3,red)
                blink(0.2,0.3,red)
    except OSError as e:
        restartClient()
```

Vorsichtshalber sichern wir wieder mit try – except ab. Im Falle eines (noch) unbekannten Fehlers würden wir neu durchstarten.

Wenn **sendNow**() True zurückmeldet, ist es Zeit für eine neue Messung und die Übermittlung der Werte. Weil es sein kann, dass aus irgendeinem Grund **readDHT**() den Wert **None** zurückgibt (Sensor-Fehler), lesen wir die Rückgabe erst einmal in eine Variable ein, die wir nachfolgend prüfen. Nur wenn wirklich Werte geliefert wurden, entpacken wir diese in die Variablen **temp** und **hum**. Nach dem Umbau zu Strings publizieren wir die Werte unter dem entsprechenden Topic. Heartbeat-LED kurz blinken lassen und den Timer neu stellen. Fertig! Das ganze Programm dhttest_mit_MQTT.py steht hier zum Download bereit.

Sie meinen, jetzt wäre es an der Zeit, für einen ersten Test. Einverstanden, es geht los! Öffnen wir auf dem Raspi oder auf dem PC über Putty ein Terminal. Dort geben wir die folgende Zeile ein.

mosquitto_sub -t "keller/temperature" -t "humidity"

Dann starten wir auf dem ESP8266 unser Programm – **F5**. Nach 5 Sekunden sehen wir im Terminalfenster und auf der Python-Shell in Thonny die ersten beiden Messwerte – nach 5 Sekunden die nächsten … In beiden Fällen können wir mit Strg + C abbrechen.

Aber – es fehlt ja noch etwas. Wir wollten doch einen Lüftermotor schalten. OK, Motor haben wir gerade keinen bei der Hand, zum Ausprobieren reicht eine LED auch. Die Beschaltung zeigt die Abbildung 7. Für das Produktionssystem können wir die LED durch eines der Relais von dem Zweiermodul ersetzen.

Einen Hinweis zur Auswahl dieses Duomoduls möchte ich an dieser Stelle nicht auslassen. Der Luftspalt zwischen der Niederspannungsseite des Moduls und dem Mittelspannungsbereich (230V) bringt uns mehr Durchschlagssicherheit. Es gibt zwei Single-Versionen. Die eine bringt zwar auch die Absicherung durch den Luftspalt, hat aber keine Befestigungslöcher. Eine zweite weist keinen Luftspalt auf obwohl der Schaltkontakt des Relais und die Eingangskontakte der Niederspannungsseite nur einen Abstand von ca. 2mm haben.

In jedem Fall besteht bei Experimenten mit Netzspannung beim Kontakt Lebensgefahr. Bitte bedenken Sie das beim Experimentieren mit Spannungen größer als 50V.

Auf geht's zur letzten Runde. Um Nachrichten vom Broker zu empfangen, müssen wir

1. ein Topic angeben das wir abonnieren möchten
2. eine Funktion bereitstellen, die ankommende Meldungen anhand des Topics einer bestimmten Aktion zuordnet
3. diese Funktion der Klasse MQTTClient bekanntmachen
4. das gewünschte Topic abonnieren
5. auf ankommende Meldungen vom Broker lauschen
6. den Erfolg der Aktion publizieren

Ad 1. und 6.
Das haben wir weiter oben bereits erledigt, als wir die Variable **topicAirflow** definiert haben. An dieser Stelle bauen wir aber noch ein weiteres Topic ein, über welches wir eine Rückmeldung an den Brocker publizieren können

topicAirflow="keller/ventilator"
topicDone="keller/ventilator/done"

Im Bereich der Variablendeklarationen ergänzen wir die Definition des Pins für den Schaltausgang.

relais=Pin(5,Pin.OUT,value=1)

Ad 2. und 6.

```
def messageArrived(topic,msg):
    topic=topic.decode()
    msg=msg.decode()
    print("Topic:",topic,"   Nachricht:",msg)
    if topic == topicAirflow:
        if msg=="an":
            relais.value(0)
            client.publish(topicDone, "AN")
        else:
            relais.value(1)
            client.publish(topicDone, "AUS")
```

Diese Funktion wandelt die empfangenen Bytes-Objekte in Strings um, vergleicht, ob das Thema stimmt und führt das aus, was es gegebenenfalls zu tun gibt. Hier wird der Ausgang zum Relais umgeschaltet. Nach dem Schaltvorgang erfolgt die Rückmeldung an den Broker, deren Topic vom Auftraggeber abonniert werden kann.

Ad 3. und 4.

```
def connect2Broker():
    # Client-Instanz erzeugen
    client = MQTTClient(myID, myMQTTserver)
    client.set_callback(messageArrived)  # (A)
    client.connect()
    client.subscribe(topicAirflow)       # (B)
    print("Connected to:", myMQTTserver)
    print("Subscribed to:",topicAirflow)
    return client
```

Die Funktion **connect2Broker**() erhält 2 wesentliche Zeilen an Zuwachs.
(**A**) Wir sagen dem client-Objekt, welche Funktion auszuführen ist, wenn eine Antwort vom Broker ankommt. Dafür übergeben wir eine Referenz auf die Funktion **messageArrived**.
(**B**) Wir abonnieren das Topic, das in der Variablen **topicAirflow** hinterlegt ist.

Ad 5.
In der while-Schleife fügen wir den Lauschbefehl für abonnierte Topics ein. Er übergibt die Ausführung an die Funktion **messageArrived**(), falls eine neue Nachricht eingetroffen ist.

```
while True:
    try:
        client.check_msg()
        if sendNow():
```

Auch hierzu gibt es wieder das vollständig ergänzte Programm dhtclient.py zum Download.

Zum Testen öffnen wir ein weiteres Terminal am Raspi. Von dort schicken wir die folgenden Nachrichten ab.

mosquitto_pub -t "keller/ventilator" -m "an"
mosquitto_pub -t "keller/ventilator" -m "aus"

Haben Sie die LED am ESP8266 beobachtet? An – aus. Außerdem wurden folgende Zeilen im Thonny-Terminal ausgegeben.

Topic: keller/ventilator Nachricht: an
Topic: keller/ventilator Nachricht: aus

In einem zweiten Terminal können wir auch die Rückmeldung an den Broker testen.

mosquitto_sub -t "keller/ventilator/done"

Antwort:
AN
AUS
…

Damit sind wir am Ende dieses Blogposts angekommen. In der nächsten Episode wird es darum gehen, die Daten, welche der Broker von den verschiedenen Messstellen sammelt, an einer Steuereinheit auf einem LCD-Display auszugeben. Die Tasten des LCD-Keypads werden wir verwenden, um Befehle an die Messstationen zu senden. Die Rückmeldungen werden wir wieder am Display sehen. Die Steuerung wird ein ESP32 übernehmen.

Doch halt, es war doch noch von einem weiteren ESP8266 die Rede. Was soll damit denn geschehen?

Nun, der andere ESP8266 soll die Heizung überwachen, Vorlauf- und Rücklauftemperatur. Nebenbei soll damit auch die Umwälzpumpe geschaltet werden können und warum eigentlich nicht auch zusätzlich die ganze Anlage? Dafür sind zwei versiegelte DS18B20-Sensoren gedacht und für jeden Schaltvorgang ein Relais von einer Zweierstufe. Die Programmierung unterscheidet sich bis auf die Sensoren nicht viel von dem heutigen Programm. Das Abfragen des DS18B20 ist ebenso

simpel wie beim DHT22. Wenn Sie Hilfestellung brauchen, finden Sie diese schon einmal vorab im Beitrag zum Frostwächter im Gewächshaus.

Im Softwarekapitel – Autostart ist beschrieben, wie man das Programm als **boot**.py zum ESP8266 schickt, um diesen autonom damit booten zu lassen. Diese beiden Punkte überlasse ich Ihnen heute als Hausaufgabe.

ESP8266 und ESP32 stellen wir nächstes Mal, zusammen mit dem Display, als autarke Steuer- und Überwachungseinheit in Dienst.

PDF-Version dieser Folge gibt es hier als Download: esp_mqtt-client1_ger.pdf.