Diesen Beitrag gibt es auch als:
PDF in deutsch
This episode is also available as:
PDF in english

Stellen Sie sich vor, Sie sind mit dem GPS-Tracker im Gelände unterwegs und werden von einem Unwetter überrascht. Wäre es da nicht besser, das Gerät könnte neben der Position auch Luftdruck und Temperatur erfassen? Wozu? Nun ja, weil vor einem Gewitter der Luftdruck meist rasant in den Keller geht. Und wenn Sie diesen Wert kennen, haben Sie Zeit, einen sicheren Unterschlupf aufzusuchen. Um Luftdruckwerte erfassen zu können, habe ich meinem GPS-Tracker einen BMP280 spendiert. Nebenbei wird damit auch die Temperatur gemessen. Beide Größen werden mit einer erstaunlichen Genauigkeit erfasst. Was es daneben noch an weiteren Neuigkeiten zu dem Gerät und zum Programm gibt, erfahren Sie in diesem Beitrag. Seien Sie herzlich willkommen zum zweiten Teil von

GPS mit MicroPython auf dem ESP32

Hardware – nur ganz wenig Zuwachs

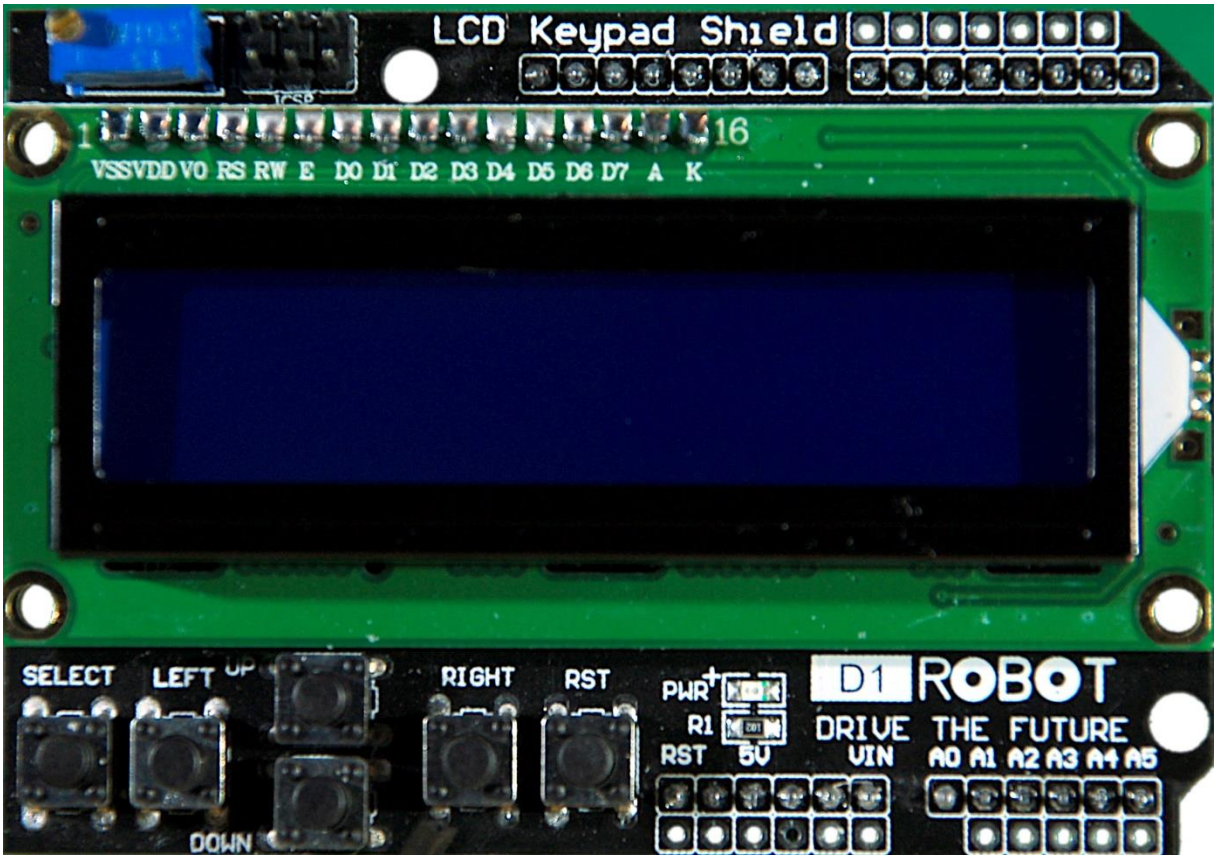
In der folgenden Liste finden Sie alle Teile für das Projekt. OK, das allermeiste davon wurde bereits im Teil 1 eingesetzt. Diese Bauteile verwenden wir natürlich wieder. Neu dazugekommen ist ein BMP280-Modul, das ich ans Ende der Liste gesetzt habe. Erwähnt muss noch werden, dass der 10kΩ-Widerstand am RST-Anschluss der Keypad-Platine durch den Pullup-Widerstand des GPIO25 ersetzt wurde. Dieser

Widerstand und ein weiterer 10kΩ dienen jetzt als Spannungsteiler für die Messung der Batteriespannung am ESP32. Da noch Analogeingänge am Controller frei sind, könnte man auch noch eine 5V-Überwachung dazu bauen. Denkbar ist auch ein Buzzer, der schreit, wenn die Spannungen unter einen Mindestpegel fallen. Aber das gehört zur Kür und nicht zur Pflicht. Die Auswahl des Controllers ESP32 gewährt Ihnen trotzdem sehr viel Freiheit beim Ausbau des Projekts. Lassen Sie der Phantasie freien Lauf!

Hier also die Liste der Zutaten, das Rezept kommt gleich danach.

1	ESP32 Dev Kit C V4 unverlötet oder ähnlich
1	LCD1602 Display Keypad Shield HD44780 1602 Modul mit 2x16 Zeichen
1	SIM 808 GPRS/GSM Shield mit GPS Antenne für Arduino
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
1	Li-Akku Typ 18650
1	I2C IIC Adapter serielle Schnittstelle für LCD Display 1602 und 2004
4	Widerstand 10kΩ
1	GY-BMP280 Barometrischer Sensor für Luftdruckmessung

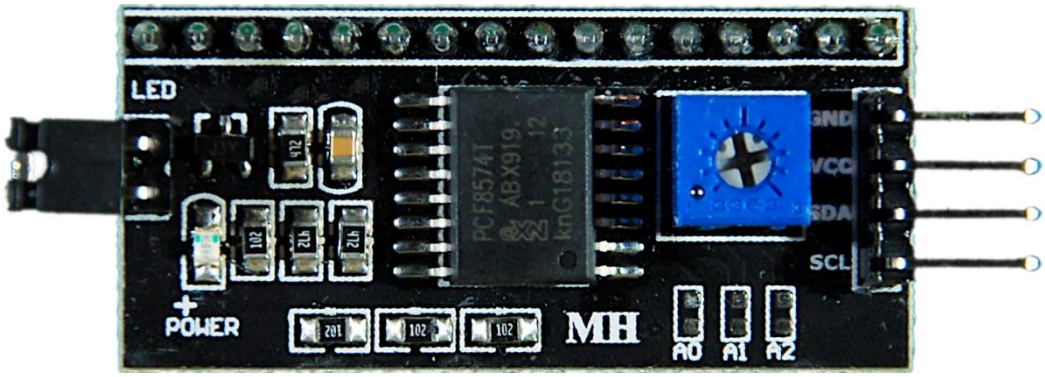
Leider habe ich immer noch keine Möglichkeit gefunden, die Hintergrundbeleuchtung des Displays zwischenzeitlich auszuschalten, um Batteriekapazität zu sparen. An den Schalttransitor kommt man nur durch Hochbiegen des Displays heran. Der Basisanschluss des Transistors müsste dann von der 5V-Versorgungsspannung getrennt und auf einen Pin herausgeführt werden.

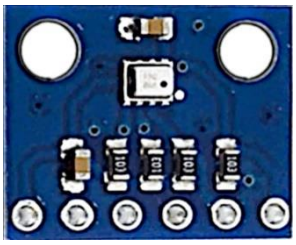


Das Display mit Keypad bietet neben der 16x2-Darstellung insgesamt 6 Tasten, von denen alle im aktuellen Projekt eine Steuerfunktion zu erfüllen haben. Fünf dieser Tasten liefern die Spannung von den Knotenpunkten einer Widerstandskaskade an einen ADC-Eingang (GPIO35) des ESP32. Die Pegel werden vom ESP32 decodiert und verschiedenen Aktionen zugeordnet. Dazu erfahren Sie später Genaueres.

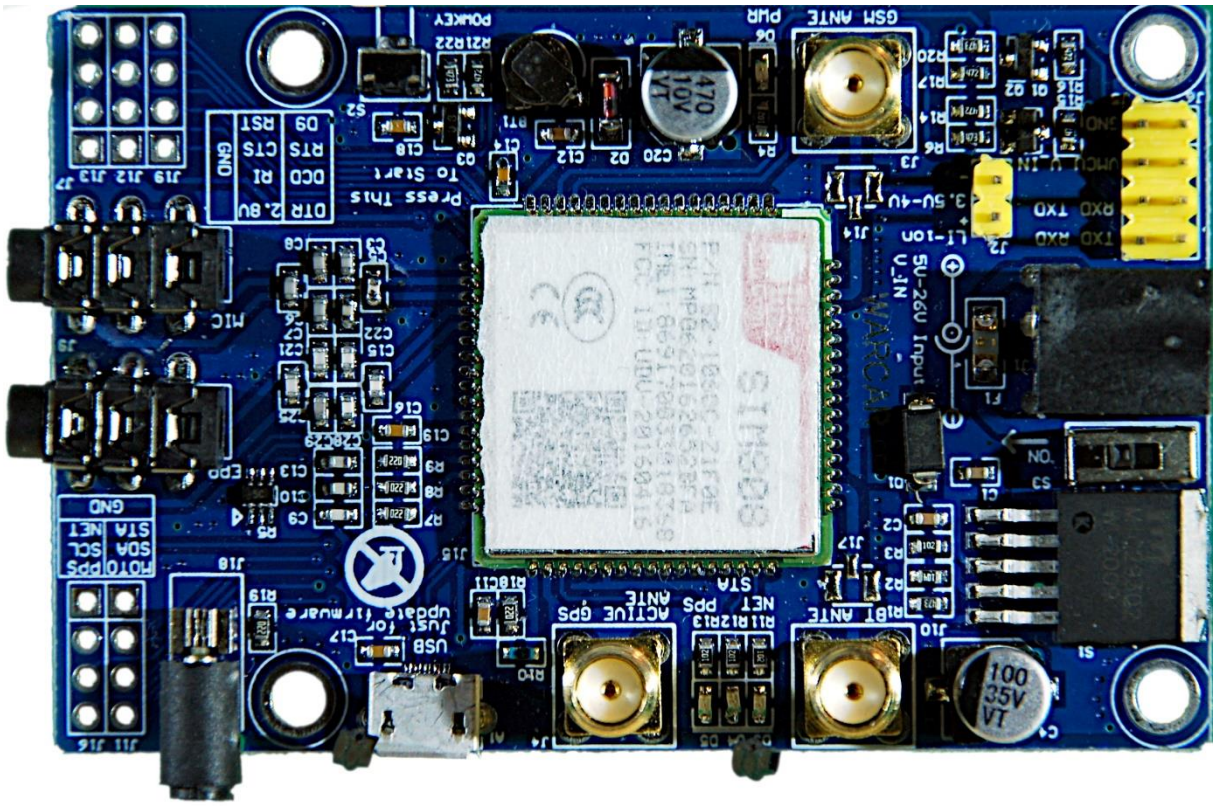
Die 6. Taste, RST, folgt nicht diesem Schema. Um diese Taste am ESP32 zu anderen Zwecken nutzen zu können, habe ich den Anschluss RST der Keypadplatine an den digitalen Eingang GPIO25 gewählt. Im ersten Teil war hier noch ein Pullup-Widerstand von 10kΩ im Einsatz. Diesen Widerstand habe ich entfernt und dafür den internen Pullup des Eingangs im ESP32 aktiviert. Wie diese Taste zur Erweiterung der Funktionalität der Schaltung eingesetzt wird, erfahren Sie weiter unten bei der Programmbesprechung.

Das Seriell-Parallel-Wandler-Modul für das LCD hat ein kleines Brüderchen in Form des BMP280 bekommen, das ebenfalls am I2C-Bus angeschlossen wird.

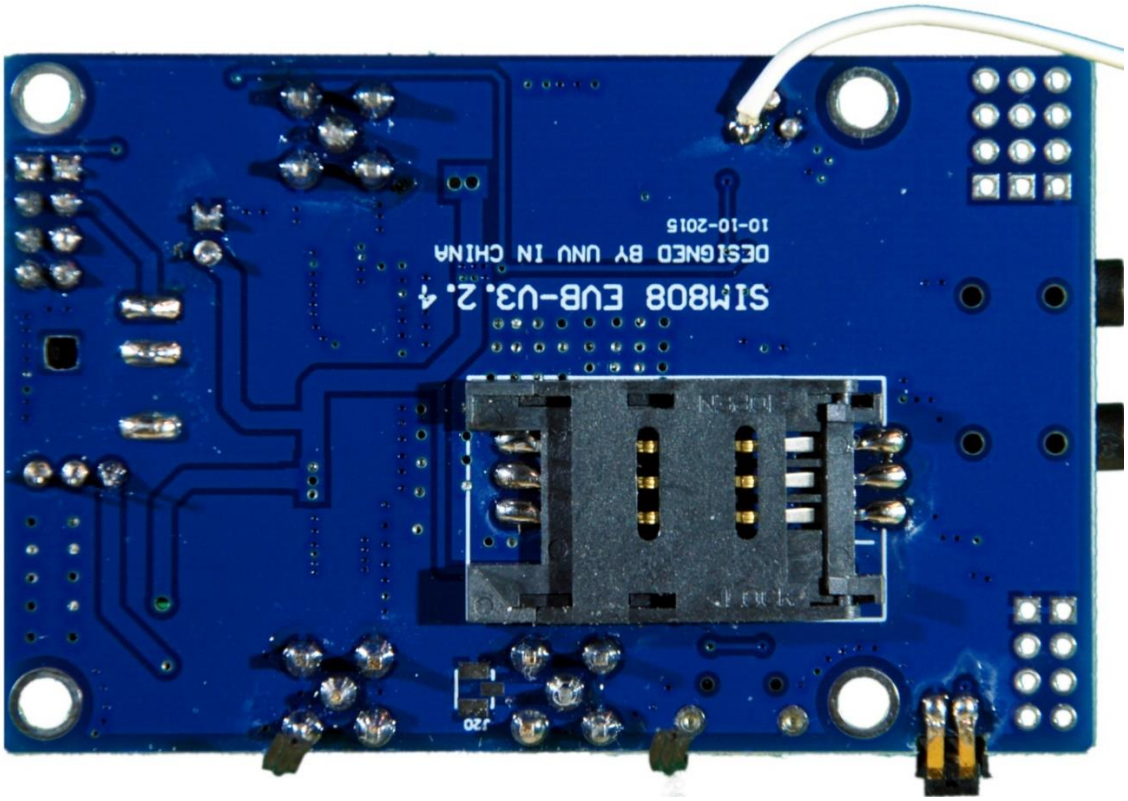




Beide Bausteine werden, wie der ESP32, mit 3,3V betrieben. Somit liegt auch der Ruhepegel der Busleitungen SCL und SDA auf 3,3V und es gibt kein Gefährdungspotential.



An der Beschaltung des SIM808 hat sich vom ersten Teil her nichts geändert. Dazu ist auch nicht viel zu sagen. Gerade mal vier Leitungen sind für die Verbindung zum ESP32 nötig, GND, TXD, RXD und die Leitung von der Starttaste des SIM808 zum GPIO4-Pin des ESP32. Die nächste Abbildung zeigt den Anschluss dieser Leitung am SIM808-Board.



Nach der Hardware werfen wir noch einen kurzen Blick auf die benötigte Software. Hier ist die Liste.

Verwendete Software:
Fürs Flashen und die Programmierung des ESP:
Thonny oder
µPyCraft
MicropythonFirmware

MicroPython-Module und Programme
GPS-Modul für SIM808 und GPS6MV2(U-Blocks)
LCD-Standard-Modul
HD44780U-I2C-Erweiterung zum LCD-Modul
Keypad-Modul
Button Modul
BMP208-Modul
i2cbus-Modul für standardisierten Zugriff auf den Bus
Das Hauptprogramm rambler.py
testkeypad.py zum Testen der Tastendecodierung

Tricks und Infos zu MicroPython

In diesem Projekt wird die Interpretersprache MicroPython benutzt. Der Hauptunterschied zur Arduino-IDE ist, dass Sie die MicroPython-Firmware auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang im ersten Teil des Blogs zu diesem Thema beschrieben.

Nachdem 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 compilieren und übertragen zu müssen. Bei der Entwicklung der Software für diesen Blog habe ich davon wieder reichlich Gebrauch gemacht. Das Spektrum reicht von einfachen Tests der Syntax bis zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen. Zu diesem Zweck habe ich mir für die verschiedenen Module jeweils ein Testprogramm erstellt, in welchem langwierige Import- und Konfigurationsaufgaben zusammenfasst sind. Aus einem davon ist schließlich das Programm rambler.py entstanden.

Gestartet werden solche Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5, das geht schneller als der Mausklick auf den Startbutton. Die Installation von Thonny habe ich schon im ersten Teil genau beschrieben.

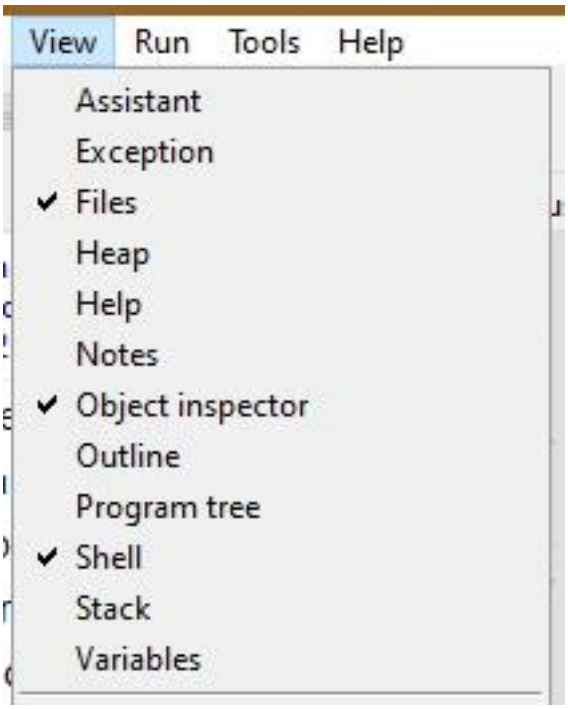
Einige weitere Shortcuts helfen beim Editieren der Programme. Python lebt von der Strukturierung des Programmtextes durch Einrückungen. Da kann es schon mal vorkommen, dass man ganze Passagen um eine (oder mehrere) Stufen ein- oder ausrücken muss. Damit nicht jede Zeile einzeln behandelt werden muss, markiert man den gesamten Block ab der ersten Spalte und drückt zum Einrücken die Tabulatortaste, zum Ausrücken Shift (Hochstellen) und Tabulatortaste. Das geht schnell und hilf Fehler zu vermeiden. Die Einrücktiefe bei Thonny ist 4. Zum Einrücken werden Leerzeichen benutzt, keine Tabulatorzeichen.

Auch als sehr hilfreich habe ich das Auskommentieren und Entkommentieren von mehreren Zeilen gleichzeitig empfunden. In Testprogrammen fasse ich meist mehrere Sequenzen für verschiedene Zwecke zusammen. Damit nicht stets alles ausgeführt wird, kommentiere ich einfach Zeilen aus, die gerade nicht verarbeitet werden sollen. Das Auskommentieren geschieht durch setzen eines "#" an den Beginn der Zeile. Für mehrere Zeilen kann das schnell lästig werden. Dann markiere ich den Zeilenblock und drücke Alt + 3 (nicht F3!). Um die Kommentarzeichen zu entfernen, drücke ich Alt + 4.

Um die Attribute von Objekten zu sehen, können Sie den Befehl dir() verwenden. So zeigt Ihnen der Befehl dir(ADC) die Attribute und Methoden der Klasse ADC, die Sie natürlich zuvor importieren müssen.

```
>>> from machine import ADC
>>> dir(ADC)
['__class__', '__name__', 'read', '__bases__', '__dict__', 'ATTN_0DB', 'ATTN_11DB', 'ATTN_2_5DB',
'ATTN_6DB', 'WIDTH_10BIT', 'WIDTH_11BIT', 'WIDTH_12BIT', 'WIDTH_9BIT', 'atten', 'read_u16', 'widt
h']
>>>
```

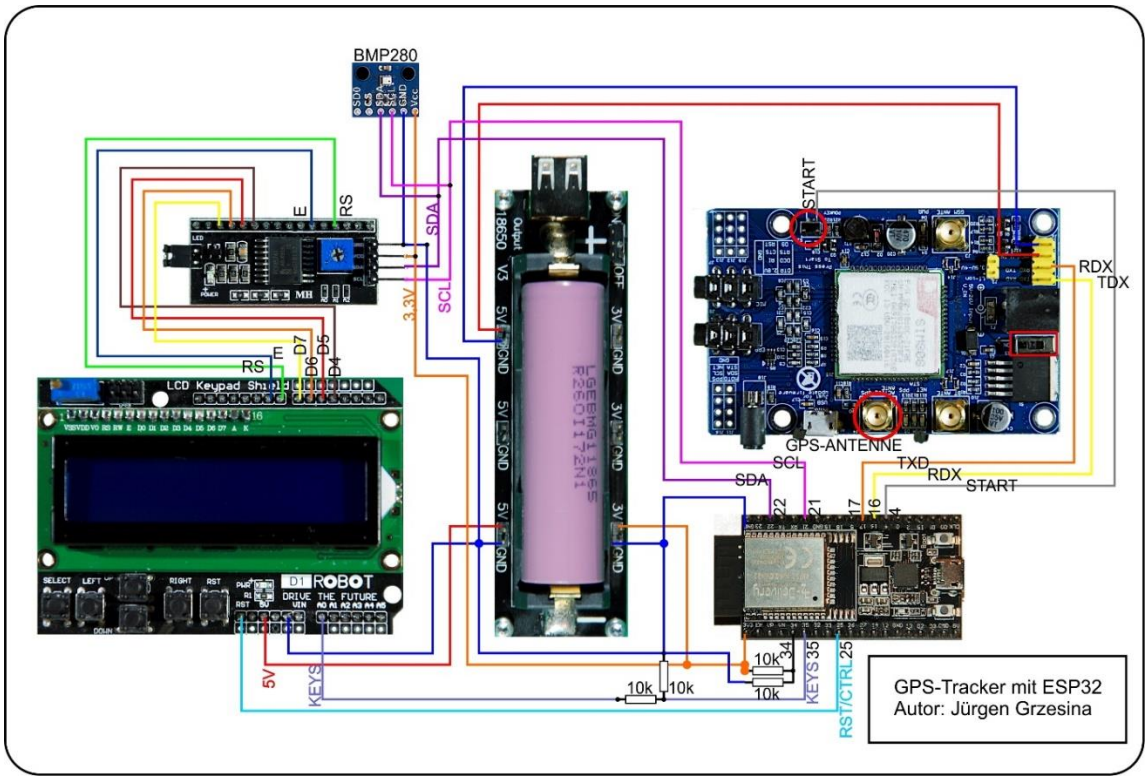
Übersichtlicher geht das über das Fenster **Object inspector**. Wenn es noch nicht dargestellt wird, können Sie es über das Menü **View** öffnen.



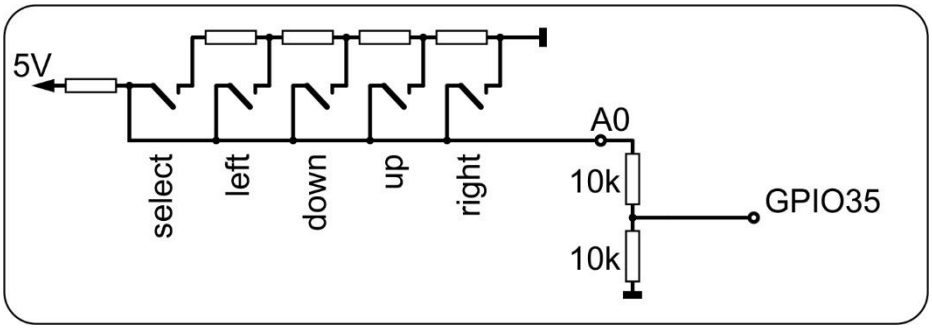
Der Object inspector hat zwei Ordner. **Attributes** ist der interessantere. Er stellt die Eigenschaften und Eigenschaftswerte von Objekten dar. Die Abbildung zeigt das für die Klasse ADC. Geben Sie im Terminalfenster ADC ein und drücken Sie Enter.

```
>>> ADC
>>>
```

Bevor wir jetzt die Programm-Module betrachten, müssen erst noch der BMP280 angeschlossen und die Spannungsmessung der 3,3V-Leitung vorbereitet werden. Orientieren Sie sich dazu bitte an folgendem Schaltschema. Ein besser lesbares Exemplar bekommen Sie mit dem Download der PDF-Datei .



Ein paar Bemerkungen zur Schaltung sollen deren Funktion klarstellen. Das LCD-Keypad wird mit 5V versorgt, das habe ich oben schon erwähnt. Der ESP32 arbeitet aber mit maximal 3,3V für die Versorgungsspannung und für die GPIO-Pins. Deshalb müssen die 5V-Pegel an den Ein- und Ausgängen des LCD und des Keypads auf ESP32-verträgliche Werte reduziert werden. Das geschieht durch den Spannungsteiler aus den beiden 10kΩ-Widerständen.



Die Abbildung zeigt eine mögliche Schaltung der Widerstandskaskade mit den Tasten SELECT, LEFT, UP, DOWN und RIGHT. Der erste Widerstand am heißen Ende liegt an 5V. Die Taster schalten den jeweiligen Pegel an den Anschluss A0 des Keypads durch, der somit im Leerlauf ca. 5V führt. Wir halbieren die hier anliegende Spannung durch einen Spannungsteiler aus zwei 10kΩ-Widerständen auf verträgliche 2,5V. Den Mittelabgriff verbinden wir mit dem Analogeingang GPIO35 des ESP32. Weitere Anpassungen nehmen wir in der Klasse KEYPAD vor.

So, die Hardware steht, dann lassen Sie uns die Tasten am LCD Keypad testen. Starten wir erst einmal zu Fuß mit REPL, der MicroPython Kommandozeile, im Terminalbereich. Wir importieren die **ADC**-Klasse und die **Pin**-Klasse vom integrierten Modul **machine**, erzeugen ein ADC-Objekt an GPIO35 und stellen dessen Eigenschaften auf 12-Bit Breite (0…4095) und maximalen Messbereich, ADC.ATTEN_11DB, womit der ESP32 eine maximale Spannung von ca. 3,3V erfassen kann.

```
>>> from machine import ADC,Pin
>>>a=ADC(Pin(35))
>>>a.atten(ADC.ATTN_11DB)
>>>a.width(ADC.WIDTH_12BIT)
>>>a.read()
```

Das Ergebnis des letzten Befehls sollte einen Wert um die 2500 liefern. Wiederholen Sie den Lesebefehl, drücken Sie aber vorher jeweils eine der Tasten. Bei mir kamen folgende Werte zum Vorschein:

SELECT:	1750
LEFT:	1150
DOWN:	670
UP:	200
RIGHT:	0

Für eine einfachere Abfrage der Tasten im Hauptprogramm, habe ich um diese Werte ein Modul gebaut. Es enthält die Klasse **KEYPAD**, die ihrerseits zwei

Methoden besitzt, den Constructor, das ist die Methode **__init__()** und die Methode **key(). __init__()** bearbeitet bei jeder Klasse grundsätzliche Dinge, erzeugt Instanzvariablen (aka Attribute), definiert Schnittstellen und erzeugt somit beim Aufruf die Umgebung für ein Objekt aus dem Bauplan dieser Klasse. Hier wird ein ADC-Objekt definiert und eine erste Messung angestoßen. Der Aufruf der Instanzmethode key() aus der Klassendefinition heraus ist an dieser Stelle noch nicht möglich, weil die Bereichsliste keyRange noch nicht definiert ist. Deshalb kann die erste Messung zum Ankurbeln des ADC nicht durch Aufruf der key-Methode erfolgen.

Danach bestimmen wir den Kalibrierfaktor k und bauen die Liste mit den Bereichen auf, die einerseits den Richtwerten der Tasten folgen und andererseits einen Spielraum unter Berücksichtigung des Kalibrierfaktors offenlassen. Die fast grenzenlose Kombinationsmöglichkeit von Objekten in MicroPython ermöglich derart übersichtliche Programmierung. Die formatierte Textausgabe unter Einbindung von numerischen Variablen schließt den Konstruktor ab.

Die Methode **key**() liefert als Rückgabewert eine Ganzzahl ab 0 bis inclusive4., die einer der Tasten entspricht. Um den Umgang damit zu erleichtern, habe ich die Klassenattribute **Right**, **Up**, **Down**, **Left** und **Select** als Konstanten definiert, die anstelle der Zahlen verwendet werden können.

```
from machine import ADC,Pin

class KEYPAD:
    Right=const(0)
    Up=const(1)
    Down=const(2)
    Left=const(3)
    Select=const(4)

    def __init__(self, pin=35):
        self.a=ADC(Pin(pin))
        self.a.atten(ADC.ATTN_11DB)
        self.a.width(ADC.WIDTH_12BIT)
        self.a.read()  # erst mal Messung initialisieren
        # keyValues: 0,200,680,1100,1750,2500
        adcMax=(self.a.read()+self.a.read()+self.a.read())//3
        k=adcMax/2500
        self.keyRange=[range(0,int(75*k)),            # right
                       range(int(100*k),int(300*k)),  # up
                       range(int(440*k),int(850*k)),  # down
                       range(int(900*k),int(1300*k)), # left
                       range(int(1450*k),int(2000*k)),# select
                       ]
        print("KEYPAD initialized, Leerlauf: {}, k=
{}".format(adcMax,k))

    def key(self):
        s=0
        for i in range(5):
            s+=self.a.read()
        m=s//5
```

```
        for i in range(5):
            if m in self.keyRange[i]: return i
        return 5
```

So arbeitet der Konstruktor
Die Tastenwerte schwanken einerseits durch Messfehler des ADC (aka Analog-Digital-Converter), andererseits durch Unterschiede in der Versorgungsspannung. Deshalb nimmt der Constructor künftiger Keypad-Objekte beim Aufruf eine Kalibrierung vor. Mit dem Faktor k werden die Grenzwerte der Tastenerkennung an den Leerlaufwert ohne Tastendruck angepasst. Die Grenzwerte habe ich nach meinen ersten Messungen mit Hilfe eines kleinen Testprogramms (testkeypad.py) so festgelegt, dass die Bereiche sich nicht überlappen. Diese **range**-Objekte (in MicroPython ist alles ein Objekt) habe ich im **Listen**-Objekt **keyRange** zusammengefasst. Durch einen Listenzeiger, genannt **Index**, werden die einzelnen Felder angesprochen.

Die Klasse als Bauplan eines Objekts (aka Instanz) legt die Zutaten für dieses Objekt (aka Instanz) fest. Jedes Objekt, das zur Instanz selbst gehören soll, erhält die Vorsilbe (aka Präfix) **self**. Self vertritt in diesem Zusammenhang den Namen des Objekts, das später von der Klasse abgeleitet wird. Bei Funktionen, die in diesem Konsens in der Objekt-Orientierten-Programmierung als Methoden bezeichnet werden, tragen dieses self als ersten Parameter in der Parameterliste. Es kann auch der einzige Parameter sein, darf aber nicht weggelassen werden. Instanz-Methoden werden, wie alle Funktionen, durch das Schlüsselwort **def** deklariert. Wird eine Instanz-Methode innerhalb der Klassendefinition aufgerufen, dann muss dem Namen der Methode auch ein self vorangestellt werden. Das Prefix self wird stets durch einen Punkt vom nachfolgenden Namen abgetrennt. Der Konstruktor in Form der Methode __init__() legt die Instanz-Variablen, deren Anfangswert und weitere Objekte beim Aufruf an. Von außen trägt der Konstruktor den Namen der Klasse. Diese Zusammenhänge können Sie an der Klassendefinition KEYPAD und am Testprogramm gut erkennen.

Wie arbeitet die Methode **key**()? Ich lasse mir, um Streuungen des ADC zu verringern, den Mittelwert von 5 einzelnen Messungen bilden. Das erledigt die erste for-Schleife. In der folgenden for-Schleife prüfe ich, ob der Mittelwert in dem Bereich liegt, der durch den Index angesprochen wird. Wenn ja, wird der Index als Funktionswert zurückgegeben, er codiert die Auswahl. Liegt der ADC-Wert nicht im adressierten Bereich, dann wird der nächste Bereich aus keyRange geprüft. Hat keiner der Bereiche entsprochen, dann wurde offenbar keine Taste gedrückt und für diesen Fall der Wert 5 zurückgegeben.

Neben weiteren Quellen sind Bereiche auch als Indexpool für for-Schleifen nötig, wie im Fall der Methode key(). Deshalb muss man wissen, dass zu einem Bereich stets alle ganzzahligen Werte vom ersten inclusive bis zum zweiten exclusive zählen. Freilich gibt es noch diverse weitere Möglichkeiten zur Festlegung von Bereichen, dazu ein anderes Mal mehr. Zur Einführung ein paar Beispiele:

Es ist
range(0,5) = range (5) = 0,1,2,3,4
range(23,24) = 23

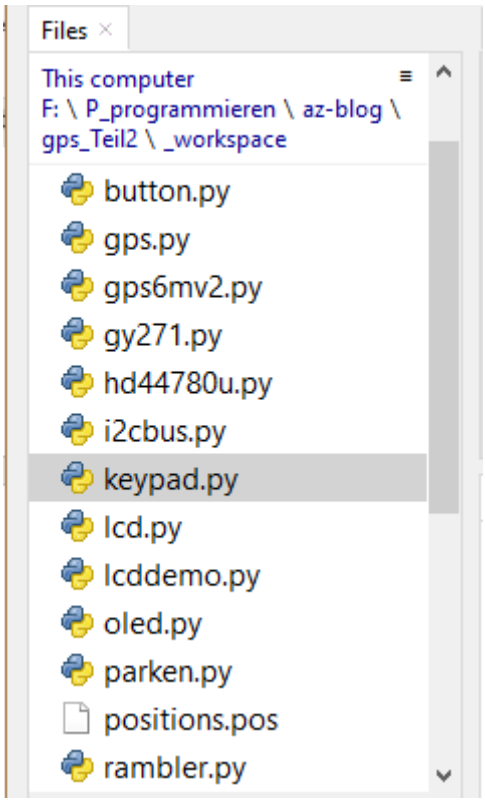
Und das geht auch

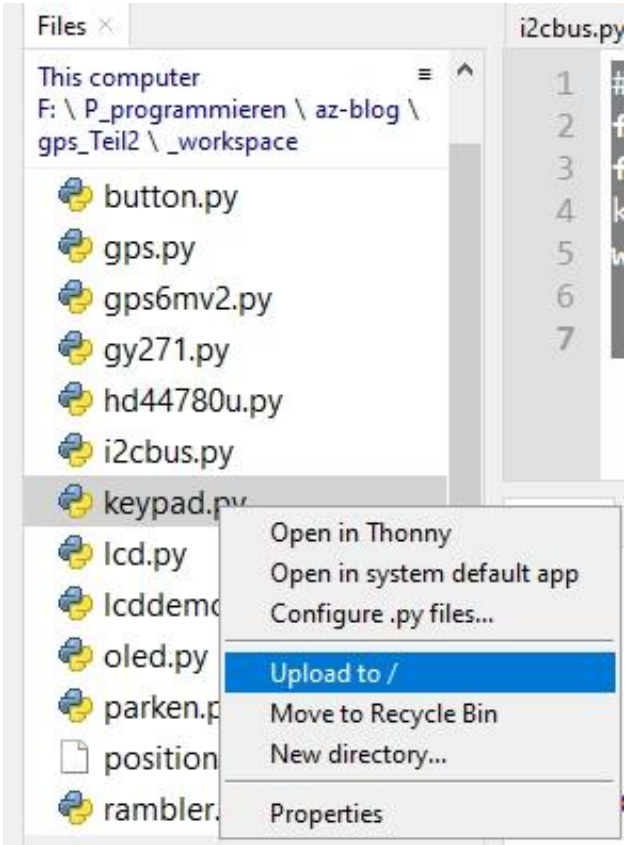
```
>>> Liste=[0,1,2,3,4,5,6]
>>> for i in range(4):
    print(Liste[i])

0
1
2
3
>>> for i in range(3,6):
    print(Liste[i])

3
4
5
>>> for i in range(1,7,2):
    print(Liste[i])

1
3
5
>>>
```

Testen wir jetzt die Klasse KEYPAD. Wir nutzen dazu das oben erwähnte Testprogramm testkeypad.py. Hier sehen Sie das sehr kurze Listing.

```
#testkeypad
from keypad import KEYPAD
from time import sleep
k=KEYPAD(35)
while 1:
    print(k.a.read(),k.key())
    sleep(1)
```

Bringen Sie keypad.py und testkeypad.py ins Arbeitsverzeichnis von Thonny und kopieren Sie keypad.py zum ESP32.





Öffnen Sie testkeypad.py in einem Editorfenster und starten Sie das Programm mit der Taste F5

Wenn Sie keine Taste gedrückt halten, erscheint jetzt im Terminal eine Spalte von Fünfen

5
5
5
5…

Drücken Sie die Tasten der Reihe nach und kontrollieren Sie die Ausgabe. Das Programm brechen Sie mit Strg + C ab, wenn der Cursor sich im Terminalfenster befindet.

k ist im Testprogramm der Name der KEYPAD-Instanz, er ersetzt das Schlüsselwort self. Die Liste **keyRange** ist eine Eigenschaft (aka Attribut) der Instanz k. In der Klassendefinition erkennen Sie das am Prefix **self**. Beim Objekt k ersetzt der Name k dieses self. Die Liste keyRange können Sie folglich wie folgt abrufen (aka referenzieren).

```
>>> k.keyRange
[range(0, 73), range(97, 292), range(428, 827), range(876, 1266), range(1412, 1948)]
>>>
```

Sollten die Zahlzuordnungen zu den Tasten nicht stimmen, RIGHT liefert 0, UP eine 1, DOWN eine 2 und LEFT und SELECT 3 und 4, dann liegt das sehr wahrscheinlich an falsch gelegten Grenzwerten im Constructor. Sie müssen dann die Tastenwerte neu bestimmen und die Grenzwerte sinnvoll festlegen. Nachdem der Konstruktor von KEYPAD das ADC-Objekt a erzeugt hat, können Sie dieses benutzen, um die ADC-Werte der Tasten abzufragen. Sie weisen damit den ADC an, die Spannung am Eingang GPIO35 zu messen und als Ganzzahl auszugeben. Mit der Einstellung **a.width(ADC.WIDTH_12BIT)** erhalten Sie Zahlenwerte zwischen 0 und 4095 inclusive.

```
>>> k.a.read()
2438
```

Kommen wir zum Display. Das Keypad hat außer dem Display die 6 Tasten zu bieten. Das war der Grund für die Auswahl dieses Bausteins. Auf der anderen Seite setze ich auch gerne OLED Displays ein; sie sind kleiner und lassen auch einfache grafische Darstellungen zu. Damit ein Programm ohne Änderungen sowohl mit LCDs als auch OLEDs zurechtkommt, habe ich für beide Fälle jeweils ein Modul entwickelt, das dieselbe Benutzeroberfläche (aka API) verwendet. Dahinter stehen dann weitere Module, die sich um hardwarespezifische Befehle einerseits und um die Kommunikation mit der Hardware andererseits kümmern. Beides ist für den Anwender meist uninteressant und wird im Hintergrund durch Libraries oder Module erledigt, die man gerne als Gerätetreiber bezeichnet.

In der Regel kommt man meist ohne diese gerätespezifischen Befehle aus. Es ist aber bei den vorliegenden Modulen stets so, dass dennoch die gesamte Befehlsstruktur quer durch alle Klassen letztlich dem Anwender zur Verfügung steht. Möglich wird das durch die Vererbung von Klassen. Damit befinden sich alle wesentlichen Befehle, vertreten durch die Methoden der Klassen, im gleichen Namensraum. Die Klasse LCD erbt von PCF8574U und diese wiederum von

HD44780U. Letztlich stehen alle Methoden aus jeder der drei Klassen zur Verfügung, wenn man lediglich die Klasse LCD in folgender Weise importiert. Die Konstruktoren des Vaters (PCF8574U) und Großvaters (HD44780U) der Klasse LCD melden sich bei der Instanzierung des Objekts d dieser Klasse der Reihe nach.

```
>>> from lcd import LCD
>>> from machine import I2C,Pin
>>> i2c=I2C(-1,Pin(21),Pin(22))
>>> d=LCD(i2c,0x27,16,2)
this is the constructor of HD44780U class
Size:16x2
Construktor of PCF8574U class
this is the constructor of LCD class
Size:16x2
>>> d
<LCD object at 3ffe9540>
>>> dir(d)
```

Mit dem letzten Befehl erhalten Sie eine umfangreiche Liste von Methoden und Attributen, deren Namen Sie in einer der drei Klassendefinitionen aufspüren können. Eine schönere Auflistung erhalten Sie im Object inspector.

Für die LCD-Klasse stehen also oberflächlich dieselben Befehle zur Verfügung wie für die Klasse OLED. Methoden, die in einer der Klassen keinen Sinn machen, sind so abgesichert, dass einfach nichts passiert, also auch kein Programmabbruch und keine Fehlermeldung. Für das vorliegende Projekt bedeutet das, dass Sie jederzeit die Anzeige auf ein OLED-Display lenken könnten, wenn Sie denn eine andere Lösung für die Tastensteuerung finden. Die Methodenaufrufe zum Display müssen nicht geändert werden, lediglich der Constructoraufruf bedarf einer Anpassung und natürlich müssen Sie die OLED-Klassen importieren

Als Nächstes können Sie das Modul **lcd.py** studieren und testen. Ich verwende für die Verdrahtung einen Adapter mit dem Chip PCF8574, der I2C-Signale des ESP32 auf die parallele Ausgabe zum LCD umsetzt. Sie steuern also über 2 Busleitungen die 2 + 4 (+2) Steuerleitungen des LCD-Moduls. Und das inklusive Pegelwandlung von 3,3V I2C des ESP32 zu 5V LCD-Eingang am Keypad. Die Anschlussverteilung ist wie folgt (siehe auch Schaltschema).

PCF8574 Bit	0	1	2	3	4	5	6	7
LCD-Bit	RS	RW	E	BL	D4	D5	D6	D7
Keypad Shield	D8		D9		D4	D5	D5	D7

Gemäß der Verwendung des oberen Daten-Nibbles am Chip wird die Klasse **PCF8574U** importiert. **U** steht für **UPPER**.

```
"""
#### Datei: lcd.py
#### Author: Juergen Grzesina
#### Verwendung:
Die Klasse LCD stellt die gleiche API wie OLED bereit, sodass
beide
```

```
Displays ohne Aenderung am Programm austauschbar sind.
Folgende Methoden stehen in der Klasse LCD bereit:
# LCD(i2c,Spalten,Zeilen)
# writeAt(string,xpos,ypos, show=True)
# clearFT(xv,yv[,xb=spalte][,yb=zeile, show=True)
# clearAll()
# (pillar(xpos,breite,hoehe, show=True))
# (setKontrast(wert))
# (xAxis(show=True), yAxis(show=True))
# switchOn(), switchOff()
Die grafischen Methoden sind aus Kompatibilitaetsgruenden
vorhanden
haben aber keine Funktion auf dem Text-LCD.
"""
from hd44780u import PCF8574U as PCF
#from hd44780u import PCF8574L as PCF

class LCD(PCF):

  CPL = const(20)
  LINES = const(4)
  HWADR =const(0x27)

  def __init__(self, i2c, adr=HWADR, cols=CPL, lines=LINES):
    #ESP32 Pin assignment
    super().__init__(i2c,adr,cols,lines)
    self.adr=adr
    self.columns = cols
    self.rows = lines
    self.name="LCD"
    self.clear()
    print("this is the constructor of LCD class")
    print("HWADR={:#X}, Size:{}x{}".format(self.adr,
self.columns, self.rows))

  # Put string s at position col x row y from left upper
corner 0; 0
  # for
  # x = column 0..19
  # y = row 0..3
  def writeAt(self,s,x,y,show=True):
    if x >= self.columns or y >= self.rows: return None
    text = s
    length = len(s)
    if x+length < self.columns:
      b = length
    else:
      b = (self.columns - x)
      text = text[0:self.columns-x]
    self.printAt(text,x,y)
    return (x+length if x+length < self.columns else None)
```

```
def clearAll(self, show=True):
  self.clear()

def setKontrast(self,k):
  pass

def pillar(self,x,b,h, show=True):
  pass

def xAxis(self, show=True):
  pass

def yAxis(self, show=True):
  pass

def switchOff(self):
  self.display(0)

def switchOn(self):
  self.display(1)

def clearFT(self,x,y,xb=None, yb=None, show=True):
  if yb!=None and yb>=self.rows: return None
  if xb==None: xb=self.columns-1
  if xb >= self.columns:
    xb = self.columns-1
  blanks=" "*(xb-x+1)
  self.printAt(blanks,x,y)
  self.position(x,y)
  return x
```

Die wichtigen Methoden aus diesem Modul sind der **Konstruktor** und **clearAll**() sowie **writeAt**().

Im Constructor __init__(), der durch LCD() im Programm aufgerufen wird, übergeben wir ein I2C-Objekt, das bereits im Hauptprogramm definiert worden sein muss. Im aktuellen Projekt wird der Bus auch für ein BMP280-Modul zur Messung von Luftdruck und Temperatur eingesetzt. Zu diesem Modul komme ich später.

i2c=I2C(-1,Pin(21),Pin(22))

Dann wird die Hardwareadresse des PCF8574 übergeben, falls diese von der Defaultadresse 0x27 abweicht. Es folgen Anzahl der Spalten und Zeilen. Im Hauptprogramm sieht das so aus.

d=LCD(i2c,0x27,cols=16,lines=2)

Die Methode **writeAt**() nimmt als ersten Parameter den auszugebenden String, gefolgt von der Spalten- und Zeilenposition im Display. Beachten Sie bitte, dass beide Zählungen bei Null beginnen.

Die Methode

clearAll()

löscht das gesamte Display und setzt den Cursor an die Position 0,0 (=linke obere Ecke = homeposition).

Gut, das Wichtigste für eine GPS-Anwendung ist: Wie spreche ich die GPS-Dienste des SIM808 an?

Drei Stufen führen zum Erfolg. Die erste Stufe ist rein manueller Art, Sie müssen den kleinen Schiebeschalter gleich neben der Rohrbuchse für die Spannungsversorgung in Richtung SIM808-Chip schieben. Eine rote LED leuchtet neben der GSM-Antennenbuchse auf.

Ein Stück weiter links davon befindet sich die Starttaste. Drücken Sie diese ca. 1 Sekunde lang, dann leuchten zwischen den anderen beiden Antennenbuchsen zwei weitere LEDs auf, die rechte davon blinkt. An die linke Schraubbuchse sollte bereits die aktive GPS-Antenne angeschlossen sein. Diese legen Sie am besten in die Nähe eines Fensters.

Damit Sie jetzt nicht jedes Mal das Gehäuse ihres GPS-Empfängers öffnen müssen, um das SIM808 zu starten, empfehle ich Ihnen, es mir gleich zu tun und ein Kabel an den heißen Anschluss des Starttasters zu löten. Von oben betrachtet ist es der rechte, wenn die Rohrbuchse ebenfalls nach rechts zeigt. Sie können nun das SIM808 starten, indem Sie einen GPIO-Pin des ESP32 als Ausgang definieren und für eine Sekunde von High nach Low und zurück auf High schalten. Ich habe dafür den Pin 4 vorgesehen.

Beim Aufruf des Konstruktors für das GPS-Objekt wird die Nummer des Pins zusammen mit dem Displayobjekt als Parameter übergeben.

```
>>> from gps import GPS,SIM808
>>> g=SIM808(4,d)
```

Wird kein Displayobjekt (d) übergeben, erfolgt auch keine Ausgabe auf LCD oder OLED. Es erfolgt keine Fehlermeldung, die Tastensteuerung arbeitet aber. Bei fast allen wichtigen Ergebnissen erfolgt auch eine Ausgabe im Terminalfenster.

Die Klasse GPS erledigt die Hauptarbeit. Der Constructor erwartet, wie erwähnt, ein Displayobjekt, das im aufrufenden Programm definiert oder bereits bekannt sein muss. Es wird ein serieller Kanal zum SIM808 auf 9600 Baud, 8,0,1 geöffnet, dann werden die Instanzvariablen für die Aufnahme der GPS-Daten eingerichtet.

Die Methode **waitForLine**() tut, was ihr Name sagt, sie wartet auf einen NMEA-Satz vom SIM808. Als Parameter wird der Typ des NMEA-Satzes angegeben, der erwartet wird. Ist der Satz vollständig und fehlerfrei, wird er an das aufrufende Programm zurückgegeben. Es können in der gegenwärtigen Ausbaustufe des Programms $GPRMC- und $GPGGA-Sätze empfangen werden. Sie enthalten alle relevanten Daten wie Gültigkeit, Datum, Zeit, geographische Breite (Latitude, vom

Äquator aus bis zu den Polen in Grad) und Länge (Longitude vom Null-Meridian aus +/- 180°) sowie Höhe über NN in Metern.

Die Methode **decodeLine**() nimmt den empfangenen Satz und versucht ihn zu decodieren. Diese Methode enthält eine lokale Funktion, die nach Vorgabe des Attributs **Mode** die Winkelangaben in die Formate **Grad Minuten Sekunden und Bruchteile**, **Grad und Bruchteile** oder **Grad Minuten und Bruchteile** umwandelt.

Die Methode **printData**() gibt einen Datensatz im Terminalfenster aus. **showData**() liefert das Ergebnis an das Display. Weil nur ein zweizeiliges Display verwendet wird, muss die Anzeige in mehrere Abschnitte aufgeteilt werden. Die Tasten des Keypads übernehmen die Steuerung.

Die Methoden der Klasse SIM808 interessieren Sie jetzt sicher am meisten, denn dadurch kommen die GPS-Daten ja erst in den ESP32 zur Verarbeitung.

Grundsätzlich erfolgt der Datenaustausch zwischen ESP32 und SIM808 über eine serielle Datenverbindung mit 9600 Baud, 8,0,1. Das bedeutet, es werden pro Sekunde 9600 Bit übertragen, wobei ein Datenrahmen (aka Frame) aus 8 Datenbits, 0 Paritätsbits und einem Stoppbit besteht. Das Startbit ist obligatorisch und wird in dieser Aufzählung nicht erwähnt. Ein Dataframe umfasst somit 10 Bits, das LSB (aka Least Significant Bit = niederwertiges Bit) wird als erstes nach dem Startbit (0) übertragen. Das Stoppbit (1) schließt die Übertragung ab. Auf TTL-Niveau entspricht eine 1 dem Pegel 3,3V, der 0 der GND-Pegel.

Weil die UART0-Schnittstelle für REPL reserviert ist, muss eine zweite Schnittstelle für die Kommunikation mit dem SIM808 vorhanden sein. Der ESP32 stellt eine solche als UART2 bereit. Die Anschlüsse für RXD (Empfang) und TXD (Sendung) können sogar frei gewählt werden. Für einen Vollduplexbetrieb (senden und empfangen gleichzeitig) müssen die Anschlüsse RXD und TXD vom ESP32 zum SIM808 gekreuzt werden. Sie können das am Schaltplan nachvollziehen. Die Defaultwerte am ESP32 sind RXD=16 und TXD=17. Die Organisation des Anschlusses übernimmt die Klasse gps.GPS.

Das beginnt mit dem Einschalten des SIM808. Wenn Sie meiner Empfehlung gefolgt sind und ein Kabel an den Einschalttaster gelötet haben, können Sie das SIM808 jetzt mit folgendem Befehl einschalten, vorausgesetzt, dass dieses Kabel am Pin 4 des ESP32 liegt.

```
>>> g.SIMOn()
```

Befehle an das SIM808 werden im AT-Format übermittelt. Es gibt eine riesige Auswahl von Befehlen, die in einer PDF-Datei nachgelesen werden können. Aber keine Sorge, für unser Projekt reichen im Prinzip wenige Befehle. Sie sind in den Methoden **init808**() und **deinit808**() zusammengefasst.

```
    def init808(self):
        self.u.write("AT+CGNSPWR=1\r\n")
        self.u.write("AT+CGNSTST=1\r\n")

    def deinit808(self):
```

self.u.write("AT+CGNSPWR=0\r\n")
self.u.write("AT+CGNSTST=0\r\n")

AT+CGNSPWR=1 schaltet die Stromversorgung zum GPS-Modul ein und AT+CGNSTST=1 aktiviert die Übertragung der NMEA-Sätze zum ESP32 über die serielle Schnittstelle UART2. Der Controller empfängt die Informationen des SIM808 und stellt sie in der oben beschriebenen Weise via Terminal und LCD bereit.

Das Modul **gps**.py enthält neben der Hardwaresteuerung des SIM808 auch noch die nötigen Befehle für das kleinere GPS-System GPS6MV2 mit dem Chip Neo 6M von UBLOX. Die Steuerung dieses Moduls erfolgt nicht über AT-Befehle sondern über eine eigene Syntax.

Zum genaueren Studium des gps-Moduls folgt nun das Listing. Es enthält eine ganze Reihe von Funktionen, die erst im dritten Teil des Blogs zum Einsatz kommen, wo es um die Versendung und den Empfang von SMS-Nachrichten geht. Für den Beitrag, den Sie gerade lesen, ist, so wie für ersten Teil, noch keine SIM-Karte nötig, die brauchen Sie erst im dritten Teil der Blogreihe.

Die Klasse SIM808 kümmert sich um die Hardwaresteuerung und um den Datentransfer zum ESP32. Die Klasse GPS enthält Methoden zur Decodierung der NMEA-Sätze vom SIM808, zur Darstellung auf dem Display und zur Kursberechnung.

```
"""
File: gps.py
Author: J. Grzesina
Rev. 1.0: AVR-Assembler
Rev. 2.0: Adaption auf Micropython
------------------
Die enthaltenen Klassen sprechen einen ESP32 als Controller
an.
Dieses Modul beherbergt die Klassen GPS, GPS6MV2 und SIM808
GPS stellt Methoden zur Decodierung und Verarbeitung der NMEA-
Saetze
$GPGAA und $GPRMC bereit, welche die wesentlichen Infos zur
Position, Hoehe und Zeit einer Position liefern. Sie werden
dann
angezeigt, wenn die Datensaetze als "gueltig" gemeldet werden.
Eine Skalierung auf weitere NMEA-Sätze ist jederzeit möglich.
GPS6MV2 und SIM808 beziehen sich auf die entsprechende
Hardware.
"""
from machine import UART,I2C,Pin
import sys
from time import sleep, time, ticks_ms
from math import *

# ********************** Beginn GSM
**************************PS
class GPS:
    #
```

```
    gDeg=const(0)
    gFdeg=const(1)
    gMin=const(1)
    gFmin=const(2)
    gSec=const(2)
    gFsec=const(3)
    gHemi=const(4)

    #DEFAULT_TIMEOUT=500
    #CHAR_TIMEOUT=200

    def __init__(self,disp=None,key=None):  # display mit
OLED-API
        self.u=UART(2,9600)
        # u=UART(2,9600,tx=19,rx=18)  # mit alternativen Pins
        self.display=disp
        self.key=key
        self.timecorr=2
        self.Latitude=""
        self.Longitude=""
        self.Time=""
        self.Date=""
        self.Height=""
        self.Valid=""
        self.Mode="DMF" # default
        self.AngleModes=["DDF","DMS","DMF"]
        self.displayModes=["time","height","pos"]
        self.DMode="pos"
        # DDF = Degrees + DegreeFractions
        # DMS = Degrees + Minutes + Seconds + Fractions
        # DMF = Degrees + Minutes + MinuteFraktions
        self.DDLat=49.123456  # aktuelle Position
        self.DDLon=11.123456
        self.DDLatOld=49.1233 # vorige Position
        self.DDLonOld=11.1233
        self.zielPtr=0
        self.course=0
        self.distance=0
        print("GPS initialized,
Position:{},{}".format(self.DDLat,self.DDLon))

    def decodeLine(self,zeile):
        latitude=["","","","","N"]
        longitude=["","","","","E"]
        angleDecimal=0
        def formatAngle(angle):  # Eingabe ist Deg:Min:Fmin
            nonlocal angleDecimal
            minute=int(angle[1])     # min als int
            minFrac=float("0."+angle[2]) # minfrac als float

angleDecimal=int(angle[0])+(float(minute)+minFrac)/60
            if self.Mode == "DMS":
```

```
                seconds=minFrac*60
                secInt=int(seconds)
                secFrac=str(seconds - secInt)

a=str(int(angle[0]))+"*"+angle[1]+"'"+str(secInt)+secFrac[1:6]
+'"'+angle[4]
            elif self.Mode == "DDF":
                minutes=minute+minFrac
                degFrac=str(minutes/60)
                a=str(int(angle[0]))+degFrac[1:]+"* "+angle[4]
            else:

a=str(int(angle[0]))+"*"+angle[1]+"."+angle[2]+"' "+angle[4]
            return a

        # GPGGA-Fields
        nmea=[0]*16
        name=const(0)
        time=const(1)
        lati=const(2)
        hemi=const(3)
        long=const(4)
        part=const(5)
        qual=const(6)
        sats=const(7)
        hdop=const(8)
        alti=const(9)
        auni=const(10)
        geos=const(11)
        geou=const(12)
        aged=const(13)
        trash=const(14)
        nmea=zeile.split(",")
        lineStart=nmea[0]
        if lineStart == "$GPGGA":

self.Time=str((int(nmea[time][:2])+self.timecorr)%24)+":"+nmea
[time][2:4]+":"+nmea[time][4:6]
            latitude[gDeg]=nmea[lati][:2]
            latitude[gMin]=nmea[lati][2:4]
            latitude[gFmin]=nmea[lati][5:]
            latitude[gHemi]=nmea[hemi]
            longitude[gDeg]=nmea[long][:3]
            longitude[gMin]=nmea[long][3:5]
            longitude[gFmin]=nmea[long][6:]
            longitude[gHemi]=nmea[part]
            self.Height,despose=nmea[alti].split(".")
            self.Latitude=formatAngle(latitude)  # mode =
Zielmodus Winkelangabe
            self.DDLat=angleDecimal
            self.Longitude=formatAngle(longitude)
            self.DDLon=angleDecimal
```

```
        if lineStart == "$GPRMC":
            date=nmea[9]
            self.Date=date[:2]+"."+date[2:4]+"."+date[4:]
            try:
                self.Valid=nmea[2]
            except:
                self.Valid="V"

    def waitForLine(self,title,delay=2000):
        line=""
        c=""
        d=delay
        if delay < 1000: d=1000
        start = ticks_ms()
        end=start+d
        current=start
        while current <= end:
            if self.u.any():
                c=self.u.read(1)
                if ord(c) <=126:
                    c=c.decode()
                    if c == "\n":
                        test=line[0:6]
                        if test==title:
                            #print(line)
                            return line
                        else:
                            line=""
                    else:
                        if c != "\r":
                            line +=c
            current = ticks_ms()
        return ""

    def showData(self):
        if self.display:
            if self.DMode=="time":

self.display.writeAt("Date:{}".format(self.Date),0,0)

self.display.writeAt("Time:{}".format(self.Time),0,1)
            if self.DMode=="height":
                self.display.writeAt("Height: {}m
".format(self.Height),0,0)

self.display.writeAt("Time:{}".format(self.Time),0,1)
            if self.DMode=="pos":
                self.display.writeAt(self.Latitude+" "*(16-
len(self.Latitude)),0,0)
                self.display.writeAt(self.Longitude+" "*(16-
len(self.Longitude)),0,1)
```

```
    def printData(self):
        print(self.Date,self.Time,sep="_")
        print("LAT",self.Latitude)
        print("LON",self.Longitude)
        print("ALT",self.Height)

    def showError(self,msg):
            if self.display:
                self.display.clearAll()
                self.display.writeAt(msg,0,0)
            print(msg)

    def storePosition(self):  # aktuelle Position als DD.dddd
merken
        lat=str(self.DDLat)+","
        lon=str(self.DDLon)+"\n"
        try:
            D=open("stored.pos","wt")
            D.write(lat)
            D.write(lon)
            D.close()
            if self.display:
                self.display.clearAll()
                self.display.writeAt("Pos. stored",0,0)
                sleep(3)
                self.display.clearAll()
        except (OSError) as e:
            enumber=e.args[0]
            if enumber==2:
                print("Not stored")
                if self.display:
                    self.display.clearAll()
                    self.display.writeAt("act. Position",0,0)
                    self.display.writeAt("not stored",0,0)
                    sleep(3)
                    self.display.clearAll()

    def chooseDestination(self, wait=3):
        if not self.display: return None
        self.display.clearAll()
        self.display.writeAt("ENTER=RST-Button",0,0)
        n="positions.pos"
        try:
            D=open(n,"rt")
            ziel=D.readlines()
            D.close()
            i = 0
            while 1:
                lat,lon=(ziel[i].rstrip("\r\n")).split(",")
                self.display.clearAll()
                self.display.writeAt("{}.
{}".format(i,lat),0,0)
```

```
                self.display.writeAt("   {}".format(lon),0,1)
                sleep(wait)
                if self.key.value()==0: break
                i+=1
                if i>=len(ziel): i=0
            self.zielPtr=i
            self.display.clearAll()
            self.display.writeAt("picked: {}".format(i),0,0)
            sleep(wait)
            self.display.clearAll()
            lat,lon=ziel[i].split(",")
            lon=lon.strip("\r\n")
            print("{}. Lat,Lon: {}, {}".format(i,lat,lon))
            lat=float(lat)
            lon=float(lon)
            return (lat,lon)
        except (OSError) as e:
            enumber=e.args[0]
            if enumber==2: print("File not found")
            self.display.clearAll()
            self.display.writeAt("There is no",0,0)
            self.display.writeAt("Positionfile",0,1)
            sleep(3)
            self.display.clearAll()
            return (0,0)

    def calcNewCourse(self,delay=3): # von letzter Position
bis hier
        lat,lon=self.chooseDestination(delay)
        if lat==0 and lon==0: return
        dy=(lat-self.DDLat)*60*1852
        dx=(lon-
self.DDLon)*60*1852*cos(radians(self.DDLatOld))
        print("Start {},{}".format(self.DDLat,self.DDLon),"
Ziel {},{}".format(lat,lon))
        #print("Ziel-Start",self.DDLon-self.DDLonOld,
self.DDLat-self.DDLatOld)
        self.calcCourse(dx,dy)

    def calcLastCourse(self): # von letzter Position bis hier
        try:
            D=open("stored.pos","rt")
            lat,lon=(D.readline()).split(",")
            D.close()
            self.DDLatOld=float(lat)
            self.DDLonOld=float(lon)
        except:
            self.DDLatOld=49.12345
            self.DDLonOld=11.12350
        dy=(self.DDLat-self.DDLatOld)*60*1852
        dx=(self.DDLon-
self.DDLonOld)*60*1852*cos(radians(self.DDLatOld))
```

```
        print("Start
{},{}".format(self.DDLonOld,self.DDLatOld),"  Ziel
{},{}".format(self.DDLon,self.DDLat))
        #print("Ziel-Start",self.DDLon-self.DDLonOld,
self.DDLat-self.DDLatOld)
        self.calcCourse(dx,dy)

    def calcCourse(self,dx,dy): # von letzter Position bis
hier
        course=0
        #print(dx,dy,degrees(atan2(dy,dx)))
        if abs(dx) < 0.0002:
            if dy > 0:
                course=0
                #print("Trace: 1")
            if dy < 0:
                course=180
                #print("Trace: 2")
            if abs(dy) < 0.0002:
                course=None
                #print("Trace: 3")
        else:  # dx >= 0.0002
            if abs(dy) < 0.0002:
                if dx > 0:
                    course=90
                    #print("Trace: 4")
                if dx < 0:
                    course=270
                    #print("Trace: 5")
            else:  ## dy > 0.0002
                course=90-degrees(atan2(dy,dx))
                #print("Trace: 6")
                if course > 360:
                    course -= 360
                    #print("Trace: 7")
                if course < 0:
                    course += 360
                    print("Trace: 8")
        self.course=int(course)
        self.distance=int(sqrt(dx*dx+dy*dy))
        print("Distance: {}, Course:
{}".format(self.distance,self.course))
        return (self.distance,self.course)

# ********************** Ende GPS
*****************************

# ********************* Beginn SIM808
*************************
class SIM808(GPS):
    DEFAULT_TIMEOUT=500
    CHAR_TIMEOUT=100
```

```
    CMD=1
    DATA=0

    def __init__(self,switch=4,disp=None,key=None):
        self.switch=Pin(switch,Pin.OUT)
        self.switch.on()
        super().__init__(disp)
        self.display=disp
        self.key=key
        print("SIM808 initialized")

    def simOn(self):
        self.switch.off()
        sleep(1)
        self.switch.on()
        sleep(3)

    def simOff(self):
        self.switch.off()
        sleep(3)
        self.switch.on()
        sleep(3)

    def simStartPhone():
        pass

    def simGPSInit(self):
        self.u.write("AT+CGNSPWR=1\r\n")
        self.u.write("AT+CGNSTST=1\r\n")

    def simGPSDeinit(self):
        self.u.write("AT+CGNSPWR=0\r\n")
        self.u.write("AT+CGNSTST=0\r\n")

    def simStopGPSTransmitting(self):
        self.u.write("AT+CGNSTST=0\r\n")

    def simStartGPSTransmitting(self):
        self.u.write("AT+CGNSTST=1\r\n")

    def simFlushUART(self):
        while self.u.any():
            self.u.read()

    # Wartet auf Zeichen an UART -> 0: keine Zeichen bis
Timeout
    def simWaitForData(self,delay=CHAR_TIMEOUT):
        noOfBytes=0
        start=ticks_ms()
        end=start+delay
        current=start
        while current <= end:
```

```
            sleep(0.1)
            noOfBytes=self.u.any()
            if noOfBytes>0:
                break
        return noOfBytes

    def
simReadBuffer(self,cnt,tout=DEFAULT_TIMEOUT,ctout=CHAR_TIMEOUT
):
        i=0
        strbuffer=""
        start=ticks_ms()
        prevchar=0
        while 1:
            while self.u.any():
                c=self.u.read(1)
                c=chr(ord(c))
                prevchar=ticks_ms()
                strbuffer+=c
                i+=1
                if i>=cnt: break
            if i>= cnt:break
            if ticks_ms()-start > tout: break
            if ticks_ms()-prevchar > ctout: break
        return (i,strbuffer) # gelesene Zeichen

    def simSendByte(self,data):
        return self.u.write(data.to_bytes(1,"little"))

    def simSendChar(self,data):
        return self.u.write(data)

    def simSendCommand(self,cmd):
        self.u.write(cmd)

    def simSendCommandCRLF(self,cmd):
        self.u.write(cmd+"\r\n")

    def simSendAT(slef):
        return self.simSendCmdChecked("AT","OK",CMD)

    def simSendEndMark(self):
        self.simSendChar(chr(26))

    def
simWaitForResponse(self,resp,typ=DATA,tout=DEFAULT_TIMEOUT,cto
ut=CHAR_TIMEOUT):
        l=len(resp)
        s=0
        self.simWaitForData(300)
        start=ticks_ms()
        prevchar=0
```

```
        while 1:
            if self.u.any():
                c=self.u.read(1)
                if ord(c) < 126:
                    c=c.decode()
                    prevchar=ticks_ms()
                    s=(s+1 if c==resp[s] else 0)
                if s == l: break
            if ticks_ms()-start > tout: return False
            if ticks_ms()-prevchar > ctout: return False
        if type==CMD:
            self.simFlushUART()
        return True

    def
simSendCmdChecked(self,cmd,response,typ,tout=DEFAULT_TIMEOUT,c
tout=CHAR_TIMEOUT):
        self.simSendCommand(cmd)
        return
self.simWaitForResponse(response,typ,tout,ctout)

# ********************** Ende  SIM808
**************************
```

Das Anwendungsprogramm hat gegenüber dem ersten Teil an Umfang zugenommen. Das liegt daran, dass die Zahl Funktionen von 4 auf nunmehr 11 gestiegen ist. Wenn Sie es in die Datei boot.py verpacken und diese zum ESP32 hochladen, startet dieser autonom, ohne den USB-Anschluss zum PC zu benötigen. Damit sind Sie im Gelände unabhängig. Die Anzeige erfolgt ja über das LCD und die Steuerung über die Keypad-Tasten. Hier das Listing des erweiterten GPS-Haupt-Programms rambler.py

.

```
import sys
from machine import ADC, Pin, I2C
from button import BUTTONS, BUTTON32
rstNbr=25
rst=BUTTON32(rstNbr,True,"RST")
ctrl=Pin(rstNbr,Pin.IN,Pin.PULL_UP)
t=BUTTONS() # Methoden für Buttons bereitstellen
from time import sleep
from gps import GPS,SIM808
from lcd import LCD
from keypad import KEYPAD
from bmp280 import BMP280

i2c=I2C(-1,Pin(21),Pin(22))
VMeterPin=34
volt=ADC(Pin(VMeterPin))
volt.atten(ADC.ATTN_6DB)
volt.width(ADC.WIDTH_10BIT)
volt.read()  # erst mal Messung initialisieren
```

```
k=KEYPAD(35)
d=LCD(i2c,0x27,cols=16,lines=2)
d.printAt("SIM808-GPS",0,0)
d.printAt("GPS booting",0,1)
sleep(1)
g=SIM808(4,disp=d,key=ctrl)
g.simGPSInit()
g.simOn()
b=BMP280(i2c)
#sleep(10)
g.mode="DDF"
# ************************* Functions
*****************************
def editPositions():
    while 1:
        s=input("Befehl|Position (Help=H): ")
        if len(s) == 1:
            s=s.upper()
            if s=="H":
                print("N: New|Clear List")
                print("L: List Positions")
                print("E: Exit Editing")
                print("H: Thish Helptext")
                print("<Latitude>,<Longitude> in DD.dddd")
            if s=="N":
                n="positions.pos"
                D=open(n,"wt")
                D.close()
            if s=="L":
                n="positions.pos"
                D=open(n,"rt")
                ziel=D.readlines()
                D.close()
                for i in ziel:
                    print(i.rstrip("\r\n"))
            if s=="E": break
        else:
            D=open("positions.pos","at")
            D.write(s+"\n")
            D.close()

def testVoltage():
    Ubat=3.26
    Umess=1.634
    kU=Ubat/Umess/4
    kM=2/1024*1.0214
    s=0
    for i in range(10):
        s=s+volt.read()
    spannung=s/10
    spannung=spannung*kM*kU
```

```
    print("Batteriespannung: ",spannung)
    return spannung

# ************************* Program Start
************************
if t.getTouch(rst):
    editPositions()

#sys.exit()

while 1:
    rmc=g.waitForLine("$GPRMC",delay=2000)
    if rmc:
        try:
            g.decodeLine(rmc)
            if g.Valid == "A":
                try:
                    gga=g.waitForLine("$GPGGA",delay=2)
                    g.decodeLine(gga)
                    g.printData()
                    g.showData()
                except:
                    g.showError("Invalid GGA-set!")
        except:
            g.showError("Invalid RMC-set!")
    wahl=k.key()
    if k.Right<=wahl<=k.Down:
        d.clear()
        if ctrl.value()==1:
            g.Mode=g.AngleModes[wahl]
            g.DMode="pos"
        else:
            if wahl==k.Right:
                g.storePosition()
            if wahl==k.Up:
                g.calcNewCourse(delay=2)

d.writeAt("Course:{}\xdf".format(g.course),0,0)
                d.writeAt("Dist:{}m".format(g.distance),0,1)
                sleep(5)
                d.clear()
            if wahl==k.Down:
                g.calcLastCourse()

d.writeAt("Course:{}\xdf".format(g.course),0,0)
                d.writeAt("Dist:{}m".format(g.distance),0,1)
                sleep(5)
                d.clear()
    if wahl==k.Left:
        d.clear()
        if ctrl.value() == 1:
            g.DMode="time"
```

```
        else:
            b.calcPressureNN()
            d.writeAt("Pres:
{}hPa".format(int(b.pressNN)),0,0)
            d.writeAt("Temp: {}\xdfC".format(b.temp),0,1)
            sleep(5)
            d.clear()
    if wahl==k.Select:
        d.clear()
        if ctrl.value() == 1:
            g.DMode="height"
        else:
            s=testVoltage()
            print("Spannung: ",s)
            d.writeAt("Batteriespannung",0,0)
            d.writeAt("{:.2f} V".format(s),0,1)
            sleep(3)
            d.clear()
            pass
    sleep(0.1)
```

Das Hauptprogramm ist in der Regel nur eine hübsche Verpackung. Die eigentliche Arbeit verrichten die Methoden in den Klassen, die um die Hardware herum gebaut werden. Sie werden auch in den Programmen in diesem Beitrag Schichten erkennen, deren Aufbau sich wiederholt und folgendem Schema folgt.

Hardware – Kommunikations-Treiber – Basisfunktionen – API – Hauptprogramm

So ist es auch mit dem letzten Modul, das sich mit dem BMP280 beschäftigt. Es umfasst alle Funktionen, die zum Ansprechen der Registerstruktur des BME280 nötig sind. Damit lassen sich Einstellungen der Betriebsweise durchführen und Messergebnisse abfragen.

Damit der Transport vom und zum ESP32 klappt, ist ein I2C-Bustreiber eingeschaltet, der die internen MicroPython-Busbefehle des ESP32 in standardisierte Befehle übersetzt und dabei nach Möglichkeit eine Anpassung zwischen numerischen und Textvariablen vornimmt. Diese Klasse I2CBus arbeitet im Hintergrund und wird von der Klasse BME280 importiert und initialisiert. Durch Vererbung verfügt die Klasse BME280 automatisch auch über den Namensraum von I2CBus. Der Benutzer der Klasse BME280 muss davon im Prinzip nichts wissen. Es genügt, die API der Klasse BME280 zu kennen. Und hier ist deren Listing.

```
# File: bmp280.py
# Author: Jürgen Grzesina
# Rev.: 0.1 AVR-Assembler
# Rev.: 2.0 MicroPython Portierung
# Stand: 12.04.2021
"""
Methoden von BME280
BME280(i2c, hwadr=0x76)
getCalibrationData()
printCalibrationData()
```

```
readDataRaw()
readControlReg()
readConfigReg()
readStatusReg()
isBussy()
isImaging()
writeContrlReg()
writeConfigReg()
setConfig(StandBy=None, Filter=None)
setControl(OST=None, OSP=None, Mode=None)
calcTfine()
calcTemperature()
calcPressureH()
calcPressureNN(self,h=465,t=20)
softReset()
"""

from i2cbus import I2CBus

class BMP280(I2CBus):
    HWADR=const(0x76)

    PShift=const(2)
    OsamP1=const(1)
    OsamP2=const(2)
    OsamP4=const(3)  # default
    OsamP8=const(4)
    OsamP16=const(5)

    TShift=const(5)
    OsamT1=const(1)  # default
    OsamT2=const(2)
    OsamT4=const(3)
    OsamT8=const(4)
    OsamT16=const(5)

    FShift=const(2)
    Filter1=const(1)
    Filter2=const(2)
    Filter4=const(3)  # default
    Filter8=const(4)
    Filter16=const(5)

    SShift=const(5)
    SBy05=const(0)
    SBy62=const(1)
    SBy125=const(2)  # default
    SBy250=const(3)
    SBy500=const(4)
    SBy1000=const(5)
    SBy2000=const(6)
    SBy4000=const(7)
```

```
    # booting to sleepMode send 0b00 to 0xF4 once
    # for entering forcedMode send 0b01 to 0xF4(control
register) each
    # for entering mormalMode with standby send 0b11 to 0xF4
once
    sleepMode=const(0)
    forcedMode=const(1)
    normalMode=const(3)  # default

    digT1R=const(0x88)
    digT2R=const(0x8A)
    digT3R=const(0x8C)
    # ------------------
    digP1R=const(0x8E)
    digP2R=const(0x90)
    digP3R=const(0x92)
    digP4R=const(0x94)
    digP5R=const(0x96)
    digP6R=const(0x98)
    digP7R=const(0x9A)
    digP8R=const(0x9C)
    digP9R=const(0x9E)

    ChipIDR=const(0xD0)
    VersionR=const(0xD1)
    ResetR=const(0xE0)
    StatusR=const(0xF3)
    ContrlR=const(0xF4)
    ConfigR=const(0xF5)
    PdataR=const(0xF7)
    TdataR=const(0xFA)

    def __init__(self, i2c, hwadr=HWADR):
        # Starting normalMode, Oversampling Press 4x, Temp 1x
        # Standbytime 125ms, Filtercoeff. 4
        super().__init__(i2c,hwadr)
        self.hwadr=hwadr
        self.bmeID=self.readUint8FromReg(ChipIDR)
        self.digT=[0]*4
        self.digP=[0]*10
        self.getCalibrationData()
        self.config=SBy125<<SShift | Filter4 << FShift
        self.writeByteToReg(ConfigR,self.config)
        self.control=OsamT1<<TShift | OsamP4<<PShift |
normalMode
        self.writeByteToReg(ContrlR,self.control)
        self.status=0
        self.tFine=0
        self.temp=0
        self.pres=0
        self.pressNN=0
```

```
        print("BMP280 initialized")

    def getCalibrationData(self):
        calVal=self.readNbytesFromReg(0x88,6,self.hwadr)
        for i in range(1,4):
            ptr=i*2
            self.digT[i]=calVal[ptr-2] | calVal[ptr-1] << 8
            if i>1:
                if self.digT[i] > 32767: self.digT[i] -= 65536
        calVal=self.readNbytesFromReg(0x8E,18,self.hwadr)
        for i in range(1,10):
            ptr=i*2
            self.digP[i]=calVal[ptr-2] | calVal[ptr-1] << 8
            if i>1:
                if self.digP[i] > 32767: self.digP[i] -= 65536

    def printCalibrationData(self):
        for i in range (1,4):
            print(self.digT[i])
        for i in range(1,10):
            print(self.digP[i])

    def readDataRaw(self):
        #read raw pressure + temp at once
        data=self.readNbytesFromReg(PdataR,6,self.hwadr)
        self.pRaw=data[0]<<12 | data[1]<<4 | data[2]>>4
        self.tRaw=data[3]<<12 | data[4]<<4 | data[5]>>4

    def readControlReg(self):
        self.control=self.readUint8FromReg(ContrlR,self.hwadr)

    def readConfigReg(self):
        self.config=self.readUint8FromReg(ConfigR,self.hwadr)

    def readStatusReg(self):
        self.status=self.readUint8FromReg(StatusR,self.hwadr)

    def isBussy(self):
        self.status=self.readStatusReg() & 0b00001000
        return (1 if self.status else 0)

    def isImaging(self):
        self.status=self.readStatusReg() & 0b00000001
        return (1 if self.status else 0)

    def writeContrlReg(self):
        self.writeByteToReg(ContrlR,self.control,self.hwadr)

    def writeConfigReg(self):
        self.writeByteToReg(ConfigR,self.config,self.hwadr)

    def setConfig(self,StandBy=None, Filter=None):
```

```
        if not StandBy is None:
            if StandBy in range (0,8):
                self.config = (self.config & 0b00011111) |
StandBy
            else: raise ValueError("Standby: 0 <= Oversampling
< 8")
        if not Filter is None:
            if Filter in range (1,6):
                self.config = (self.config & 0b11100011) |
Filter
            else: raise ValueError("Filter: 0 <= Oversampling
< 8")

    def setControl(self,OST=None, OSP=None, Mode=None):
        if not OST is None:
            if OST in range (1,6):
                self.control = (self.control & 0b00011111) |
OST
            else: raise ValueError("Temperatur: 1 <=
Oversampling <6")
        if not OSP is None:
            if OSP in range (1,6):
                self.control = (self.control & 0b11100011) |
OSP
            else: raise ValueError("Druck: 1 <= Oversampling
<6")
        if not Mode is None:
            if (Mode==0) or (Mode==1) or (Mode==3):
                self.control = (self.control & 0b11111100) |
Mode
            else: raise ValueError("Mode: 0(sleep), 1(forced)
oder 3(normal)")

    def calcTfine(self):
        self.readDataRaw()
        # nur berechnen, wenn tFine=0 ist
        var1=(((self.tRaw>>3)-
(self.digT[1]<<1))*self.digT[2])>>11
        var2=(((((self.tRaw>>4)-self.digT[1]) *
                ((self.tRaw>>4)-self.digT[1]))>>12)*
                self.digT[3])>>14
        self.tFine=var1+var2

    def calcTemperature(self):
        self.calcTfine()
        self.temp=((self.tFine*5+128)>>8)/100.
        return self.temp

    def calcPressureH(self):
        self.calcTfine()
        var1 = self.tFine - 128000
        var2 = var1 * var1 * self.digP[6]
```

```
        var2 = var2 + ((var1 * self.digP[5]) << 17)
        var2 = var2 + (self.digP[4] << 35)
        var1 = ((var1 * var1 * self.digP[3]) >> 8) + ((var1 *
self.digP[2]) << 12)
        var1 = (((1 << 47) + var1) * self.digP[1]) >> 33
        if var1 == 0: return 0
        p = 1048576 - self.pRaw
        p = int((((p << 31) - var2) * 3125) / var1)
        var1 = (self.digP[9] * (p >> 13) * (p >> 13)) >> 25
        var2 = (self.digP[8] * p) >> 19
        p = ((p + var1 + var2) >> 8) + (self.digP[7] << 4)
        self.pres=p/256
        return self.pres

    def calcPressureNN(self,h=465,temp=None):
        self.calcPressureH()
        t=(temp if temp else self.calcTemperature())
        T=t+273
        self.pressNN=(self.pres*pow((T/(T+0.0065*h)),-
5.255))/100
        return self.pressNN

    def softReset(self):
        self.writeByteToReg(ResetR,0xB6,self.hwadr)
```

Zur Bedienung des Hauptprogramms noch ein paar Anmerkungen.

- Es kann einige Minuten dauern, bis das SIM808 brauchbare Ergebnisse liefert.
- Das Programm startet mit der Anzeige Grad, Minuten und Bruchteile.
- Nach Tastendruck bitte abwarten, bis das Display gelöscht wird, dann die Taste loslassen. Die nächste Anzeige erfolgt im neuen Modus.

Tasten:
RST beim Start Eingabe von Wegpunkten als Zielvorgabe
right Grad und Dezimalen
Up Grad, Minuten Sekunden und Bruchteile
Down Grad, Minuten und Bruchteile
Left Datum und Uhrzeit
Select Höhe und Uhrzeit
RST+Right Aktuelle Position als Wegpunkt speichern
RST+Up Abrufen von Wegpunkten und Kurs und Distanzanzeige
RST+Down Kurs und Distanz vom zuletzt gespeicherten Wegpunkt aus
RST+Left Luftdruck und Temperatur
RST+Select Spannung auf der 3,3V-Leitung
Nach den RST-Funktionen kehrt die Anzeige zur letzten Normalanzeige zurück.

Jetzt brauchen Sie das ganze Equipment nur noch in einer Schachtel verstauen und dann geht die Post ab ins Gelände. Und wer weiß, vielleicht verpacken Sie nach erfolgtem Test ja auch alles in ein formschönes Gehäuse.

Wie schon angedeutet, geht es im nächsten Beitrag um den Einsatz des SIM808 zum Übermitteln von Geodaten und anderen Informationen via SMS. Damit entsteht ein Trackingmodul oder ein stationäres Messmodul, das per Zeitsteuerung oder durch einen Anruftrigger Daten über das Mobilfunknetz übertragen kann. So ist endlich leicht festzustellen, wo der Sohnemann mit Papas Nobelkutsche grade rumkurvt oder wo der entlaufene Hund eingefangen werden kann oder…

Viel Spaß bei der Umsetzung des Projekts!

Weitere Downloadlinks:
PDF in deutsch
PDF in english