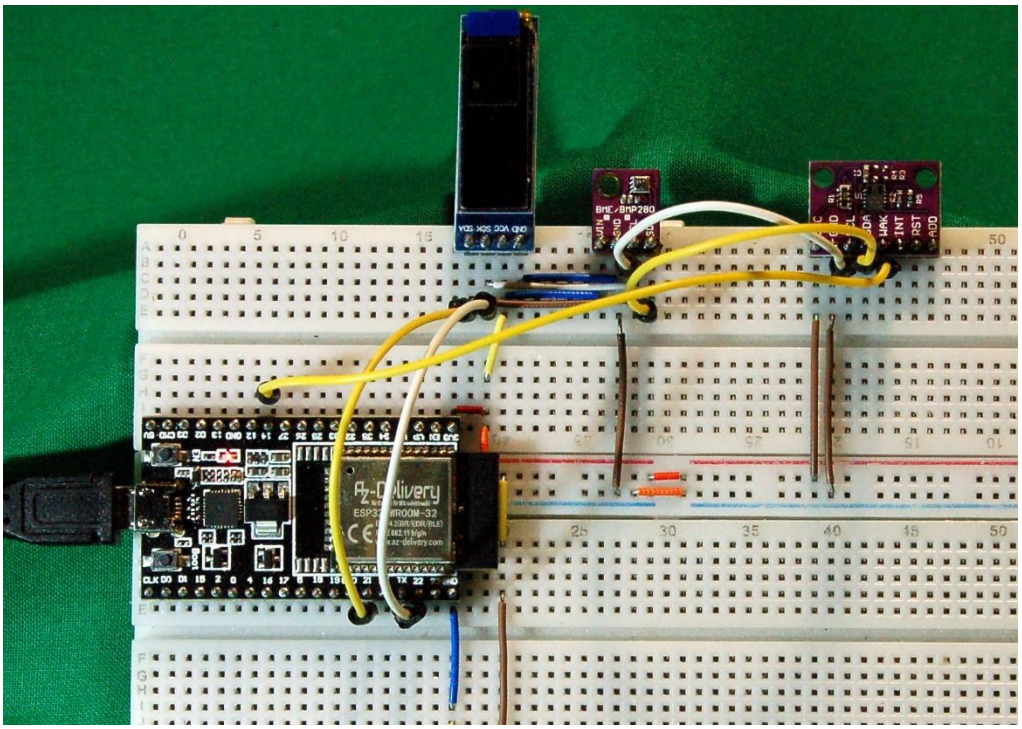

CO2-Meter - Aufbau

Diesen Beitrag gibt es auch als PDF-Dokument.

Neue Module reizen mich immer dazu, dieselben auszuprobieren. Meistens ist das damit verbunden, einen Treiber in MicroPython dafür zu schreiben. Genau das habe ich mit einem CCS811 getan. Es ist ein digitaler Gassensor, der, im Vergleich zu seinen fetten, großen, analogen Brüdern, um Welten weniger Energie durch die Heizung verbrät. Da stehen 45mW gegen 750mW, zum Beispiel beim MQ-3 oder MQ-135, das ist ein Faktor 16! Und auch die räumliche Ausdehnung ist sehr unterschiedlich, vor allem, was die Bauhöhe anbelangt.

Abbildung 1: MQ-135_Draufsicht_wahre Größe

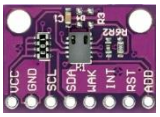

Abbildung 2: ccs811_wahre Größe

Diese SnO_2 – Gassensoren (Zinn-Dioxid) reagieren alle auf organische Verbindungen, sind aber letztlich doch Spezialisten, mit Vorliebe für einzelne Stoffgruppen. So mag der MQ-3 am liebsten Ethanol und andere Alkohole, der MQ-2 liebt Rauchgas und der CCS811 ist auf CO_2 spezialisiert. Neben der Arbeit mit dem Sensor verrate ich in diesem Beitrag wieder das ein oder andere Bonbon im Zusammenhang mit der Programmierung in MicroPython. Ich lade Sie ein zu einem neuen Streifzug durch die Programmierung mit

MicroPython auf dem ESP32 und ESP8266

heute

Der CO_2-Schnüffler CCS811

Die eingesetzten Bausteine, es sind neben dem ESP32 deren drei, ein OLED-Display, ein CSS811 und ein BME280, werden alle über den I2C-Bus angesprochen. Zu diesem Thema habe ich erneut ein Fehlverhalten des MicroPython-Kerns festgestellt. Dem Fehler bin ich schließlich mit Hilfe des DSO (Digitales Speicher Oszilloskop) und eines Logic Analyzers auf die Schliche gekommen. Ohne dieses kleine nützliche Ding sucht man sich einen Wolf. Doch davon später. Werfen wir zuerst einen Blick auf die Hardware für das Projekt CO_2-Schnüffler.

1	ESP32 Dev Kit C unverlötet oder ESP32 NodeMCU Module WLAN WiFi Development Board oder NodeMCU-ESP-32S-Kit
1	0,91 Zoll OLED I2C Display 128 x 32 Pixel
1	GY-BME280 Barometrischer Sensor für Temperatur, Luftfeuchtigkeit und Luftdruck
1	Kohlendioxid-Gassensor Metalloxid CCS811 Sensor mit hochempfindlichem On-Board-Detektionssensormodul für die Luftqualität
1	Breadboard Kit - 3 x 65Stk. Jumper Wire Kabel M2M und 3 x Mini Breadboard 400 Pins
diverse	Jumperkabel
evtl.	Logic Analyzer

Für dieses Projekt ist der kleine Bruder des ESP32, der ESP8266, nicht geeignet, weil sein Speicher einfach zu schmalbrüstig ist. Bereits beim Importieren des BME280-Treiber-Moduls macht er die Grätsche. Ansonsten kann es ein beliebiger ESP32-er sein, denn wir brauchen grade mal drei GPIO-Pins, zwei für den I2C-Bus und einen für den WAKE-Eingang des CCS811, damit dieser überhaut geneigt ist, uns zuhören zu wollen. Das erinnert ein wenig an die Chip-Adressierung auf dem SPI-Bus. Dennoch benötigen wir für den CCS811 zusätzlich eine Hardware-Adresse, damit er sich gekitzelt fühlt. Für das Display und den BME280 entfällt diese Hallo-Wach-Leitung, denen genügt die richtige Hausnummer auf dem I2C-Bus.

Die Verdrahtung ist sehr übersichtlich, was es Einsteigern einfach macht, den Aufbau zu stemmen. Beim Programm liegt das Schwergewicht auf dem Treibermodul des CCS811, denn das Hauptprogramm besteht grade mal aus knapp 90 Zeilen mit viel Luft dazwischen.

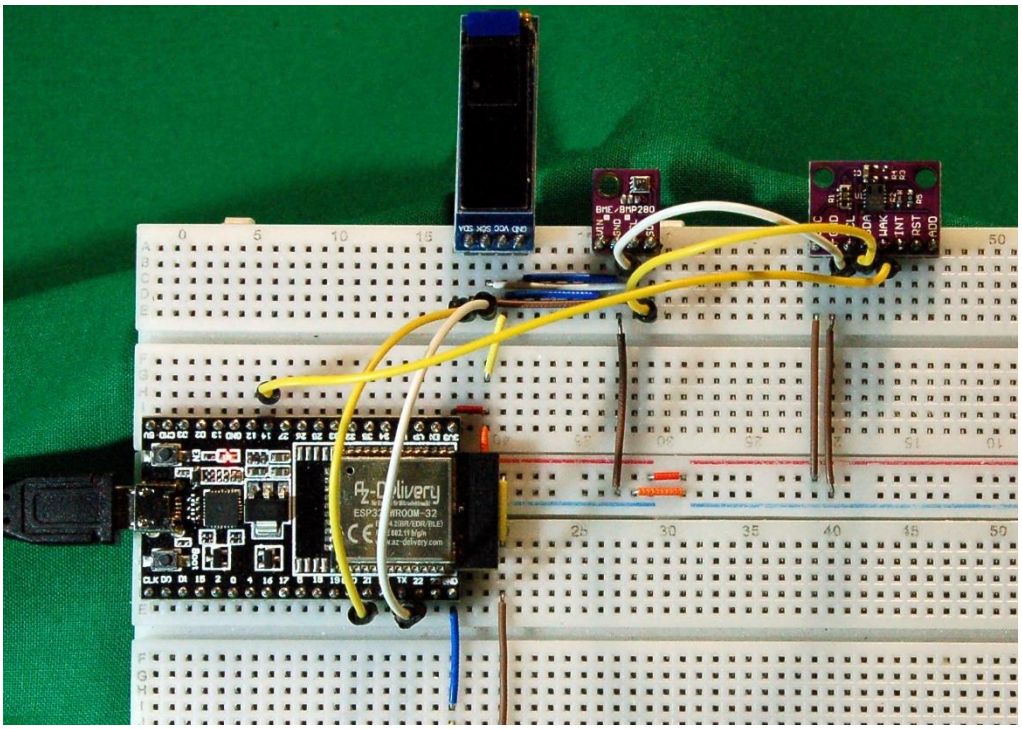

Abbildung 3: CO2-Meter - Aufbau

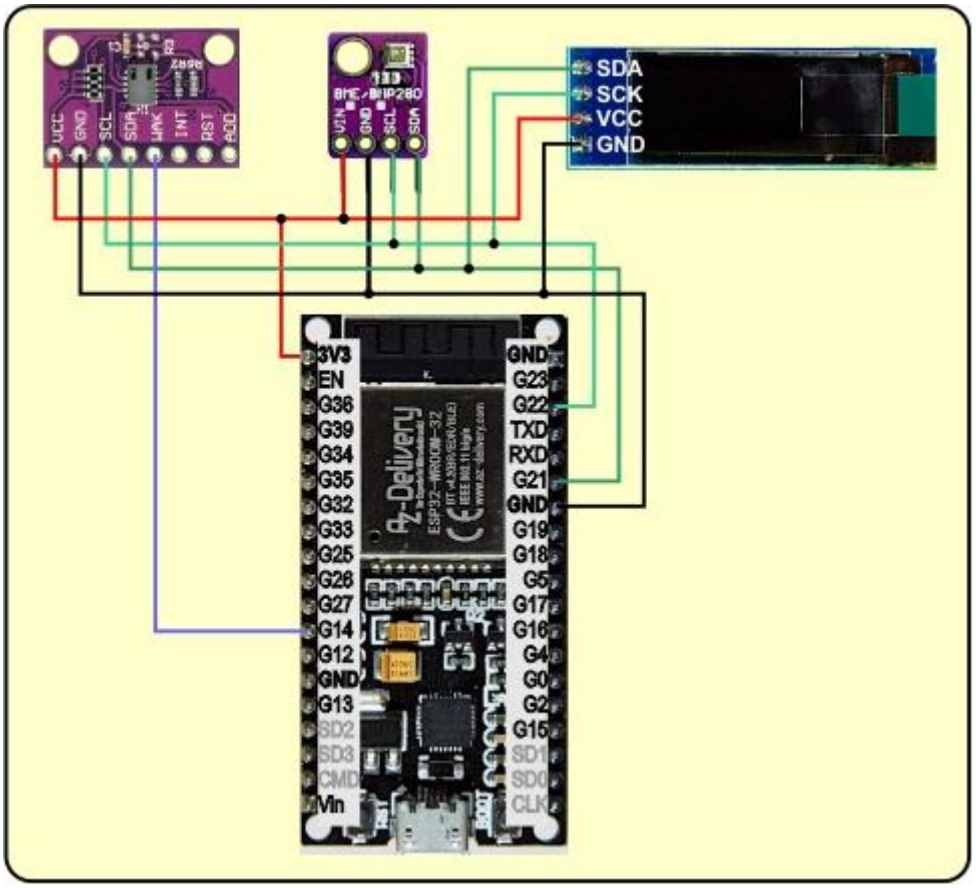

Abbildung 4: CO2-Schnüffler - Schaltung

Die Software

Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware für den ESP32:

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

Die MicroPython-Programme zum Projekt:

ssd1306.py Hardwaretreiber für das OLED-Display
oled.py API für OLED-Displays
ccs811.py Treibermodul
co2sensor.py Betriebssoftware
bme280.py Treibermodul

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.

Das Treibermodul CCS811

Nun findet man vielleicht ein Modul im Web, stellt dann aber fest, dass es für ein anderes Board geschrieben wurde, oder dass es auf einer wesentlich älteren Kernelversion beruht. Und was für MicroPython 9 noch funktioniert hat, muss für Version 19 nicht mehr taugen. Deshalb hole ich mir gerne das Datenblatt zum BOB (Break Out Board) und mache mich selber ans Werk.

Es wird ja ständig am Betriebssystem-Kern von MicroPython herumgedoktert. Grade in den letzten zwei Jahren wurde dabei einiges verschlimmbessert. Besonders davon betroffen sind das PWM-Modul und, wie ich dieses Mal festgestellt habe, das I2C-Modul. Letzteres beinhaltet laut Dokumentation "Primitive I2C operations", "Standard Operations" und "Memory Operations". Bei der ersten und dritten Familie hakt es beim ESP32. Mit der Funktion **readfrom_mem**() Daten von einem bestimmten Register einer Peripherie-Einheit einzulesen, wie es in der Dokumentation steht, ist unmöglich.

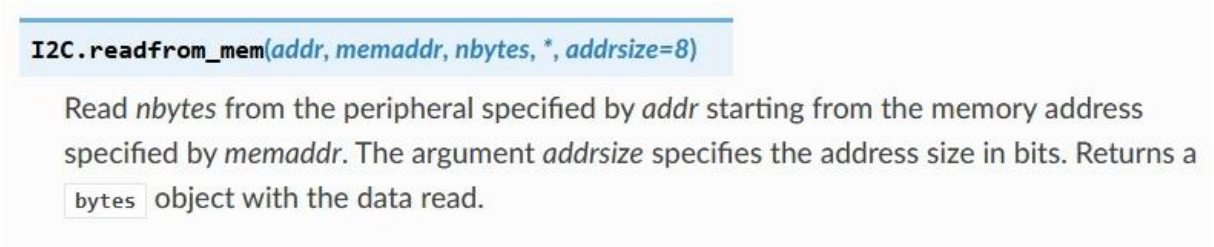

Abbildung 5: Auszug aus der MicroPython-Dokumentation

addr ist die Hardware-Adresse, **memaddr** die Registernummer oder das Kommando und **nbytes** spezifiziert die Anzahl der einzulesenden Bytes. Nun verlangt das I2C-Protokoll eine bestimmte Vorgehensweise, um an die Daten zu kommen. Zuerst muss die Hardware-Adresse auf dem Bus übertragen werden, der Peripheriebaustein quittiert mit einem Acknowledge (ACK), dann folgt die Speicher- oder Registeradresse. Im zweiten Gang folgt nach einer Stop- und Start-Condition erneut

die Hardware-Adresse mit gesetztem Read-Bit. Dann sendet der Slave seine Daten. So sollte es sein, auch beim CCS811. Aber so ist es nicht, wie der folgende Plot meines Logic Analyzers zeigt.

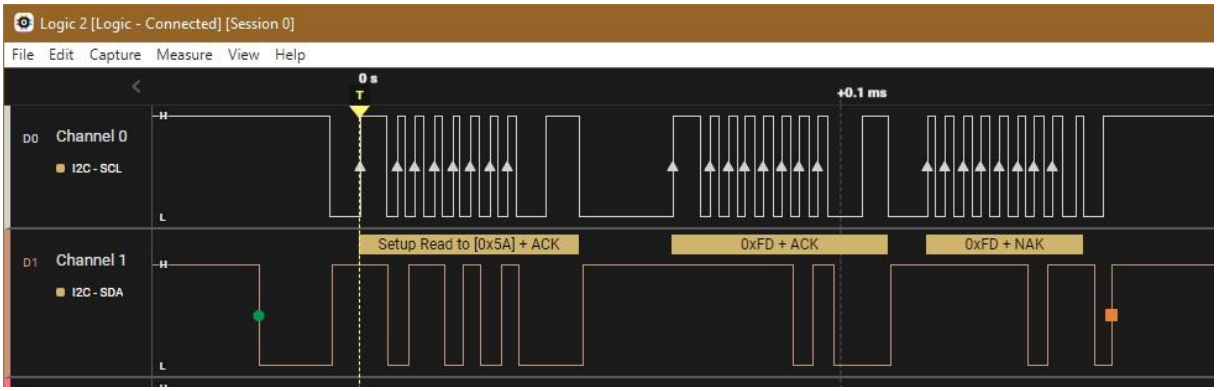

Abbildung 6: readfrom_mem-0x5a-0x20-1_nach einschalten

Der Auftrag an MicroPython lautete: lies vom Baustein mit der Hardware-Adresse 0x5a (der CCS811) von Adresse 0x20 ein Byte ein.

i2c.readfrom_mem(0x5a, 0x20, 1)

Es fehlt eindeutig der erste Teil, das Spezifizieren der Speicherstelle. Stattdessen wird nur die Hardware-Adresse mit gesetztem Read-Bit (LSB) gesendet, worauf der CCS811 zweimal mit 0xFD (???) antwortet. Wenn Sie also nicht graue Haare kriegen und wutentbrannt alles hinschmeißen wollen, dann verwenden Sie diesen Befehl besser nicht. Bis zum Einsatz des Logic Analyzers hat mich der ganze Mist erst einmal gute zwei Stunden gekostet, in denen ich meinen Aufbau kontrolliert und dies und das verändert habe, was ich vielleicht übersehen oder falschgemacht haben könnte. Als ich dann eigene Routinen implementiert hatte, lief alles wie am Schnürchen. So sollte der Plot nämlich aussehen.

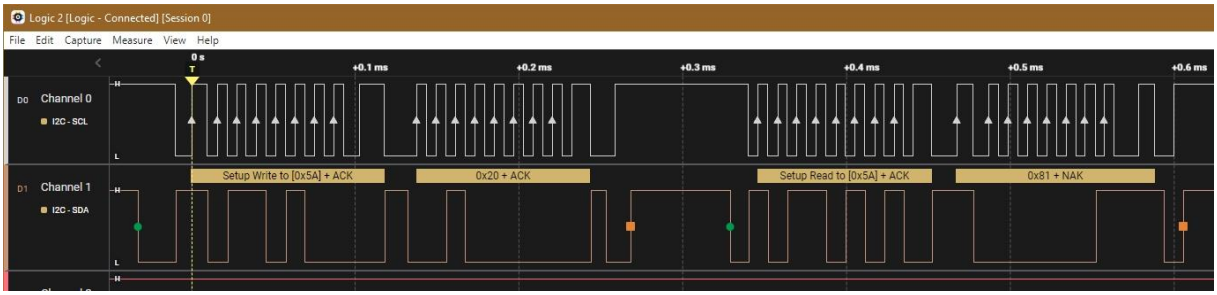

Abbildung 7: So sollte es aussehen

Sehen wir uns nun das fertige Treibermodul für den CCS811 an. Grundlage ist das Datenblatt.

Für das Einhalten der Wartezeit nach dem Erwecken des CCS811 importieren wir die Funktion **sleep** vom Modul **time**.

```
from time import sleep
```

Zur Fehlerbehandlung bei der Initialisierung deklariere ich ein paar Klassen, deren Chef CCS811_Error ist. Die anderen Klassen erben davon.

```
class CCS811_Error(Exception):
    pass

class No_CCS811_DeviceError(CCS811_Error):
    def __init__(self):
        super().__init__("I2C-Fehler","Kein CCS811 gefunden")

class Wrong_HW_ID_Error(CCS811_Error):
    def __init__(self):
        super().__init__("CCS811-Fehler","Falsche HW-ID")

class ApplicationNotValidError(CCS811_Error):
    def __init__(self):
        super().__init__("CCS811-Fehler","App not valid")

class No_baseline_found_Error(CCS811_Error):
    def __init__(self):
        super().__init__("File-Fehler","baseline not found")
```

Die Klassen-Deklaration von CCS811 startet mit der Festlegung der Konstanten für die Registerbezeichner und Bitpositionen.

```
class CCS811:
    STATUS = const(0x00) # datasheet p15
    MEAS_MODE = const(0x01)
    ALG_RESULT_DATA = const(0x02)
    RAW_DATA = const(0x03)
    ENV_DATA = const(0x05)
    NTC = const(0x06)
    THRESHOLDS = const(0x10)
    BASELINE = const(0x11)
    HW_ID = const(0x20)
    HW_VERSION = const(0x21)
    FW_BOOT_VERSION = const(0x23)
    FW_APP_VERSION = const(0x24)
    ERROR_ID = const(0xE0)
    APP_START = const(0xF4)
    SW_RESET = const(0xFF)

    dataReady = const(0x08)
    appValid = const(0x10)
    error = const(0x01)
```

Es folgen die Bedeutungen der Bits des Registers 0xE0 = ERROR_ID.

```
    errorCode={
        1: "Invalid Write Address ",
        2: "Invalid Read Address ",
        4: "Invalid Measure Mode ",
        8: "Resistance Exeeded ",
        16: "Heater Current ",
        32: "Heater Voltage "
        }
```

Dann kommt meine Interpretation der Funktion **readfrom_mem**(), die genau nach der Vorgabe von AMS arbeitet.

Figure 13:
I²C Register Read

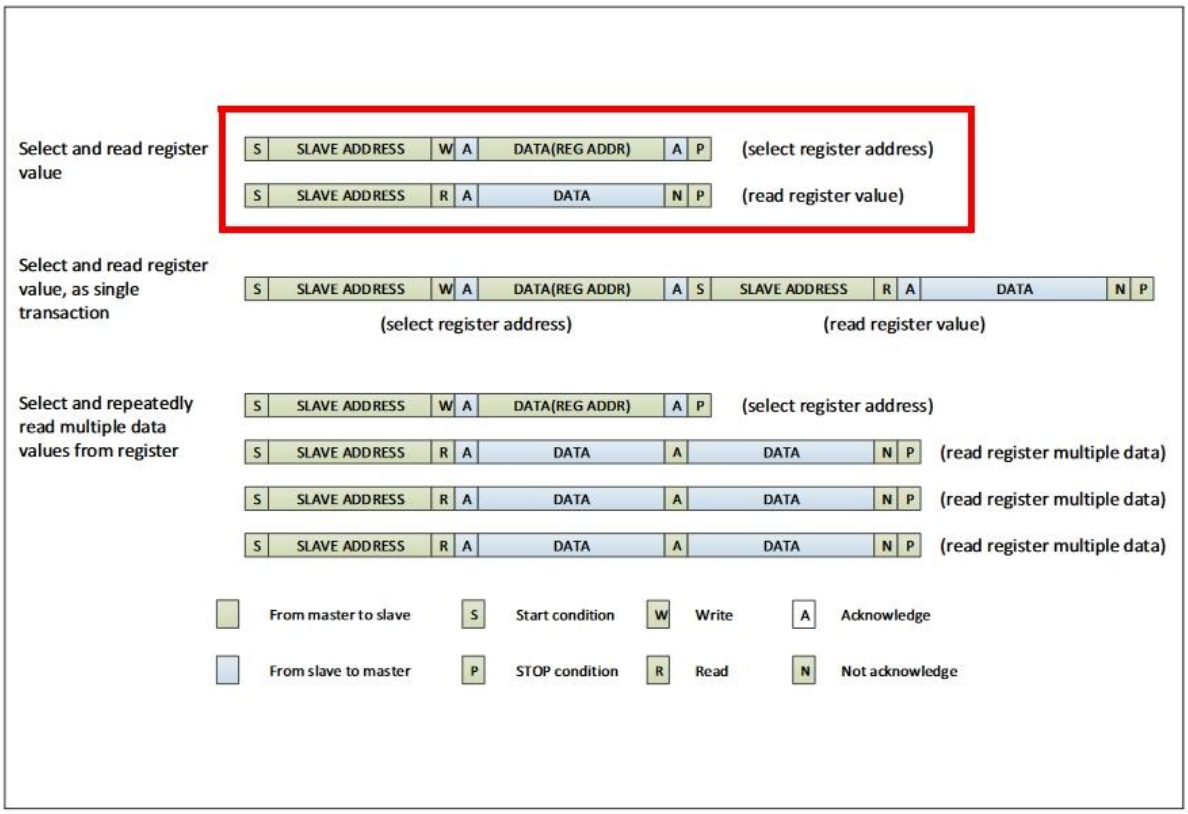

Abbildung 8: Lesen von einem Peripheriebaustein

```
    def readBytesFromReg(self,reg,num):
        buf=bytearray(num)
        try:
            buf=bytearray(num)
            buf[0]=reg
            self.i2c.writeto(self.hwadr,buf[:1])
            self.i2c.readfrom_into(self.hwadr,buf)
            return buf
        except OSError:
            return buf
```

Sporadisch auftretende Übertragungsfehler zwangen mich, die Operation mit try-except abzufangen. Im Falle eines Zugriffsfehlers wird statt dem Registerinhalt der Puffer **buf** mit **num** Nullbytes als Inhalt zurückgegeben, so wie er eingangs erzeugt wurde.

>>> num=3
>>> bytearray(num)
bytearray(b'\x00\x00\x00')

Das I2C-Protokoll verlangt für die Übertragung Objekte, die dem Bufferprotokoll folgen. Das sind **bytes**-Objekte oder **bytearray**s. Weil Ganzzahlen nicht zu dieser Gruppe gehören, muss ich die Register-Nummer im ersten Byte des Arrays **buf** verpacken, **buf[0] = reg**. Danach sende ich ein Slice (=Scheibe) des Arrays, welches nur das erste Byte enthält, **buf[:1**] an die Hardware-Adresse, die ich in **self.hwadr** verpackt habe.

>>> num=3
>>> bytearray(num)
bytearray(b'\x00\x00\x00')
>>> buf[0]=0xE0
>>> buf
bytearray(b'\xe0\x00\x00')
>>> buf[:1]
bytearray(b'\xe0')

Danach erwarte ich vom CCS811 so viele Bytes, wie der Puffer Elemente enthält, im Beispiel also drei.

Wieso das die Leute vom MicroPython-Gremium nicht umsetzen können, wie es in der Doku steht, bleibt mir ein Rätsel.

Auch die Methoden zum Senden von Registerinhalten arbeiten natürlich mit **bytearray**s. Um ein Byte zu senden, brauche ich ein Array mit zwei Elementen

```
    def writeByteToReg(self,reg,data):
        buf=bytearray(2)
        buf[0]=reg
        buf[1]=data & 0xFF
        self.i2c.writeto(self.hwadr,buf)
```

Das erste Element enthält wieder die Registernummer, die hier besser als Kommandocode zu verstehen ist. Das zweite Element bekommt dann das Datenbyte. Um sicher zu gehen, undiere ich mit 0xFF, damit **buf[1]** wirklich nur 8 Bits abbekommt. Ich habe einen Test mit einem 16-Bit-word gemacht und festgestellt, dass tatsächlich nur das Low-Byte zugewiesen wird, aber wer garantiert mir, bei dem heute festgestellten Fehler beim I2C-Transfer, dass das bei zukünftigen Releases der Firmware auch noch so ist. In analoger Weise arbeitet auch das Versenden eines words, also einer 16-Bit-Ganzzahl. Diese wird in das höherwertige und niederwertige Byte aufgeteilt.

```
    def writeWordToReg(self,reg,data):
        buf=bytearray(3)
        buf[0]=reg
        buf[1]=data >> 8
        buf[2]=data & 0xFF
        self.i2c.writeto(self.hwadr,buf)
```

>>> data=0xa3C4
>>> hex(data>>8)
'0xa3'
>>> hex(data&0xFF)
'0xc4'

Der **Konstruktor** erzeugt die nötigen Attribute des Objekts und prüft, ob der CCS811 ordnungsgemäß ansprechbar ist. Zwingend zu übergeben ist ein I2C-Objekt. Optional ist die 7-Bit-Hardware-Adresse. Sie muss beim Aufruf nicht angegeben werden, falls sie dem Defaultwert 0x5A entspricht. Sie kann, je nach Hersteller des Boards auch 0x5B sein. Den Anschluss nWake des CCS811 kann man, anstatt ihn über ein GPIO-Pin anzusteuern, auch fest auf GND-Potenzial legen. Dann entfällt die Übergabe des Pin-Objekts, und der CCS811 ist ständig aktiviert. Andernfalls übergebe ich das Pin-Objekt, mit dem ich den Eingang auf LOW ziehen kann. Wichtig ist in diesem Zusammenhang, dass der CCS811 nur dann über den I2C-Bus kommunizieren kann, wenn der nWake-Eingang LOW ist. Bei anderen Bausteinen ist der Bus stets ansprechbar, auch wenn die sonstigen Funktionen gerade ein Schläfchen machen.

```
    def __init__(self, i2c, HWADR=0x5A, nwake=None):
        self.i2c = i2c
        self.hwadr = HWADR
        self._tVOC = 0
        self._eCO2 = 0
        self._mode = 1
        self.baseline=[0,0]
        if nwake is not None:
            self.wakePin = nwake
        self.error = False
        self.dataRdyInt=1
        self._state = 0
        self.buf = bytearray(6)

        # datasheet p4 HWADR=0x5a alt. 0x51
        if self.hwadr not in i2c.scan():
            raise No_CCS811_DeviceError

        # datasheet p21 (HW_ID)=0x81
        hwID=self.readBytesFromReg(HW_ID,1)
        if hwID[0] != 0x81:
            raise Wrong_HW_ID_Error

        # datasheet p16 valid App?
        status = self.readBytesFromReg(STATUS,1)
```

```
        if not status[0] & appValid:
            raise ApplicationNotValidError

        # datasheet p24 Start App
        self.buf[0]=0xF4
        self.i2c.writeto(self.hwadr,self.buf[:1])

        # datasheet p17 Set mode 1 (1-second-cycle + Int)
        self.buf[0]=0x18
        self.i2c.writeto(self.hwadr,self.buf[:1])

        print("CCS811 @ {0:#X}".format(self.hwadr))
```

Der CCS811 kann in verschiedenen Modi arbeiten. Die Modusauswahl realisiert die Methode **measMode**(), der als Argument eine Nummer von 0 bis 4 übergeben wird. Die Modi sind auf Seite 17 im Datenblatt beschrieben. In meinem Beispielprogramm arbeite ich mit Modus 1, jede Sekunde wird ein Messwert geliefert, der CCS811 ist also dauernd unter Strom.

```
    def measMode(self,mode=None):
        self.aWake(0)
        if mode is None:
            return (self.readBytesFromReg(MEAS_MODE,1)[0] \
                   & 0b01110000) >> 4
        else:
            if mode not in range(5): mode = 1
            reg=self.readBytesFromReg(MEAS_MODE,1)[0]
            reg = (reg & 0b10001111) | mode << 4
            self._mode=mode
            self.writeByteToReg(MEAS_MODE,reg)
```

Die Anweisung **aWake**(0) stellt sicher, dass der CCS811 überhaupt ansprechbar ist. Wurde kein Argument übergeben, dann meldet die Methode den gegenwärtigen Zustand des Registers 0x01 zurück. Der Modus ist in den Bits 4 bis 6 codiert. Die Modusnummer erhalte ich, wenn ich diese Bits maskiere und um 4 Positionen nach rechts verschiebe. **readBytesFromReg**() liefert ein bytearray zurück. Im Element 0 steht der Inhalt des Registers.

Ein übergebenes Argument wird auf den Bereich 0..4 überprüft und auf 1 gesetzt, wenn der Wert außerhalb liegt. Dann lese ich den Registerinhalt von 0x01 ein und setze die Mode-Bits durch Undieren mit der Maske 0x8F = 0b10001111 auf 0. Im selben Durchgang oderiere ich die um 4 Positionen nach links geschobene Modusnummer dazu. Das Attribut _mode wird angepasst und das Register neu geschrieben.

Den Messwert einzulesen macht nur Sinn, wenn einer bereitsteht. Das kann man prüfen, wenn man den Ausgang **int** des CCS811 mit einem interruptfähigen GPIO-Pin des ESP32 verbindet. Eine steigende Flanke an **int** zeigt dann an, dass ein Messwert abgeholt werden kann. Diese Funktion wird eingeschaltet, indem das Bit 3 im Register **MEAS_MODE** auf 1 gesetzt wird. Die Methode **intDataReady**() managet das in ähnlicher Weise wie **measMode**().

```
    def intDataReady(self,enint=None):
        self.aWake(0)
        if enint is None:
            return (self.readBytesFromReg(MEAS_MODE,1)[0] \
                   & 0b00001000) >> 3
        else:
            if enint not in range(2): enint = 1
            reg=self.readBytesFromReg(MEAS_MODE,1)[0]
            reg = (reg & 0b11110111) | enint << 3
            self.dataRdyInt=enint
            self.writeByteToReg(MEAS_MODE,reg)
```

Die Methode **aWake**() fragt den wakePin-Status ab oder setzt den Ausgang auf den Wert des übergebenen Arguments, das natürlich auf den korrekten Bereich, 0 oder 1, überprüft wird. Bis der CCS811 aus seinem Schlummer erwacht ist, warten wir eine Millisekunde.

```
    @property
    def status(self):
        state= self.readBytesFromReg(STATUS,1)
        self.state=state
        return "{0:#x}".format(state[0])
```

Die Methode **status**() muss nur einen Wert zurückgeben und benötigt außer dem Parameter **self** keinen weiteren. Der Decorator **@property** erlaubt auf den zurückgegebenen Wert, wie auf eine Variable zuzugreifen. Auf die Attribute eines Objekts sollte in der OOP (Object Oriented Programing) nicht direkt zugegriffen werden. Daher gibt es das Konzept der Getter- und Setter-Routinen. Einen Getter erhält man, indem man in die Zeile vor der Methoden-Deklaration den Decorator @property schreibt. Würde ich **_state** direkt abrufen, bekäme ich einen Zahlenwert. Über den Getter status kann ich das Format der Ausgabe beliebig verändern und an meine Vorstellungen anpassen. Setter erlauben zum Beispiel eine Plausibilitätskontrolle, bevor die Zuweisung an ein Attribut oder eine Variable erfolgt. MicroPython bietet leider keine Kombination aus Getter und Setter, es müssen immer Zwei Routinen sein. Ich kombiniere dennoch gerne beide, etwa wie in **measMode**() oder **aWake**(). Gebe ich kein Argument an, habe ich den Getter-Teil, der mir ein Register ausliest und die Information aufbereitet, bevor ein Wert zurückgegeben wird. Gebe ich ein Argument an, wird dessen Wert geprüft und für die Übergabe an den CCS811 umgewandelt.

Um den Inhalt des Statusregisters eines Objekts ccs abzufragen geben Sie also nicht print(ccs._state) ein, obwohl das möglich wäre, sondern print(ccs.status). Die Rückgabe ist übrigens ein als Hexadezimalzahl formatierter String.

Auch die Methode **checkDataReady**() ist in derselben Weise dekoriert. Wenn **status** eine 1 an Bitposition 3 meldet, wird **state & dataReady** ungleich 0 und somit vom Interpreter als True gewertet. Wir sprechen Register 0x02 (ALG_RESULT_DATA) an, was der CCS811 als Aufforderung auffasst, uns bis zu acht Bytes als Ergebnis zu senden: High- und Low-Byte des berechneten eCO_2-Werts (equivalent CO_2), High- und Low-Byte des TVOC-Werts (Total Volatile Organic Components), Status, Error_ID und zwei Bytes Raw Data. Wir holen aber nur die sechs ersten ab.

```
    @property
    def checkDataReady(self): # Zuordnung datasheet p18
        self.aWake(0)
        state=int(self.status)
        if state & dataReady:
            coH,coL,tvH,tvL,st,er = \
                    self.readBytesFromReg(ALG_RESULT_DATA,6)
            self._eCO2=(coH << 8 ) | coL
            self._tVOC=(tvH << 8) | tvL
            self._state=st
            self.error=er
            return True
        else:
            return False
```

Durch achtmaliges Linksschieben des High-Bytes und anschließendes Oderieren mit dem Low-Byte entsteht der 16-Bitwert von eCO_2 und TVOC. Die Werte werden den entsprechenden Attributen des CCS811-Objekts zugewiesen. Konnten Werte gelesen werden, geben wir **True** zurück, sonst **False**.

Es folgen die Getter für die Messwerte.

```
    @property
    def eCO2(self):
        return self._eCO2

    @property
    def TVOC(self):
        return self._TVOC
```

In den beiden Baseline-Bytes, die über das Kommando-Byte **BASELINE**=0x11 abgerufen und geschrieben werden können, befindet sich ein Wert, der vom CCS811 intern berechnet wird und der die momentanen Temperatur- und Luftfeuchte-Werte in die Messung mit einbezieht. Die Methode **baseLine**() ruft die beiden Bytes ab und steckt sie in eine Liste, die zurückgegeben wird.

```
    @property
    def baseLine(self):
        self.aWake(0)
        self.baseline=list(self.readBytesFromReg(BASELINE,2))
        return self.baseline
```

storeBaseLine() schreibt die beiden Bytes in die Datei **baseline** im Flash-Speicher des ESP32. Beim Neustart des Programms **co2sensor.py** werden sie eingelesen und dienen bis zum nächsten Klima-Update als Näherungswerte. Zum speichern werden die Zahlen in Strings umgewandelt und mit einem Zeilenvorschub versehen.

```
    def storeBaseLine(self,bl):
        with open("baseline","w") as f:
            f.write(str(bl[0])+"\n")
            f.write(str(bl[1])+"\n")
```

loadBaseLine() erledigt den Ladevorgang, wenn denn die Datei **baseline** existiert. Wir stellen das fest. Indem wir die Liste der Dateien im Root-Verzeichnis anfordern und nachsehen, ob der Name **baseline** in der Liste enthalten ist. Wenn ja, dann lesen wir die beiden Zeilen, entfernen den Zeilenvorschub und wandeln das Ergebnis in eine Zahl um. Damit füttern wir die Liste, die zurückgegeben wird.

Beim ersten Start gibt es noch keine Datei. In diesem Fall muss der CCS811 mit den Default-Werten 50% relative Luftfeuchte und 25°C auskommen.

```
    def loadBaseLine(self):
        dateien=os.listdir("/")
        if "baseline" in dateien:
            with open("baseline","r") as f:
                hb=int((f.readline()).strip())
                lb=int((f.readline()).strip())
            self.baseline=[hb,lb]
            return [hb,lb]
        else:
            print("No baseline stored")
            raise No_baseline_found_Error
```

Mit **setBaseLine**() gelangen die Baseline-Bytes zum CCS811. Wir lassen einen 16-Bit-Wert zusammensetzen und schicke ihn mit **writeWordToReg**() ins Register 0x11.

```
    def setBaseLine(self, bl):
        data=bl[0] << 8
        data|=bl[1]
        self.aWake(0)
        self.writeWordToReg(BASELINE,data)
```

Unser BME280 kann recht genau die Klimadaten erfassen, deshalb haben wir ihn mit an Bord genommen. Die Daten müssen an den CCS811 übermittelt werden, das macht die Methode **setEnvData**(), der wir die rel. Luftfeuchtigkeit und Temperatur übergeben.

Die Werte müssen in einer speziellen Weise an den CCS811 gesendet werden. Es sind immer 2 Bytes. Aber die Übergabe erfolgt in Form eines Zwei-Byte-Werts, der eine Festkomma-Notierung eines float-Werts erfordert. Das klingt kompliziert, bedeutet aber letztlich nichts anderes, als dass der ganzzahlige Anteil eines

Temperaturwerts die Bits 1 bis 7 des High-Bytes und der Nachkommaanteil die Bits 0 bis 7 des Low-Bytes plus dem Bit 0 des High-Bytes belegt. Das Datenblatt zeigt das so.

Relative Humidity

Figure 16:
Relative Humidity Fields and Byte Order

Byte 0								Byte 1							
Humidity High Byte								Humidity Low Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512
Humidity %								Humidity % Fraction							

Humidity is stored as an unsigned 16 bits in 1/512%RH. The default value is 50% = 0x64, 0x00. As an example 48.5% humidity would be 0x61, 0x00.

Temperature

Figure 17:
Temperature Fields and Byte Order

Byte 2								Byte 3							
Temperature High Byte								Temperature Low Byte							
7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
64	32	16	8	4	2	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256	1/512
Temperature 25°C								Temperature 25°C Fraction							

Abbildung 9: Environmentdata setzen

Dazu kommt noch die Information aus dem Datenblatt, dass eine 0 im ganzzahligen Teil, für die Umgebungstemperatur von -25°C steht. Daher ist zur BME280-Temperatur 25 zu addieren. Was rauskommt, wird um eine Position nach links geschoben. Das LSB des Nachkommaanteils ist ein 512-tel °C. Die Bitfolge ergibt sich somit durch die Multiplikation des Bruchanteils mit 512. Das Ergebnis wird dann mit dem ganzzahligen Anteil oderiert. Für die Übertragung werden daraus zwei Bytes zu 8 Bit fabriziert.

Beispiel: 28,741°C
ganzzahliger Anteil: 28 + 25 = 53
Bruchanteil: 0,741 • 512 = 379,392

```
          00110101 = 53
          00110101  << 9
0110101000000000
       101111011 = 379
0110101101111011
```

aufteilen in zwei Bytes:

```
0110101101111011 >> 8
          01101011 High
0110101101111011 & 0xFF
0000000011111111
          01111011 LOW
```

Abbildung 10: Logeleien

Von der Luftfeuchte wird nur der ganzzahlige Anteil übertragen, der Bruchanteil wird auf 0 gesetzt.

```
    def setEnvData(self,hum,temp):
        tempH=(int(temp)+25) << 9
        tempL=int((temp-int(temp))*512)
        tempW= tempH | tempL
        self.buf[0]= ENV_DATA # Register-Adresse
        self.buf[1]= hum < 1
        self.buf[2]= 0
        self.buf[3]= tempW >> 8
        self.buf[4]= tempW & 0xFF
        self.aWake(0)
        self.i2c.writeto(self.hwadr,self.buf[:5])
```

Die letzte Methode des Moduls gibt eventuelle Fehlermeldungen im Klartext aus. Wenn das Fehlerbit im Statusregister gesetzt ist, wird das Fehlerregister zur genaueren Feststellung des Fehlers ausgelesen. Weil mehrere Bits gesetzt sein können, muss der Ausgabestring zusammengesetzt werden. Das geschieht in der for-Schleife. Bei jedem gesetzten Bit wird der entsprechende Text aus dem Dictionary **errorCode** geholt und an **code** angehängt.

```
    @property
    def anyErrors(self):
        self.aWake(0)
        state= self.readBytesFromReg(STATUS,1)[0] & 0x01
        if state :
            fehler=self.readBytesFromReg(ERROR_ID,1)[0]
            code=""
            for f in range(6):
                if fehler & 1<< f:
                    code += CCS811.errorCode[f]
            if code == "":
                return None
            else:
                return code+"Fehler"
```

Das Hauptprogramm

Es spannt die beiden Zugpferde, CCS811- und BME-Klasse, vor unseren Wagen und lässt noch weitere Ponis, wie oled, sleep und timeout mitlaufen.

```
# co2sensor.py

from machine import Pin, SoftI2C
import sys
from oled import OLED
from ccs811 import CCS811
from time import sleep
from bme280 import BME280
from timeout import *
```

Damit ich nicht jedes Mal in irgendwelchen Programmen danach suchen muss, habe die Sequenz für den Softwaretimer TimeOut() jetzt in ein Modul verpackt und dabei noch etwas aufgehübscht. Es ist manchmal recht nützlich, einen Timer zu haben, der nie abläuft.

```
from time import ticks_ms

# Nicht blockierender Softwaretimer gibt False zurueck, wenn
# die Zeitdauer in t noch nicht abgelaufen ist, sonst True

def TimeOut(t):
    start=ticks_ms()

    def compare():
        nonlocal start
        if t==0:
            return False
        else:
            return int(ticks_ms()-start) >= t

    return compare
```

Mein Standardschnippsel für den I2C-Bus.

```
if sys.platform == "esp8266":
    i2c=SoftI2C(scl=Pin(5),sda=Pin(4),freq=100000)
elif sys.platform == "esp32":
    i2c=SoftI2C(scl=Pin(22),sda=Pin(21),freq=100000)
else:
    raise RuntimeError("Unknown Port")
```

Dann richten wir das CCS811-Objekt ein.

```
wakePin=Pin(14,Pin.OUT,value=0)

ccs=CCS811(i2c,nwake=wakePin)
ccs.aWake(0)
ccs.intDataReady(1)
ccs.measMode(1)
```

Es folgen Display- und BME280-Ojekt.

```
d=OLED(i2c,heightw=32) # 128x32-Pixel-Display
d.writeAt("CO2-Meter",0,0)

bme=BME280(i2c)
```

Letzteres kurbeln wir schon mal an, um die Variablen Hum0 und Temp0 zu deklarieren, die später referenziert werden und daher schon bekannt sein müssen..

```
Hum0=bme.calcHumidity()
Temp0=bme.calcTemperature()
```

Der CCS811 soll uns sagen, wie seine Einstellungen wirklich aussehen.

```
print("Mode:",ccs.measMode(), "nWake:",ccs.aWake())
```

Die Taste für den sauberen Ausstieg aus der Hauptschleife.

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

Der CCS811 braucht Zeit zum warm werden. Das Datenblatt gibt da 20, besser 30 Minuten vor. Vor dem ersten Betrieb sollte das Teil 24 - 48 Stunden eingebrannt werden. Außerdem wird empfohlen, gespeicherte Baseline-Daten erst nach der Warmlaufzeit zu restaurieren. Im Projekt habe ich dafür drei Softwaretimer installiert. Während der Testphase liegen die Zeiten im Sekundenbereich.

```
refreshPeriod=5*1000 # nach 60 Sek. EnvData updaten
over=TimeOut(refreshPeriod)

loadPeriod=10*1000
loadIt=TimeOut(loadPeriod) # nach >30 Min. Baseline laden

savePeriod=20*1000
saveIt=TimeOut(savePeriod) # nach 60 Min. Baseline sichern
```

In der Hauptschleife gibt es 5 Jobs. Der erste ist das periodische Einlesen von Temperatur und relativer Luftfeuchte, wenn der Timer **over**() abgelaufen ist. Die Werte werden im Terminal und im OLED-Display ausgegeben. Dann stellen wir den Timer neu.

```
while 1:
    if over():
        Hum=bme.calcHumidity()
        Temp=bme.calcTemperature()
        print("Temperatur: {0:5.2f}; RelFeuchte: {1:5.2f}".\
              format(Temp,Hum))
        d.clearFT(0,0,15,0,False)
        d.writeAt("{:3.2}* {:5.2f}%    ".\
                  format(Temp,Hum),0,0,False)
        over=TimeOut(refreshPeriod)
        if (abs(Temp - Temp0) > 3) or (abs(Hum - Hum0) > 5):
            Temp0=Temp
            Hum0=Hum
            ccs.setEnvData(Hum,Temp)
            print("EnvData set")
            d.writeAt("XX",13,0)
```

Wenn sich die Absolutwerte der Differenzen aus altem und neuem Messwert um mehr als ein bestimmtes Limit unterscheiden, werden die neuen Werte in die Variablen der alten übernommen und durch **setEnvData**() an den CCS811 geschickt. Welche Limits Sie verwenden wollen, entscheiden Sie selbst. Das Programm informiert uns über das Terminal und im Display erscheint rechts oben "XX".

Dann schauen wir nach, ob eine neue Messung durchgeführt wurde. Ist das der Fall, dann hat die Methode die Werte bereits an die Attribute _**eCO2** und _**TVOC** übergeben und wir können sie über die Getter-Methoden abholen.

```
    if ccs.checkDataReady:
        print("eCO2: {}; TVOC: {}; Baseline: {}".\
              format(ccs.eCO2,ccs.TVOC,ccs.baseLine))
        d.writeAt("eCO2 {:5} ppm   ".\
              format(ccs.eCO2),0,1,False)
        d.writeAt("TVOC {:5} ppb   ".format(ccs.TVOC),0,2)
```

Ist der Timer **loadIt**() abgelaufen, wird es Zeit, die zuletzt gespeicherte Baseline einzulesen. Was die Baseline eigentlich ist, fragen Sie? Nun, wenn Sie die Messwerte für reine Luft in einem Diagramm auftragen und verbinden würden, bekämen Sie eine Parallele zur Zeitachse, die Basislinie oder Baseline. Die beiden Bytes im Register BASELINE = 0x11 werden mit Hilfe der Temperatur und Luftfeuchte in den ENV_DATA-Registern berechnet und beeinflussen die Berechnung des eCO_2- und TVOC-Werts. Wenn der CCS811 in CO_2-haltiger Luft gestartet wird, braucht er für die Berechnung korrekter Messwerte als Basisbezug die Informationen im BASELINE-Register. Das darf aber erst beschrieben werden, wenn der Sensor stabil arbeitet, und das ist nach ca. 30 Minuten der Fall. Weil ständiges Nachladen keinen Sinn macht, es würden ja die vom CCS811 neuberechneten Werte laufend überschrieben, Stelle ich den Timer **loadIt**() mit der 0 auf Dauerlauf. Damit tritt er fürderhin nicht mehr in Erscheinung.

```
    if loadIt():
        try:
            bl=ccs.loadBaseLine()
            ccs.setBaseLine(bl)
            print("Baseline geladen und gesetzt\n")
        except :
            print('File-Fehler', 'baseline not found\n')
        loadIt=TimeOut(0)
```

```
    if saveIt():
        ccs.storeBaseLine(ccs.baseLine)
        print("Baseline gesichert\n")
        saveIt=TimeOut(savePeriod)
```

Das Datenblatt gibt auch Empfehlungen zum Speichern der Baseline-Werte. Die Zeiträume dafür hängen von der Einsatzdauer des CCS811 ab. In jedem Fall sollte vor Programmende gespeichert werden. Deshalb ist der Programmausstieg mit Strg+C keine gute Lösung, weil das irgendwo im Programm passieren kann. Besser ist der Ausstieg mit der Flash-Taste. Nach dem Sichern der Baseline schicke ich den CCS811 schlafen und beende nach den Meldungen das Programm mit **sys.exit**().

```
    if taste.value()==0:
        ccs.storeBaseLine(ccs.baseLine)
        ccs.aWake(1)
        d.clearAll()
        d.writeAt("Prog. cancelled",0,0)
        d.writeAt("Baseline saved",0,1)
        sys.exit()
```

Kurzes Durchschnaufen und Ring frei für die nächste Runde.

```
    sleep(1)
```