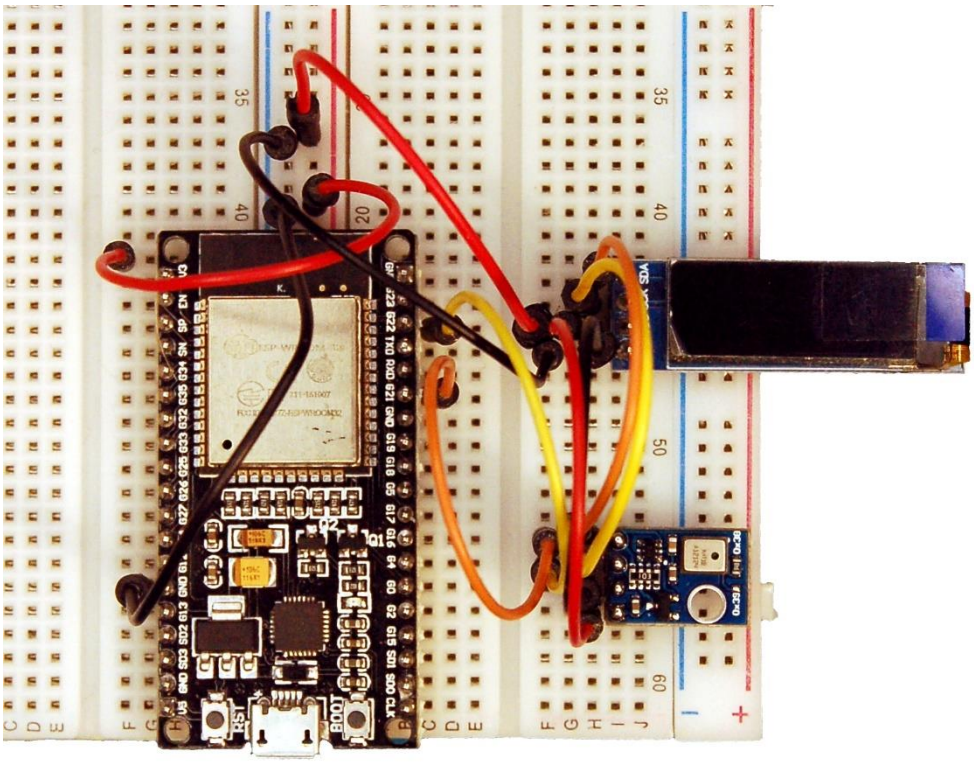

AHT10 mit ESP32 - Aufbau

Diese Blogfolge, die für Anfänger gut geeignet ist, gibt es auch als PDF-Dokument.

Die Menschen freuen sich aufs Frühjahr und manche planen bereits die Bepflanzung ihres Gartens oder Balkons. Die Pflanzen dafür kann man kaufen oder – selbst heranziehen. Aber, was tun, wenn man mal eine Woche nicht zuhause ist. Wer gießt die Pflänzchen, und kuschelt sie, wenn es an der Fensterbank nachts frisch wird? Und wenn es dunkel wird, wer macht Licht in der besten spektralen Zusammensetzung an? Die Antworten darauf und diverse Tipps zu MicroPython bekommen Sie in der neuen Staffel von

MicroPython auf dem ESP32 und ESP8266

heute

Teil 1 – Kleinklima testen mit dem AHT10

Mit den oben genannten Schwierigkeiten hatte ich in den letzten Jahren immer wieder zu tun. Deshalb habe ich beschlossen, dieses Jahr einen Assistenten für die Anzucht und Pflege meiner Sämlinge zu bauen. Das Kleinklima in der Saatbox wird ein AHT10 überwachen, dessen Programmierung ich Ihnen in dieser Blogfolge vorstellen werde. Weitere Beiträge zu den einzelnen Baugruppen folgen.

Als Zuhause für die Setzlinge habe ich mir aus dem Baumarkt eine Klarsichtbox besorgt. Dort hinein passt eine Schale, die eigentlich für Besteckteile gedacht ist. Ich habe sie mit einer Platte aus wasserfest verleimtem Sperrholz abgedeckt, in welches

ich 15 Löcher von 4cm Durchmesser gebohrt habe. Sie dient als Halter für die stabilen, wieder verwendbaren Kunststoff-Schnapsgläschen aus dem Supermarkt, deren Böden ich mit einigen Bohrungen versehen habe, damit die Pflänzchen, die dort wohnen sollen, auch Wasser aus der Schale aufnehmen können.

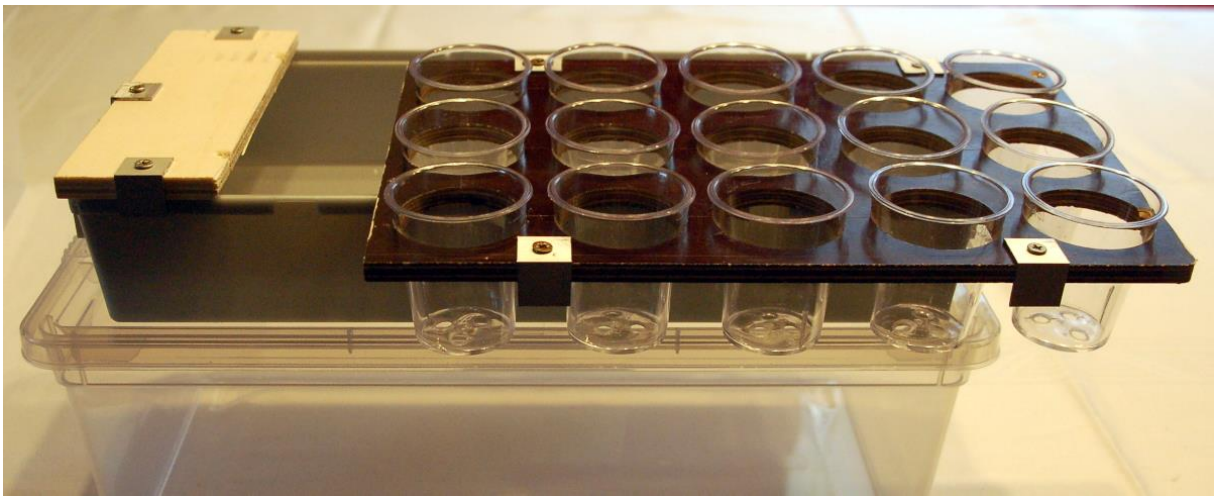

Abbildung 1: Wasserschale mit Einsatz

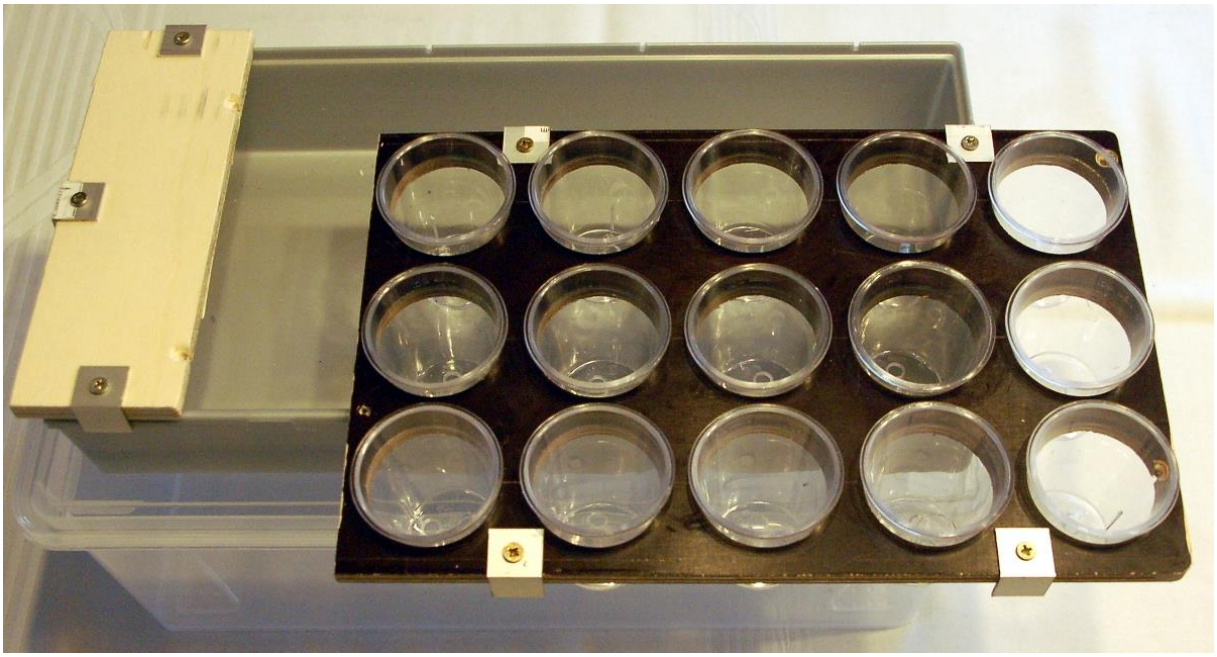

Abbildung 2: Platz für 15 Anzuchtbecher und die Klimahardware

In den folgenden Beiträgen werde ich die verschiedenen Stationen für die Umsetzung des Projekts beleuchten. Hardware und Software sind modular gehalten, sodass sie jeweils eine bestimmte Thematik behandeln. Die Module sind aber voneinander unabhängig. Damit haben Sie die Möglichkeit, die Teile auszuwählen, die Sie persönlich umsetzen möchten. Beginnen wir heute damit, dem Kleinklima in der Saatbox auf die Schliche zu kommen, das durch die Größen Temperatur und relative Luftfeuchtigkeit bestimmt wird. Der AHT10 kann beides messen und über den I2C-Bus an den Controller, einen ESP32, übermitteln.

Hardware

Als Controller habe ich einen ESP32 gewählt, weil der mit genügend frei wählbaren GPIO-Anschlüssen aufwarten kann, davon brauchen wir beim Vollausbau 10 Stück aufwärts.

Die ESP32-Modelle in der Teileliste sind alle brauchbar. Lediglich beim ESP32-Lolin-Board muss für den I2C-Anschluss SDA statt GPIO21 der GPIO25-Pin genommen werden.

1	ESP32 Dev Kit C unverlötet oder ESP32 Dev Kit C V4 unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder NodeMCU-ESP-32S-Kit oder ESP32 Lolin LOLIN32 WiFi Bluetooth Dev Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	AHT10 Feuchte und Temperatursensor oder AHT10 Feuchte und Temperatursensor
1	MB-102 Breadboard Steckbrett mit 830 Kontakten
	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F evtl. auch 65Stk. Jumper Wire Kabel Steckbrücken für Breadboard

Die Software

Fürs Flashen und die Programmierung des ESP32:

Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

v1.19.1 (2022-06-18) .bin

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für das OLED-Display
aht10.py API für das Temperatur-Feuchte-Modul
temphum.py Demoprogramm

MicroPython - Sprache - Module und Programme

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung (english version). Darin gibt es auch eine Beschreibung, wie die Micropython-Firmware (Stand 18.06.2022) 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.

Schaltung und Aufbau

Unsere Schaltung ist sehr übersichtlich, sie enthält nur den ESP32, einen AHT10 und ein 3-zeiliges OLED-Display.

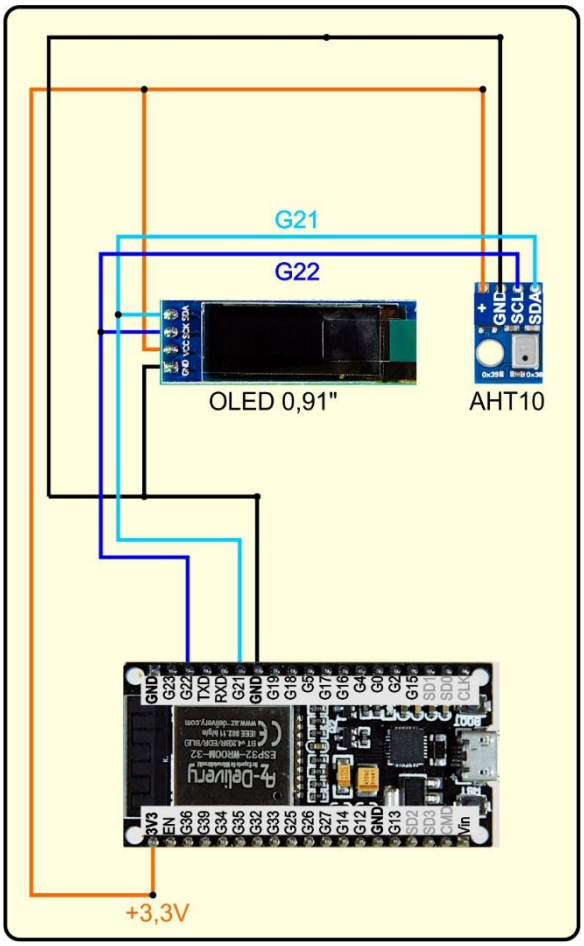

Abbildung 3: AHT10 am ESP32

Der Aufbau erfolgt auf zwei längs aneinander gesteckten Breadboards mit einer mittigen Stromschiene. Die Anordnung bietet genug Platz für die künftigen Erweiterungen des Projekts.

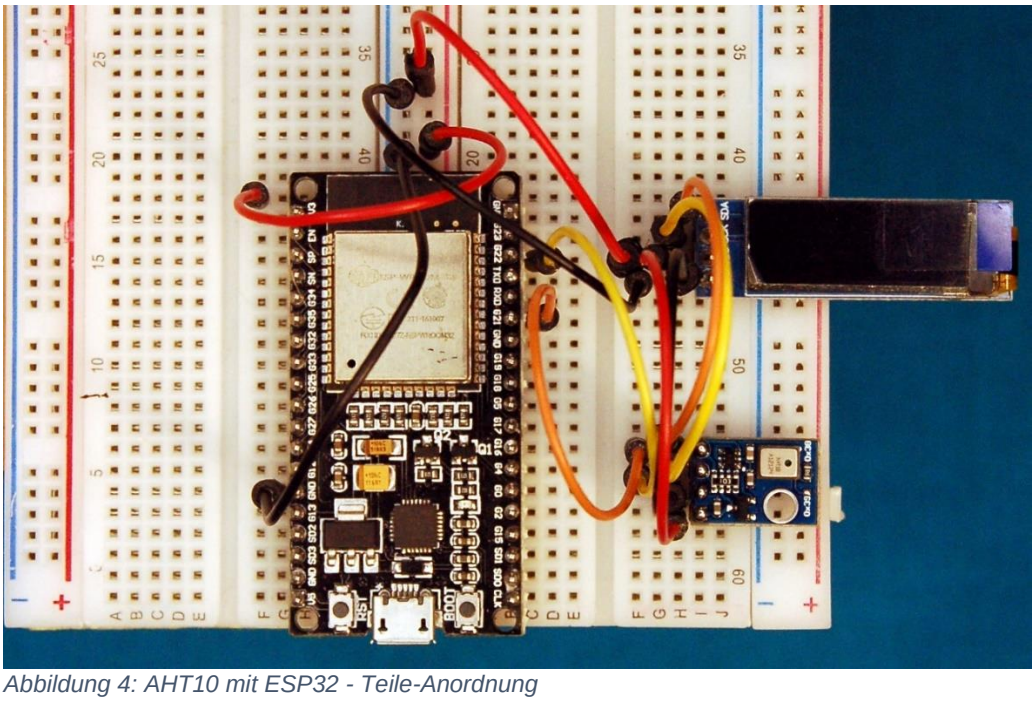
Abbildung 4: AHT10 mit ESP32 - Teile-Anordnung

Wir werden in Kürze ein paar Dinge ausprobieren, da ist es gut, wenn Sie schon einmal die Schaltung zusammenstöpseln.

AHT10 versus DHT11, DHT22

Was macht den Unterschied zwischen ATH10 und den DHT-Brüdern aus? Ganz entscheidend ist die Anbindung an einen Microcontroller. Die DHTs brauchen pro Stück einen eigenen GPIO-Pin, denn die haben keine Adresse wie ein I2C-Baustein, zum Beispiel der AHT10. An einem DHT-GPIO kann natürlich auch kein andres Gerät angeschlossen sein.

I2C ist das Akronym für Inter-Integrated Circuit. Dieser Bus wurde von Philips eingeführt, um verschiedene Baugruppen in einem Gerät miteinander zu vernetzen. Bus heißt, dass mehrere Bausteine über die Leitungen SDA und SCL (Serial Data und Serial Clock) miteinander verbunden sind und über eine sogenannte Hardware-Adresse zu Beginn der Kontaktaufnahme angesprochen werden.

Uns ermöglicht dieses Verfahren, dass AHT10 und OLED-Display von denselben GPIO-Pins versorgt werden. Leider lässt sich die Hardware-Adresse beider Module nicht verändern, sodass jeweils nur ein Exemplar jeder Sorte am Bus liegen darf.

Weitere I2C-Bausteine, die auch Temperatur und relaltive Luftfeuchte messen können sind der SHT21 oder der BME280. Der BME280 wird in einem der nächsten Blogposts der Star of the county down. Die Verwendung des SHT21 habe ich schon einmal in der Blogfolge zum Mammut-Matrixdisplay vorgestellt. Dort gab es Probleme mit einem solchen Baustein, der, wie sich herausstellte ein korruptes Innenleben besaß.

Natürlich gibt es neben dem Interface noch weitere Unterschiede, von denen ich einige in der Tabelle 1 zusammengestellt habe.

Eigenschaft	DHT11	DHT22	AHT10	HTU21	BME280
Temperaturbereich	0..80°C	-40..100°C	0..100°C	40..125°C	-40..85°C
Auflösung	1°C	0,1°C	0,01°C	0,01°C	k. A.
Rel Feuchte	0..80%	0..100%	0..100%	0..100%	0..100%
Auflösung	1%	0,1%	0,024%	0,04%	0,008%
Preis	*	* .. **	*	*	***
I2C	-	-	Ja	Ja	Ja
Komplexität der MicroPython-Programmierung nach Datenblatt	*	*	**	**	****

Tabelle 1: Bausteinvergleich

Signale auf dem I2C-Bus

Immer wenn es Probleme bei der Datenübertragung gibt, setze ich gerne das DSO (Digitales Speicher Oszilloskop) ein, oder ein um Welten billigeres, kleines Tool, einen Logic-Analyzer (LA) mit 8 Kanälen. Das Ding wird an den USB-Bus angeschlossen und zeigt mittels einer kostenlosen Software, was auf den

Busleitungen los ist. Dort, wo es nicht auf die Form von Impulsen ankommt, sondern lediglich auf deren zeitliche Abfolge ist ein LA Gold wert. Und, während das DSO nur Momentaufnahmen des Kurvenverlaufs liefert, kann man mit dem LA über längere Zeit abtasten und sich dann in die interessanten Stellen hineinzoomen. Eine Beschreibung zu dem Gerät finden Sie übrigens in dem Blogpost "Logic Analyzer - Teil 1: I2C-Signale sichtbar machen" von Bernd Albrecht. Dort ist auch beschrieben, wie man den I2C-Bus abtastet.

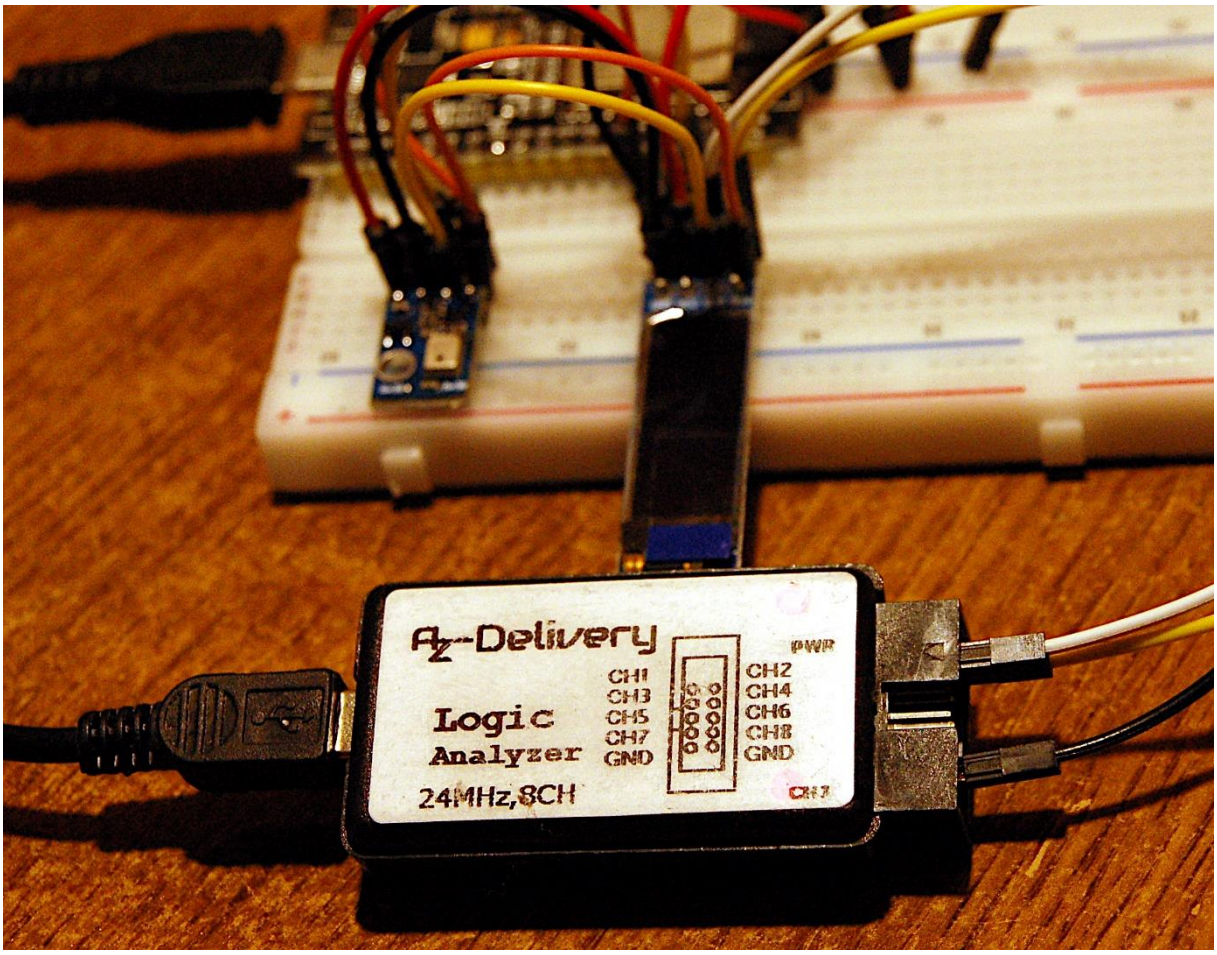

Abbildung 5: Logic Analyzer am I2C-Bus

Ich stelle Ihnen hier einmal auszugsweise die Übertragung der Hardware-Adresse (HWADR) an den AHT10 gefolgt von einem Daten-Byte vor. Dazu lege ich den Pin 1 des Logic Analyzers mit einem Jumperkabel an die SCL-Leitung und den Pin 2 des Logic Analyzers an die SDA-Leitung des I2C-Busses. GND verbinde ich mit GND.

Nun starte ich Thonny, lege eine neue Datei im Editor an und gebe den folgenden Text ein.

```
from machine import Pin,SoftI2C
import sys

if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
else:
    raise RuntimeError("Unknown Port")
```

Die Variable sys.platform sagt uns, welchen Controllertyp wir verwenden. Davon abhängig wird ein I2C-Objekt instanziiert, welches wir gleich zu einem ersten Test verwenden werden. Ist der ESP32 und seine Beschaltung bereit und der Controller mit dem PC verbunden? Dann starten wir das Programm mit der F5-Taste. Das geht schneller als mit der Maus den grünen Startbutton mit dem weißen Dreieck anzufahren und zu klicken. Läuft das Programm ohne Fehlermeldung durch, dann ist alles OK. Im Terminal geben wir jetzt den ersten I2C-Befehl ein, wollen sehen, wer denn so alles da ist. Eingaben formatiere ich fett, die Antworten vom System kursiv.

>>> i2c.scan()
[56, 60]

Die eckigen Klammern stellen in MicroPython eine sogenannte Liste dar. Sie enthält als Elemente die 7-Bit-Hardwareadressen der gefundenen I2C-Bausteine. 56 =0x38 ist die Nummer auf die der AHT10 reagiert, 60 =0x3C muss daher das OLED-Display sein. Der Bus steht bereit und wartet auf die Kommunikation des ESP32 mit den angeschlossenen Parteien.

Damit die Buschtrommel funktioniert, muss es einen geben, der den Takt angibt, das ist der ESP32, er ist der Chef und der heißt bei I2C Master. Der AHT10 und das OLED-Display sind die Sklaven, die Slaves. - Ohhh! Ich dachte, das Zeitalter der Sklaverei ist schon lange vorbei! – Sei' drum, der Master gibt an, mit welchem Slave er zu parlieren wünscht. Dazu erzeugt er eine **Start condition** als eine Art "Achtung an alle"-Nachricht. Dann legt er die Hardware-Adresseauf den Bus, an die er als LSB (Least Significant Bit) eine 0 anhängt, wenn er dem Slave Daten schicken möchte (Schreiben) oder eine 1, wenn er vom Slave eine Antwort erwartet (Lesen). Der Hardware-Adresse folgt im Falle eines Schreibzugriffs das zu sendende Datenbyte. Zum Abschluss kommt als "OK, das war's", eine **Stop condition**. Wie die Signalfolge aussieht, das schauen wir uns gleich an.

Ich gebe im Terminalbereich von Thonny den folgenden Befehl ein, aber schicke ihn noch nicht ab. Der Logic Analyzer ist angeschlossen wie oben beschrieben.

>>> i2c.writeto(0x38,b"\xBA")

Als nächstes starte ich das Programm Logic 2. Im Menü am rechten Fensterrand klicke ich auf **Analyzers** und dann auf das Pluszeichen. Aus der Liste wähle ich **I2C**.

Jetzt ist alles vorbereitet, wir starten mit der Taste R die Aufzeichnung und wechseln schnell zu Thonny und drücken die Entertaste, um den Befehl abzuschicken. Danach wieder schnell zurück zu Logic 2 und mit R die Aufzeichnung stoppen.

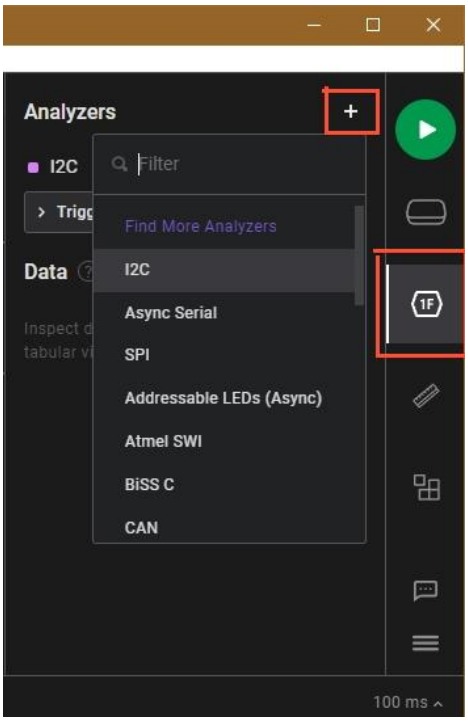

Abbildung 6: Logic 2 - Analyzers - I2C

Um die gesamte Signalfolge zu sehen, setze ich den Mauszeiger auf eine der Signalbahnen und drehe das Mausrad zu mir. Irgendwann taucht ein vertikaler Strich in den Aufzeichnungsbahnen auf. Dann setze ich den Mauszeiger auf diesen Strich und drehe das Mausrad von mir weg. Der Strich wird immer breiter, bis ich die Signalpulse erkennen kann. Ich habe die wichtigsten Stellen mit Zeitmarken gekennzeichnet.

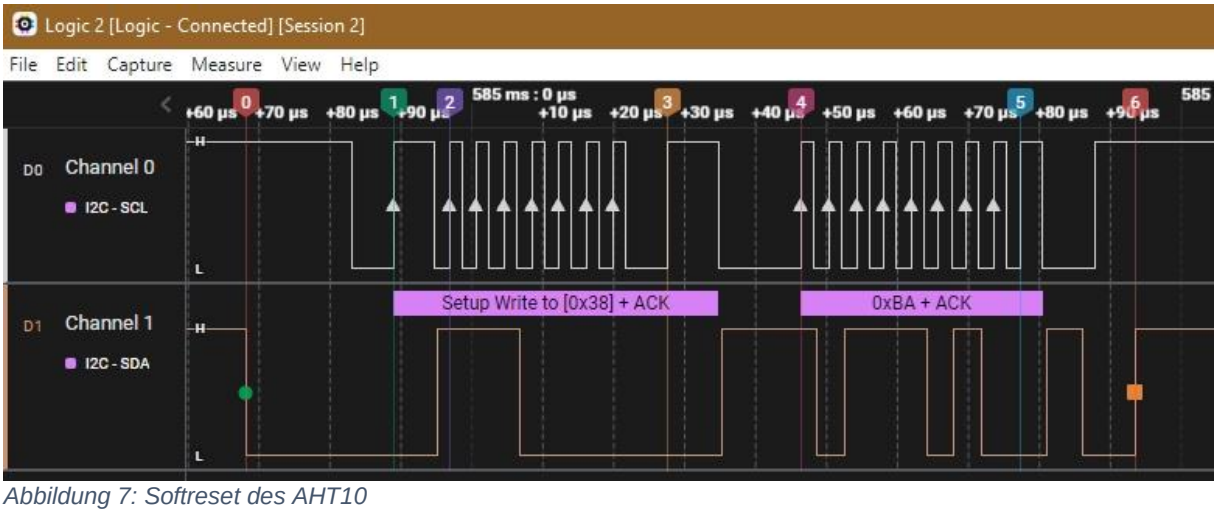

Abbildung 7: Softreset des AHT10

Die Marke 0 kennzeichnet die **Start Condition**, SDA geht auf LOW, während SCL HIGH ist. Dann wartet der Master ca 15µs, damit die Slaves aus den Federn kommen, um auf eine Hardware-Adresse zu empfangen. Wir haben 0x38 = 0b00111000 angegeben. Der ESP32 macht daraus durch Linksschieben der Bits 0b011100 und hängt als LSB eine 0 an, weil er dem AHT10 ein Byte senden möchte. Dann legt er das Adressbyte auf den Bus. Mit jeder steigenden Flanke (zum Beispiel

Marke 1 und 2) seines Taktsignals auf SCL sagt er den Slaves, dass sie sich den Zustand der SDA-Leitung jetzt merken sollen. In Abbildung 4 können Sie ablesen, dass der Master das Byte 0x70 = 0b01110000 gesendet hat. Mit dem Acknowledge-Bit (ACK) signalisiert der Slave an der neunten steigenden Flanke, ob er das Byte vom Master empfangen und als seine Adresse erkannt hat. In diesem Fall zieht er, wie hier der AHT10, die SDA-Leitung auf 0. Die anderen Slaves am Bus ziehen ihre Schlafmützen wieder über die Ohren und pennen weiter. Auf die gleiche Weise sendet der Master als nächstes das Byte 0xBA, das der AHT10 wieder mit einem ACK quittiert. Danach gibt der Slave die SDA -Leitung wieder frei. Der Master erzeugt keinen neuen Taktimpuls, SCL bleibt auf 1. Wenn jetzt der Master die SDA-Leitung auch freigibt, freigegebene Leitungen gehen durch die Pullup-Widerstände auf HIGH-Pegel, ist das die **Stop Condition**.

Nach diesem Schema arbeiten die I2C-Routinen des MicroPython-Moduls **aht10**.py. Bevor wir diese näher anschauen, muss ich noch auf eine MicroPython-spezifische Sache hinweisen. MicroPython kennt zwar keine Typdeklarationen für Variablen wie sie in C üblich sind. Dennoch gibt es ein paar einfache Datentypen, die MicroPython zum Teil transparent behandelt, ohne zu murren.

```
>>> a="123"
>>> a
'123'
>>> a=123
>>> a
123
```

Aus einer Zeichenkette wird durch die zweite Anweisung flugs eine Integer-Zahl. Aber während in LUA Node-MCU oder Perl folgende Zeile den Wert 246 ergibt,

```
a+"123"
246
```

erhalten wir in MicroPython eine Fehlermeldung, weil MicroPython keine impliziten Typumwandlungen kennt.

```
>>> a+"123"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't convert 'int' object to str implicitly
```

MicroPython kommt aber mit einer expliziten Typumwandlung zurecht:

```
>>> a+int("123")
246
```

Im Fall der I2C-Anweisung

```
>>> i2c.writeto(0x38,b"\xBA")
```

erwartet MicroPython als Hardware-Adresse eine Integerzahl. Statt 0x38 könnten wir auch die 56 (dezimal) verwenden. Als Daten-Byte(-s) erwartet die Methode **writeto**()

aber einen Datentyp, der das Bufferprotokoll unterstützt, aber das tun Integer-Zahlen nicht, Strings auch nicht. Deshalb muss man die Daten entweder als bytes-Objekt übergeben, wie ich es hier gemacht habe, oder als bytearray. Das sähe dann so aus:

>>> buf=bytearray(6)
>>> buf[0] = 0xBA
>>> i2c.writeto(0x38,buf[0:1])
>>> buf[0]
186
>>> buf[0:1]
bytearray(b'\xba')

buf ist ein bytearray und das unterstützt das Bufferprotokoll. Während **buf**[0] den Integerwert 0xBA darstellt, der das nicht tut. Ich darf also nicht **buf**[0] übergeben sondern ich muss den Teil (Slice = Scheibe) von **buf** nehmen, der nur das erste Element enthält, **buf**[0:1].

Nach diesen Spitzfindigkeiten kommen wir jetzt zur Klasse **AHT10**, die im MicroPython-Modul **aht10**.py deklariert wird.

Das Modul zum AHT10

```
# aht10.py
from time import sleep_ms
from sys import exit
from math import log
```

Einige Methoden der Klasse **AHT10** müssen Wartezeiten einhalten, bis der AHT10-Chip den empfangenen Auftrag ausgeführt und die Daten bereitgestellt hat. Dafür importieren wir die Funktion **sleep_ms**() vom Modul **time**. Zur Berechnung des Taupunkts benötigen wir die Funktion **log**, die den Logarithmus Naturalis berechnen kann. Sie steckt im Modul **math**.

Es folgt die Deklaration der Klasse **AHT10**.

```
class AHT10:
    HWADR   = const(0x38)

    INITCMD = (0xE1, 0x08, 0x00)
    TRIGGERCMD = (0xAC, 0x33, 0x00)
    SOFTRSTCMD = const(0xBA)
    BUSSY = const(0x80)
    CALIBRATED = const(0x08)
    NENNER = const(2**20)
    K2 = 17.62
    K3 = 243.12
```

Die Funktion des AHT10 wird in ungewöhnlicher Weise gesteuert. Es werden keine Register angesprochen, sondern Kommandos gesendet. Für die Initialisierung und die Triggerung einer Messung werden drei Bytes als Kommando schreibend

übertragen für den Softreset, den wir schon kennengelernt haben nur eines. Bei einem Lesebefehl überträgt der AHT10 stets sechs Bytes, wovon das erste das Statusbyte ist, die folgenden fünf enthalten Temperatur- und Feuchte-Rohwerte.

Solange das **Busy**-Flag (MSB im Status-Byte) auf 1 ist, hat der AHT10 noch zu tun und der ESP32 muss warten. Außerdem funktioniert die Messung nur dann, wenn das **calibrated**-Flag im AHT10 gesetzt ist. Das wird durch die Initialisierung erledigt. Das Datenblatt gibt leider nicht die umfassende Auskunft, die man als Programmierer erwartet. Ein Teil ist überdies mit fernöstlichen Hieroglyphen durchsetzt, deren Bedeutung man durch Experimente herausfinden muss. Das hat hier viel Zeit gekostet und nur vereinzelt etwas gebracht.

Zur Berechnung der wahren Messwerte benötigt man einen Teiler von 2 hoch 20, den ich in die Konstante NENNER gesteckt habe. K2 und K3 sind Konstanten, die man unter Anwendung der Magnus-Formel braucht, um aus Temperatur und rel. Luftfeuchte die Taupunkttemperatur berechnen zu können.

```
    def __init__ (self, i2c):
        self.i2c=i2c
        sleep_ms(20)
        self.buf=bytearray(6)
        self.reset()
        self.temp= 9999
        self.hum = 9999
        print("AHT10 bereit")
        try:
            self.begin()
        except:
            print("Device Error")
            sys.exit()
```

Dem Konstruktor der Klasse ist bei der Instanziierung nur ein I2C-Objekt zu übergeben, das im aufrufenden Programm erzeugt werden muss. Bis der AHT10 malochen kann, braucht er eine Startzeit von 20ms.

Ein bytearray der Länge 6 wird deklariert, mit dem wir später den I2C-Verkehr abwickeln werden. Standardmäßig erfolgt ein Softreset. Liefert die Initialisierung durch die Methode **begin**() **True** zurück, dann ist der AHT10 einsatzbereit, andernfalls bekommen wir eine Fehlermeldung und das Programm bricht ab.

```
    def reset(self):
        self.buf[0]=SOFTRSTCMD
        self.i2c.writeto(HWADR,self.buf[0:1])
        sleep_ms(20)
```

Der Softreset wird in der bereits oben beschriebenen Art und Weise mit dem globalen bytearray **buf** erledigt. Danach wieder, wie beim Start, 20ms Ruhepause.

```
    def begin(self):
        for i in range(0,3):
            self.buf[i]=AHT10.INITCMD[i]
        self.i2c.writeto(HWADR,self.buf[0:3])
        self.waitRDY()
        if not(self.status() & CALIBRATED):
            raise RuntimeError ("AHT10 nicht initialsiert")
```

Wir befüllen das bytearray **buf** mit den drei Bytes des Initialisierungskommandos **INITCMD** in der for-Schleife, um dann auch nur die die Scheibe mit den ersten drei Elementen des Arrays zum AHT10 zu senden. Denken Sie daran, dass MicroPython sequenzielle Datentypen ab der 0 indiziert und das letzte Element einen Index besitzt, der um 1 kleiner ist wie die angegebene Obergrenze, [0:3] adressiert demnach die Elemente 0,1 und 2 als Teilbereich (Slice) des Arrays **buf**. Wir warten darauf, dass das BUSSY-Flag 0 wird, und prüfen das CALIBRATED-Flag durch Undieren mit der Maske. Ist es gesetzt, hat der Ausdruck den Wert 16 und wird als **True** eingestuft. Andernfalls kommt 0 heraus, was einem **False** entspricht. Eine **Runtime-Exception** zu werfen, ist eine andere Möglichkeit einen Programmabbruch zu provozieren, wenn die Exception nicht durch das aufrufende Programm abgefangen wird.

```
    def waitRDY(self):
        s=self.status()
        while s & BUSSY:
            sleep_ms(2)
            s=self.status()
```

waitRDY() ruft fortlaufend den Status des AHT10 durch Aufruf der Methode **status**() ab. Die while-Schleife wird erst verlassen, wenn der Ausdruck **Status-Byte & BUSSY** eine 0 liefert.

```
    def status(self):
        self.readSensor() # First Byte is status byte
        return self.buf[0]
```

status() lässt über **readSensor**() die 6 Bytes vom Sensor in das bytearray **buf** einlesen und gibt nur den Zahlenwert des ersten Arrayelements, **buf**[0], zurück.

```
    def readSensor(self):
        self.i2c.readfrom_into(self.HWADR, self.buf)
```

readSensor() bildet die unterste Hierarchie-Ebene beim Einlesen. **i2c.readfrom_into**() liest so viele Bytes vom AHT10, wie in das bytearray **buf** passen und das sind 6 Stück.

```
    def triggerDevice(self):
        for i in range(0,3):
            self.buf[i]=AHT10.TRIGGERCMD[i]
        self.i2c.writeto(HWADR,self.buf[0:3])
```

Jede neue Messung muss getriggert werden, es gibt keinen Freilaufmodus. **triggerDevice**() sendet in bekannter Weise die drei Kommandobytes von **TRIGGERCMD**.

```
    def getValues(self):
        self.waitRDY()
        self.readSensor()
```

getValues() wartet artig auf das Ende des Messvorgangs und lässt dann die Werte einlesen. Diese Methode ist eigentlich überflüssig, denn Letzteres hat ja **waitRDY**() bereit getan, das seinerseits nichts anders macht, als fortlaufend den AHT10 mittels **readSensor**() abzufragen. Das Attribut **buf** ist in der Klasse AHT10 global und daher von überall innerhalb AHT10 auf jeder Ebene erreichbar. **waitRDY**() wird verlassen, wenn das BUSSY-Flag 0 ist. Dann sind die Messwerte aber auch bereits in den Elementen 1..5 des bytearrays **buf** enthalten. Es wäre aber mystisch, und schwer nachvollziehbar, wenn allein nach einem **waitRDY**() die Nutzdaten bereits aus den Rohdaten berechnet würden. Diese drei Zeilen gönnen wir uns also der Klarheit wegen.

```
    def getTemp(self):
         self.getValues()
         raw = ((self.buf[3] & 0xF) << 16) | \
               (self.buf[4] << 8) | self.buf[5]
         return raw * 200 / NENNER - 50
```

Die Zellen 3,4 und 5 von **buf** enthalten den Temperaturrohwert. Genau genommen sind nur die unteren 4 Bits von **buf**[3], das Low-Nibble, der Temperatur zuzurechnen. Die bilden aber die vier höchstwertigen Bits des Zwischenwerts **raw**. Weil 16 Bitpositionen darunterliegen, isoliere ich das Low-Nibble von **buf**[3] durch Undieren mit 0x0F und schiebe die Bits 16 Stellen nach links. Die nächsten 8 Bits liefert **buf**[4], ich schiebe sie um 8 Positionen nach links und oderiere das mit dem bisherigen Wert. Die untersten 8 Bits können dann mit **buf**[5] durch Oderieren aufgefüllt werden.

Folgende Darstellung kann den Vorgang vielleicht besser vermitteln.
Nehmen wir an, buf[3:6] hat die Form (0b????xxxx,0byyyyyyyy, 0bzzzzzzzz), dann passiert folgendes, wobei ?,x,y und z Bitpositionen darstellen.

```
     0b????xxxx
&    0b00001111
=    0b0000xxxx
                    0b0000xxxx << 16
     0bxxxx 00000000 00000000
                    0byyyyyyyy << 8
          0byyyyyyyy 00000000

     0bxxxx 00000000 00000000
|         0byyyyyyyy 00000000
|                    0bzzzzzzzz
=    0bxxxx yyyyyyyy zzzzzzzz
```

Diese 20-Stellige Binärzahl ist nun laut Datenblatt mit 200 zu multiplizieren und durch 2 hoch 20 zu dividieren. Wenn dann noch 50 abgezogen wird, erhält man die Temperatur in Grad Celsius. Genau das passiert in **getTemp**().

```
    def getHum(self):
        self.getValues()
        raw=(self.buf[1] << 12) | (self.buf[2] << 4) | \
             (self.buf[3] >> 4)
        return raw * 100 / NENNER
```

Ähnlich arbeitet **getHum**(), nur sind hier die oberen vier Bits von **buf**[3] das niederwertigste Nibble des Rohwerts. Ein Schieben um 4 Positionen nach rechts verfrachtet die Bits dorthin. **buf**[1] und **buf**[2] gesellen sich durch Linksschieben um 12 beziehungsweise um 4 Positionen dazu. Wieder entsteht eine 20-stellige Binärzahl, die aber jetzt nur mit 100 zu multiplizieren und durch die Konstante Nenner = 2^{20} zu dividieren ist, um den Wert der relativen Luftfeuchte zu erhalten.

```
    def getDewPoint(self):
        temp=self.getTemp()
        hum=self.getHum()
        dp= AHT10.K3 * ((AHT10.K2 * temp)/ \
                         (AHT10.K3 + temp) + log (hum/100))/\
                        ((AHT10.K2 * AHT10.K3) /  \
                         (AHT10.K3 + temp) - log (hum/100))
        return dp
```

Der Taupunkt ist die Temperatur, bei der der unsichtbare Wasserdampf in der Luft beginnt, zu kondensieren. Es bildet sich dann Nebel, Scheiben beschlagen. Die Formel (15) auf der angegebenen Wikipedia-Seite habe ich verwendet, um den Taupunkt zu berechnen, nachdem ich die Temperatur- und Feuchte-Werte eingelesen habe.

$$\tau(\varphi,\ \vartheta) = K_3 \cdot \frac{\frac{K_2 \cdot \vartheta}{K_3 + \vartheta} + \ln \varphi}{\frac{K_2 \cdot K_3}{K_3 + \vartheta} - \ln \varphi}$$

Abbildung 8: Taupunktberechnung nach Magnus

Denken Sie bitte daran, vor Aufruf von **getTemp**(), **getHum**() und **getDewPoint**() eine Messung zu triggern, **triggerDevice**(). Temperatur, Feuchte und Taupunkt werden dann aus demselben Datensatz berechnet. Das Demoprogramm **temphum**.py zeigt die Anwendung der Klasse. Die Diskussion folgt nach dem Listing.

```
# temphum.py

from aht10 import AHT10
from machine import Pin,SoftI2C
import sys
from time import sleep#, ticks_ms
from oled import OLED

if sys.platform == "esp8266":             # (1)
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4))
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21))
else:
    raise RuntimeError("Unknown Port")

d=OLED(i2c,heightw=32)                    # (2)
d.writeAt("HYGRO-THERM",2,0)
sleep(3)

aht=AHT10(i2c)                            # (3)

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

while 1:                                  # (4)
    aht.triggerDevice()
    d.clearAll(False)
    temp=aht.getTemp()
    hum=aht.getHum()
    dp=aht.getDewPoint()
                                          # (5)
    print("TEMP: {:0.2f}; HUM: {:0.2f}; DP: {:0.2f}".\
          format(temp,hum,dp))
                                        # (6)
    d.writeAt("TEMP: {:>6.2f} *C   ".format(temp),0,0,False)
    d.writeAt("HUM:  {:>6.2f} %    ".format(hum),0,1,False)
                                          # (7)
```

```
    d.writeAt("DP:    {:>6.2f} *C    ".format(dp),0,2)
    sleep (1)

    if taste.value() == 0:                # (8)
        d.clearAll()
        d.writeAt("PROGRAM CANCELED",0,0)
        sys.exit()
```

Bevor wir vom Modul **aht10** die Klasse **AHT10** importieren können, muss die Datei **aht10**.py in den Flash des ESP32 hochgeladen werden. Rechtsklick auf die Datei und **Upload to /**.

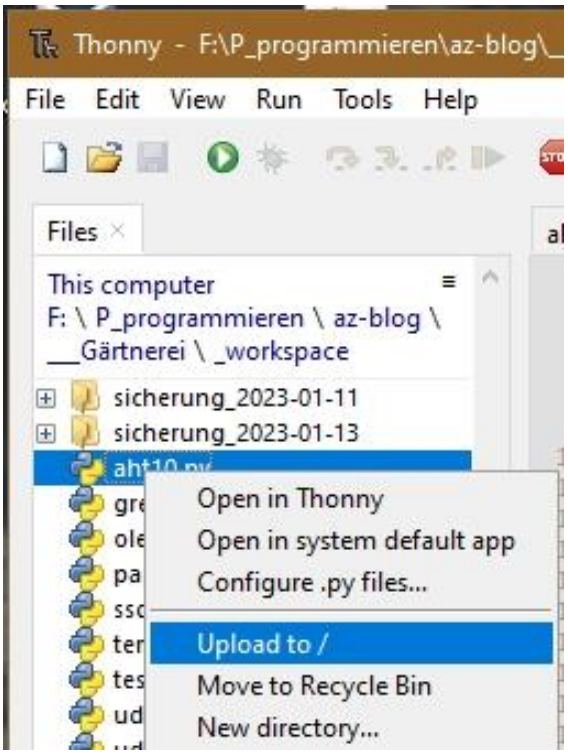

Abbildung 9: Modul hochladen

Von **machine** brauchen wir **Pin** und **SoftI2C**. Pin zum Deklarieren der GPIOs für **SCL** und **SDA** sowie **taste**. **SoftI2C** verwende ich gerne, weil ich die Anschlüsse für den Bus dann frei wählen kann.

sys liefert uns die Textkonstante **platform** und die Funktion **exit**(). Damit haben wir Zugriff auf den Controllertyp – "esp32" oder "esp8266" – und die Möglichkeit zum geordneten Verlassen des Programms.

Mit **sleep**() aus dem Modul **time** schicken wir den ESP32 für die übergebene Sekundenanzahl zum Schnarchen. Das Argument darf auch vom Typ **float** und auch kleiner als 1 sein.

Die Klasse **OLED** bietet uns eine komfortable API für die Steuerung des OLED-Displays. Mehr dazu später.

@ (1)
Die if-Konstruktion erlaubt mir, ohne Änderungen am Programm, plattformübergreifend zu arbeiten. Jeder Controllertyp sucht sich selbst die richtigen Pins für den I2C-Bus aus. Hier könnten auch weitere portabhängige Deklarationen eingebaut werden, zum Beispiel für ADC- oder PWM-Objekte.

@ (2)
Der Konstruktor der Klasse OLED braucht für den Positionsparameter **i2c** die eben deklarierte I2C-Instanz und die Displayhöhe **heightw** =32. Der optionale Parameter het den Defaultwert 64, die Defaultbreite ist widthw = 128. Sie muss nicht angegeben werden, weil der Wert unserem Display entspricht. Wir lassen den Anwendungstitel für drei Sekunden anzeigen.

@ (3)
Wir instanziieren das **AHT10**-Objekt **aht** und das GPIO-Objekt **taste**. Als Hardware nutzen wir dafür die Flash-Taste des ESP32 an GPIO0. Der Anschluss besitzt bereits einen externen Pullup. Dennoch schalte ich auch den internen ein, denn sollte jemals ein anderer GPIO-Pin gewählt werden, dann brauche ich an der Deklaration selbst keine Änderung mehr vornehmen. Ich ändere nur die Zuweisung an **tPin**. Hiermit und mit (1) verhält es sich so, wie mit den Textbausteinen eines Textverarbeitungsprogramms – einmal festlegen, wiederholt gebrauchen.

@ (4)
Die while-Schleife ist die Main-Loop und läuft endlos, zumindest solange der ESP32 Saft hat und die Flash-Taste nicht gedrückt wird.

Wir triggern eine Messung. Während der AHT10 arbeitet, löschen wir die Anzeige. Wegen des Arguments **False** passiert das im Hintergrund. Das heißt es wird nur der Pufferspeicher im RAM des ESP32 mit Nullen beschrieben. Der Inhalt wird aber noch nicht an das Display-Modul geschickt, das zu diesem Zeitpunkt immer noch den Titel anzeigt.

Dann holen wir die Werte ab. Falls der AHT10 mit der Messung noch nicht fertig ist, erledigt das Warten darauf im Hintergrund und für uns transparent die Methode **AHT10.getValues**(), die von **getTemp**(), **getHum**(), und **detDP**() aufgerufen wird.

@ (5)
Der print-Befehl nutzt Formatierungsanweisungen, um Text und Zahlenwerte, hier vom Typ float, gezielt zu vermischen. Das Zahlenformat wird in geschweiften Klammern angegeben. Nach dem Doppelpunkt steht der Wert für die minimale Breite der auszugebenden Zahl. Die Null ist der Defaultwert und könnte auch ohne Änderung der Ausgabe weggelassen werden. Die Angabe wird sowieso ignoriert, wenn die Zahl, inclusive Trennzeichen, mehr Stellen aufweist als die minimale Breite angibt. Mit dem Punkt und der folgenden Zahl wird die Anzahl von Nachkommastellen bei Fließkommazahlen eingestellt, hier 2. Das f gibt den Typ der Zahl an.

@ (6)
Die Werte im Display sollen Dezimalpunkt unter Dezimalpunkt stehen. Weil kein Wert mit Vorzeichen breiter als 6 Stellen sein kann, gebe ich als minimale Breite 6 an und

sorge mit dem ">" dafür, dass die Formatierung rechtsbündig erfolgt. Dieses Vorgehen ersetzt quasi einen rechten oder dezimalen Tabulator.

@ (7)
Erst dieser Schreibbefehl schickt den Inhalt des Zeichen-Puffers zum OLED-Display. Dieses Vorgehen verhindert das Flackern der Anzeige. Probieren Sie ruhig aus, was passiert, wenn **False** bei den Schreibanweisungen in der Schleife weggelassen oder durch das optionale **True** ersetzt werden.

@ (8)
Ist die Flash-Taste gedrückt, wenn das Programm an der if-Konstruktion ankommt, dann wird die Anzeige gelöscht und der Text "PROGRAM CANCELED" ausgegeben. Danach beendet **exit**() die Programmausführung.

Damit sind wir am Ende des Klimateils angekommen. Der AHT10 kann dem ESP32 jetzt die Information liefern, ob das Kleinklima in der Saatbox passt, ob geheizt oder gekühlt werden muss und ob die Luftfeuchtigkeit für das Wachstum passt.

In der nächsten Folge werden wir Licht in die Sache bringen, es wird um die Beleuchtung der Plantage gehen. Sie können schon einmal auf die Suche nach einem kräftigen 5V-Netzteil gehen. Zusammen mit der Heizung und der Pumpe kommen wir im Endausbau auf 3,5 bis 4 Ampere, wenn alle drei Hauptverbraucher gleichzeitig aktiviert sind.