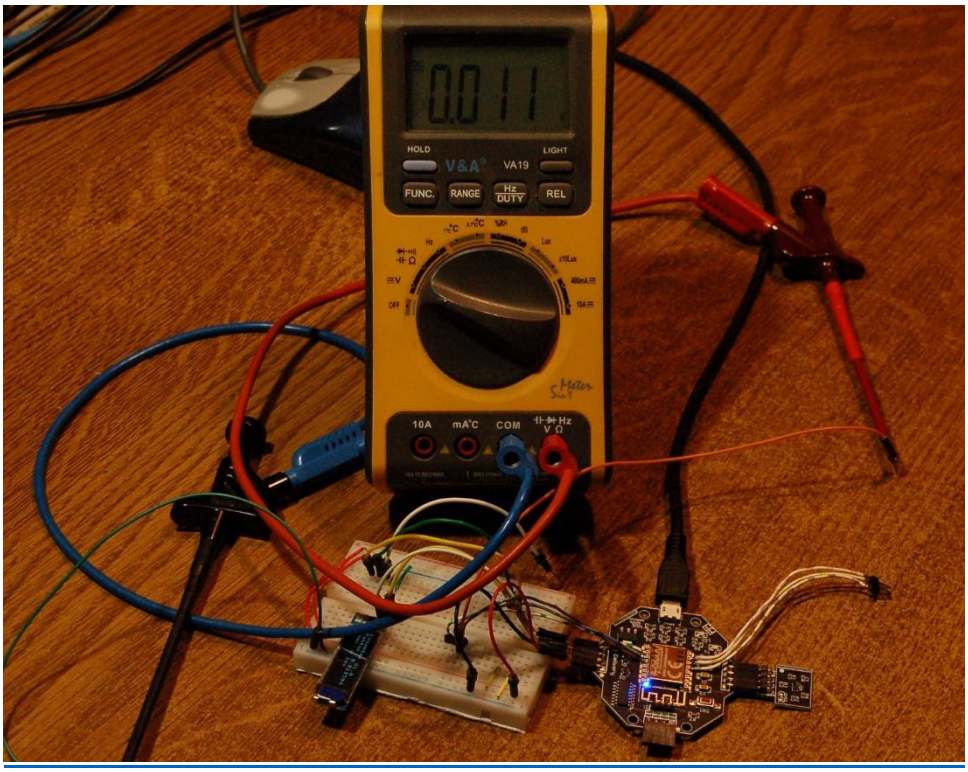

AZ-ONEBOARD im erweiterten Test

Diesen Beitrag gibt es auch als PDF-Dokument.

Im ersten Beitrag zum Thema AZ-Oneboard hatten wir uns um das Break Out Board mit dem BH1750 gekümmert und den Zugang zu den vom System nicht genutzten GPIOs geschaffen. Heute geht es um die Programmierung eines Moduls für den Zugriff auf das SHT30-Board, mit dem Temperatur und relative Luftfeuchte erfasst werden können. Grundlage dafür ist das Datenblatt von Sensirion. Der Chip kann Temperaturen von 0°C bis 65°C erfassen und wird, ähnlich wie der BH1750, über Kommandos gesteuert. Allerdings sind das hier 16-Bit-Words, dargestellt durch zwei Bytes. Wir werden für den SHT30 eine eigene Exception-Klasse bauen, eine CRC-Test-Routine programmieren und die Erkenntnisse über Decorators erneut nutzen. Außerdem gibt es einen Tipp zum bequemen Testen von Modulen. Genaueres erfahren Sie in dieser neuen Folge aus der Reihe

MicroPython auf dem ESP32, ESP8266 und Raspberry Pi Pico

heute

Das AZ-Oneboard und der SHT30

Die Steckleiste des Break Out Boards mit dem SHT30 hat dieselbe Pinfolge wie der BH1750 und kann daher auch an eine beliebige der drei Buchsenleisten angesteckt werden.

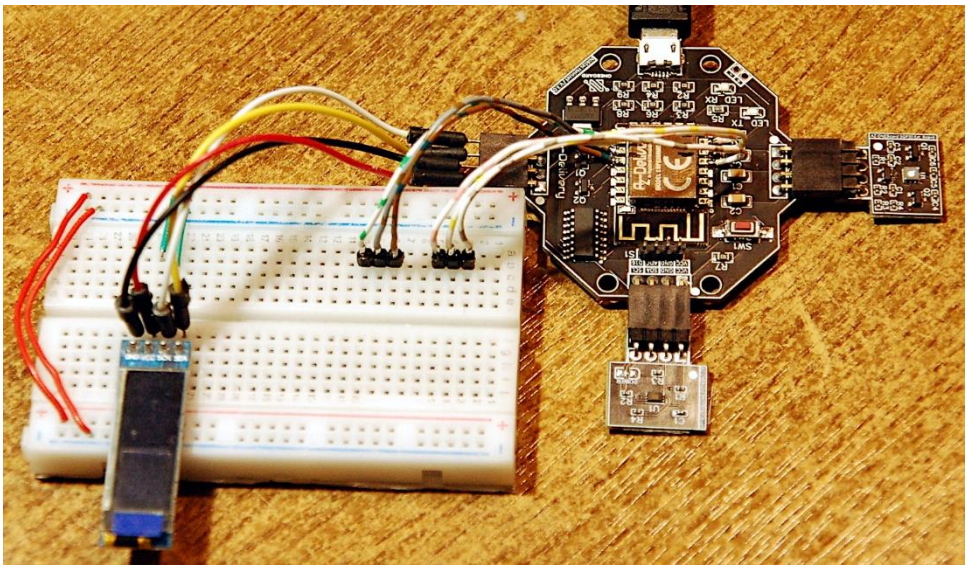

Abbildung 1: AZ-Oneboard mit zwei Satelliten

Die Hardware

Das sonst übliche Board mit dem Controller, hier ein ESP8266, wird durch das AZ-Oneboard ersetzt. An der Hardwareliste aus der ersten Episode hat sich nichts geändert. Das SHT30-Board ist ja bereits in dem Kit enthalten.

1	AZ-ONEBoard Entwicklungsboard inklusive Extensionboards SHT30, BH1750 & SGP30
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	Mini Breadboard 400 Pin mit 4 Stromschienen
1	Jumper Wire Kabel 3 x 40 STK. je 20 cm M2M/ F2M / F2F
optional	Logic Analyzer

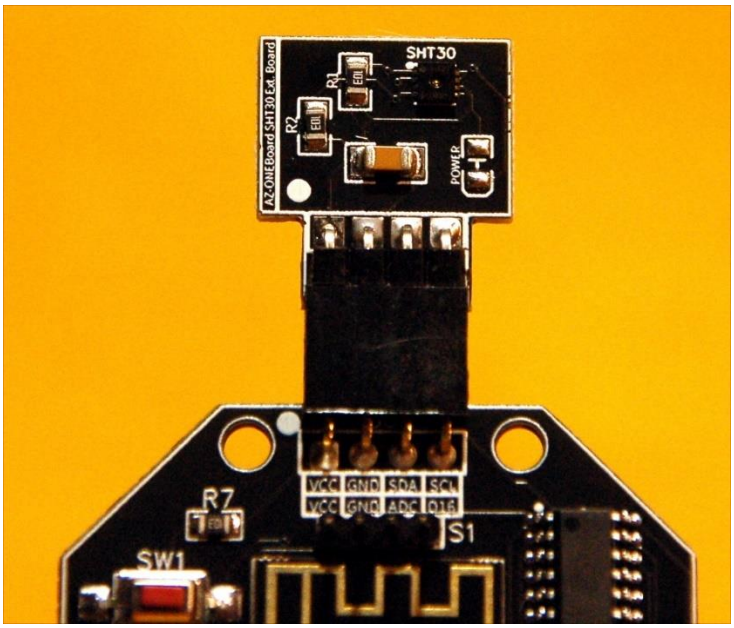

Abbildung 2: Anschluss des SHT30-Boards

Die Software

Fürs Flashen und die Programmierung des Controllers:

Thonny oder
µPyCraft

Signalverfolgung:

Saleae Logic 2

Verwendete Firmware für einen ESP32:

Micropython Firmware Download
v1.19.1 (2022-06-18) .bin

Verwendete Firmware für einen ESP8266:

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

Die MicroPython-Programme zum Projekt:

timeout.py: Nichtblockierender Software-Timer
oled.py: OLED-API
ssd1306.py: OLED-Hardwaretreiber
bh1750.py: Hardwaretreiber-Modul
bh1750_test.py: Demoprogramm
bh1750_kal.py: Programm zum Kalibrieren der Lux-Werte
sht30.py: Hardwaretreiber-Modul
sht30_test.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. Wie Sie den **Raspberry Pi Pico** einsatzbereit kriegen, finden Sie hier.

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.

Die Vorbereitung des AZ-Oneboards

Das Board kommt mit einem fertig programmierten ESP8266. Das Programm demonstriert die Möglichkeiten der drei Sensoren. Abbildung 2 zeigt die Ausgabe des Programms mit dem Terminalprogramm Putty.

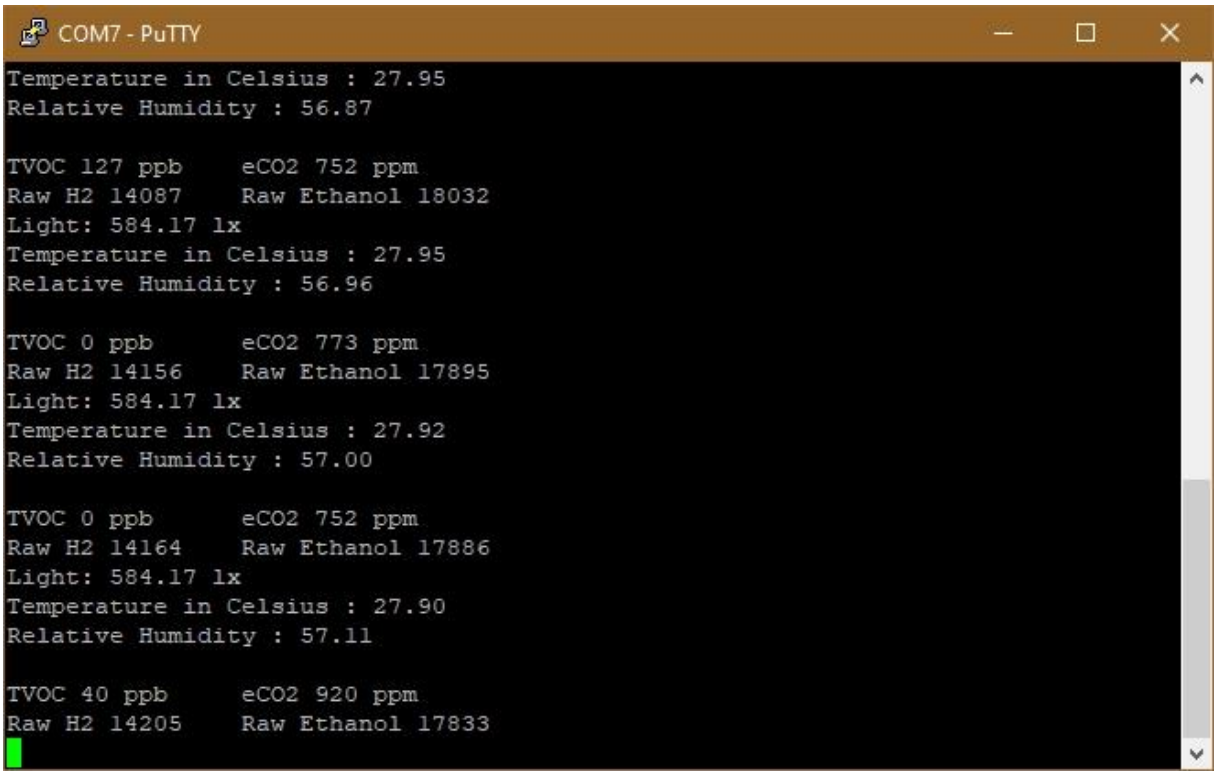

Abbildung 3: Ausgabe mit der Demosoftware

Wir wollen aber unser eigenes Programm in MicroPython schreiben und laufen lassen. Der erste Schritt zum Ziel ist, dass wir einen entsprechenden MicroPython-Kern auf den ESP8266 brennen, der das Demoprogramm überschreibt. Laden Sie also erst einmal die Firmware herunter und folgen Sie dann bitte dieser Anleitung. Der ESP8266 meldet sich dann im Terminal von Thonny etwa so:

MicroPython v1.23.0 on 2024-06-02; ESP module (1M) with ESP8266

Type "help()" for more information.
>>>

Die Programmierung des Moduls

Am AZ-Oneboard sind jetzt angesteckt: das SHT30-Board, das BH1750-Board und das OLED-Display. Wir starten wieder mit der händischen Überprüfung der Devices. Eingaben in REPL werden in diesem Skript **fett** formatiert, die Antworten vom ESP8266 *kursiv*.

>>> from machine import Pin,SoftI2C
>>> i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
>>> i2c.scan()
[35, 60, 68]

Die 35 = 0x23 kennen wir aus der letzten Folge als Geräteadresse des BH1750. 60 = 0x3C kommt vom OLED-Display. Also ist 68 = 0x44 die Hardwareadresse des SHT30, und alle drei Peripherie-Geräte funktionieren so weit, also legen wir los.

Ein paar Dinge importieren.

```
from machine import Pin
from time import sleep_ms
from sys import exit
```

Eine eigene Exception-Klasse

MicroPython hat zur Fehlerabsicherung die Basisklasse Exception mit einer großen Anzahl von Unterklassen, die bestimmte Bereiche abdecken. Wir wollen heute eine eigene Ausnahmeklasse für die Behandlung von Fehlern erstellen, die speziell den SHT30 betreffen. Sie soll **SHTError** heißen und muss von Exception erben.

```
class SHTError(Exception):
```

Folgende Fehlerarten können auftreten. Wir weisen den Bezeichnern Konstanten zu.

```
    SHTTempCRCError=const(1)
    SHTHumCRCError=const(2)
    SHTBoardError=const(3)
    SHTStatusCRCError=const(4)
```

Wie bei jeder Klasse brauchen wir einen Konstruktor, dem beim Aufruf eine der Konstanten übergeben wird. Den Wert weisen wir dem Instanz-Attribut **code** zu und initialisieren die Top-Klasse **Exception**, der wir eine Referenz auf die Methode **message**() übergeben. Diese wird sich um die Fehlermeldungen kümmern.

```
    def __init__(self, code=None):
        self.code=code
        super().__init__(self.message())

    def message(self):
        if self.code == TempCRCError:
            return "CRC-Fehler Temperatur"
        elif self.code == HumCRCError:
            return "CRC-Fehler rel. Feuchte"
        elif self.code == BoardError:
            return "Kein SHT3X-Board gefunden"
        else:
            return "nicht identifizierter Fehler"
```

Starten wir jetzt das Programm schon einmal im Editorfenster von Thonny – F5.

>>> %Run -c $EDITOR_CONTENT

Mit der folgenden Zeile lösen wir eine Fehlermeldung aus, welche die genaue Position und die Art des Fehlers mitteilt.

>>> raise SHTError(0x01)
Traceback (most recent call last):
* File "<stdin>", line 1, in <module>*
* File "<stdin>", line 13, in __init__*
* File "<stdin>", line 16, in message*
NameError: name 'TempCRCError' isn't defined

Als Nächstes gehen wir ans Ende des Programms. Durch den folgenden Trick können wir das Modul, welches wir gleich erstellen werden, ganz einfach testen, ohne dass wir die Datei nach jeder Änderung auf den ESP8266 hochladen müssen. Das erspart eine Menge Zeit und Umstände. Ergänzen wir das Programm also durch folgende Zeilen.

```
if __name__ == "__main__":
    from machine import SoftI2C
    i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
#     sh=SHT3X(i2c)
```

Wird das Modul als Hauptprogramm im Editorfenster gestartet, weist MicroPython dem Attribut **__name__** den String **"__main__"** zu. Wird eine Klasse dagegen regelrecht importiert, dann bekommt **__name__** den Namen der Klasse zugewiesen. Wenn wir unser bisheriges Programm abspeichern und auf den ESP8266 hochladen, können wir uns durch folgende Eingaben davon überzeugen.

>>> from sht30 import SHTError
>>> SHTError.__name__
'SHTError'

Mit der Funktionstaste F5 wird das Modul im Editorfenster als Hauptprogramm gestartet.

>>> __name__
'__main__'

Wir importieren somit die Klasse **SoftI2C** und instanziieren ein I2C-Bus-Objekt, so wie wir es in einem Hauptprogramm tun würden. Sobald die ersten Methoden unserer Klasse SHT3X existieren, können wir die unterste Zeile entkommentieren und haben damit nach einem Druck auf F5 unmittelbaren Zugriff auf die Routinen.

Die Klasse SHT3X

Beim BH1750 hatten wir die Kommandos durch Konstanten definiert. Prinzipiell ginge das hier auch mit Hilfe von 16-Bit-Worten. Dennoch habe ich einen anderen Weg gewählt. Wie wir aus dem vorangegangenen Beitrag wissen, können über den I2C-Bus nur Bytes-Objekte oder Bytearrays übertragen werden, weil diese Datentypen das Buffer-Protokoll unterstützen. Die Kommandos für den SHT30 setzen sich grundsätzlich aus zwei Bytes zusammen. Warum also nicht gleich Bytes-Objekte für die Festlegung verwenden? Das erspart auch noch die Umwandlung. Der Definitionsblock sieht damit aus wie folgt.

```
class SHT3X():

    SHTHWADR = const(0x44) # adr -> GND
    SHTPOLY  = const(0x131) # x^8 + x^5 + x^4 + 1 = 100110001
    SHTCMD_BREAK = b'\x30\x93'
    SHTCMD_SOFTRST   = b'\x30\xA2'
    SHTCMD_HEATER_ENABLE = b'\x30\x6D'
    SHTCMD_HEATER_DISABLE = b'\x30\x66'
    SHTCMD_READ_STATUS = b'\xF3\x2D'
    SHTCMD_CLEAR_STATUS = b'\x30\x41'
    SHTCMD_ART = b'\x2B\x32'
    SHTCMD_READ_VAL = b'\xE0\x00'
    SHTCMD_HEATER_ON = b'\x30\x6D'
    SHTCMD_HEATER_OFF = b'\x30\x66'
    SHTCMD_ONESHOT = b'\x2C\x24'
    SHTONESHOT = [b'\x06\x0D\x10',b'\x00\x0B\x16']
    SHTCMD_PERIODIC = b'\x20\x21\x22\x23\x27'
    SHTPERIODIC = [b'\x32\x24\x2F',
                b'\x30\x26\x2D',
                b'\x36\x20\x2B',
                b'\x34\x22\x29',
                b'\x37\x21\x2A']
```

Ein paar weitere Definitionen folgen.

```
    stretch=const(0)
    nostretch=const(1)
    mtime = [12,5,2]
    r_high =const(0)
    r_med = const(1)
    r_low = const(2)
    QueryDelay=(2000,1000,5000,250,100) #ms
```

Damit sind wir mit **__init__**() auch schon beim Konstruktor der Klasse SHT3X. Obligatorisch wird beim Aufruf ein I2C-Bus-Objekt übergeben.

```
    def __init__(self, i2c, hwadr=SHTHWADR):
        self.i2c=i2c
        self.hwadr=hwadr
        self.name="SHT30"
        self.repMode=0
        self.rawTemp=b'\x00\x00'
        self.rawHum=b'\x00\x00'
        self.temp=0.0
        self.hum=0.0
        if self.hwadr in i2c.scan():
            print(self.name,"initialisiert")
        else:
            raise SHTError(SHTBoardError)
        self.clearStatus()
        self.status=0
        sleep_ms(2)
```

Nach der Festlegung einiger Instanzattribute prüfen wir, ob sich die Hardware-Adresse des SHT30 in der Liste befindet, die **scan**() zurückgibt. Ist das nicht der Fall, werfen wir eine **SHTBoardError-Exception**. Das aufrufende Programm kann darauf mit try – except reagieren. Tut es das nicht, erfolgt der Programmabbruch. Dann setzen wir das Statusregister im SHT30 zurück und definieren das Attribut **status**.

Der Methode **sendCommand**() übergeben wir den Kommando-Code, der mittels **i2c.writeto**() nach der Hardware-Adresse über den I2C-Bus gesendet wird.

```
    def sendCommand(self,cmd):
        self.i2c.writeto(self.hwadr,cmd)
```

Jetzt können wir die Methode **clearStatus**() definieren, die auf **sendCommand**() zurückgreift.

```
    def clearStatus(self):
        self.sendCommand(SHT3X.SHTCMD_CLEAR_STATUS)
```

Aufmerksame Leser stellen hier fest, dass das Kommando **SHTCMD_CLEAR_STATUS** zusammen mit dem Klassenbezeichner referenziert wird. Beim BH1750 war das nicht so. Der Grund ist, dass die Kommandos hier als Bytes-Objekte definiert sind und nicht als Konstanten. Konstanten werden direkt im Programmcode abgelegt, bei Variablen ist das nicht möglich. Würde man alle Kommandodefinitionen als Konstanten von Bytes-Objekten definieren, dann könnte **SHT3X.** wegbleiben. Als Konstanten werden einige Basistypen akzeptiert, also zum Beispiel Ganzzahlen, Fließkommazahlen, Bytes-Objekte und String-Literale.

Jetzt können wir die letzte Zeile im Programm entkommentieren.

```
    sh=SHT3X(i2c)
```

Als Nächstes soll eine Routine entstehen, die einen Einzelschuss triggert und anschließend die Messwerte einliest. Die Tabelle in Abbildung 3 zeigt uns die Möglichkeiten auf. Repeatability übersetzt Onkel Google als Wiederholgenauigkeit. So heißt es auf Seite 2 im Kleingedruckten, dass es sich dabei um den dreifachen Wert der Standardabweichung von mehreren Messungen handelt. Erreicht wird das wohl durch mehrfaches Sampeln der Messgrößen aus denen dann ein Mittelwert berechnet wird. Das würde auch die verschiedenen Messzeiten der einzelnen Modi erklären, die zwischen 4,5ms (Low) und 15,5ms (High) liegen

Condition		Hex. code	
Repeatability	Clock stretching	MSB	LSB
High	enabled	0x2C	06
Medium			0D
Low			10
High	disabled	0x24	00
Medium			0B
Low			16

Abbildung 4: Singleshot Modi

Clockstretching **enabled** bedeutet im I2C-Bus-Protokoll, dass ein Peripheriegerät die Taktleitung auf 0 ziehen kann, wenn es gerade beschäftigt ist und keine Daten senden kann. Beim SHT30 verhält sich das ähnlich. Wenn nach einem empfangenen Kommando der Wandlerprozess getriggert wurde und gleich danach eine Leseanfrage vom ESP8266 gesendet wird, antwortet der SHT30 darauf mit einem Acknowledge-Bit (ACK), legt aber die Taktleitung so lange auf 0, bis der Prozess beendet ist und Messdaten vorliegen. Dann gibt er die Taktleitung frei und sendet die Datenbytes mit dem wieder einsetzenden Takt vom ESP8266. Die Blöcke links in Abbildung 5 entsprechen den Senden des Kommandos und einer Leseanfrage, die der SHT30 aber nicht bedienen kann, weil noch keine Daten vorliegen. Die Wandlung dauert etwa 12ms. Danach werden die Daten übertragen. Die Abbildungen 6 und 7 zeigen die Details.

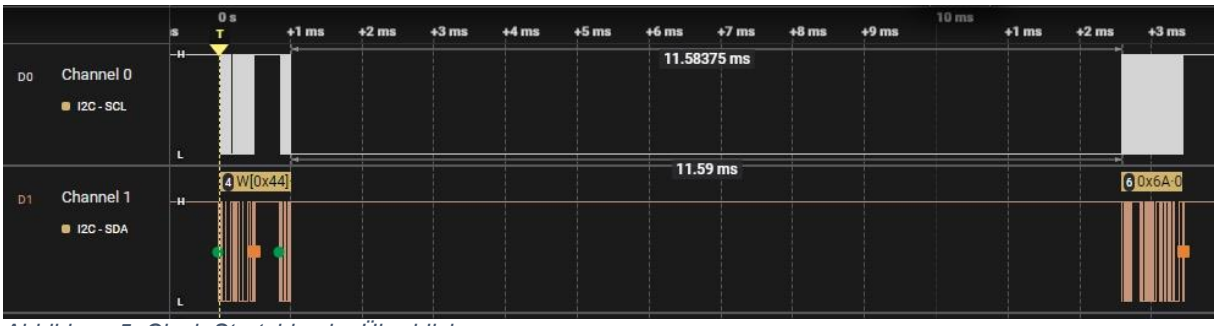

Abbildung 5: Clock-Stretching im Überblick

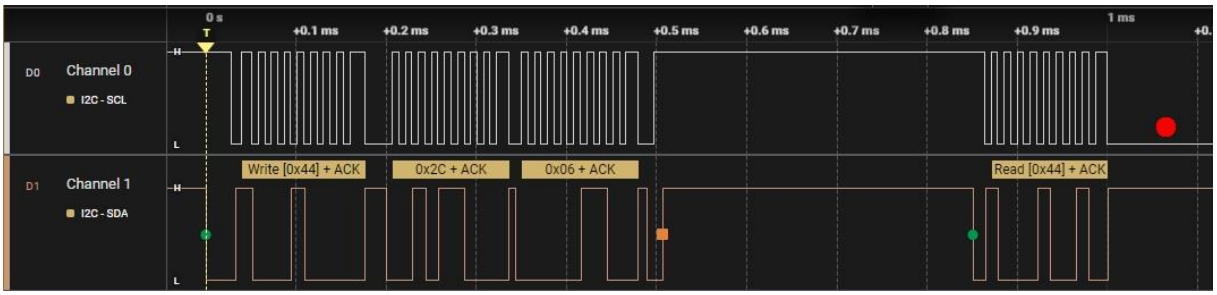

Abbildung 6: Clock-Stretching – Kommando 0x2C06 und Leseanfrage

Nach der Leseanfrage beginnt beim roten Punkt das Clockstretching. Für ca. 11,6 ms hält der SHT30 die CLK-Leitung auf 0.

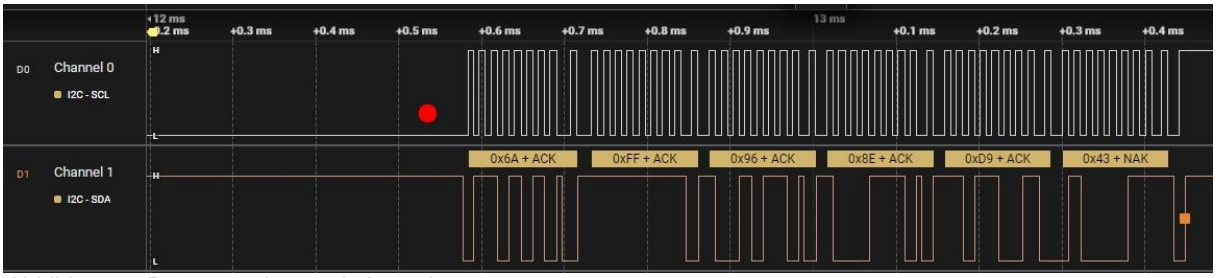

Abbildung 7: Daten senden nach Auszeit

Danach kann der ESP8266 die Taktleitung wieder übernehmen und die sechs Datenbytes einlesen.

Ist Clockstretching nicht enabled, beantwortet der SHT30 eine Leseanfrage mit einem Not Acknowledge-Bit (NACK). Diesen Umstand muss unser Programm

handeln. Am besten man wartet mit dem Senden der Leseanfrage so lange, bis der SHT30 laut Datenblatt fertig sein müsste.

Die Routine muss nun aus den Vorgaben durch die Parameter **stretching** und **rep** das Kommandowort zusammenbauen, versenden, warten, falls stretching disabled ist und dann die Daten einlesen.

Die übergebenen Argumente werden auf Einhaltung des Wertebereichs geprüft. Dann verwenden wir sie als Indices in das Bytes-Objekt SHT_ONESHOT und in die Liste mit den LOW-Bytes, die die Wiederholgenauigkeit festlegen. Die herausgepickten Werte senden wir als Bytearray zum SHT30.

```
    def getOneshot(self, stretching=0, rep=0):
        assert stretching in [stretch, nostretch]
        assert rep in [r_high, r_med, r_low]
        buf=bytearray(2)
        buf[0]=SHT3X.SHTCMD_ONESHOT[stretching]
        buf[1]=SHT3X.SHTONESHOT[stretching][rep]
        self.sendCommand(buf)
```

Wenn das Stretching deaktiviert ist, müssen wir selber eine Pause einhalten, deren Dauer vom Repeatability-Wert abhängt und aus der Liste **mtime** mit **rep** als Index geholt werden kann. Ist Stretching aktiviert, kümmert sich der ESP8266 selbst im Verborgenen um die korrekte Abwicklung.

Aber wieso holen wir dann sechs Bytes vom SHT30 ab, Temperatur und Feuchte sind doch nur 16-Bit-Werte? Nun, nach den beiden Bytes für den Messwert wird jeweils ein Byte für eine Prüfsumme drangehängt, um die Korrektheit der Werte prüfen zu können.

```
        if stretching==nostretch:
            sleep_ms(SHT3X.mtime[rep])
        echo=self.i2c.readfrom(self.hwadr,6)
```

Diesen Cyclic Redundancy Check oder kurz CRC übernimmt die Routine **crcTest**(), der wir jeweils die beiden Datenbytes und die Prüfsumme in einem Slice des empfangenen Bytes-Objekts übergeben, und zu der wir gleich im Anschluss kommen. Sie meldet True zurück, wenn kein Fehler festgestellt wurde, sonst False. In diesem Fall werfen wir die entsprechende Exception.

```
        if not self.crcTest(echo[:3]): # die ersten drei Bytes
            raise SHTError(SHTError.SHTTempCRCError)
        if not self.crcTest(echo[3:6]): # die letzten drei
            raise SHTError(SHTError.SHTHumCRCError)
        self.rawTemp=echo[:2]
        self.rawHum=echo[3:5]
        return [echo[:2],echo[3:5]]
```

Die Rohwerte werden in Instanz-Attribute verfrachtet und als Liste zurückgegeben.

Für den CRC brauchen wir drei Dinge, die Daten, die Prüfsumme und ein sogenanntes Polynom. Beim SHT30 lautet es auf 0x131. Der im Handbuch

angegebene Wert von 0x31 ist falsch, wie man an dem daneben angegebenen Polynom ersehen kann! Jeder vorkommenden Potenz von x entspricht nämlich eine 1 an der Bitposition, die der Exponent angibt. Eine Beschreibung des Prüfvorgangs finden Sie hier, zusammen mit einem Beispiel.

SHTPOLY = const(0x131) # x^8 + x^5 + x^4 + 1 = 0b100110001

```
    def crcTest(self, blobb):
        start=0xFF
        for h in blobb[:-1]:
            start ^= h
            for i in range(8):
                if start & 0x80:
                    start = (start << 1) ^ POLY
                else:
                    start <<= 1
        return blobb[-1] == start
```

Durch Slicing und Indexing trennen wir das Bytes-Objekt in die beiden Datenbytes und das Prüfsummenbyte auf. **blobb[:-1]** liefert Byte 0 und Byte 1 und **blobb[-1]** liefert die Prüfsumme. Programm starten – F5.

>>> blobb=b'\x8e\xd9\x43'
>>> blobb[:-1]
b'\x8e\xd9'
>>> blobb[-1]
67   (= 0x43)

>>> sh.crcTest(blobb)
True

Für die kontinuierliche Erfassung von Messwerten, sind das Anstoßen der Messreihe und das Einlesen der Werte in getrennten Routinen realisiert. Es gibt dabei kein Clockstretching, dafür kann man aber verschiedene Messintervalle einstellen. Die Wartezeiten sind in dem Tupel **QueryDelay** untergebracht, auf die ein Hauptprogramm zugreifen kann.

Wie beim Single Shot wird das Befehlswort gemäß unseren Wünschen aus zwei Bytes zusammengesetzt. **mps** gibt die Anzahl Messungen pro Sekunde an.

Condition		Hex. code	
Repeatability	mps	MSB	LSB
High	0.5	0x20	32
Medium			24
Low			2F
High	1	0x21	30
Medium			26
Low			2D
High	2	0x22	36
Medium			20
Low			2B
High	4	0x23	34
Medium			22
Low			29
High	10	0x27	37
Medium			21
Low			2A

Abbildung 8: Das sind die Dauerläufer

startPeriodic() entspricht dem ersten Teil von **getOneshot**(). **mps** und **rep** sind Indices in die jeweiligen Listen.

```
    def startPeriodic(self, mps=0, rep=0):
        assert mps in range(5)
        assert rep in range(3)
        buf=bytearray(2)
        buf[0]=SHT3X.SHTCMD_PERIODIC[mps]
        buf[1]=SHT3X.SHTPERIODIC[mps][rep]
        self.sendCommand(buf)
```

Das Einlesen von Werten übernimmt **readPeriodic**(), das ähnlich wie der zweite Teil von **getOneshot**() aufgebaut ist. Wenn die Serienmessung getriggert ist, braucht man nur noch **readPeriodic**() auzurufen.

```
    def readPeriodic(self):
        self.sendCommand(SHT3X.SHTCMD_READ_VAL)
        echo=self.i2c.readfrom(self.hwadr,6)
        if not self.crcTest(echo[:3]):
            raise SHTError(SHTError.SHTTempCRCError)
        if not self.crcTest(echo[3:6]):
            raise SHTError(SHTError.SHTHumCRCError)
        self.rawTemp=echo[:2]
        self.rawHum=echo[3:5]
        return [echo[:2],echo[3:5]]
```

Wir starten das Programm erneut - F5

>>> sh.startPeriodic(2,0)
>>> sh.readPeriodic()
[b'kY', b'\x92\xfc']
>>> sh.readPeriodic()
[b'kd', b'\x92\xaa']
>>> sh.readPeriodic()
[b'kn', b'\x92\xd2']

Alle jetzt noch fehlenden Kommandos haben feste Bytes-Objekte, die nicht zusammengesetzt werden müssen, und daher weisen sie auch weitgehend dieselbe einfache Struktur auf.

Scanvorgänge lassen sich mit **sendBrake**() abbrechen.

```
    def sendBrake(self):
        self.sendCommand(SHT3X.SHTCMD_BREAK)
```

sendReset() setzt den Sensor zurück und bewirkt das neue Laden von Kalibrierungsdaten.

```
    def sendReset(self):
        self.sendCommand(SHT3X.SHTCMD_SOFTRST)
```

Der Chip enthält eine Wärmequelle, mit deren Hilfe man die Funktion überprüfen kann. Ohne Argument aufgerufen, liefert **heater**() den Status, der wird durch Bit13 des Statusregisters angegeben. Wir isolieren das Bit durch Undieren und schieben es an die Position 0. Mit den Argumentwerten 0 oder 1 schalten wir den Heater aus oder ein.

```
    def heater(self, status=None):
        if status is None:
            s=self.readStatus()
            return (s & 0x2000) >> 13
        if status:
            self.sendCommand(SHT3X.SHTCMD_HEATER_ENABLE)
        else:
            self.sendCommand(SHT3X.SHTCMD_HEATER_DISABLE)
```

readStatus() folgt in der Struktur dem Anfordern und Einlesen von Messwerten. Auch hier wird eine Prüfsumme mitgeliefert.

```
    def readStatus(self):
        self.sendCommand(SHT3X.SHTCMD_READ_STATUS)
        sleep_ms(15)
        echo=self.i2c.readfrom(self.hwadr,3)
        if not self.crcTest(echo[:3]):
            raise SHTError(SHTError.SHTStatusCRCError)
        self.status=(echo[0]<<8) | echo[1]
        return self.status
```

Eine ganze Reihe von Statusbits steht zur Verfügung. Durch den Decorator **@property** sparen wir uns die Eingabe von Klammern.

```
    @property
    def resetDetected(self):
        return (self.readStatus() & 0x0010) >> 4

    @property
    def commandStatus(self):
        return (self.readStatus() & 0x0002) >> 1

    @property
    def rhTrackingStatus(self):
        return (self.readStatus() & 0x0800) >> 11

    @property
    def tTrackingStatus(self):
        return (self.readStatus() & 0x0400) >> 10

    @property
    def checksumStatus(self):
        return self.readStatus() & 0x0001
```

Programmstart – F5

>>> sh.heater()
0
>>> sh.heater(1)
>>> sh.heater()
1
>>> sh.heater(0)
>>> sh.heater()
0

>>> sh.resetDetected
0
>>> sh.commandStatus
0
>>> sh.rhTrackingStatus
0

Statusregister löschen:

```
    def clearStatus(self):
        self.sendCommand(SHT3X.SHTCMD_CLEAR_STATUS)
```

Die Methoden **getOneshot**() und **readPeriodic**() liefern nur die Rohwerte für Temperatur und relative Luftfeuchte. **calcValues**() berechnet daraus die richtigen Messwerte. Die Formeln dafür verrät uns das Datenblatt. Zur Berechnung wird auf die bereits eingelesenen Rohwerte in den Attributen **rawTemp** und **rawHum** zurückgegriffen.

```
    def calcValues(self):
        self.temp=175.0 * ((self.rawTemp[0]<<8) |
self.rawTemp[1]) / 0xFFFF - 45
        self.hum= 100.0 * ((self.rawHum[0]<<8) |
self.rawHum[1]) / 0xFFFF
        return [self.temp,self.hum]
```

Die Werte werden in Instanzattributen gespeichert und auch direkt in Form einer Liste zurückgegeben. **temp**, **hum** und **values** ermöglichen das nachträgliche Auslesen die berechneten Werte.

```
    @property
    def Temp(self):
        return self.temp

    @property
    def Hum(self):
        return self.hum

    @property
    def Values(self):
        return [self.temp,self.hum]
```

Programmstart - F5

>>> sh.getOneshot()
[b'#', b'\x90>']
>>> sh.calcValues()
[28.9227, 56.3455]
>>> sh.Temp
28.9227
>>> sh.Hum
56.3455
>>> sh.Values
[28.9227, 56.3455]

Manchmal will man den ganzzahligen und den Bruchanteil der Messwerte getrennt haben. Das erledigt die Methode **readValuesDecimal**(). Sie holt sich die Werte aus den Instanzattributen und produziert je eine Liste für Temperatur und Feuchte.

```
    def readValuesDecimal (self):
        temp=[int(self.temp),int((self.temp-
int(self.temp)+0.005)*100)]
        hum=[int(self.hum),int((self.hum-
int(self.hum)+0.005)*100)]
        return [temp,hum]
```

Programmstart – F5

>>> sh.readValuesDecimal()
[[28, 92], [56, 35]]

Für den Test des Moduls müssen wir die Datei als **sht30.py** abspeichern und zum ESP8266 hochladen.

Testprogramm

Das Programm **sht30_test.py** enthält vier Routinen, die die Funktion des Moduls im Programmkontext demonstrieren. Die Routinen fassen zusammen, was wir bereits händisch auf REPL ausprobiert haben.

Eine fünfte Funktion bezieht die Helligkeitsmessung mit dem Modul BH1750 mit ein. Die Datei **bh1750.py** muss dazu in den Flash des ESP8266 hochgeladen sein. Wir lassen das Programm im Editorfenster von Thonny ausführen und können dann jede der Funktionen einzeln aufrufen.

```
from machine import Pin,SoftI2C
from sht30 import SHT3X,SHTError
from time import sleep_ms

i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
sht=SHT3X(i2c)

def getRawVals(mps):
    sht.startPeriodic(mps,0)
    sleep_ms(16)
    print(sht.readPeriodic())
    sleep_ms(SHT3X.QueryDelay[mps])
    print(sht.readPeriodic())
    sleep_ms(SHT3X.QueryDelay[mps])
    print(sht.readPeriodic())

def getTempHum():
    read=False
    while not read:
        try:
            sht.getOneshot(SHT3X.nostretch,SHT3X.r_low)
            temp,hum=sht.calcValues()
            st=sht.readStatus()
            print(temp,hum,"{:16b}".format(st))
            print(sht.readValuesDecimal())
            read=True
        except:
            sleep_ms(50)

def getPeriodic(mps):
    sht.startPeriodic(mps,SHT3X.r_high)
    sleep_ms(16)
    while 1:
        sht.readPeriodic()
```

```
        temp,hum=sht.calcValues()
        st=sht.readStatus()
        print("{:04.2f},{:04.2f}
{:016b}".format(temp,hum,st))
        sleep_ms(SHT3X.QueryDelay[mps])

def thermoHygroMeter(mps):
    sht.startPeriodic(mps,SHT3X.r_high)
    sleep_ms(16)
    d.writeAt("THERMO-HYGRO-01",1,0,False)
    while 1:
        d.clearFT(0,1,15,2,False)
        sht.readPeriodic()
        temp,hum=sht.calcValues()
        d.writeAt("Temp: {:04.2f} *C".format(temp),1,1,False)
        d.writeAt(" Hum: {:04.2f} %".format(hum),1,2)
        sleep_ms(SHT3X.QueryDelay[mps])

def duotest(mps):
    from bh1750 import BH1750
    bh=BH1750(i2c)
    sht.startPeriodic(mps,SHT3X.r_high)
    sleep_ms(16)
    while 1:
        d.clearAll(False)
        lumi=bh.luminanz()
        d.writeAt("Lumi: {} lux".format(lumi), 1,0,False)
        sht.readPeriodic()
        temp,hum=sht.calcValues()
        d.writeAt("Temp: {:04.2f} *C".format(temp),1,1,False)
        d.writeAt(" Hum: {:04.2f} %".format(hum),1,2)
        sleep_ms(SHT3X.QueryDelay[mps])
```

In der nächsten Folge schließen wir die Untersuchung des AZ-Oneboards mit einem Modul für das SGP30-Board zunächst ab. Auch da gibt es wieder interessante Dinge zu erfahren.

Also, bleiben Sie dran!