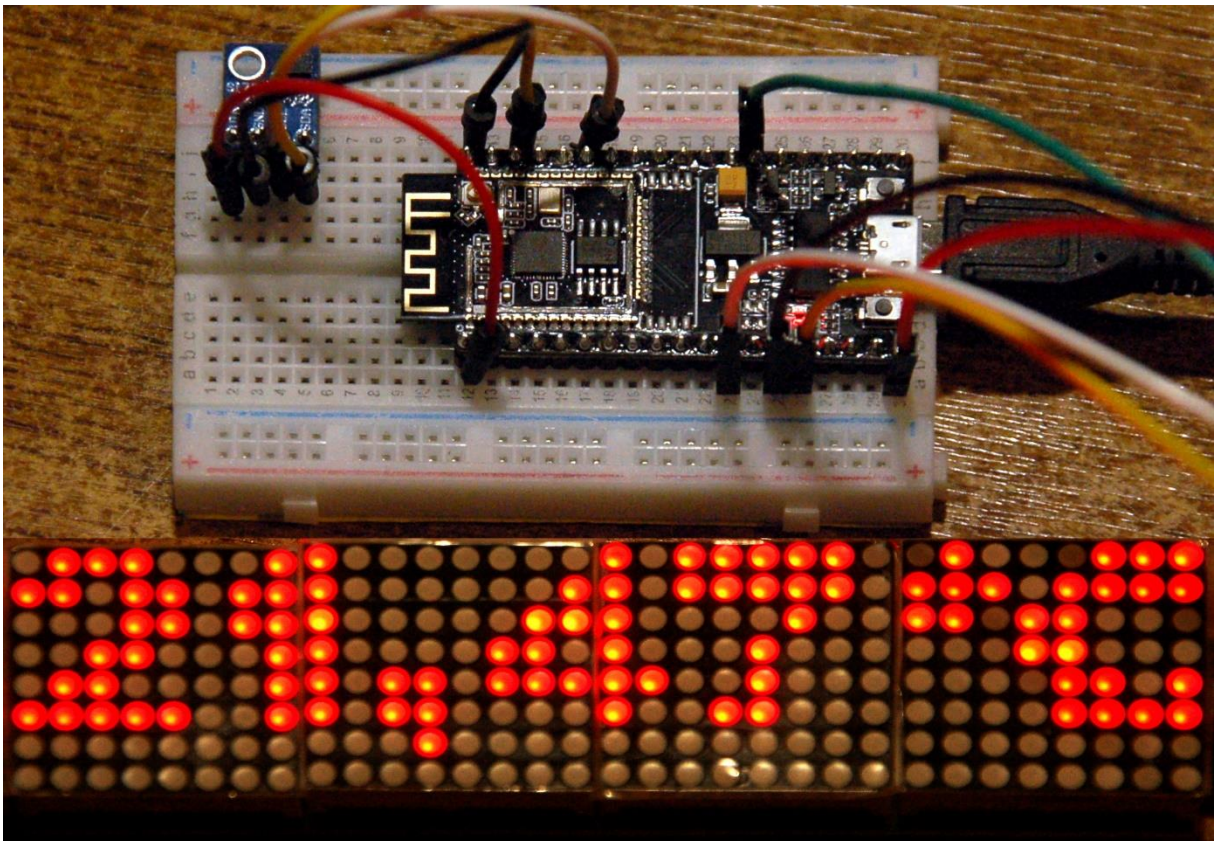

Thermo- Hygrometer

Dieser Beitrag ist auch als PDF-Dokument verfügbar.

Eine schöne große Anzeige allein kann noch nicht alles sein. Man kauft sich ja schließlich auch kein Auto, um es nur in die Garage zu stellen, damit die sich nicht so alleine fühlt. Also werde ich Ihnen in dieser Folge und in den nächsten ein paar nette Anwendungsbeispiele des Mammut-Matrix-Displays vorstellen. Aufgrund der wenigen Teile ist dieses Projekt, ebenso wie das Matrixdisplay selbst, gut für Anfänger geeignet. Ich beginne heute mit einer kleinen Wetterstation aus nur zwei Teilen, die Anzeige nicht mitgerechnet, denn die wurde ja, zusammen mit dem MicroPython-Modul matrix8x8.py, bereits in der vorangegangenen Episode detailliert besprochen. In dieser Folge erfahren Sie auch, wie eine CRC-Prüfsumme berechnet wird und wie ein CRC-Test abläuft. Aber, bevor wir loslegen, erst einmal willkommen bei

MicroPython auf dem ESP32 und ESP8266

heute

Das Mammut-Matrix-Display und der HTU21

Der HTU21 ist ein Zwerg von 3mm x 3mm, der sich auf einem Platinchen mit 10mm x 13mm befindet, welches den Namen GY-21 trägt. Auf der Unterseite der Platine entdecke ich einen Spannungsregler und diverses "Hühnerfutter". Der Baustein kann also mit Spannungen von 5V versorgt werden, arbeitet aber mit 3,3V. Das ist sehr

gut, denn dadurch liegt der Pegel auf den I2C-Bus-Leitungen, mit denen das Teil angesteuert wird, im sicheren Bereich für den ESP32. Der hat ja bereits die Aufgabe, das Display anzusteuern und jetzt den Zusatzjob, mit dem HTU21 alias SHT21 zu plaudern.

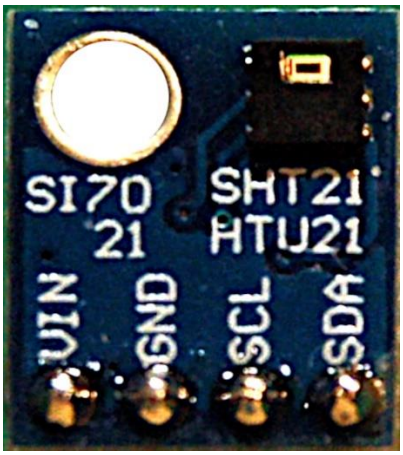

Abbildung 1: : HTU21 alias GY-21

Die Themen dieser Unterhaltungen werden die Temperatur und die relative Luftfeuchtigkeit sein. Das Ergebnis der Unterhaltungen soll dann auf dem Matrixdisplay landen. Letzteres sollte für diesen Zweck schon aus wenigstens 8 Elementen bestehen.

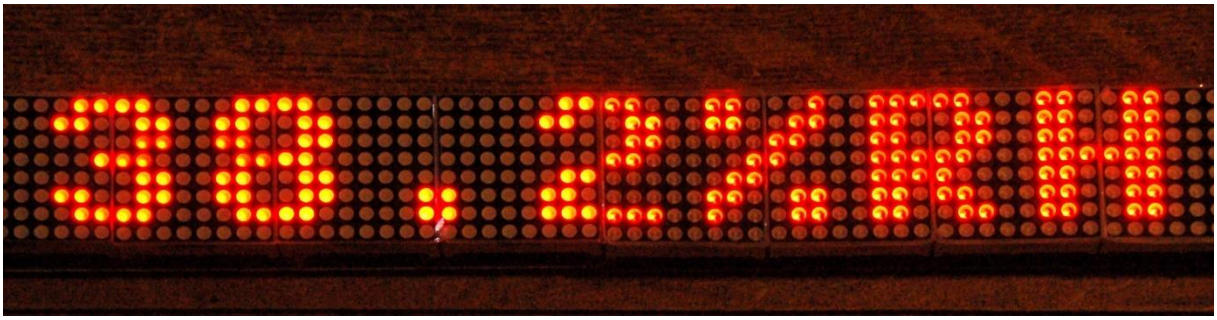
Abbildung 2: Feuchteanzeige

Das Datenblatt des HTU21 ist zum Beispiel hier oder hier zu finden. Es stellt sich nämlich heraus, dass sich HUT21 und SHT21, was das I2C-Bus-Verhalten und die Register angeht, identisch sind.

Bezüglich der Controller, ESP32 und ESP8266, hat mich interessiert, inwieweit beide Familien für die genannten Aufgaben taugen. Mit einer Ausnahme verfügen beide sowohl über einen SPI-Bus als auch über einen I2C-Bus und haben sich daher als gut brauchbar erwiesen. Die Ausnahme ist der keine ESP8266-01. Bei ihm sind nur GPIO0 und GPIO2 herausgeführt. Damit könnte man einen HTU21 anflanschen, was wir auch in der nächsten Folge tun werden.

Hardware

1	D1 Mini NodeMcu mit ESP8266-12F WLAN Modul oder NodeMCU Lua Amica Modul V2 ESP8266 ESP-12F oder ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
nach Bedarf	MAX7219 8x8 1 Dot Matrix MCU LED Anzeigemodul oder MAX7219 8x32 4 in 1 Dot Matrix LED Anzeigemodul
diverse	Jumperkabel
1	Minibreadboard oder Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
1	Sperrholzstreifen 5x 25cm …

Beim Aneinanderreihen von Matrixeinheiten, einzeln oder in Gruppen, hat sich bei mir ein Sperrholzstreifen von 5cm Breite und der Länge des gesamten Displays bewährt. Da drauf habe ich die Platinen mit Eternit Dichtungsmasse fixiert. Die Knete hält wo sie soll, ist aber auch wieder restlos ablösbar.

Die Schaltskizzen für den ESP32 und den ESP8266 sind sehr überschaubar.

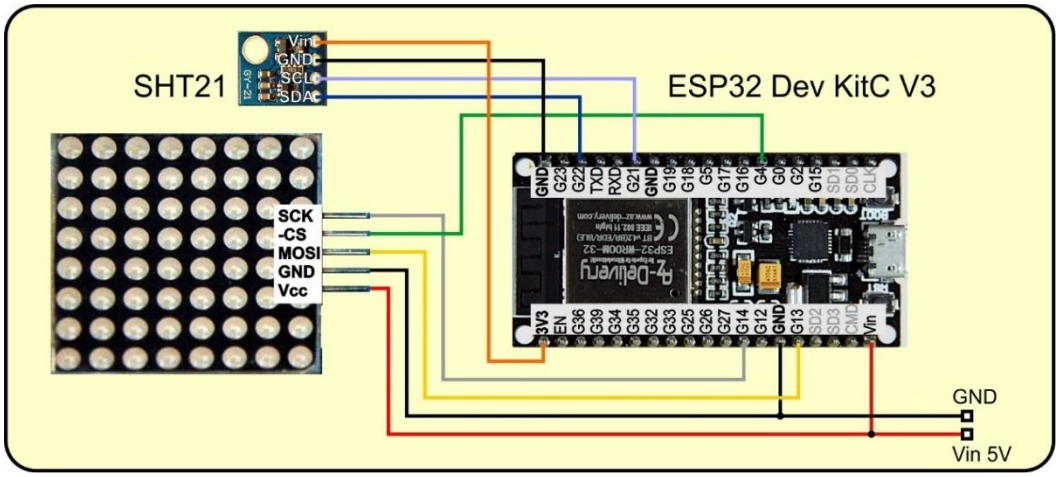

Abbildung 3: SHT21-Thermo+Hyrdometer mit ESP32

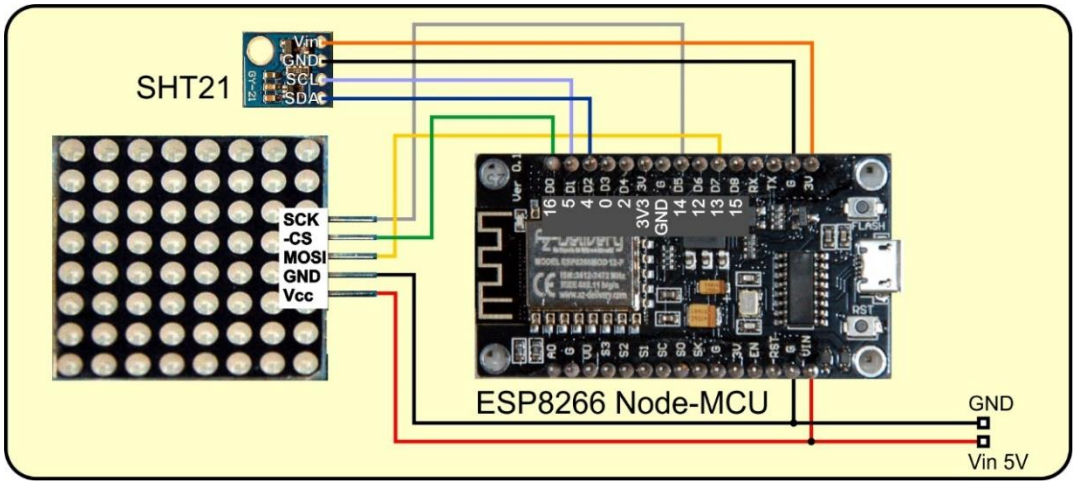

Abbildung 4: SHT21-Thermo+Hyrdometer mit ESP8266

Ein Amica Node-MCU passt auf ein Mini-Breadboard. Gleiches gilt für einen ESP32S, der einen schmaleren Fußabdruck hat wie beispielsweise ein ESP32 Dev Kit C V3

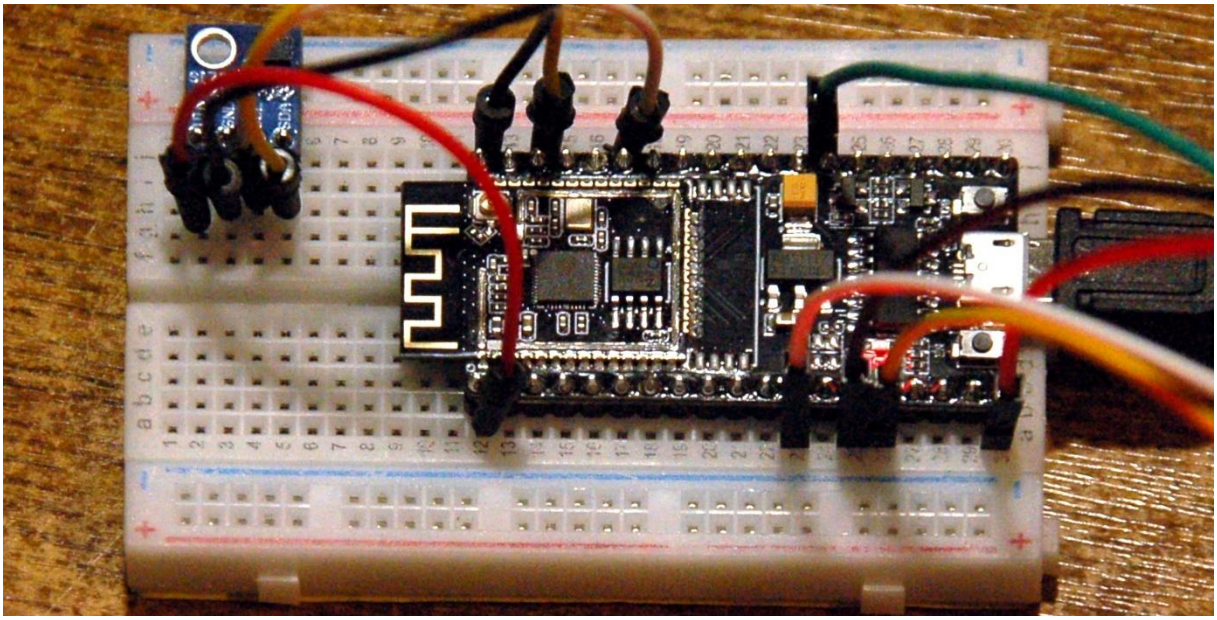

Abbildung 5: ESP32S mit GY-21

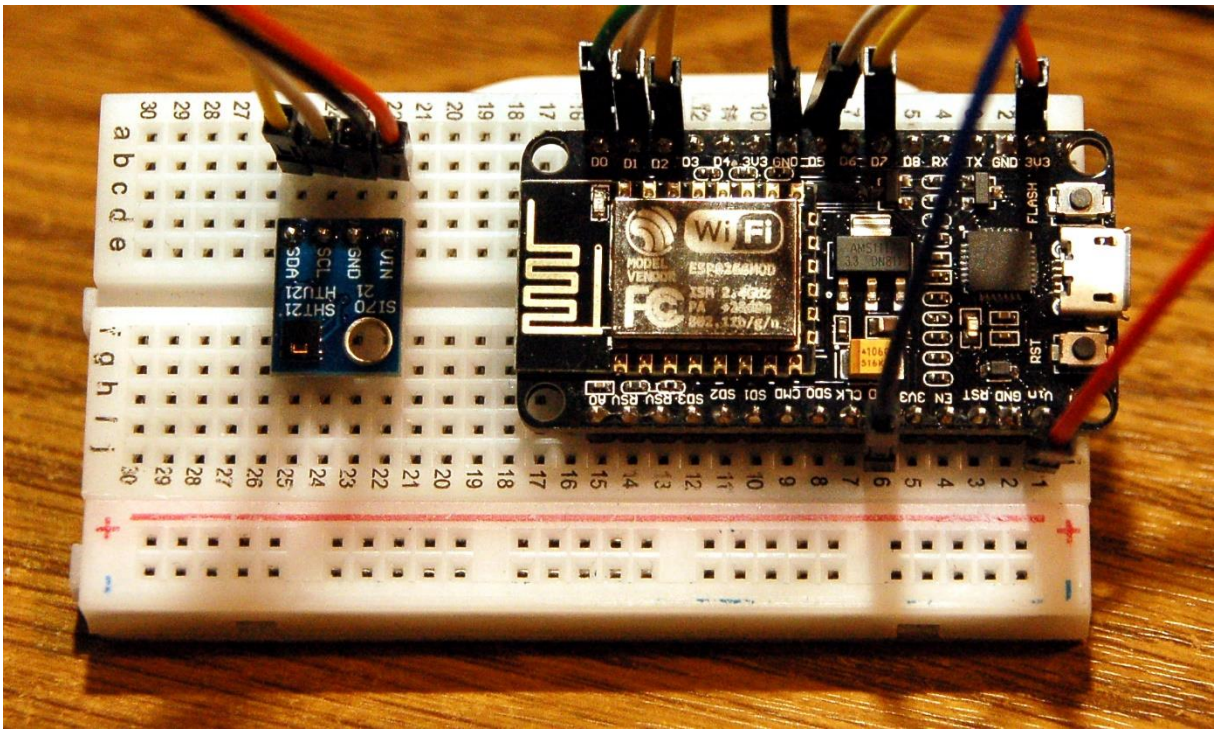

Abbildung 6: Amica Node-MCU aus der ESP8266-Familie

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP8266/ESP32:

MicropythonFirmware
Bitte eine Stable-Version aussuchen
ESP8266 mit 1MB Version 1.18 Stand: 25.03.2022 oder
ESP32 mit 4MB Version 1.18 Stand 25.03.2022

Die MicroPython-Programme zum Projekt:

matrix8x8.py Treibermodul für den MAX7219
matrixtest.py Demoprogramm für ein x mal 8 -Matrixdisplay
sht21.py Treibermodul für das GY-21-Modul
sht21_32_test.py Testprogramm für den ESP32
sht21_8266_test.py Testprogramm für den ESP8266
shtdisplay.py Thermo- Hygrometer Software
i2cbus.py Routinen für den I2C-Bus

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 05.02.2022) 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.

Signale auf dem I2C-Bus

In den zurückliegenden Episoden zum Thema I2C-Bus oder SPI-Bus habe ich mitunter die Signale auf den Busleitungen mit dem DSO (aka digitales Speicher-Oszilloskop) erfasst und abgebildet. Erst kürzlich bekam ich Anfragen von Lesern, die Probleme mit dem I2C-Bus hatten. Der Anlass war in einem Fall ein defektes Kabel, im anderen Fall waren es schlechte Kontakte. Die Fehler waren durch ein paar MicroPython-Befehle rasch eingegrenzt. Andere Fehler halten sich hartnäckiger.

Bei den Vorbereitungen zu diesem Beitrag hatte ich einen HTU21 im Einsatz, der sich ohne Probleme auf dem I2C-Bus lokalisieren ließ. Aber er weigerte sich strickt, seine Registerinhalte auslesen zu lassen. Änderungen meiner Treibersoftware blieben erfolglos. In solchen Fällen setze ich dann das DSO ein, oder ein weitaus 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.

Nun, dieses Gerät habe ich nach einigen Stunden Datenblattstudium und vergeblicher Fehlersuche am Programm dann auch eingesetzt. Hätte ich das nur viel früher getan… Kurzum, die Situation war eindeutig. So sehen die Kurvenformen beim defekten HTU21 aus:

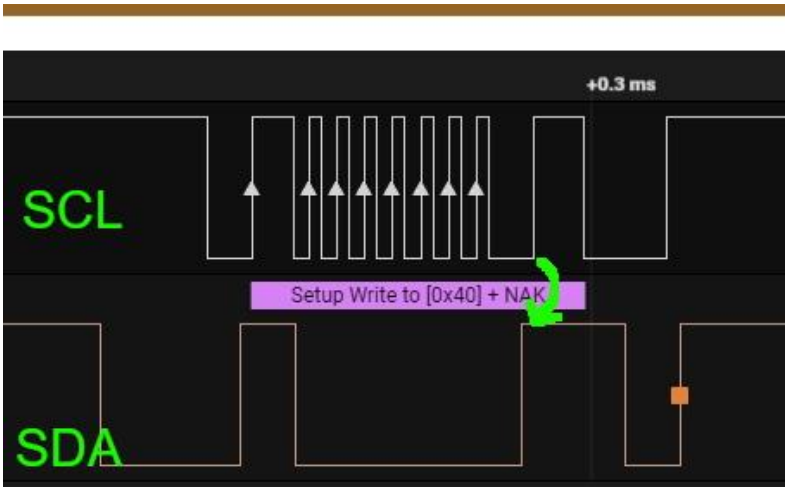

Abbildung 7: Adressierung des HTU21 mit NACK-Antwort

Und so hätte das aussehen sollen. Sehen Sie den Unterschied beim ersten Frame? Bereits die Hardwareadresse wird mit einem NACK- statt einem ACK-Bit quittiert

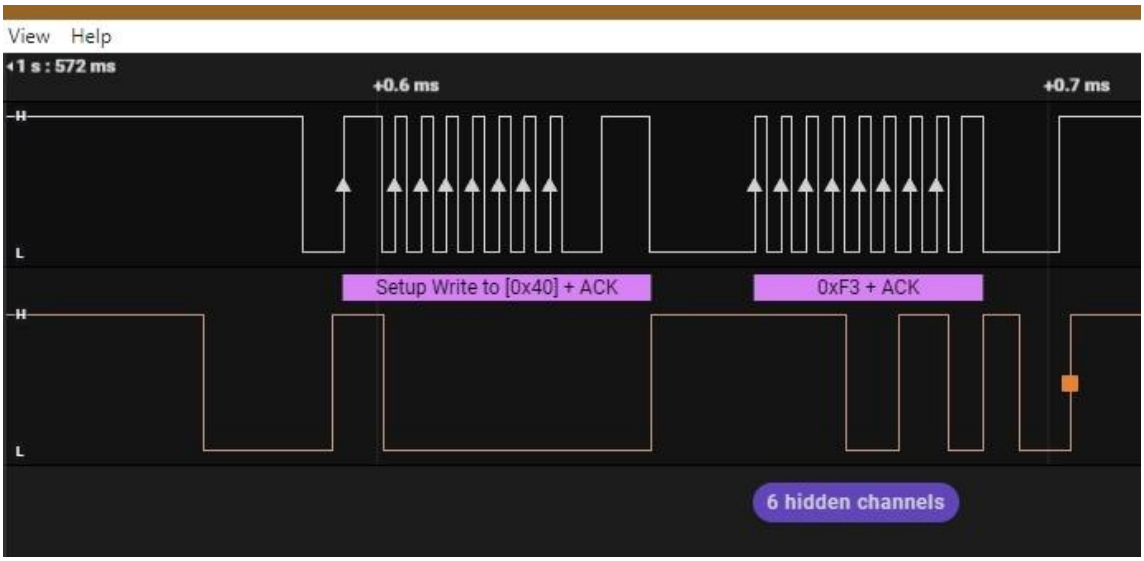

Abbildung 8: Auftrag für einen Temperatur-Scan

Und so hätte der Lesebefehl aussehen sollen, zu dem es aufgrund des Fehlers im Chip nicht mehr gekommen ist.

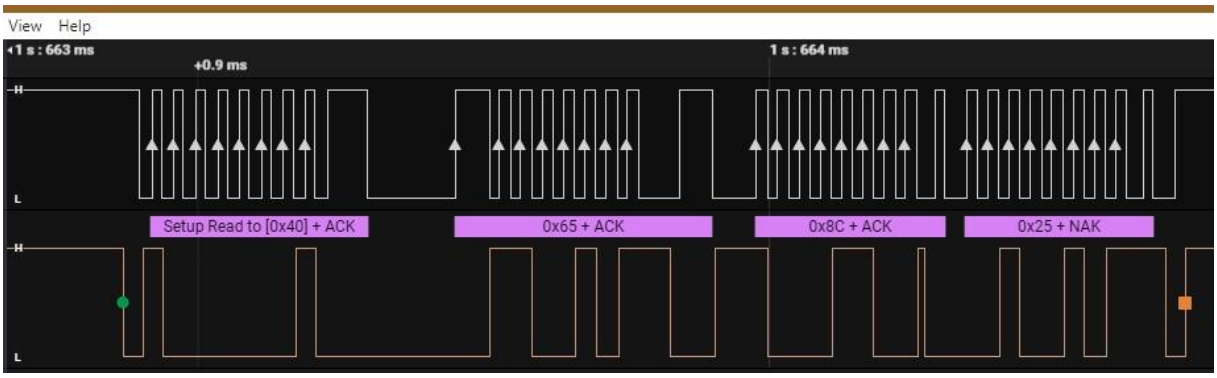

Abbildung 9: Temperaturrohwert 91ms später abholen

Stattdessen beim fehlerhaften Modul wieder ein NACK (Not Acknowledge), das dadurch hervorgerufen wird, dass der Slave, hier der HTU21, die SDA-Leitung während des neunten Taktimpulses vom Controller nicht auf GND zieht. Fatalerweise kam in lockerer Folge einmal ein ACK an der erwarteten Position aber meistens eben nicht. Da sucht man sich einen Wolf. Mit dem Austausch des Bausteins war das Problem behoben.

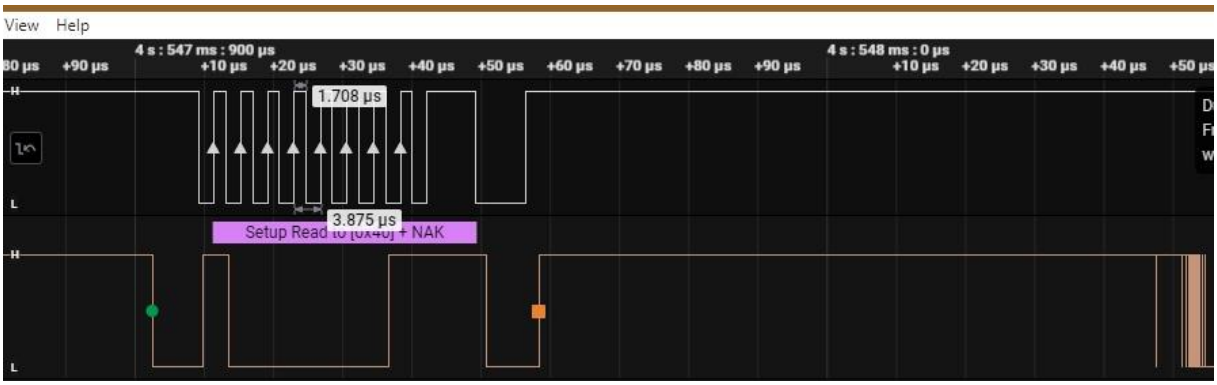

Abbildung 10: Lesebefehl an den HTU21 mit NACK

Die Klasse SHT21

Nach diesem kurzen Ausflug zur Fehlersuche, bauen wir jetzt die Klasse SHT21. Im Vergleich zur Klasse MATRIX fällt sie um einiges schmäler aus. Auch hier ist das Datenblatt des SHT21 die Quelle aller Weisheit.

Ich importiere die Klasse I2CBus und die Funktion **sleep_ms**(). In **I2CBus** habe ich eine Reihe von Methoden deklariert, welche die Parameterübergabe an die Methoden der Klasse **SoftI2C** vereinheitlichen. Die Umwandlung von den Zahlenformaten Byte und Integer zum Typ bytearray wird dort automatisch vorgenommen.

```
from i2cbus import I2CBus
from time import sleep_ms

class SHT21(I2CBus):
```

Meine Klasse SHT21 erbt von der Klasse I2CBus. Damit stehen alle Methoden von I2CBus im Namensraum von SHT21 ohne Prefix zur Verfügung. Dass dabei ein Bezeichner in SHT21 einen Namensvetter in I2CBus überschreiben könnte, ist nicht zu erwarten.

Im Abschnitt **5.3 Sending a Command** auf Seite 7 des Datenblatts finde ich die Hardwareadresse des SHT21.

*After sending the Start condition, the subsequent I2C header consists of the **7-bit I2C device address '1000'000'** and an SDA direction bit (Read R: '1', Write W: '0').*

Wie die meisten I2C-Devices ist die Hardwareadresse (device address) auch hier ein 7-Bit-Wert. 0b1000000=0x40=64. Durch Hinzufügen des Richtungsbits R/-W an der LSB-Position wird daraus die 8-Bit-Adresse, die wir auf den Plots in den Abbildungen 7 bis 10 sehen. Der SHT21 tastet die SDA-Leitung bei steigender Flanke an SCL ab. Acht steigende Flanken für 8 Bits, die neunte steigende Flanke liest das ACK (0) oder NACK-Bit (1) auf der SDA-Leitung.

Auf der Seite 8 im Datenblatt, oben links, folgt die Tabelle der Register des SHT21. Ich übertrage sie sinngemäß ins Programmlisting.

Command	Comment	Code
Trigger T measurement	hold master	1110'0011
Trigger RH measurement	hold master	1110'0101
Trigger T measurement	no hold master	1111'0011
Trigger RH measurement	no hold master	1111'0101
Write user register		1110'0110
Read user register		1110'0111
Soft reset		1111'1110

Table 6 Basic command set, RH stands for relative humidity, and T stands for temperature

Abbildung 11: Register oder Kommando-Tabelle

```
    HWADR=const(0x40)
    # Kommandos
    TriggerT = const(0xE3)
    TriggerRH= const(0xE5)
    TriggerTnoHold=const(0xF3)
    TriggerRHnoHold=const(0xF5)
    WriteUser=const(0xE6)
    ReadUser=const(0xE7)
    SoftReset=const(0xFE)
```

Der Konstruktor braucht das im Hauptprogramm erzeugte I2C-Objekt und nimmt optional eine Hardwareadresse. Ihr Wert wird durch die Konstante HWADR per Default vorbelegt und auch gleich in eine Instanzvariable **self.hwadr** übertragen, damit sie allen Methoden der Klasse zur Verfügung steht.

Der nächste Befehl **super().__init__(i2c,self.hwadr)** instanziiert die Klasse I2CBus. Sie entspricht dem sonst notwendigen Konstruktoraufruf **I2CBus(i2c,0x40)**.

Es folgen die Deklarationen für die Wartezeiten nach dem Absetzen eines Wandlerbefehls für Temperatur und relative Feuchte bis zum Einlesen der Werte. Die entnehme ich den Spalten **RH max** und **T max** aus der Tabelle 7 auf Seite 9 im Datenblatt.

Resolution	RH typ	RH max	T typ	T max	Units
14 bit			66	85	ms
13 bit			33	43	ms
12 Bit	22	29	17	22	ms
11 bit	12	15	9	11	ms
10 bit	7	9			ms
8 bit	3	4			ms

Abbildung 12: Wandler-Wartezeiten

```
        self.Twait=(86,22,43,11) # ms
        self.Hwait=(30,4,9,15) # ms
        self.resFlags=(0x00,0x01,0x80,0x81)
        self.resText=("12/14", "8/12", "10/13", "11/11")
        self.resolution=0 # RH/T=12/14
```

Es gibt vier Kombinationen für die Auflösung der Temperatur- und Feuchtemessung. Bit 7 und Bit 0 im User-Register sind dafür zuständig. Diese Kombis stehen im Tuple **resFlags**. **resText** gibt die Kombis im Klartext an und **resolution** hält die aktuelle Einstellung der Auflösung. Die Info dazu steht in Tabelle 8 im Datenblatt, Seite 9.

Bit	# Bits	Description / Coding	Default
7, 0	2	Measurement resolution '00': RH 12 bit, T 14 bit '01': RH 8 bit, T 12 bit '10': RH 10 bit, T 13 bit '11': RH 11 bit, T 11 bit	'00'
6	1	Status: End of battery[15] '0': VDD > 2.25V '1': VDD < 2.25V	'0'
3, 4, 5	3	Reserved	
2	1	Enable on-chip heater	'0'
1	1	Disable OTP Reload	'1'

Table 8 User Register

Abbildung 13: User Register und Wandler-Auflösung

```
        self.sht21Reset()
        self.TRaw=0
        self.HRaw=0
        self.Temp=0
        self.Hum =0
        print("SHT21 initialized @ {:#X}".format(self.hwadr))
        sleep_ms(15)
```

Dann setze ich den SHt21 mit meiner Routine **sht21Reset**() zurück und erzeuge die globalen Variablen für Temperatur und Feuchte. Es folgt die Ausgabe des Konstruktors. Dann warten wir noch auf den Abschluss des Reset-Kommandos. Darüber informiert der Abschnitt 5.5 auf Seite 9.

…
The soft reset takes less than 15ms.

Die Methode für den Chip-Reset besteht aus einem Schreibbefehl an den SHT21. Wir greifen auf das Register **SoftReset = 0xFE** schreibend zu, wie es das Datenblatt im Abschnitt 5.5 auf Seite 9 vorgibt.

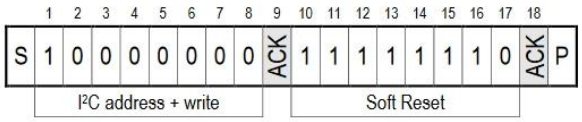

Figure 17 *Soft Reset* – grey blocks are controlled by SHT2x.

Abbildung 14: Reset-Befehl

```
    def sht21Reset(self):
        self.writeToBus(SoftReset,self.hwadr)
```

Die Wandlung eines Temperaturwerts und das Einlesen erledigt die Methode **readTemperatureRaw**().

```
    def readTemperatureRaw(self):
        self.writeToBus(TriggerTnoHold,self.hwadr)
        sleep_ms(self.Twait[self.resolution])
        data=self.readBytesFromBus(3,self.hwadr)
        if self.calcChecksum(data,3) == 0:
            self.TRaw=data[0]<<8 | (data[1] & 0xFC)
        else:
            self.TRaw=0
        return  data
```

Und das steckt dahinter – Datenblatt Seite 9:

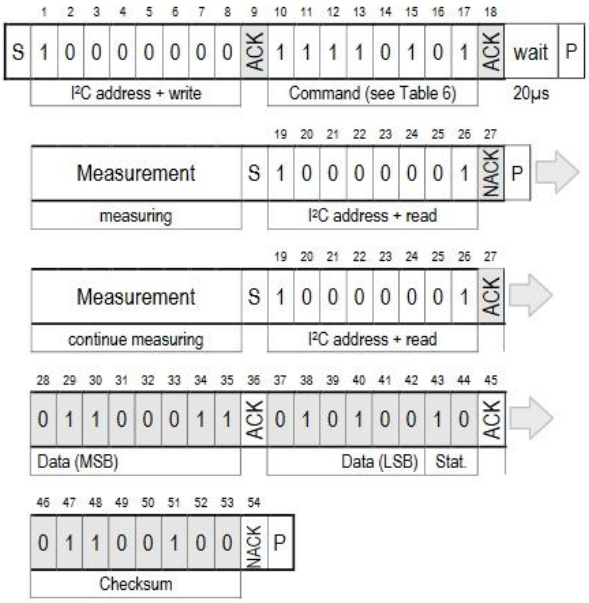

Figure 16 *No Hold master* communication sequence – grey blocks are controlled by SHT2x.

Abbildung 15: Ablauf einer Konversation

Ich sende den Befehl zum Starten einer Temperaturmessung, **TriggerTnoHold** = 0xF3. **noHold** bezieht sich auf das Verhalten der SCL-Leitung während der Messung und besagt, dass SCL vom SHT21 nicht auf GND-Pegel gehalten wird. So bleibt der Bus während der Messdauer frei.
Entweder ich frage immer wieder nach, ob es schon so weit ist, oder ich warte einfach die von Sensirion vorgegebene Zeit. Ich habe mich für die zweite Lösung entschieden und schicke den ESP für ein paar Millisekunden in die Heia. Wie lang das ist, hängt von der aktuellen Einstellung der Auflösung in **resolution** ab. Danach werden drei Bytes vom Bus abgeholt, die der SHT senden muss. Wenn die Prüfsumme, das dritte Byte, 0 ist, setze ich den Rohwert der Temperatur aus den ersten beiden Bytes zusammen. **data[0]** enthält das MSB. Ich schiebe die Bites um 8 Stellen nach links, das entspricht einer Multiplikation mit 256 und oderiere mit dem LSB in **data[1]**, von dem ich die beiden niederwertigsten Bits lösche. Das steht so auf Seite 8 im vorletzten Absatz:

For the calculation of physical values Status Bits must be set to '0' – see Chapter 6.

Falls ein Übertragungsfehler vorliegt, setze ich **TRaw** auf 0

Es folgt die Berechnung der wahren Temperatur, wie es der Abschnitt 6.2 auf Seite 10 des Datenblatts vorgibt. Ferner lasse ich das Ergebnis auf zwei Stellen nach dem Komma runden. Dazu multipliziere ich den berechneten Wert mit 100, addiere 0,5, damit ab ,5 korrekt aufgerundet wird und mache aus der Summe eine Ganzzahl, die ich abschließend wieder durch 100 teile.

```
    def calcTemperature(self):
        self.Temp=int((-46.85+175.72*\
                       self.TRaw/pow(2,16))*100+0.5)/100
        return self.Temp
```

Für die relative Luftfeuchte sind die Vorgänge dieselben, nur die Formel für die Berechnung des Endwerts ist eine andere.

Das Vorgehen für die Berechnung der Prüfsumme (crc = cyclic redundancy check) habe ich in CRC8 Berechnung.pdf anhand eines Beispiels genau beschrieben. Hier wird es für drei Bytes in Folge angewandt. Weil das vom SHT21 übermittelte crc-Byte in die Prüfung integriert ist, muss als Ergebnis 0x00 herauskommen. **data** gibt das Bytearray und **nob** die Anzahl zu prüfender Bytes (number of bytes) an die Methode. Bei dieser Testmethode wird quasi der Rest einer Polynomdivision durch fortgesetzte Subtraktion bestimmt. Jeder vorkommenden Potenz im Prüf-Polynom und im Rest entspricht eine 1.

```
    def calcChecksum(self,data,nob):
        poly=0x131 # Polynom: x^8+x^5+x^4+1 = 100110001
        crc=0
        for i in range (nob):
            crc ^= data[i]
            for pos in range(8,0,-1):
                if crc & 0x80:
                    crc = (crc << 1) ^ poly
                else:
                    crc = (crc << 1)
        return crc
```

Erinnern Sie mich daran, einen Test dieser Methode durchzuführen, wenn alles Restliche in der Kiste ist.

Um die Auflösung verändern zu können, muss ich lesend und schreibend auf das User-Register zugreifen können. Das erledigen die beiden einschlägigen Methoden **readUser**() und **writeUser**(). Vom Bus wird ein Bytes-Objekt gelesen und als Ganzzahl durch **readUser**() zurückgegeben.

```
    def readUser(self):
        self.writeToBus(ReadUser,self.hwadr)
        data=self.readBytesFromBus(1,self.hwadr)# Bytesobjekt
        return data[0] # Inhalt User-Register als Ganzzahl
```

```
    def writeUser(self,data):
        r=self.writeByteToReg(WriteUser,data,self.hwadr)
        return r
```

Die Methode **resolve**() greift auf die beiden Mainzelmännchen zurück. Das Datenblatt klärt auf Seite 9 im Abschnitt 5.6 darüber auf, dass die reservierten Bits 3, 4 und 5 nicht verändert werden dürfen. Deshalb lese ich das User-Register ein, lösche die Bits 7 und 0 und oderiere sie mit dem Muster, das durch den Index **res** aus dem Tuple **resFlags** geholt wird. Wird der Parameter **res** nicht angegeben, dann fragt **resolve**() die aktuelle Auflösung ab und gibt den Index zurück.

```
    def resolve(self, res=None):
        w=self.readUser()
        if res is not None:
            assert 0 <= res <= 3
            w &= 0x7E
            w |= self.resFlags[res]
            self.writeUser(w)
            self.resolution =res
            return res
        else:
            w &= 0x81
            return self.resFlags.index(w)
```

Die Methode **tellResolution**() geht noch einen Schritt weiter. Sie gibt den Index und und die Klartextmeldung aus dem Tuple **resText** zurück.

```
    def tellResolution(self):
        return self.resolution,self.resText[self.resolution]
```

Das User-Register kann auch noch über den Batteriestatus aufklären. Das ist eventuell hilfreich, wenn die Spannungsversorgung nicht aus einem Netzteil kommt. Fällt die Spannung unter 2,25V, dann geht das Bit 6 auf 1. Die Abfrage erledigt die Methode **tellBattStatus**(). Ich hole den Registerinhalt undiere mit **BatteryMask**=0x40 und schiebe das Bit in Position 6 nach Position 0. Dadurch erhalte ich einen Index 0 oder 1, den ich als Zeiger in das Tuple ("OK","BAD") verwenden kann.

```
    def tellBattStatus(self):
        w=(self.readUser() & BatteryMask)>>6
        return ("OK","BAD")[w]
```

Gut, alles in der Box, dann haben Sie jetzt entweder den Programmtext des Moduls sht21.py selbst eingegeben oder sie haben es bereits vorher oder gerade eben heruntergeladen. Wenn nicht, dann sollten Sie das jetzt spätestens tun. Denn, war da nicht noch etwas, an das Sie mich erinnern sollten? Genau, der Test von **calcChecksum**(). Aber so schnell geht das nicht.

Zuerst muss das Modul sht21.py in den Flash des ESP hochgeladen werden. Die Schaltung haben Sie ja sicher schon aufgebaut? Dann brauchen Sie ein Testprogramm, das die umfangreichen Vorarbeiten erledigt, damit Sie nicht die vielen Zeilen jedes Mal von Hand eingeben müssen. Für einen ESP32 heiß das Programm sht21_32_test.py und für einen ESP8266 ist es sht21_8266_test.py. Der Unterschied liegt nur in den Pins für den I2C-Bus.

```
# sht21_32_test.py
from machine import Pin, SoftI2C
from i2cbus import I2CBus
from time import sleep_ms
from sht21 import SHT21
import os,sys
taste=Pin(0,Pin.IN,Pin.PULL_UP)
blinkLed=Pin(2,Pin.OUT)

# Pintranslator fuer ESP8266-Boards
# LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
# ESP8266 Pins 16  5  4  0  2 14 12 13 15
#                 SC SD

SCL=Pin(21)
SDA=Pin(22)
i2c=SoftI2C(SCL,SDA)

sleep_ms(15) # for booting SHT-Device from power up
s=SHT21(i2c)
s.readTemperatureRaw()
print(s.calcTemperature(),"°C")
s.readHumidityRaw()
print(s.calcHumidity(),"%RH")
```

Öffnen Sie das Programm in einem Editorfenster und starten Sie es mit F5. Wenn Sie alles richtig gemacht haben, bekommen Sie die erste Meldung vom HTU21.

>>> %Run -c $EDITOR_CONTENT
I2CBus-Tools OK @ 0X40
SHT21 initialized @ 0X40
20.78 °C
36.82 %RH

Jetzt testen wir die crc-Funktion. Angenommen, der HTU will für die Temperatur die Bytes 37 und 142 senden. Dann muss er eine Prüfsumme berechnen. Das macht er genauso wie die Methode **calcChecksum**(). Wir erzeugen ein Bytearray mit diesen Werten, weil wir auch einen solchen Datentyp vom HTU21 bekommen. Das übergeben wir der Methode **calcChecksum**(). **s** ist das HTU21- oder SHT21-Objekt. 206 ist die Prüfsumme der beiden Bytes.

>>> data=bytearray([37,142])
>>> s.calcChecksum(data,2)
206

Der SHT21 hängt die 206 an die anderen Bytes dran und schickt sie uns. Wir nehmen die Bytefolge und jagen sie durch den Redundancy-Check. Ist das nicht eine großartige Magie? Tatsächlich kommt 0 heraus. also kein Fehler bei der Übertragung.

>>> data=bytearray([37,142,206])
>>> s.calcChecksum(data,3)
0

Das Klimaprogramm mit Großanzeige

Vielleicht fragen Sie sich, warum ich ein Modul mit einer Klasse SHT21 gestrickt habe. Hätte es nicht gereicht, die ganzen Methoden einfach als Funktionen in ein entsprechendes Programm einzubauen? Es gibt mindestens zwei gute Gründe, die für die Klassen-Lösung sprechen.

- Module und Klassen sind universell wiederverwendbar.
- Module und Klassen entschlacken ein Programm und machen es besser lesbar, übersichtlicher und leichter zu pflegen.
- Sowohl MATRIX wie auch SHT21 werden als Klassen in zwei weiteren Episoden dieser Reihe wieder auftauchen. Der Import ist mit einer Zeile erledigt. Andernfalls müsste ich eine Vielzahl von Programmzeilen kopieren.
- Und was ist, wenn ich eine Funktion etwas abändern möchte und habe das Paket in 20 anderen Programmen verwendet? Dann müsste ich die Funktion in 20 Programmen ändern. Als Methode in einer Klasse mache ich das genau einmal und die Änderung wird in 20 Programme transparent übernommen.

Jetzt haben Sie nicht nur zwei, sondern gleich vier gute Gründe. Und es gibt noch mehr!

Gut lassen Sie uns jetzt alles zu einem Programm zusammenfügen, für ESP32 und ESP8266 und das Matrix-Display und den HTU21-Sensor: **shtdisplay**.py

```
# shtdisplay.py
import sys, os
from time import sleep_ms,sleep
from matrix8x8 import MATRIX
from machine import Pin, SoftI2C, SPI
from i2cbus import I2CBus
from sht21 import SHT21

chip=sys.platform
if chip == 'esp8266':
    # Pintranslator fuer ESP8266-Boards
    # LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
    # ESP8266 Pins 16  5  4  0  2 14 12 13 15
    #                 SC SD
    bus = 1
    MISOp = Pin(12) # D6
    MOSIp = Pin(13) # D7
    SCKp  = Pin(14) # D5
    spi=SPI(1,baudrate=4000000)   #ESP8266
    # # alternativ virtuell mit bitbanging
    # spi=SPI(-1,baudrate=4000000,sck=SCK,mosi=MOSI,\
```

```
    #         miso=MISO,polarity=0,phase=0)  #ESP8266
    CSp = Pin(16, mode=Pin.OUT, value=1) # D0
    SCL=Pin(5) # D1
    SDA=Pin(4) # D2
elif chip == 'esp32':
    bus = 1
    MISOp= Pin(15)
    MOSIp= Pin(13)
    SCKp = Pin(14)
    spi=SPI(1,baudrate=10000000,sck=Pin(14),mosi=Pin(13),\
            miso=Pin(15),polarity=0,phase=0)  # ESP32
    CSp = Pin(4, mode=Pin.OUT, value=1)
    SCL=Pin(21)
    SDA=Pin(22)
    taste=Pin(0,Pin.IN,Pin.PULL_UP)
    blinkLed=Pin(2,Pin.OUT)
else:
    # blink(led,800,100,inverted=True,repeat=5)
    raise OSError ("Unbekannter Port")

print("Hardware-Bus {}: Pins fest vorgegeben".format(bus))
print("MISO {}, MOSI {}, SCK {},
CS{}".format(MISOp,MOSIp,SCKp,CSp))
print("SCL {}, SDA {}\n".format(SCL,SDA))

numOfDisplays=16
d=MATRIX(spi,CSp,numOfDisplays)
i2c=SoftI2C(SCL,SDA)
sleep_ms(15) # for booting SHT-Device
s=SHT21(i2c)

delay=3

while 1:
    d.clear()
    s.readTemperatureRaw()
    tempString="{0:3.1f} C"
    humString="{0:3.1f} %"
    s.calcTemperature()
    print(s.Temp,"°C")
    d.text(tempString.format(s.Temp),0,0,1)
    d.show()
    sleep(delay)
    d.clear()
    s.readHumidityRaw()
    s.calcHumidity()
    print(s.Hum,"%RH")
    d.text(humString.format(s.Hum),0,0,1)
    d.show()
    sleep(delay)
```

Das Programm hat 84 Zeilen. Mit allen Importen kommen über 500 Zeilen zusammen. Ein weiterer guter Grund für Module und Klassen, oder?

Die interessanteste Zeile in diesem Listing ist

```
chip=sys.platform
```

Das liefert uns nämlich den Typ des eingesetzten Controllers. Mit dieser Info ist es ein Leichtes, die GPIO-Pins für SPI- und I2C-Bus korrekt vorzubelegen, bevor damit die entsprechenden Objekte instanziiert werden. Wenn das weitere Programm keine typspezifischen Befehle nutzt, kann man es für mehrere Controller verwenden, ohne Anpassungen daran vornehmen zu müssen.

Ein Schönheitsfehler ist der Umfang dieses Abschnitts. Aber vielleicht habe ich Sie ja mit dem Klassenvirus infiziert und Sie basteln bis zum nächsten Mal ein Modul mit einer Klasse, wodurch man diese Geschichte auslagern kann. Viel Vergnügen beim Ausprobieren, Forschen und mit der Hausaufgabe.

Bis dann!