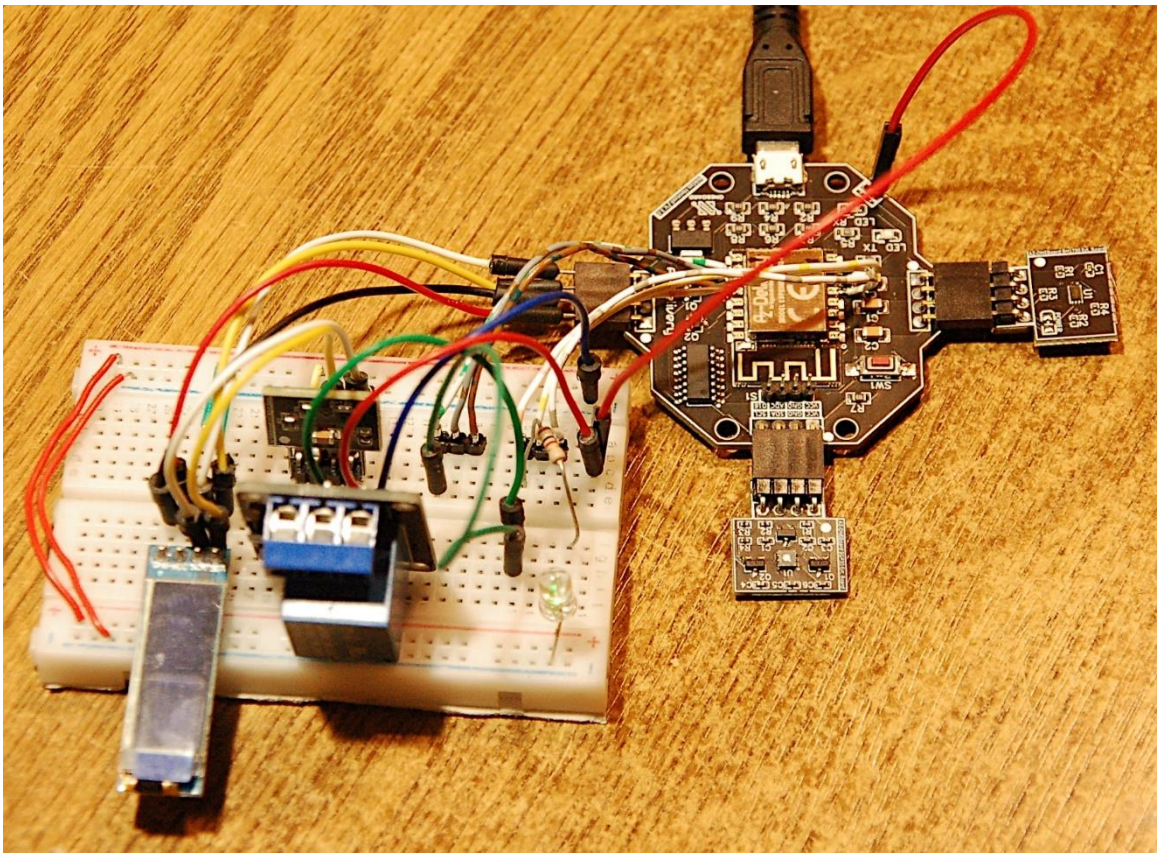

AZ-Oneboard - Vollausbau

Dieser Beitrag ist auch als PDF-Dokument erhältlich.

In den vorangegangenen Beiträgen az-oneboard mit bh1750 und az-oneboard mit SHT30 hatten wir uns mit dem Beleuchtungsstärke-Sensor BH1750 und dem SHT30 beschäftigt, der Temperatur und relative Luftfeuchte misst. Heute kommt der dritte Sensor des Kits an die Reihe, der SGP30. Er ist zuständig für die Messung der Luftqualität in Innenräumen. Das Akronym IAQ leitet sich genau von dieser Eigenschaft ab, Indoor Air Quality. Für diesen Sensor werden wir ein MicroPython-Modul stricken.

Mit der Erweiterung des AZ-Oneboards aus der zweiten Episode können wir ferner ein Relais und zwei LEDs ansteuern. Dabei bleiben immer noch drei GPIOs zur weiteren freien Verfügung. Anhand von LED und Relais, schauen wir uns an, wie man programmtechnisch einen Schalter mit Zeitverzögerung beim Ausschalten (Treppenhaus-Automat, Monoflop) und einen Ein-Aus-Schalter mit Hysterese, also mit versetzter Ein-Ausschaltschwelle, realisieren kann. Außerdem versuchen wir, mit dem BH1750 ein nettes kleines Gadget aufzubauen. Haben Sie schon einmal eine LED mit einem Feuerzeug angemacht und dann später einfach ausgeblasen? All diese Themen besprechen wir in dieser neuen Episode aus der Reihe

MicroPython auf dem ESP32, ESP8266 und Raspberry Pi Pico

heute

Das AZ-Oneboard im Vollausbau

Alle drei Satelliten-Boards zum AZ-Oneboard haben dieselbe Pinbelegung und sind daher gegeneinander austauschbar. Die Reihenfolge der Pins am OLED weicht davon ab. Beim Verbinden mit Jumperkabeln muss man darauf achten, um die Boards nicht zu beschädigen. Wie man die vom AZ-Oneboard ungenutzten GPIOs herausführen kann. Ist im zweiten Teil beschrieben. Zur I2C-Bus-Erweiterung habe ich mir eine kleine Lochrasterplatine mit 4 Pins Breite und 12 Pins Länge zugeschnitten und an den Enden mit einer gewinkelten Steckleiste sowie einer gewinkelten Buchsenleiste versehen. Dazwischen habe ich zwei gerade, vierpolige Buchsenleisten eingelötet. So kann ich neben den drei Standard-Sensoren noch weitere Satelliten, wie das OLED-Display, anklemmen. Die Kontakte sind längs miteinander verbunden.

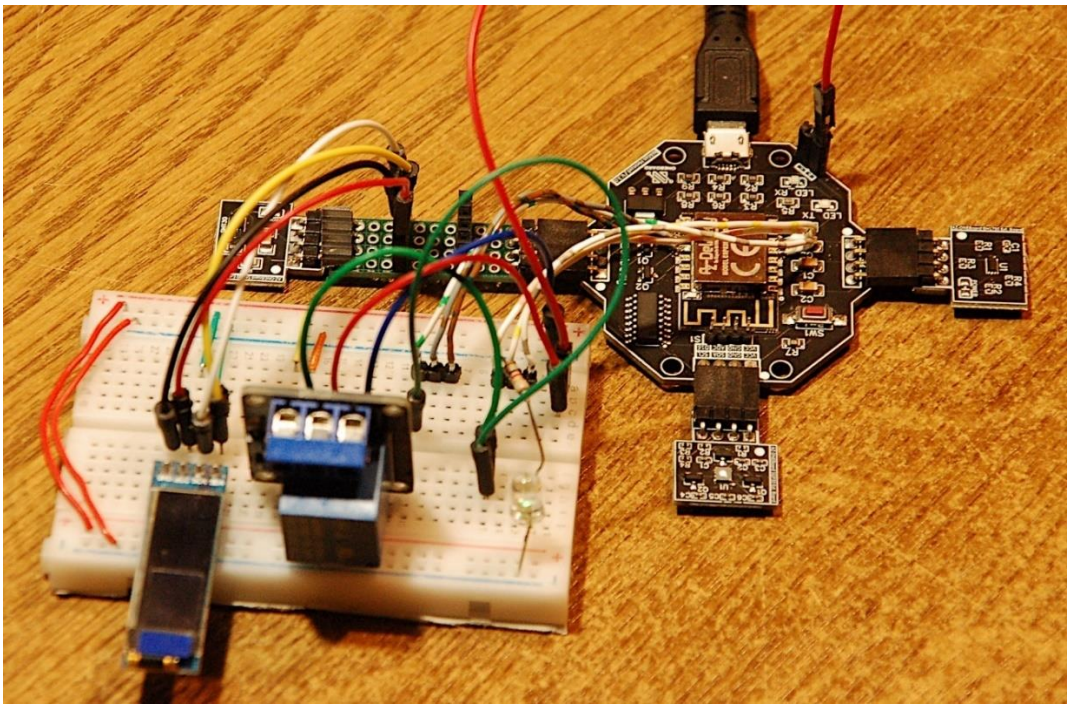

Abbildung 1: AZ-Oneboard mit Sensoren, Relais und LED

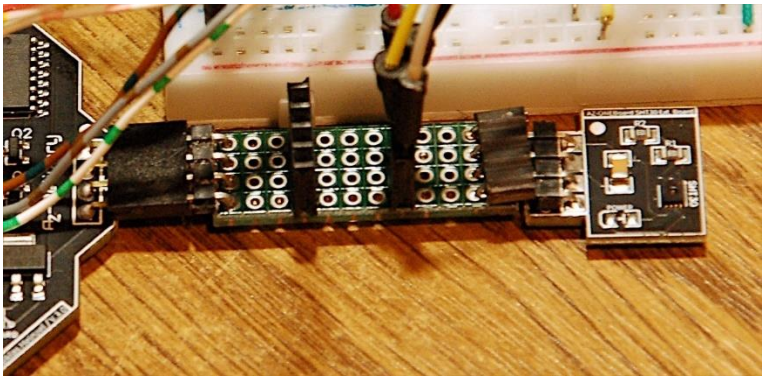

Abbildung 2: I2C-Bus-Erweiterungsplatine

Das Relais liegt mit dem "+"-Anschluss an +5V des AZ-Oneboards, "-" geht an GND und der "S"-Anschluss wird mit GPIO15 verbunden. GPIO13 des AZ-Oneboards führt über einen 1,0kΩ-Widerstand an die Anode der LED (längerer Anschluss), die Kathode liegt an GND.

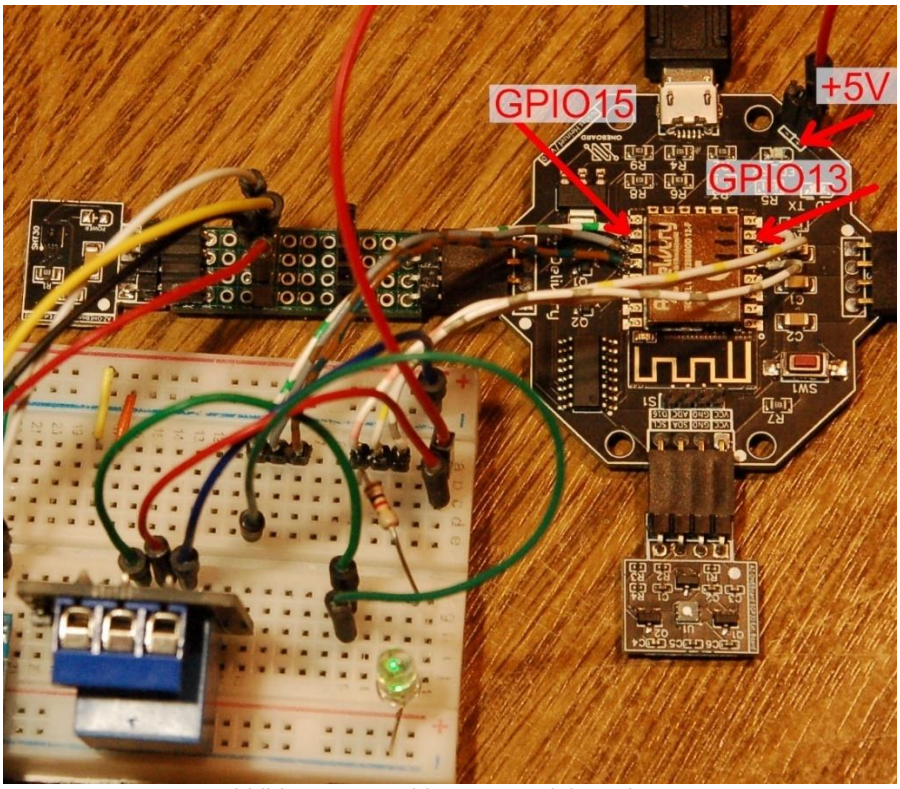

Abbildung 3: Anschluss von Relais und LED

Die Hardware

Das sonst übliche Board mit dem Controller, hier ein ESP8266, wird durch das AZ-Oneboard ersetzt. Zur Hardwareliste aus der ersten Episode haben sich ein Relais, zwei Widerstände und zwei LEDs gesellt. Die Sensor-Boards sind 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	1-Relais 5V KY-019 Modul High-Level-Trigger
1	LED grün zum Beispiel LED Leuchtdioden Sortiment Kit, 350 Stück, 3mm & 5mm, 5 Farben - 1x Set
1	LED weiß
1	Widerstand 1,0kΩ zum Beispiel Widerstände Resistor Kit 525 Stück Widerstand Sortiment, 0 Ohm -1M Ohm
1	Widerstand 150Ω
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

Die Software

Fürs Flashen und die Programmierung des Controllers:

Thonny oder
µPyCraft

Für den ersten Test des AZ-Oneboards:

Terminal-Programm Putty

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
sgp30.py: Hardwaretreiber-Modul
sgp30_test.py: Demoprogramm
sensortest.py: Demoprogramm
gadget.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.

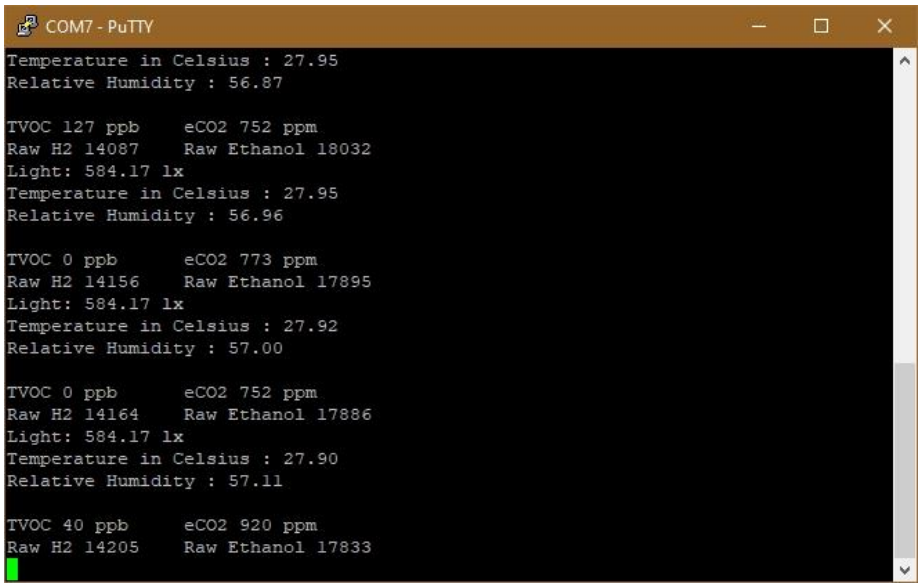

Abbildung 4: Ausgabe mit der Demosoftware

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.
>>>

Das SGP30-Modul

Wie die beiden anderen Sensoren, wird auch der SGP30 durch einen Satz von Kommandos angesteuert und abgefragt. Beim BH1750 waren das Bytes, beim SHT30 waren es Words, also 16-Bit-Werte. Die zu übertragenden Daten hatten aber jeweils gleiches Format. Immerhin mussten wir bei diesen Sensoren verschiedene Wandlungszeiten berücksichtigen, die sich aber aus dem jeweiligen Betriebsmodus durch Berechnung oder aus Tabellen ableiten ließen.

Beim SGP30 gibt es auch Zwei-Byte-Kommandowörter aber die Rückmeldungen des Chips können 0, 3, 6 oder gar 9 Bytes lang sein. Dazu kommen auch noch verschiedene Verzögerungen bis die Daten bereitstehen. Wenn man nicht für jedes Kommando eine eigene Routine schreiben will, muss man tricksen.

Zwei weitere Dinge, die wir bereits bei den anderen Sensoren eingesetzt hatten, finden wir auch in diesem Modul wieder. Das ist eine Exception-Klasse, die genau auf die Klasse SGP30 zugeschnitten ist und außerdem treffen wir auch wieder auf die Prüfsummenberechnung. Für den ESP8266 habe ich auch das Modul **sgp30.py** gesundschrumpfen müssen, weil das Programm für die drei Sensoren sonst nicht zum Laufen zu bringen war - Speichermangel. Hier kommt also die abgespeckte Version **sgp30_s.py**.

Die Klasse SGPError

Die Exception-Klasse SGPError gleicht im Aufbau der Klasse SHTError aus dem Modul sht30.py, wo diese auch ausführlich diskutiert wurde. Ich gehe hier nicht mehr näher darauf ein.

Einleitend zur Klasse SGP30 das Importgeschäft. **pack**() aus dem Modul **struct** wird uns bei der Typänderung von Zahlen helfen. Zur Berechnung der absoluten Feuchte der Luft in g/L brauchen wir die Exponentialfunktion e^x = exp().

```
from machine import Pin
from time import sleep_ms
from sys import exit
from struct import pack
from math import exp
```

Dann folgen einige Konstanten, die Geräteadresse des SGP30 ist 0x58 = 88 und das Polynom zur CRC-Berechnung ist 0x131. Obacht, im Datenblatt ist ein Fehler, dort heißt es 0x31, das ist Quark! Das Startbyte zur CRC-Berechnung ist 0xFF und der Chip-Test liefert stets das gleiche Ergebnis 0xD400.

```
class SGP30():

    SGPHWADR = const(0x58)
    SGPPOLY = const(0x131) # !!! Fehler im Datenblatt!!!
                           # Seite 12; Tabelle 13; Zeile 5
    SGPCRCSTART = const(0xFF)
    SGPTestResult = const(0xD400)
```

Der Trick bei den Kommandos ist einfach. Anstelle einer eigenen Routine für jedes Kommando, ein verwende ich ein Bytes-Objekt mit vier Elementen welches ich innerhalb der Sende-Routine aufdröseln werde. Dier ersten beiden Bytes stellen das Kommandowort, das dritte Byte nennt die Anzahl zu empfangender Bytes inclusive Prüfsumme und das vierte Byte codiert die Wartedauer bis zur Bereitstellung der Daten in Millisekunden.

```
    # Command word, response bytes, delay in ms
    SGPCMD_getSerial=b'\x36\x82\x09\x02'
    SGPCMD_iaq_init=b'\x20\x03\x00\x0A'
    SGPCMD_measure_iaq=b'\x20\x08\x06\x0c'
    SGPCMD_get_iaq_baseline=b'\x20\x15\x06\x0A'
    SGPCMD_set_iaq_baseline=b'\x20\x1E\x00\x0A'
    SGPCMD_set_absolute_humidity=b'\x20\x61\x00\x0A'
    SGPCMD_measure_test=b'\x20\x32\x03\xDC'
    SGPCMD_get_feature_set=b'\x20\x2F\x03\x0A'
    SGPCMD_measure_raw=b'\x20\x50\x06\x19'
    SGPCMD_get_tvoc_inceptive_baseline=b'\x20\xB3\x03\x0A'
    SGPCMD_set_tvoc_baseline=b'\x20\x77\x00\x0A'
```

Der Konstruktor nimmt eine Reihe von Argumenten. Zwingend anzugeben ist ein I2C-Bus-Objekt, optional können abweichende Werte für die Hardware-Adresse sowie für die Parameter **test** und **iaq_init** angegeben werden.

Wir reichen die Referenz auf das I2C-Bus-Objekt an das entsprechende Instanzattribut weiter und prüfen, ob die Hardware-Adresse in der Liste enthalten ist, die **i2c.scan**() liefert. Wir der SGP30 nicht gefunden werfen wir eine SGPBoardError-Exception. Sonst merken wir uns die Hardware-Adresse. Die absolute Luftfeuchte belegen wir mit 13.355 g/L vor Dieser Wert sollte zyklisch aus der Raumtemperatur und der relativen Luftfeuchte (gemessen mit SHT30) mit Hilfe der Methode **rel2absHumidity**() berechnet werden, zu der wir später kommen.

```
    def __init__(
            self, i2c,
            adr=SGPHWADR,
            test=False,
            iaq_init=True):

            self.i2c = i2c
            if adr not in self.i2c.scan():
                raise SGPError(SGPBoardError)
            self.hwadr = adr
            self.abshum = 13.355
```

Die Seriennummer wird eingelesen, ebenso die Version des Feature-Sets. Gegebenenfalls stoßen wir eine Testmessung an, deren Ergebnis 0xD400 sein muss. Die erhaltenen Daten stellen wir in REPL dar. Abschließend initialisieren wir die IAQ-Messung.

```
            self.serial = self.readSerial()
            self.featureSet = self.getFeatureSet()
            if test:
                if SGPTestResult != self.measure_test():
                    raise SGPError(SGPChipError)
            print(
                "SGP30 initialisiert @ {}\n".for-
mat(hex(self.hwadr)),
                "Serial ID: {}\n".format(self.serial),
                "Feature set: {}\n".format(self.featureSet),
                "Algoritmus init: {}".format(iaq_init))
            if iaq_init:
                self.iaqInit()
```

crcTest() berechnet die Prüfsumme der empfangenen Datenbytes und liefert True zurück, wenn diese mit der Prüfsumme vom SGP30 übereinstimmt. Der Chip sendet immer 2 Datenbytes gefolgt von der Prüfsumme. Die drei Bytes werden an den Parameter **blobb** übergeben und zunächst in Datenteil und Prüfsumme aufgetrennt. Die for-Schleifen führen den CRC (Cyclic Redundancy Check) durch.

```
    def crcTest(self,blobb): # 6.6 data = 2 bytes + CRC-Byte
            start = SGPCRCSTART
            for h in blobb[:-1]:
                start ^= h
                for i in range(8):
                    if start & 0x80:
                        start = (start << 1) ^ SGPPOLY
                    else:
                        start <<= 1
            return start == blobb[-1]
```

Weil wir auch Daten mit einem CRC-Byte an den SGP30 senden wollen, brauchen wir eine Routine, welche die Prüfsumme berechnet. Wir übergeben die beiden Datenbytes und erhalten das CRC-Byte zurück.

```
    def createCRC(self,blobb): # 6.6 data = 2 bytes
        start=SGPCRCSTART
        for h in blobb:
            start ^= h
            for i in range(8):
                if start & 0x80:
                    start = (start << 1) ^ SGPPOLY
                else:
                    start <<= 1
        return start & 0xFF # CRC-Byte
```

Die Seriennummer des Chips holt die Routine **readSerial**(). Den Zahlenwert kriegen wir durch Schieben und Oderieren der drei empfangenen Bytes. Das MSB wird als erstes Byte empfangen, steht also in **words**[0] und so weiter. Das Ausgabeformat ist hexadezimal.

```
    def readSerial(self): # 6.5
        words=self.commandReadResult(SGP30.SGPCMD_getSerial)
        serial=(((words[0] << 16)| words[1]) << 16) | words[2]
        return hex(serial)
```

Die Routinen für das Senden von Kommentaren und den Empfang der Daten sind hier in der Methode **commandReadResult**() zusammengefasst. Den Grund dafür kennen wir schon, die unterschiedliche Länge der zu empfangenden Daten und die abweichenden Wartezeiten. Nun können neben dem Kommandowort aber weitere Daten an den SGP30 gesendet werden, statt welche zu empfangen. In diesem Fall nimmt der Parameter **param** diese als Bytesfolge entgegen.

```
    def commandReadResult(self,command,param=None): # 6.3
        cmd=command[:2]
        if param is not None:
            cmd += param
        length=command[2]
        delay=command[3]
```

Das Kommandowort befindet sich in den ersten beiden Bytes von **command**. Falls **param** nicht **None** enthält, sind darin Datenbytes, die an das Kommandowort anzuhängen sind. Wir filtern außerdem die Anzahl der zu empfangenden Bytes heraus und die Wartezeit. Dann schreiben wir das Kommando und warten **delay** ms. Wenn jetzt **length** = 0 ist, werden keine Bytes vom SGP30 erwartet und wir sind fertig.

Sonst holen wir die entsprechende Anzahl Bytes ab und erzeugen eine leere Liste für empfangenen Datenworte.

```
        self.i2c.writeto(self.hwadr, cmd)
        sleep_ms(delay)
        if length == 0:
            return None
        blobb=self.i2c.readfrom(self.hwadr,length)
        words=[]
        for i in range(length//3):
            if self.crcTest(blobb[3*i:3*(i+1)]):
                words.append((blobb[3*i] << 8) | blobb[3*i+1])
            else:
                raise SGPError(SGPCRCError)
        return words
```

Weil pro Datenwort jeweils zwei Datenbytes und ein CRC-Byte gelesen werden, kommen **length** // 3 Pakete an. Jedes Paket wird dem CRC-Test unterzogen. Der Index i läuft von 0 bis maximal 2 und adressiert die Slices **blobb**[0:3], **blobb**[3:6] und **blobb**[6:9]. Verlief der CRC-Test positiv, schrauben wir das Datenwort zusammen und fügen es in die Liste **words** ein, die wir zurückgeben.

Die folgenden sechs Methoden greifen nun einfach auf die Methode **commandReadResult**() und auf die Kommando-Bytesfolge zurück.

```
    def getFeatureSet(self):
        return self.commandReadResult(SGP30.SGPCMD_get_fea-
ture_set)[0]

    def measureTest(self):
        return self.commandReadResult(SGP30.SGPCMD_mea-
sure_test)[0]

    def iaqInit(self):
        self.commandReadResult(SGP30.SGPCMD_iaq_init)

    def measureIaq(self):
        return self.commandReadResult(SGP30.SGPCMD_mea-
sure_iaq)

    def getRawValues(self):
        return self.commandReadResult(SGP30.SGPCMD_mea-
sure_raw)

    def getTVOCInceptiveBaseline(self):
        return self.commandReadRe-
sult(SGP30.SGPCMD_get_tvoc_inceptive_baseline)
```

Für einen Reset des SGP30 wird die General Call Adresse 0 zusammen mit dem Byte 0x06 gesendet. Alle Chips, welche dieses Feature unterstützen werden dadurch zurückgesetzt. Dazu zählt auch der SHT30.

```
    def softReset(self):
        self.i2c.writeto(0, b'\x06')
```

Der SGP30 braucht für seine korrekte Funktion den Wert der absoluten Luftfeuchte in Gramm Wasser pro Liter Luft. Mit der Methode **setAbsoluteHumidity**() wird der Wert im 8.8 Fixpunktformat an den SGP30 geschickt. Über das Format sprechen wir später noch genauer. Der Wert besteht aus einem Datenwort (H), das wir durch Anwendung der Methode **pack**() in zwei Bytes zerlegen, Reihenfolge Big Endian (>), also MSB first. Die Methode **createCRC**() liefert ein (1) Byte, das in ein Bytes-Objekt umgewandelt werden muss, bevor es an das Bytesobjekt in **blobb** angehängt werden kann. **big** steht für Big Endian, was in diesem Zusammenhang zwar ohne Bedeutung ist, aber wegen der Syntax zwingend angegeben werden muss. Die drei Bytes schicken wir zum SGP30.

```
    def setAbsoluteHumidity(h): # abshum type 8.8
        blobb = pack(">H",h)
        blobb += self.createCRC(blobb).to_bytes(1,"big")
        self.commandReadResult(SGP30.SGPCMD_set_absolute_humi-
dity,blobb)
```

Die Formel für die Berechnung der absoluten Feuchte ist im Datenblatt vorgegeben. Mit Hilfe des SHT30 können die Eingabewerte für Temperatur und rel. Feuchte bestimmt werden.

```
    def rel2absHumidity(self, t, rh): # 6.3 - p 10

self.abshum=216.7*((rh/100*6.112*exp(17.62*t/(243.12+t)))
(273.15+t))
        abshum88=(int(self.abshum) << 8 ) | int((self.abshum %
1) * 256)
        return self.abshum88
```

In **abshum** wird der dezimale Wert abgelegt, so wie er auf einem Taschenrechner herauskommen würde. Der SGP30 will aber ein anderes Format, Festkomma 8.8. Das heißt, die 8 Bits des MSB beinhalten den ganzzahligen Anteil und die zweiten 8 Bits den Zähler eines Bruches, mit dem Nenner 256 und dem Wert des Nachkommanteils.

Sei **abshum** = 13,355, dann ist 13 = 0x0D der ganzzahlige Anteil. Den Nachkommateil erhalten wir durch **abshum** % 1, also als Teilungsrest von 13,355 : 1 = 0,355 oder 355 Tausendstel. Den Zähler des Bruches mit dem Nenner 256 erhalten wir durch die Multiplikation von 0,355 mit 256, das ist 90,88. 90/256 ist also ungefähr 0,355 und 90 = 0x5A ist das zweite Byte. Zusammengesetzt kommt 0x0d5A heraus und wird zurückgegeben. Einen derartigen Wert will die Methode **setAbsoluteHumidity**() als Argument haben.

Den dezimalen Wert der absoluten Feuchte können wir mit **absHumidity** abrufen. Der Decorator @property erlaubt den Abruf wie bei einer Variablen, die Klammern fallen beim Aufruf weg.

```
    @property
    def absHumidity(self):
        return self.abshum
```

>>> sgp.absHumidity
13.355

Ähnlich funktioniert die Abfrage der Messwerte für CO_2equ in ppm (Parts per Million) und tvoc in ppb (Parts per Billion (Milliarde)). Diese Werte liefert die Methode **measureIaq**() in einer Liste, auf die wir durch die Indices 0 und 1 zugreifen.

```
    @property
    def co2e(self):
        return self.measureIaq()[0]

    @property
    def tvoc(self):
        return self.measureIaq()[1]
```

Fehlt noch eine Methode, um den dezimalen Wert der Feuchte in das 8.8-Format umzuwandeln.

```
    def abshum2fix88(self, h=None): # dec zu 8.8 fixpoint
        if h is None:
            h=self.abshum
        else:
            self.abshum=h
        abshum88=(int(h) << 8 ) | int((h % 1) * 256)
        return abshum88
```

Wird kein Argument beim Aufruf angegeben, dann verwenden wir den Wert in **abshum**. Haben wir händisch einen Wert berechnet, können wir ihn übergeben und an **abshum** weiterreichen. Wir berechnen die beiden Fixkomma-Bytes und geben das 8.8-Datenwort zurück.

SGP30-Testlauf

Mit dem Programm **sgp30_test.py** können wir das neue Modul testen. Importgeschäft, Bus-Objekt definieren und an die Konstruktoren der Klasse SGP30 und OLED übergeben.

```
from machine import Pin,SoftI2C
from sgp30_s import SGP30
from time import sleep_ms
from oled import OLED
from sys import exit

i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
sgp=SGP30(i2c)
d=OLED(i2c,heightw=32)

af88=sgp.abshum2fix88()
sgp.setAbsoluteHumidity(af88)
```

```
while 1:
    print("\nCO2eq; TVOC",sgp.measureIaq())
    d.clearAll(False)
    d.writeAt("AIR-QUALITY",2,0,False)
    d.writeAt("CO2e {:>5} ppm".format(sgp.co2e),0,1,False)
    d.writeAt("TVOC {:>5} ppb".format(sgp.tvoc),0,2)
    sleep_ms(300)
```

Wir verwenden den im Konstruktor vorgelegten Wert für die absolute Feuchte, stellen ihn ins 8.8-Format um und senden ihn an den SGP30. In der while-Schleife fragen wir die Messwerte ab und geben sie in REPL und im Display aus.

Gesamttest aller Sensoren

Das Testprogramm für alle drei Sensoren sieht ähnlich aus. Natürlich müssen alle drei Klassen importiert und Objekte damit erzeugt werden.

```
from machine import Pin,SoftI2C
from sgp30_s import SGP30
from sht30_s import SHT3X
from bh1750 import BH1750
from oled import OLED
from time import sleep_ms, sleep
from sys import exit
from timeout import TimeOut

i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
d=OLED(i2c,heightw=32)

sht=SHT3X(i2c)
sgp=SGP30(i2c)
bh=BH1750(i2c)
```

Wir definieren zwei Pin-Objekte für das Relais und die grüne LED.

```
rel=Pin(15,Pin.OUT,value=0)
green=Pin(13,Pin.OUT,value=0)
```

Ebenso die beiden Variablen für das Monoflop, den Verzögerungsschalter. Zu der Methode **TimeOut**() kommen wir noch weiter unten.

```
triggered=False
abgelaufen = TimeOut(0)
```

Schließlich initialisieren wir noch den Arbeitsmodus des SHT30 und setzen die absolute Feuchte wie oben.

```
mps=3
sht.startPeriodic(mps,0)
sleep_ms(16)
```

```
af88=sgp.abshum2fix88()
sgp.setAbsoluteHumidity(af88)
```

In der Schleife lesen wir die Werte ein und geben sie im Display aus.

```
while 1:
    sht.readPeriodic()
    temp,hum=sht.calcValues()
    co2e,tvoc=sgp.measureIaq()
    lum=bh.luminanz()
    d.clearAll(False)
    d.writeAt("Lumi: {:>5} lux".format(lum),0,0,False)
    d.writeAt("t: {:0.1f};h: {:0.1f}".format(temp,hum),0,1,
              False)
    d.writeAt("CO2 {:>3}".format(co2e),0,2,False)
    d.writeAt("TVOC{:>3}".format(tvoc),8,2)
```

Ist die Temperatur höher als 25 °C, schalten wir das Relais ein. Damit es nicht ständig flattert, wenn die Temperatur leicht schwankt, wählen wir für das Ausschalten einen etwas niedrigeren Wert und erzeugen damit eine sogenannte Hysterese, eine Umschaltverzögerung.

```
    if temp >= 25:
        rel(1)
    elif temp <=24.5:
        rel(0)
```

Fällt die Beleuchtungsstärke unter 150 Lux, soll die grüne LED einschaltet werden und 5 Sekunden an bleiben. Das regeln wir mit dem nicht blockierenden Timer über die Methode **TimeOut**(). Sie gibt eine Referenz auf die darin gekapselte Funktion **compare**(), eine Closure, zurück, die wir dem Bezeichner **abgelaufen** zuweisen. Der Aufruf von **abgelaufen**() liefert **True** zurück, wenn die 5 Sekunden um sind. In der Zwischenzeit wird die while-Schleife aber weiterhin ausgeführt. **triggered** = True zeigt an, dass die LED bereits eingeschaltet ist. Der Block darf nur ausgeführt werden, wenn die LED noch nicht leuchtet.

```
    if lum < 150 and not triggered:
        green(1)
        abgelaufen = TimeOut(5)
        triggered = True
```

Ist der Timer abgelaufen und die LED an, dann muss sie ausgeschaltet werden. **triggered** setzen wir wieder auf False.

```
    if abgelaufen() and triggered:
        green(0)
        triggered = False
    sleep_ms(SHT3X.QueryDelay[mps])
```

Wir warten noch, bis der Messzyklus des SHT30 beendet ist und starten in die nächste Runde.

LED anzünden und ausblasen

Im grauen vordigitalen Zeitalter hatte ich als Gag eine Schaltung mit einem LDR, einem Transistor und einem Glühlämpchen aufgebaut.

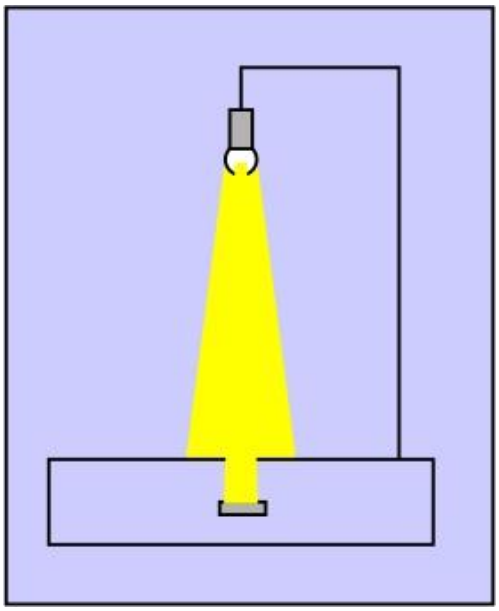

Abbildung 5: Lampen-Gadget

Wenn man zwischen Glühbirne und LDR ein brennendes Streichholz hält, geht die Lame an. Bläst man die Lampe an, so dass sie zur Seite schwingt, geht die Lampe aus.

Mit dem AZ-Oneboard und dem BH1750 kann man die Anordnung als digitale Variante aufbauen. Die Glühlampe wird durch eine weiße LED ersetzt, und der LDR durch den BH1750. Der ESP8266 kümmert sich um die Umsetzung des sehr einfachen Steuerprogramms.

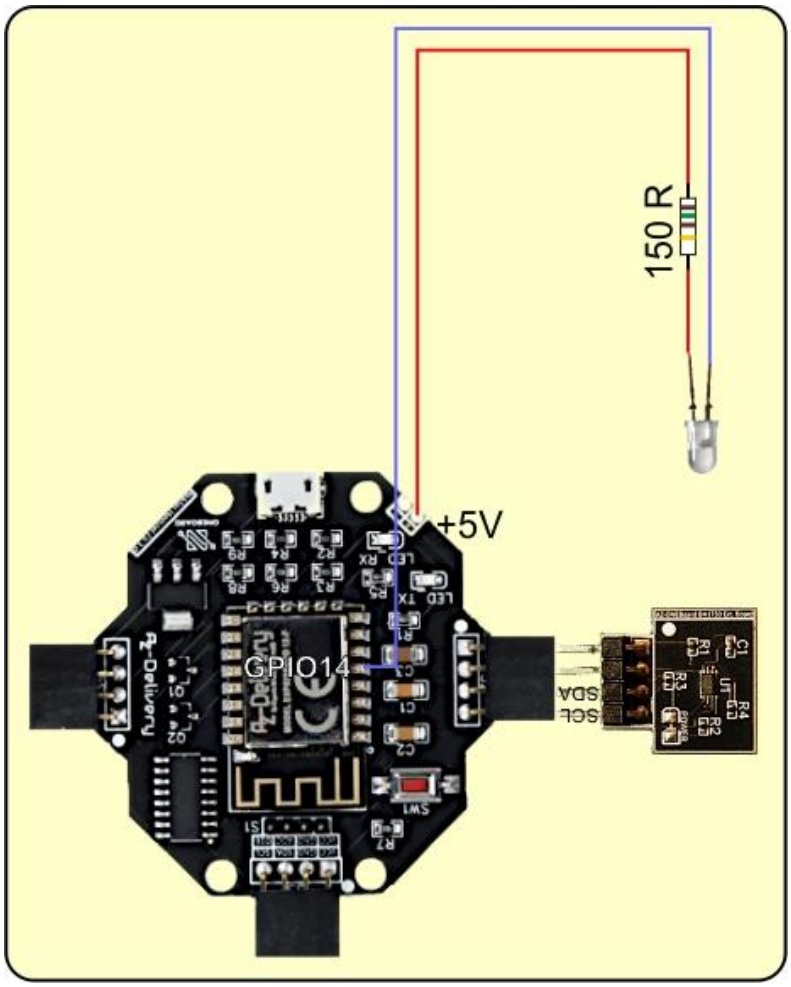

Abbildung 6: Lampen-Gadget mit AZ-Oneboard, BH1750 und LED

Vorab messen wir ein paar Beleuchtungsstärken. Aus Sicherheitsgründen verwenden wir kein offenes Feuer, sondern eine Taschenlampe.

Normales Umgebungslicht am BH1750: 467
Bei eingeschalteter LED in 5cm Entfernung: 3853
Taschenlampe in 20cm Entfernung: 8310

An diesen Werten orientieren wir uns im Programm bei der Angabe der Grenzwerte.

```
from machine import Pin,SoftI2C
from bh1750 import BH1750
from time import sleep_ms

i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
bh=BH1750(i2c)

white=Pin(14,Pin.OUT,value=1)

while 1:
    lum = bh.luminanz()

    if lum > 3000:
```

```
        white(0)

    if lum < 1000:
        white(1)

    sleep_ms(120)
```

Nach den Importen instanziieren wir wieder ein I2C-Bus-Objekt und weisen es dem BH1750-Konstruktor zu. Die Kathode der LED schließen wir an GPIO14 über einen Widerstand an, die Anode legen wir an den +5V-Anschluss des AZ-Oneboards. Die LED ist aus, wenn der Ausgang auf logisch 1 liegt. Die Spannung an GPIO14 beträgt jetzt 3,1V und ist daher absolut im grünen Bereich. Wenn wir den Ausgang auf logisch 0 legen, leuchtet die LED. Es fließt ein Strom von 10,8mA, auch in Ordnung!

Der BH1750 arbeitet im continuous highresolution Mode. Zwischen zwei Messungen braucht er daher 120ms Pause. Wir holen einen Wert ab und prüfen auf den Bereich. Der Wer von 3000 sorgt dafür, dass die LED beim kurzen Beleuchten des BH1750 mit der Taschenlampe an geht und an bleibt, um vor allem weiterhin mit 3800 Lux den Sensor zu beleuchten.

Wird die LED aus ihrer Richtung zum BH1750 weit genug ausgelenkt, zum Beispiel durch Anblasen, dann fällt der Belichtungswert unter 1000 und die LED wird ausgeschaltet.