

AZ-ONEBOARD im erweiterten Test

Diesen Beitrag gibt es auch als [PDF-Dokument](#).

Vor vier Jahren hatte ich mit einem Environment-System namens Envy zu tun, mit einem ESP8266-12F, einem MQ2 als Gassensor und einem SHT30 zur Erfassung von Temperatur und relativer Luftfeuchte. Die Heizstromstärke des Sensors betrug ca. 170mA bei 5V. Das ergibt eine Leistung von 850 mW! Die Energieversorgung erfolgte über USB, aber für die Programmierung brauchte man zusätzlich einen USB zu TTL-Adapter.

Vor ein paar Tagen erfuhr ich von einer Nachfolgeversion des Boards. Das AZ-Oneboard kommt wieder mit einem ESP8266 12F und besitzt jetzt einen echten USB-Zugang mit einem CH340C. Außerdem hat das Board einen optionalen Anschluss für 5V, an dem auch die Busspannung anliegt. In dem Kit sind neben der Hauptplatine noch drei Break Out Boards enthalten: ein SHT30, ein BH1750 (Luxmeter) und ein SGP30, der CO₂, sowie den TVOC-Anteil (Total Volatile Organic Compounds), also flüchtige organische Verbindungen in der Raumluft, misst. Der im SGP30 enthaltene Sensor ist um Galaxien kleiner als der MQ2 und arbeitet bei 1,8V mit 48mA, das sind grade mal 86mW. Das CO₂-Equivalent wird in ppm (Parts per million) und TVOC in ppb (Parts per Billion) gemessen.

Angesteckt werden die Sensorboards, deren Chips alle über den I2C-Bus angesteuert werden, über jeweils eine der Buchsenleisten auf dem AZ-Oneboard. An diese Buchsenleisten kann man natürlich auch andere I2C-Bus-Bausteine, wie zum Beispiel ein OLED-Display, anschließen. Allerdings muss man dafür Jumperkabel verwenden, um die Pinzuordnung anzupassen.

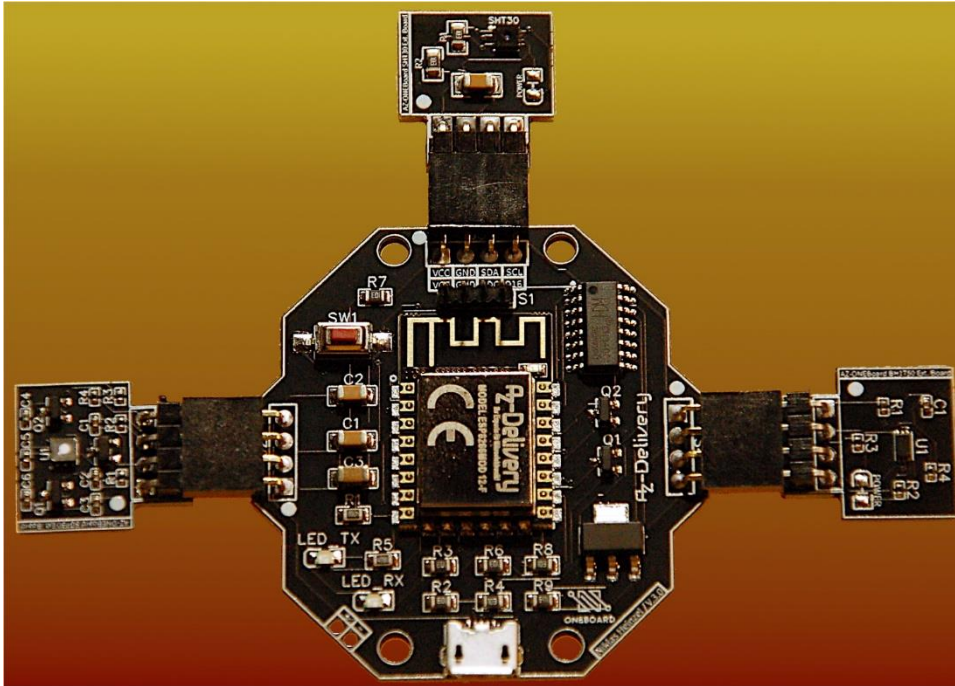


Abbildung 1: envy3_0

Heute werden wir uns den Lichtsensor BH1750 vornehmen und eine Schwachstelle des AZ-Oneboards beseitigen. Worum es sich dabei handelt, was man tun kann und wie der BH1750 über MicroPython angesprochen wird, das erfahren Sie in dieser Folge aus der Reihe

MicroPython auf dem ESP32, ESP8266 und Raspberry Pi Pico

heute

Ein Luxmeter mit dem ESP8266

Das Luxmeter habe ich deshalb als ersten Beitrag ausgewählt, weil das zugehörige MicroPython-Modul nicht so umfangreich ausfällt. Dadurch wird es für Einsteiger einfacher und es bleibt Raum für die Erweiterung des Boards sowie die Integration eines OLED-Displays. Was wird gebraucht?

Die Hardware

Das sonst übliche Board mit dem Controller, hier ein ESP8266, wird durch das AZ-Oneboard ersetzt.

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
optional	Luxmeter zum Kalibrieren

Der Aufbau ist so einfach, dass ein Schaltplan überflüssig ist. Wir stecken das Break Out Board mit dem BH1750 einfach auf eine der Buchsenleisten der Controller-Platine, so wie in Abbildung 2 dargestellt.

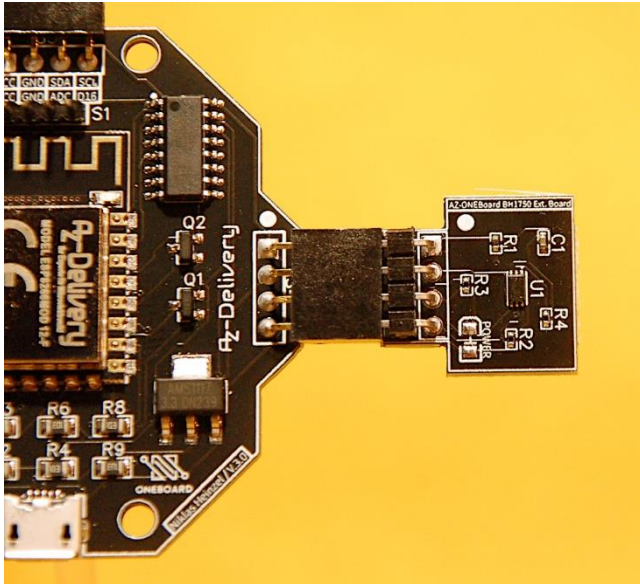


Abbildung 2: AZ-Oneboard mit BH1750

Die Software

Fürs Flashen und die Programmierung des ESP32:

[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

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

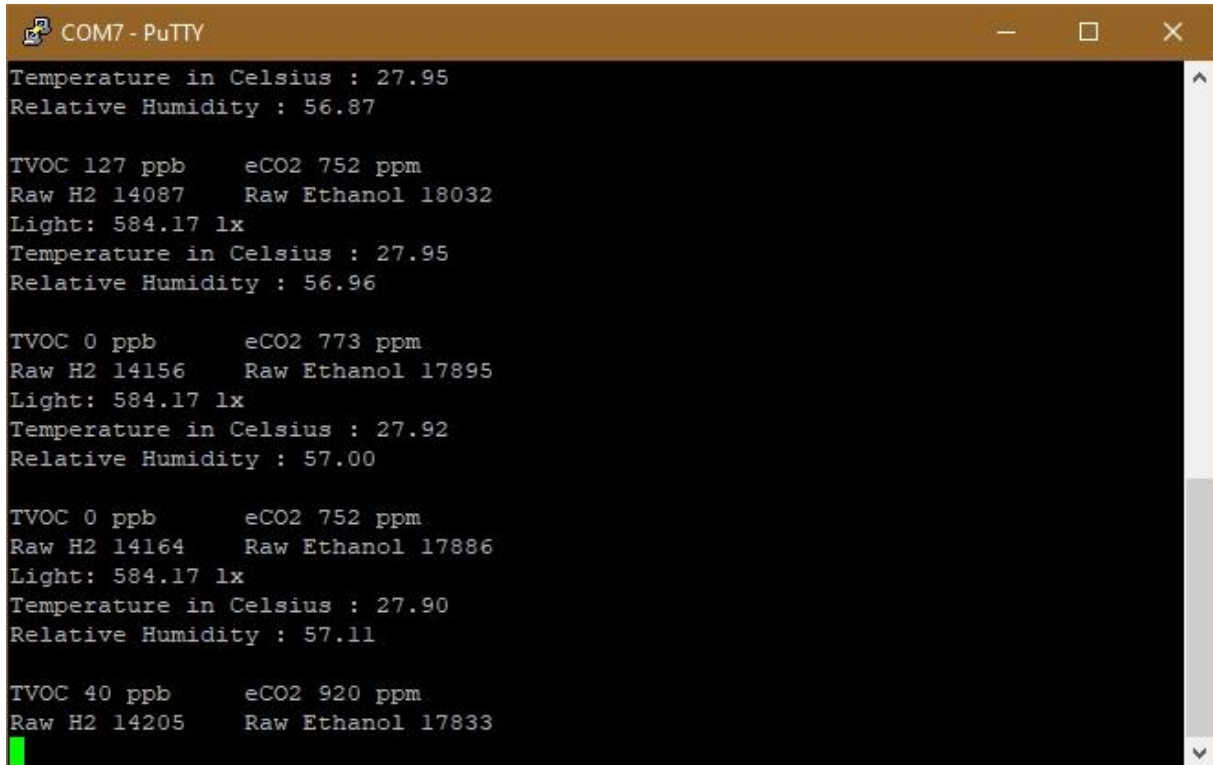
Manuell werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5 gestartet. Das geht schneller als der Mausklick auf den Startbutton, oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

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.

Vorbereitung des AZ-Oneboards

Das Board wird mit einem Sketch ausgeliefert, der die Funktion der Sensoren überprüft und die Messwerte in einem Terminalprogramm wie [Putty](#) ausgibt. Das funktioniert aber nur, wenn alle drei Break Out Boards angesteckt sind.



```
COM7 - PuTTY
Temperature in Celsius : 27.95
Relative Humidity : 56.87

TVOC 127 ppb      eCO2 752 ppm
Raw H2 14087      Raw Ethanol 18032
Light: 584.17 lx
Temperature in Celsius : 27.95
Relative Humidity : 56.96

TVOC 0 ppb       eCO2 773 ppm
Raw H2 14156      Raw Ethanol 17895
Light: 584.17 lx
Temperature in Celsius : 27.92
Relative Humidity : 57.00

TVOC 0 ppb       eCO2 752 ppm
Raw H2 14164      Raw Ethanol 17886
Light: 584.17 lx
Temperature in Celsius : 27.90
Relative Humidity : 57.11

TVOC 40 ppb      eCO2 920 ppm
Raw H2 14205      Raw Ethanol 17833
```

Abbildung 3: Ausgabe mit der Demosoftware

Auch mit unserem Entwicklungstool Thonny funktioniert das, wenn wir zuerst das AZ-Oneboard am Bus anschließen und danach Thonny starten.

Aber wir wollen ja 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 in [REPL](#), dem Terminal von Thonny, etwa so:

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

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

```
>>>
```

Dann ist es auch schon Zeit für die ersten Gehversuche. Eingaben in [REPL](#) werden in diesem Skript **fett** formatiert, die Antworten vom ESP8266 *kursiv*. Wir wollen sehen, ob der ESP8266 Kontakt mit dem Sensorboard aufnehmen kann. Dazu müssen wir die Klassen **Pin** und **SoftI2C** importieren. Dann instanziiieren wir ein Bus-Objekt und verwenden es für einen Rundruf, wer denn grade so da ist.

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

In der [Liste](#), die die Funktion `scan()` zurückgibt, findet sich die Hardware- oder Geräteadresse des BH1750-Chips, 35 dezimal oder 0x23 hexadezimal. Die Verbindung klappt also.

Schließen wir nun an einer anderen Buchsenleiste am AZ-Oneboard unser OLED-Display mit vier Jumperkabeln an. Die Kabel sind nötig, weil die Pins am Display eine andere Reihenfolge haben wie am AZ-Oneboard. Die Zuordnung ist folgende:

Display : AZ-Oneboard
Vcc : +
GND: -
scl: scl
sda: sda

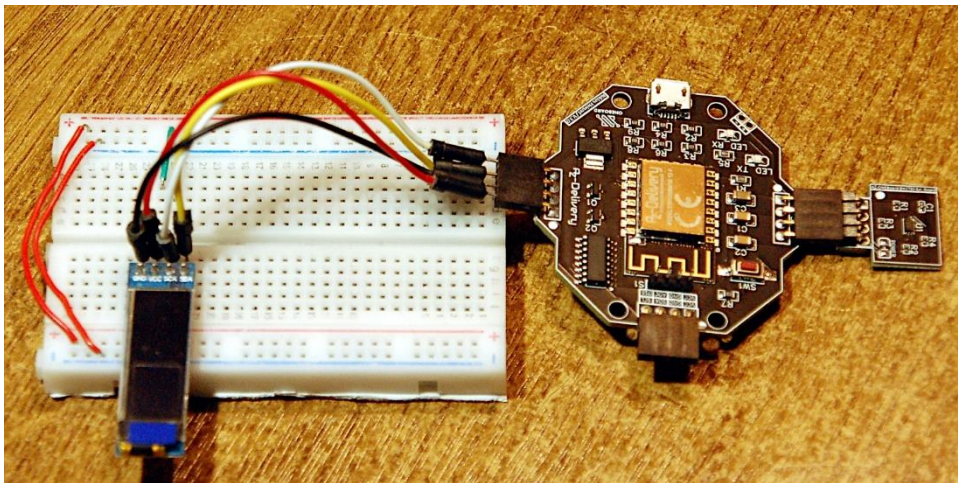


Abbildung 4: OLED-angeschlossen

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

Wir sehen, dass sich ein zweites Gerät mit der Adresse 60 = 0x3C gemeldet hat, das Display. Das testen wir auch gleich mal. Aber damit die nächsten Befehle auch funktionieren, müssen wir zuerst die Dateien [oled.py](#) und [ssd1306.py](#) in den Flash des ESP8266 hochladen. Wir importieren die Klasse OLED und erzeugen damit ein Display-Objekt.

```
>>> from oled import OLED
>>> d=OLED(i2c, heightw=32)
this is the constructor of OLED class
Size:128x32
>>> d.writeAt("Luxmeter 1.0",2,0)
14
```

Der Text erscheint mittig in der ersten Displayzeile, weil wir ab Spalte 2 geschrieben haben und der Text aus 14 Zeichen besteht. Eine Zeile fasst maximal 16 Zeichen.

Als Nächstes machen wir uns an das Treibermodul für den BH1750.

Die Klasse BH1750

Für den Zugriff auf die Innerei des BH1750 müssen wir zwei Sachen erledigen. Eine Geschichte ist die Konversation mit dem Chip über den I2C-Bus. Die andere, umfangreichere Sache ist die Steuerung von Abläufen auf dem Chip, entweder durch Zugriff auf die internen Register (Speicherstellen) oder, wie hier, das Senden von Kommandos und das Abholen der Daten. Beide Stufen sind in der Regel von Chip zu Chip unterschiedlich gelöst. Wir beginnen wie immer mit dem Importgeschäft.

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

class BH1750:
    HWADR = const(0x23) # ADDR=LOW

    PWRdown = const(0x00)
    PWRon = const(0x01)
    Reset = const(0x07)
    ContHres = const(0x10) # 120ms; 0,5 lux
    ContHres2= const(0x11) # 120ms; 1,0 lux
    ContLres = const(0x13) # 16ms; 4,0 lx
    OnceHres = const(0x20)
    OnceHres2= const(0x21)
    OnceLres = const(0x23)
    ChangeMTH= const(0x40) # or with MT.7:5 as 2:0
    ChangeMTL= const(0x60) # or with MT.4:0 as 4:0
    Wait=[180,180,0,24]
```

Der Deklaration der Klasse BH1750 folgt die Festlegung einiger Konstanten, voraus die Geräteadresse und danach die Codes für verschiedene Kommandos. Alle Daten können wir aus dem [Datenblatt des BH1750](#) entnehmen.

Instruction	Opecode	Comments
Power Down	0000_0000	No active state.
Power On	0000_0001	Waiting for measurement command.
Reset	0000_0111	Reset Data register value. Reset command is not acceptable in Power Down mode.
Continuously H-Resolution Mode	0001_0000	Start measurement at 1lx resolution. Measurement Time is typically 120ms.
Continuously H-Resolution Mode2	0001_0001	Start measurement at 0.5lx resolution. Measurement Time is typically 120ms.
Continuously L-Resolution Mode	0001_0011	Start measurement at 4lx resolution. Measurement Time is typically 16ms.
One Time H-Resolution Mode	0010_0000	Start measurement at 1lx resolution. Measurement Time is typically 120ms. It is automatically set to Power Down mode after measurement.
One Time H-Resolution Mode2	0010_0001	Start measurement at 0.5lx resolution. Measurement Time is typically 120ms. It is automatically set to Power Down mode after measurement.
One Time L-Resolution Mode	0010_0011	Start measurement at 4lx resolution. Measurement Time is typically 16ms. It is automatically set to Power Down mode after measurement.
Change Measurement time (High bit)	01000_MT[7,6,5]	Change measurement time. ※ Please refer "adjust measurement result for influence of optical window."
Change Masurement time (Low bit)	011_MT[4,3,2,1,0]	Change measurement time. ※ Please refer "adjust measurement result for influence of optical window."

※ Don't input the other opecode.

Abbildung 5: Kommando-Übersicht

Die Methode `__init__()` stellt den Konstruktor `BH1750()` der Klasse `BH1750` dar. Als Argumente werden beim Aufruf im Positions-Parameter `i2c` das I2C-Objekt, und optional in den Schlüsselwortparametern `adr`, `mod` und `fak` weitere Daten an die Routine übergeben. Werden die Parameter beim Aufruf nicht aufgeführt, dann nehmen sie die Defaultwerte an.

```
def __init__(self, i2c,
              adr=HWADR,
              mod=ContHres,
              fak=0.81):
    self.i2c = i2c
    self.hwadr = adr
    self.powerOn()
    self.mode(mod)
    self._latency=69
    self._faktor=1
    self.faktor=fak
    print("BH1750 Luxmeter is @",hex(adr))
```

Damit die Werte auch in anderen Methoden der Klasse verfügbar sind, weisen wir sie Instanzattributen zu. Alle Objekte, die zu einer Instanz gehören, werden durch ein vorangestelltes `self` definiert und referenziert. Die Instanzattribute sind also `i2c`, `hwadr`, `latency` und `faktor`. Mit `powerOn()` und `mode()` werden weiter unten definierte Methoden aufgerufen, referenziert. Der `print`-Befehl meldet uns, dass das Objekt erstellt wurde und verrät uns die hexadezimale Darstellung der Hardwareadresse des BH1750.

Das Abholen von Messwerten ist beim BH1750 besonders einfach, weil der Chip stets Datenworte, die Kombination aus zwei Bytes, sendet. Wir brauchen also nur eine Methode zu definieren, die zwei Bytes vom I2C-Bus einliest. Das MSB, das höherwertige Byte, kommt zuerst.

```
def readBytes(self):
    buf=bytearray(2)
    self.i2c.readfrom_into(self.hwadr,buf)
    return buf
```

Der ESP8266 empfängt die beiden Bytes und verpackt sie für die weitere Verarbeitung in ein Objekt, das dem Bufferprotokoll folgt. Das kann ein `bytes`-Objekt sein, oder wie hier, das `bytearray buf`. Die Funktion `i2c.readfrom_into()` sendet zuerst die Hardware-Adresse und schaufelt dann die empfangenen Bytes in den Buffer. Wie das bei der Übertragung auf den Leitungen aussieht, das zeigt der Plot in Abbildung 6, den ich mit meinem Logic Analyzer und dem Programm Logic2 von Saleae aufgenommen habe.

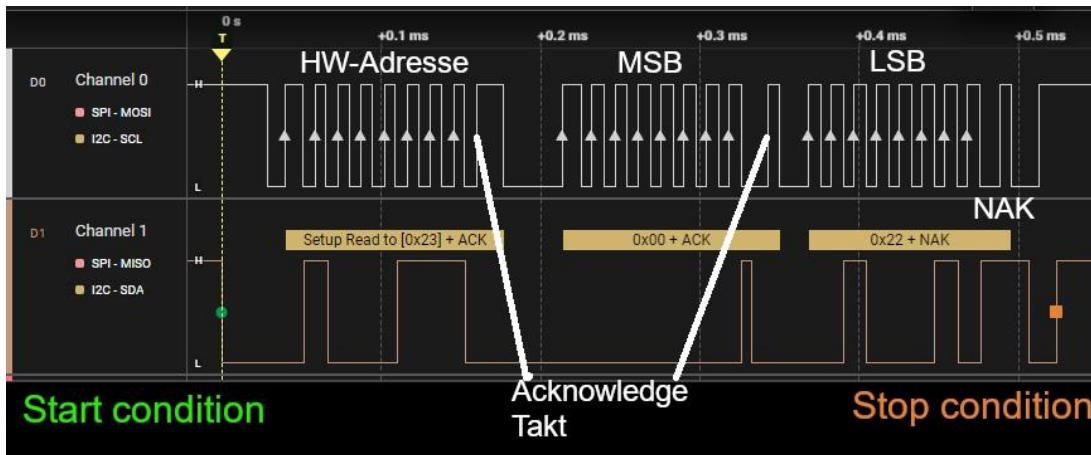


Abbildung 6: Daten vom BH1750 einlesen

Die Übertragung beginnt mit einer Start Condition, scl = 1, sda geht auf 0. Die Hardware-Adresse wird immer mit 7 Bits angegeben, das achte Bit wird rechts an die Adresse angehängt, das Write-Read-Bit. Hier ist es eine 1, und das sagt dem BH1750, dass er Daten senden soll. Beim Senden eines Befehls an den Chip ist dieses Bit 0. Aus 0x23 wird also 0x46.

Mit jeder steigenden Taktflanke sampelt der Controller die sda-Leitung, das heißt, er schaut nach, welcher Pegel auf der Leitung anliegt. Nach acht Taktflanken zieht der BH1750 die sda-Leitung auf 0. Mit diesem Acknowledge-Bit zeigt er an, dass er die Adresse als seine erkannt hat. Der ESP8266 weiß jetzt, dass er sich auf den Empfang der Daten vorbereiten muss. Mit jeder fallenden Taktflanke legt nun der BH1750 ein Datenbit auf die sda-Leitung, das der Controller mit der steigenden Flanke sampelt. Hat der Empfang fehlerfrei geklappt, sendet jetzt der ESP8266 das ACK-Bit. Der BH1750 bringt daraufhin das zweite Byte auf den weg, das der ESP8266 mit einem NACK-Bit, not acknowledge, quittiert. Mit einer Stop Condition beendet der Controller die Übertragung.

Mit der Methode **command()** senden wir einen Befehl an den BH1750, den wir in **cmd** als Ganzzahl übergeben. Die Werte haben wir eben als Konstanten abgelegt.

```
def command(self, cmd) :
    buf=bytearray(1)
    buf[0]=cmd
    self.i2c.writeto(self.hwadr,buf)
```

Jetzt kann MicroPython aber diesen Wert, der eigentlich als Integer-Wert eine 32-Bitzahl ist, nicht senden. Erstens wüsste der BH1750 mit den vier Bytes nichts anzufangen und zweitens weigert sich MicroPython, Objekte wie Ganzzahlen Fließkommazahlen, Listen und andere höhere Objekte über den Bus zu senden. Die Klasse SoftI2C müsste für diesen Zweck Routinen für die automatische Typumwandlung in Buffer-Protokoll konforme Objekte zur Verfügung stellen. Also stopfen wir selber das Kommando-Byte in die Zelle eines Bytearrays, damit kann MicroPython etwas anfangen.

```
>>> i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
```

Das senden einer Ganzzahl schlägt fehl.

```
>>> i2c.writeto(0x23,0x01)
```

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

TypeError: object with buffer protocol required

Ein bytearray wird akzeptiert.

```
>>> buf=bytearray(1)
```

```
>>> buf[0]=0x01
```

```
>>> i2c.writeto(0x23,buf)
```

```
1
```

Ebenso ein bytes-Objekt.

```
>>> i2c.writeto(0x23,b'\x01')
```

```
1
```

Oder ein String

```
>>> i2c.writeto(0x23,"\x01")
```

```
1
```

Das hier Gesagte gilt in derselben Weise für den SPI-Bus, die serielle UART-Schnittstelle und das WLAN-Interface.

Was wir eben von Hand an den BH1750 gesendet haben, ist der Power-On-Befehl. Die Methode **powerOn()** tut das Gleiche, indem das **PWRon**-Kommando an die Methode **command()** übergeben wird. Analog funktioniert **powerDown()**.

```
def powerOn(self):  
    self.command(PWRon)
```

```
def powerDown(self):  
    self.command(PWRdown)
```

Einen Überblick über die Zustände des BH1750 und die Übergänge vom einen in den anderen zeigt Abbildung 7.

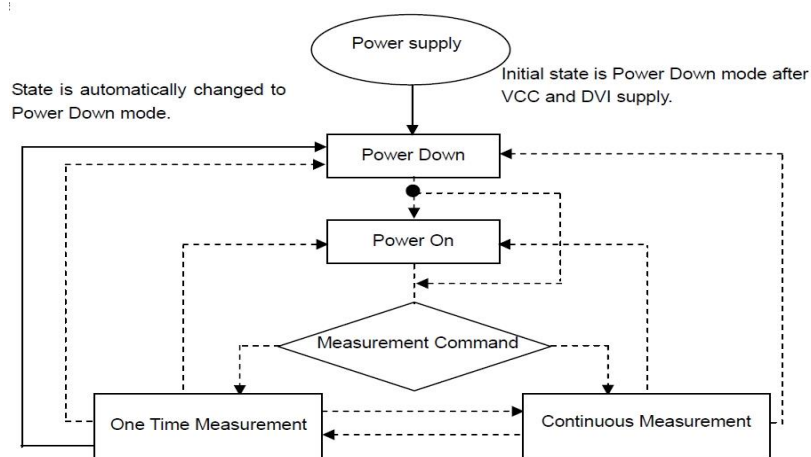


Abbildung 7: Die Zustände des BH1750

Der BH1750 kennt verschiedene Messmethoden. Grundsätzlich gibt es einen Einzelschuss-Modus und einen Dauerlauf-Modus. Jeder Modus verfügt über drei verschiedene Stufen der Auflösung Lowres, Highres und Highres2, entsprechend 4 Lux, 1 Lux und 0,5 Lux. Mit der Methode **mode()** können wir den entsprechenden Modus einstellen. Um nichts falsch zu machen, prüfen wir zuerst, ob der Übergebene Wert in der Liste der zulässigen Kommandos enthalten ist. Wenn dem nicht so ist, wird ein `AssertionError` gemeldet.

```
def mode(self, modus= None):
    if modus is not None:
        assert modus in [
            ContHres,
            ContHres2,
            ContLres,
            OnceHres,
            OnceHres2,
            OnceLres]
        self.modus=modus
        self.command(modus)
    else:
        return self.modus
```

Weil wir den aktuellen Modus vom BH1750 nicht abfragen können, merken wir uns den Wert in dem Instanzattribut **modus** und senden daraufhin den Befehl. Wird der Parameter beim Aufruf weggelassen, dann liefert die Methode den Wert des Instanzattributs **modus** zurück.

```
>>> bh.mode(0x81)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 57, in mode
AssertionError:
```

Bevor das Reset-Kommando gesendet werden kann, muss sich der BH1750 im Power On-Zustand befinden. Der Hinweis steht unten auf Seite 5 des [Datenblatts](#). Durch das Kommando werden lediglich die Register mit dem letzten Messwert auf 0 gesetzt.

```
def reset(self): # p 5
    self.powerOn()
    self.command(Reset)
```

Mit **start()** stoßen wir eine Einzelmessung an. Den gewünschten Modus können wir im Parameter **mode** übergeben. Vorgelegt ist der Defaultwert `None`. Wird beim Aufruf kein Argument für **modus** übergeben, dann wird der Wert des Instanzattributs **modus** verwendet, um die Messung zu starten. Die Messdauer wird der [Liste Wait](#) entnommen. Den Index auf den Listen-Wert, erhalten wir aus den beiden niederwertigsten Bits des Kommandobytes, also 0, 1 oder 3. Der Index 2 kann nicht auftreten. Damit die Zuordnung der Listenelemente auf die Modi stimmt, brauchen wir ein 3. Element, das ich einfach auf 0 gesetzt habe.

```

def start(self, modus=None):
    if modus is not None:
        self.mode(modus)
    index= self.modus & 0x03
    if self.modus & 0x20:
        self.command(self.modus)
        wartezeit=max(BH1750.Wait[index],
                      BH1750.Wait[index] * self.faktor)
        sleep_ms(wartezeit)
    else:
        raise ValueError("Falscher Modus-Wert")

```

Das Setzen eines Einzelschuss-Modus kann nur erfolgen, wenn Bit5 im Kommando-Byte gesetzt ist. Die if-Abfrage stellt das sicher. Ist es nicht gesetzt, werfen wir eine ValueError-Exception. Das Hauptprogramm muss diese abfangen, sonst wird das Programm abgebrochen.

Wenn alles gepasst hat, wird der Modus gesetzt und damit die Einzelmessung getriggert. Die minimale Wartezeit bis zur Bereitstellung des Ergebnisses liefert die Liste **Wait**, 180 ms für Highres-Modi und 24 ms für den Lowres-Modus.

Nun kann man aber die Messzeit in gewissen Grenzen verstellen. Das wird nötig, wenn der Sensor mit einem Schutzglas abgedeckt wird. Jedes Glas absorbiert trotz der Durchsichtigkeit einen Teil der Lichtenergie, was durch eine Verlängerung der Messzeit ausgeglichen werden muss. Dieser Verlängerungsfaktor muss beim Berechnen der Wartezeit natürlich berücksichtigt werden. Wir nehmen sicherheitshalber also immer den größeren der beiden Werte. **sleep_ms()** schickt den ESP8266 dann kurz ins Traumland.

Egal ob Einzelschuss oder automatische Dauermessung, die Lux-Werte werden durch die Methode Luminanz abgeholt und aufbereitet.

```

def luminanz(self):
    hb, lb=self.readBytes()
    teiler= 2 if self.modus & 0x03 == 1 else 1
    lux=int(((hb << 8) | lb) / (1.2 * (69/self._latency))
/ teiler) # p11
    return lux

```

Das Bytearray, das **readBytes()** liefert, entpacken wir gleich in die lokalen Variablen **hb** und **lb**. Wenn der Modus Highres2 aktiv war, wird ein zusätzlicher Teiler 2 nötig, so sagt es das Datenblatt auf Seite 11 unten. Wir lösen das durch eine conditional Expression. Die Formeln zur Berechnung der Beleuchtungsstärke auf Seite 11 sind mathematisch gesehen falsch, es fehlen Klammern.

Illuminance per 1 count (lx / count) = 1 / 1.2 *(69 / X) muss heißen:

Illuminance per 1 count (lx / count) = 1 / (1.2 *(69 / X)), denn mit einer längeren Belichtungszeit, muss auch ein höherer Wert für die Luminanz herauskommen.

Wir schieben das MSB um 8 Positionen nach links und [oderieren](#) mit dem LSB. So entsteht aus zwei Datenbytes ein 16-Bit-Datenwort. Das ist durch das Produkt aus

1,2 und dem Kehrwert des Korrekturfaktors und durch **teiler** zu dividieren. Heraus kommt die Beleuchtungsstärke in Lux.

Um die Messdauer zu verändern, wir haben das weiter oben schon kurz diskutiert, gibt es zwei Kommandos. Die höherwertigen Bits sagen dem BH1750 was er mit dem Rest machen soll.

Change Measurement time (High bit)	01000_MT[7,6,5]	Change measurement time. ※ Please refer "adjust measurement result for influence of optical window."
Change Masurement time (Low bit)	011_MT[4,3,2,1,0]	Change measurement time. ※ Please refer "adjust measurement result for influence of optical window."

※ Don't input the other opecode.

Abbildung 8: Damit wird die Messdauer verändert.

Die Routine **period()** nimmt einen Wert aus dem Intervall [31 ... 254] und legt ihn in dem Instanz-Attribut **_latency** ab, das wir in der Methode **luminanz()** brauchen. Dann wird der Korrekturfaktor der Messzeit berechnet. Der folgenden beiden Zeilen bereiten den übergebenen Wert so zu, dass die beiden Kommando-Bytes daraus entstehen. Im Datenblatt fragt man sich an dieser Stelle (Abbildung 8) wie das wohl gemeint ist. Auf Seite 11 wird es dann schon klarer. Hintergrund für das Vorgehen ist wohl der, dass jedes Kommando nur aus einem Byte bestehen soll. Also muss neben der Nutzlast, dem Korrekturwert, auch ein Schlüssel übermittelt werden, der dem BH1750 sagt, was mit der Nutzlast zu tun ist. Letztlich werden aus einem Nutzlast-Byte dann doch zwei Kommando-Bytes.

```
def period(self, val=None): # p 11 messzeit-Ausgleich
    if val is not None:
        assert val in range(31,255)
        self._latency=val
        self._faktor=val / 69 # 69 is default
        hb=((val & 0xE0) >> 5) | ChangeMTH
        lb=(val & 0x1f) | ChangeMTL
        self.command(hb)
        self.command(lb)
    else:
        return self._latency
```

Wir schieben also die oberen drei Bits des Datenteils, die wir durch [Undieren](#) mit $0xE0 = 0b11100000$ bekommen, um 5 Positionen nach rechts und [oderieren](#) sie auf den Schlüsselteil **ChangeMTH** = $0b01000000$. Die unteren fünf Bits gewinnen wir durch Undieren mit $0x1F = 0b00011111$ und oderieren sie auf **ChangeMTL** = 01100000 . Danach werden die Kommandos zum BH1750 übertragen. Wird **period()** ohne Argument aufgerufen, dann liefert die Methode den aktuellen Wert des Korrekturbytes zurück.

Ein wenig Mystik zum Programmschluss gefällig? - Wir haben eine Messung ohne Sensorabdeckung gemacht und als Ergebnis den Wert 587 Lux erhalten. Mit Glasabdeckung messen wir 391 Lux. Um wieder auf 587 Lux zu kommen, müssen wir die Messzeit verlängern und zwar mit dem Faktor $1,5 = 587 / 391$. Was für ein Korrektur-Byte gehört denn nun dazu? Schön wäre es, wenn es eine Methode zur Umrechnung gäbe. Hier kommt sie, zusammen mit ein bisschen MicroPython-Mystik.

```

@property
def faktor(self):
    return self._faktor

@faktor.setter
def faktor(self, val): # use: faktor = Wert
    assert 0.45 <= val <= 3.68
    self._faktor = val
    self.period(int(val*69+0.5)) # setzt Messzeitfaktor
    print(self._latency)

```

Zum Abfragen des Korrekturfaktors dient die Methode **faktor()**. **@property** ist ein sogenannter [Decorator](#). Er macht die Abfrage in der Form möglich, wie sie bei Variablen üblich ist, man gibt nur den Namen an, die Klammern fallen weg. Sei also das Instanzattribut **_faktor** vom Wert 1,5 dann liefert **faktor** eben diesen Wert. Ohne den Decorator müsste man schreiben **faktor()**. **@property** macht also den Rückgabewert einer Funktion so referenzierbar wie eine einfache Variable.

Warum tun wir das? Wir könnten doch einfach das Instanzattribut **_faktor** direkt abfragen, zum Beispiel so.

```
>>> print(bh._faktor)
```

Zwei Gründe sprechen dagegen. Erstens gehört es in der Objekt-Orientierten-Programmierung zur feinen Art, Variablen nicht direkt zu referenzieren. Man fragt nicht direkt ab und man weist nicht direkt Werte zu. MicroPython hat diesen Zugriffsschutz nicht eingebaut, aber wir können das erzwingen. Es ist nicht nötig, dass der Benutzer eines Programms die Bezeichner von Attributen und Variablen kennt. Er soll nur mit den dahinterstehenden Werten arbeiten.

Zweitens soll ein User/Programmierer nicht irgendwelche Werte zuweisen können, die möglicherweise das Programm zum Absturz bringen können. In **mode()** und **period()** haben wir mit der **assert**-Anweisung für Sicherheit gesorgt. Indem wir unser Modul kompilieren, können wir aus dem Skript eine nichtlesbare Datei machen, die genauso ausführbar ist wie das Skript. Welche Methoden und Attribute sich dahinter verbergen ist dann nicht mehr einsehbar. Wir verraten dem User/Programmierer nur das, was wir freigeben wollen. Indem wir mit dem Decorator **@property** arbeiten, können wir den wahren Namen der abzurufenden Variablen verbergen. Mehr noch, es könnte ja sein, dass für das Abrufen noch weitere Schritte durchgeführt werden müssen, die zum Beispiel die Erscheinungsform verändern sollen.

Ähnlich sieht es aus, wenn wir den Faktor wissen und daraus das Korrektur-Byte berechnen wollen. Auch hierfür verwenden wir einen Decorator, **@faktor.setter**. Auch er versteckt die darunter stehende Methode so, dass sie wie eine Variable behandelt werden kann. Natürlich könnten wir das Attribut **_faktor** direkt belegen. Aber wer stellt dann sicher, dass der Wert im zulässigen Bereich liegt?

In **val** übergeben wir den gewünschten Korrekturfaktor, der sofort auf gültige Werte überprüft wird. Das Instanzattribut **_faktor** wird upgedatet, dann rufen wir die Methode **period()** mit dem berechneten Korrektur-Byte auf, die den Rest erledigt. Den Wert von **_latency** lassen wir uns zurückgeben. Würde die Methode keinen

Decorator haben, müssten wir schreiben **faktor(1,5)**, so schreiben wir wie bei einer Variablen **faktor = 1,5** und lassen dennoch die Überprüfung durchlaufen und einen anderen Wert berechnen.

Was ich hier mit faktor gezeigt habe, ließe sich übrigens auch mit mode, luminanz und period machen. Probieren sie es gerne aus!

Demo und Test

Ein einfaches [Demo-Programm](#) zeigt die Verwendung des Moduls **bh1750.py**.

```
# bh1750_test.py

from machine import Pin,SoftI2C
from bh1750 import BH1750
from time import sleep_ms
from oled import OLED

i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
bh=BH1750(i2c)
d=OLED(i2c,heightw=32)

while 1:

    d.clearAll(False)
    d.writeAt("Luxmeter V1.0",0,0, False)
    d.writeAt("Faktor: {:02f}".format(bh.faktor),0,1, False)
    d.writeAt("Lumi: {} lux".format(bh.luminanz()),0,2)
```

Das I2C-Objekt **i2c** wird zweifach verwendet, wir übergeben es an die Konstruktoren der BH1750- und der OLED-Klasse. In der Endlosschleife löschen wir das Display, lassen die Überschrift ausgeben, gefolgt vom aktuellen Korrektur-Faktor und dem Luminanz-Wert. Das Argument **False** führt dazu, dass die Daten erst einmal nur in den Anzeige-Puffer geschrieben werden. In der letzten Zeile fehlt dieses Argument, das bewirkt, dass jetzt der gesamte Daten-Puffer zum Display übertragen wird. Das spart Rechenzeit und unterdrückt das Flackern der Anzeige.

Ein zweites Programmchen hilft beim Ermitteln des richtigen Korrekturfaktors durch Trial und Error.

```
# bh1750_test.py

from machine import Pin,SoftI2C
from bh1750 import BH1750
from time import sleep_ms
from oled import OLED
from sys import exit
```



```

i2c=SoftI2C(Pin(5),Pin(4),freq=100000)
bh=BH1750(i2c)
d=OLED(i2c,heightw=32)

while 1:
    d.clearAll(False)
    d.writeAt("Luxmeter V1.0",0,0, False)
    d.writeAt("Faktor: {:.02f}".format(bh.faktor),0,1, False)
    d.writeAt("Lumi: {} lux".format(bh.luminanz()),0,2)
    inp=input("Faktor (e:exit; s:speichern): ")
    if inp=="e":
        exit()
    elif inp=="s":
        with open("bh1750_kal.txt","w") as file:
            file.write(str(f)+"\n")
            d.writeAt("gespeichert      ",0,2)
            exit()
    f=float(inp)
    bh.faktor=f
    sleep_ms(200)

```

Nach dem Anzeigen der Werte wird in REPL ein Wert für den Faktor verlangt. Nach der Eingabe wandeln wir den String in eine Fließkommazahl um und rufen den Setter **faktor** auf – kurze Pause – nächste Runde. Mit einem Luxmeter neben unserem Sensor erfassen wir nun den Helligkeitswert und ändern den Faktor so lange, bis beide Geräte in etwa denselben Wert anzeigen. Mit der Taste "s" können wir den Wert abspeichern und mit "e" wird die Schleife verlassen.

Ein wenig Löten zum Schluss

Nun können wir prächtig Messwerte aufnehmen, aber Aktor können wir keinen anschließen. Ein Ausweg wäre die Verwendung eines Portexpanders vom Typ PCF8574 oder MCP23017, die beide über den I2C-Bus angesteuert werden.

Ich habe einen anderen Weg gewählt, um das AZ-Oneboard aufzumuffen. Ich habe mal kurz den LötKolben aufgeheizt und auf dem Board die nicht genutzten und auch nicht herausgeführten GPIO-Anschlüsse mit kurzen Drahtstücken und Steckleisen versehen. Damit stehen mir die GPIOs 0, 2, 15 sowie 13, 12 und 14 zur Verfügung. Zur Not könnte man auch noch den I2C-Bus erweitern, indem man an den Pins 4 und 5 ebenfalls Drähte anlötet.

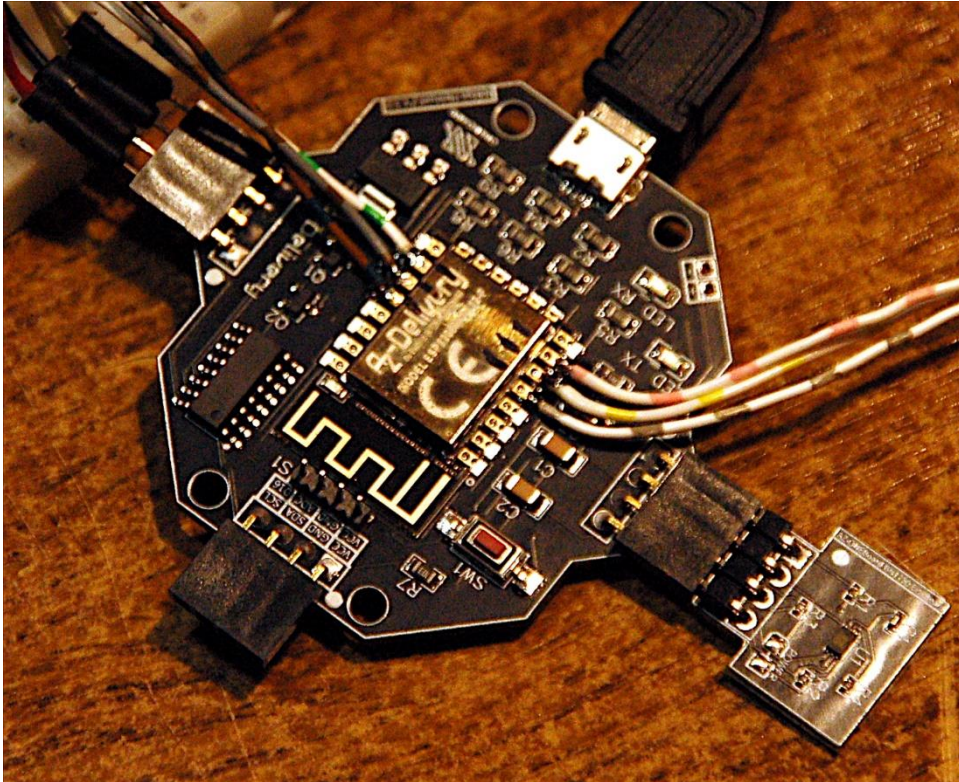


Abbildung 9: Porterweiterung

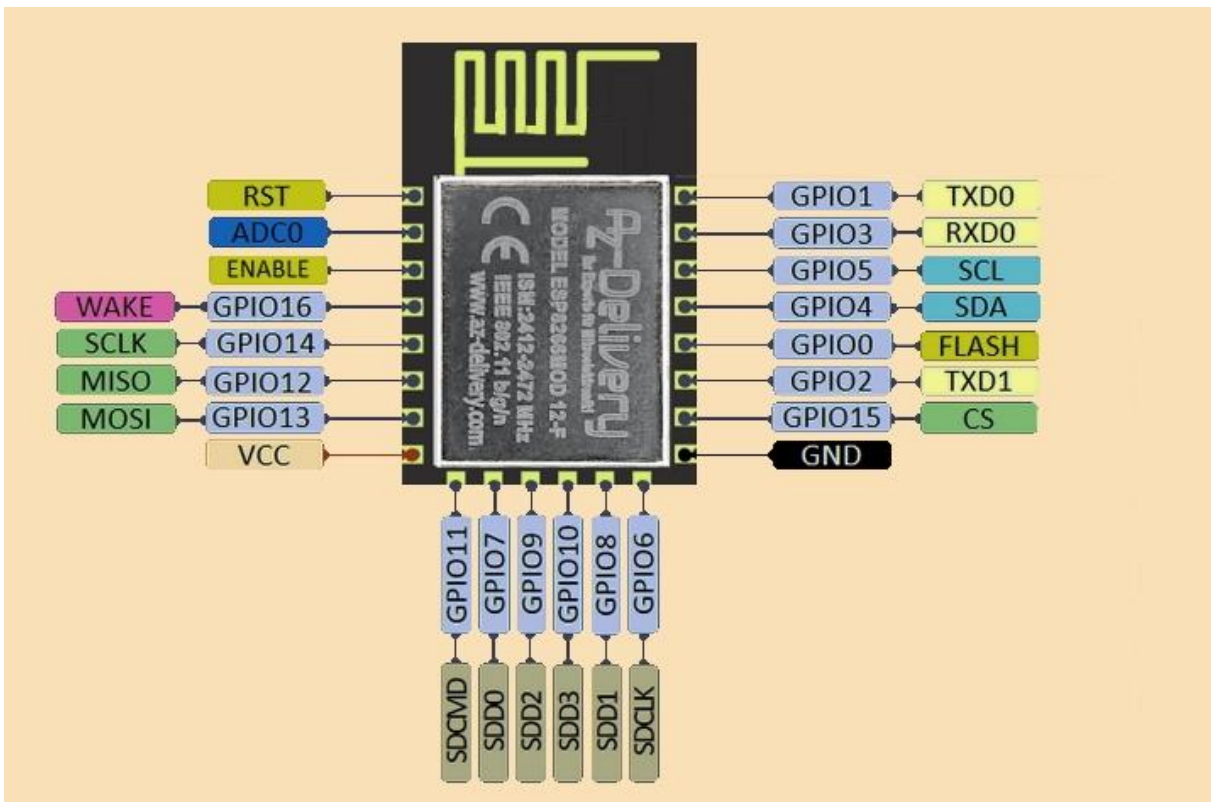


Abbildung 10: Pinbelegung ESP8266 12F

In der nächsten Folge werden wir uns mit dem SHT30 am AZ-Oneboard beschäftigen und ein Modul dafür schreiben. Bis dann!