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

Die Beiträge von Bernd Albrecht zum Thema "Ostereiersuche mit GPS" haben mich dazu inspiriert zu prüfen, ob das Vorhaben nicht auch mit Micropython zu machen wäre. Nun, natürlich ist es das, wenn man sich das nötige Werkzeug zusammengesucht hat. Und noch etwas verrate ich Ihnen gleich zu Beginn, das Programm fällt sehr kurz und übersichtlich aus – dank zweier Klassen, die ich speziell für das SIM808 und das Keypad gebaut habe. Die eigene Suche nach Quellen will ich Ihnen gerne ersparen, denn in diesem Beitrag verrate ich Ihnen, welche Tools Sie brauchen, wo sie diese herbekommen und wie sie eingesetzt werden. Damit willkommen zum ersten Teil von

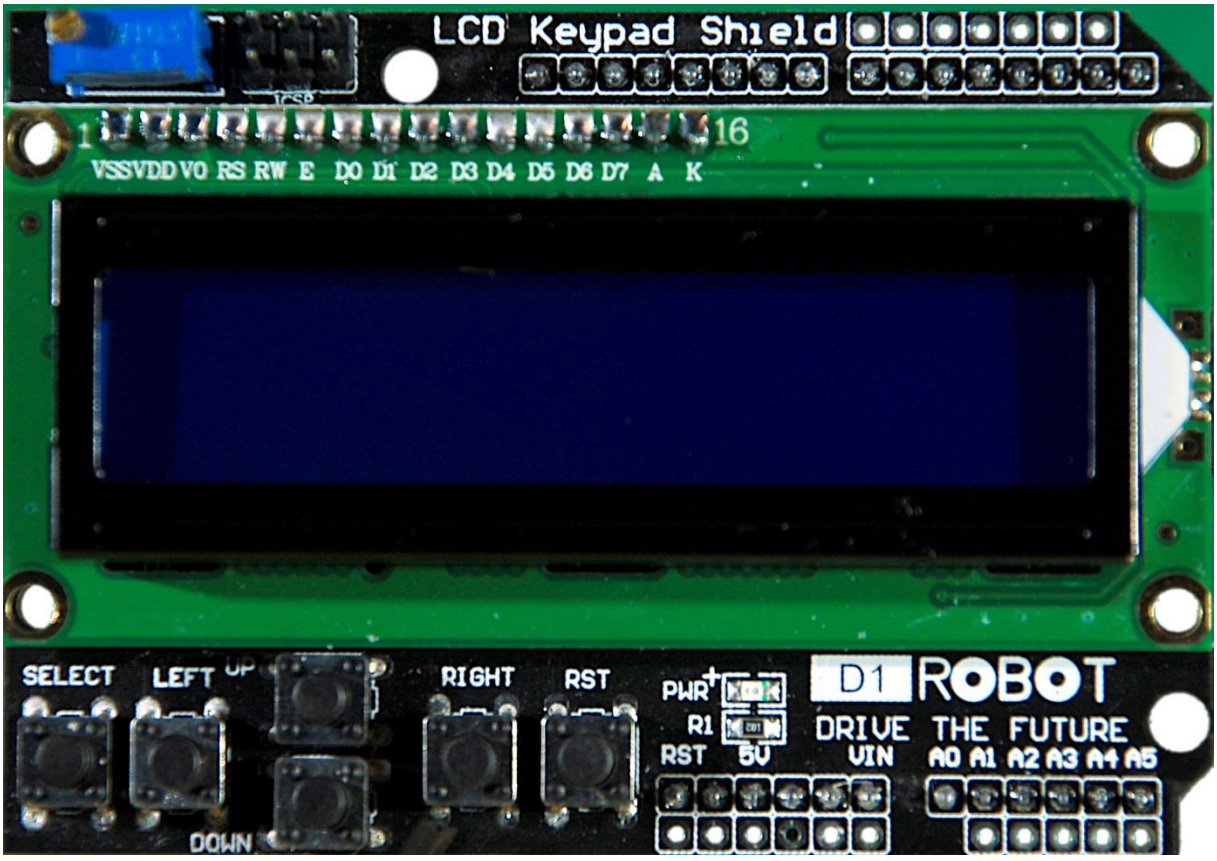
GPS mit MicroPython auf dem ESP32

Ein paar Vorüberlegungen zur Hardware

Wenn Sie dem Osterhasen beim Verstecken der Eier mit dem GPS zur Hand gegangen sind, haben sie die benötigte Hardware schon zum großen Teil. Welche Änderungen ich an der Liste vorgenommen habe und warum, das erfahren Sie gleich im Anschluss.

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
3	Widerstand 10kΩ

Die Wahl des Controllers fiel eindeutig auf den ESP32, zum einen wegen der Möglichkeit, dort MicroPython als Firmware zu etablieren, was auf dem Arduino nicht geht. Auch ein ESP8266 scheidet aus, weil keine zweite serielle Hardwareschnittstelle zur Verfügung steht. Und Raspi wäre eindeutig überdimensioniert. Die Softwarelösungen, wie am Arduino, arbeiteten nicht zuverlässig. Außerdem reichen die In-Out-Leitungen am ESP8266 für mein komplettes Vorhaben nicht aus.

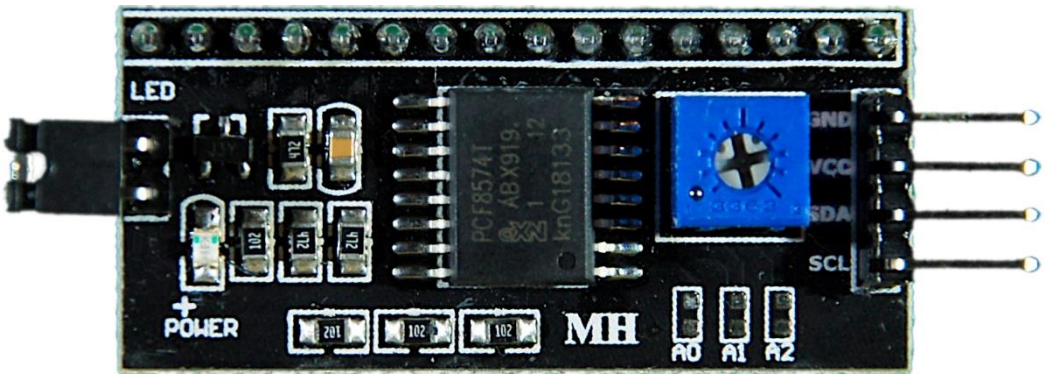


Das Display mit Keypad bietet neben der 16x2-Darstellung insgesamt 6 Tasten, von denen fünf im aktuellen Projekt eine Steuerfunktion zu erfüllen haben. Diese Tasten liefern die Spannung von den Knotenpunkten einer Widerstandskaskade an einen ADC-Eingang (GPIO35). Die Pegel werden vom ESP32 decodiert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

Die 6. Taste, RST, folgt nicht diesem Schema. Wird das Keypad als Shield am Arduino benutzt, dann legt die Taste den RST-Eingang des AT-Mega328 (mit 10kΩ-Pullup-Widerstand) auf GND-Potential und löst damit einen Kaltstart aus. Um diese Taste am ESP32 zu anderen Zwecken nutzen zu können, müssen wir ihr einen

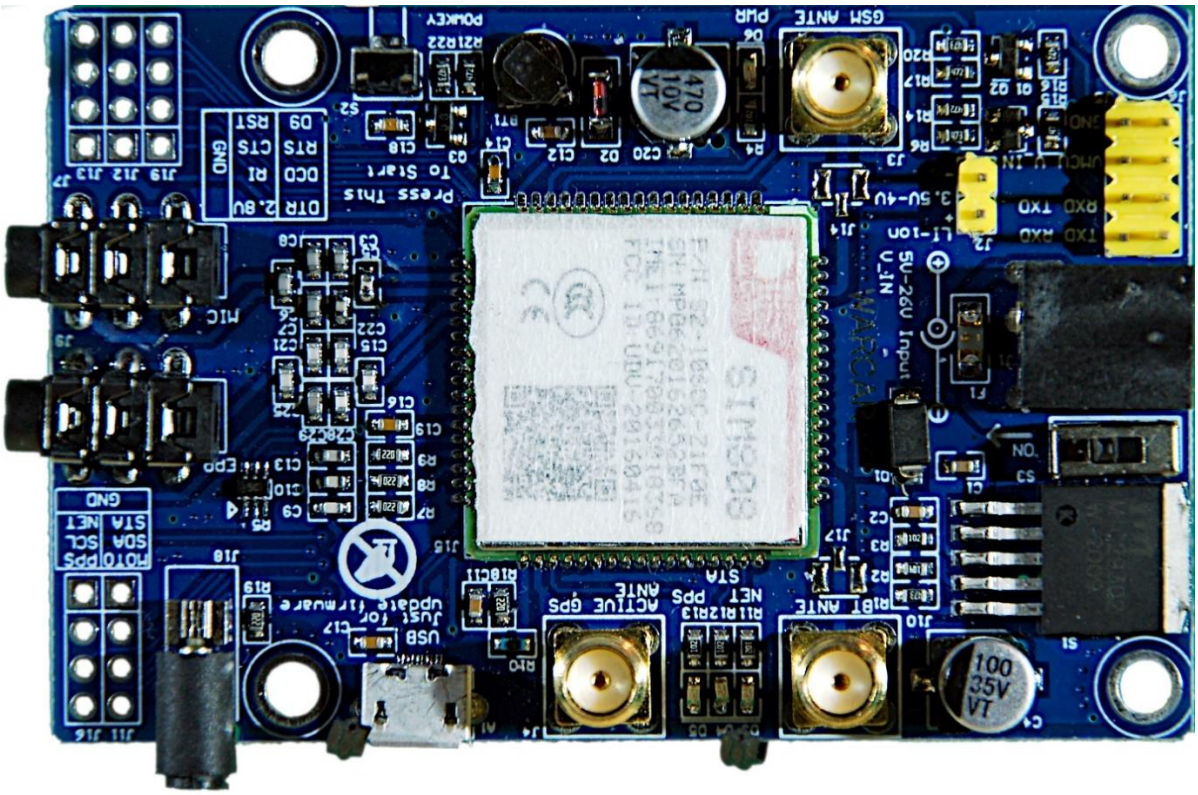
Pullup-Widerstand gegen 3,3V (nicht 5V!) spendieren. Dann kann man einen beliebigen Eingang des ESP32 an den RST-Ausgang der Keypadplatine anschließen und diese Taste mit dem Controller abfragen. Ich habe für diesen Zweck den GPIO25 gewählt. Es wäre schaltungstechnisch einfacher gewesen, statt des Keypads mit LDC ein OLED-Display einzusetzen, ich wollte aber nicht auf die Tasten verzichten, die ich sonst anderweitig hätte ersetzen müssen. Denn irgendwie muss man den Aufbau steuern können, wenn man in der Pampa unterwegs ist.

Mit dem LCD habe ich mir allerdings noch ein anderes Problem aufgehalst. Das Display braucht eine Versorgungsspannung von 5V, damit es funktioniert. Damit liegen auch die Eingänge des Displays, 4-mal Daten, RS und E, über Pullups an 5V. Die verträgt aber der ESP32 nicht an seinen Ausgängen. Die Spannungspegel an den Pins des Controllers sollen 3,3V nicht übersteigen. Der Ansatz über Spannungsteiler war mir zu aufwendig. Auch ein ULN2803 als Pegelwandler hätte einen höheren Kabelaufwand hervorgerufen. Weil außerdem bereits ein MicroPython-Modul für ein I2C-LCD in meiner Werkzeugkiste lag, habe ich mich entschlossen, dem LCD einen I2C-Adapter zu spendieren.



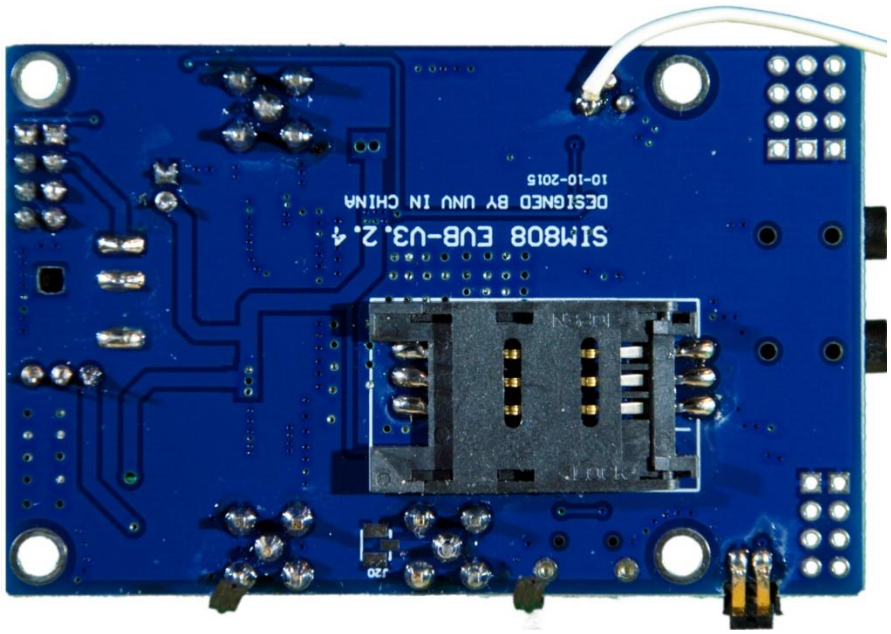
Dessen Eingänge sind 3,3V-kompatibel und die Ausgänge vertragen die 5V vom Display. Zum Anschluss an den ESP32 sind jetzt nur noch 2 Leitungen, SCL (21) und SDA (22) statt der 6 Direktverbindungen nötig. Ferner steht der I2C-Bus auch noch anderen Sensoren, wie zum Beispiel einem BMP/BME280, zur Verfügung, der als Erweiterung der Schaltung vorgesehen ist.

Für die Energieversorgung habe ich einen Batteriehalter für eine 18650-er Lithiumzelle im Einsatz, weil darauf die beiden Betriebsspannungen 3,3V und 5V über Steckpins bereits zu Verfügung stehen. Das SIM808 könnte sogar direkt aus der Li-Zelle versorgt werden. In diesem Fall müsste aber an die Lötstellen der Halterung auf der Platine jeweils ein Drahtstück angelötet werden. Ich bevorzuge jedoch die 5V-Anschlüsse, die direkt an das SIM808 sowie das LCD Keypad geführt werden können. 3,3V gehen an den I2C-Adapter und den ESP32.



Bleibt noch das SIM808. Es wird normalerweise über die Rohrbuchse (2,1x5,5) mit Spannungen von 5..12V versorgt, hat aber auch einen Anschluss für 3,5 bis 4,2V für eine Lithiumzelle im Angebot. Für externe 5V steht an der gelben Stiftleiste ein Anschluss neben einem der GND-Pins zur Verfügung. Am größeren gelben Stiftsockel links oben liegt GND und genau darunter +5V Vin.

Ferner ist es sinnvoll, ein Kabel an die Pins der Einschalttaste zu löten, die dann an einen Pin (hier GPIO4) des ESP32 gelegt wird. Der Controller kann somit beim Booten automatisch das SIM808 starten. Die Taste muss dann nicht mehr bedient werden.



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

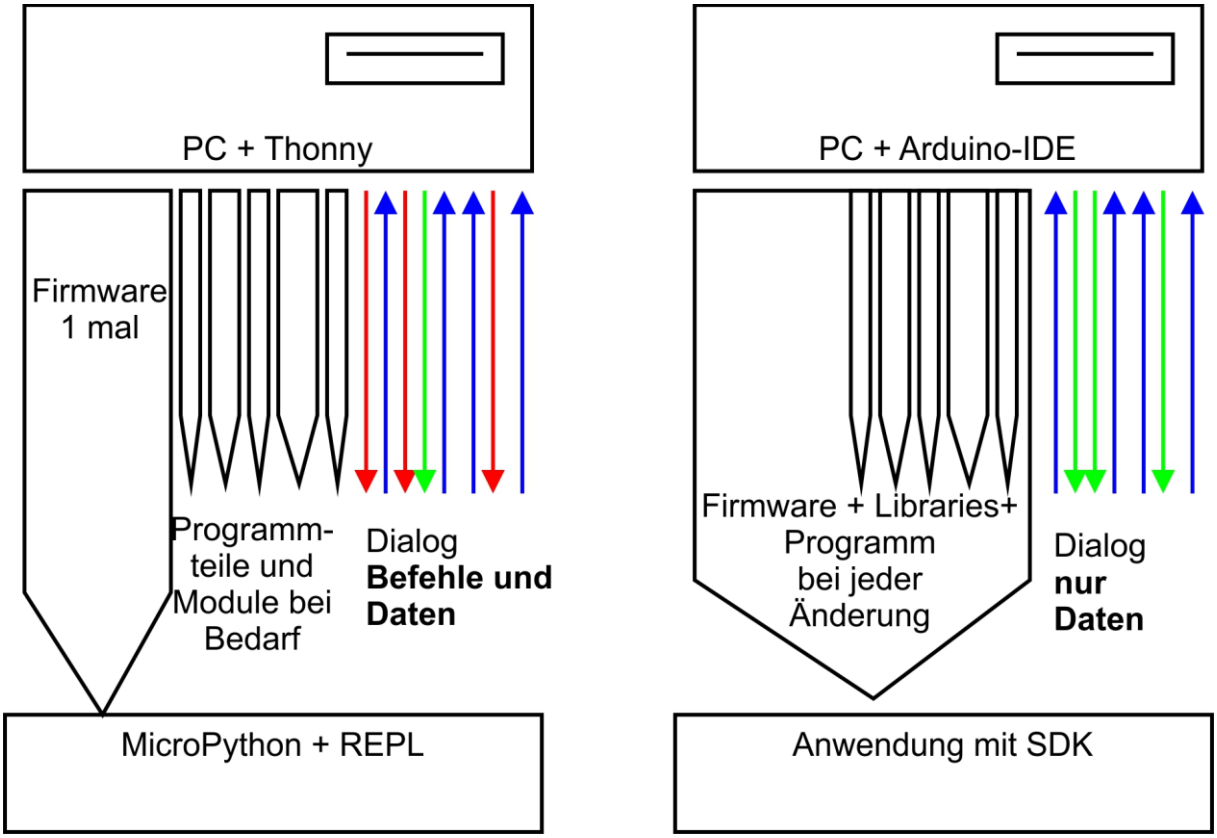
Verwendete Software:
Fürs Flashen und die Programmierung des ESP:
Thonny oder
µPyCraft
MicropythonFirmware

MicroPython-Module
GPS-Modul für SIM808 und GPS6MV2(U-Blocks)
LCD-Standard-Modul
HD44780U-I2C-Erweiterung zum LCD-Modul
Keypad-Modul
Button Modul

Ein paar Gedanken 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.

Die Grafik zeigt diesen Unterschied, aber auch noch einen vielleicht noch wichtigeren zweiten.



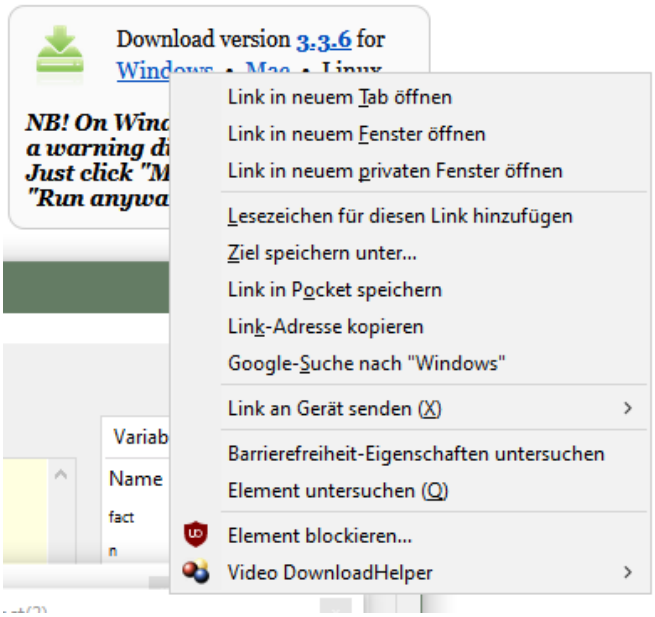
Nachdem die Firmware geflasht ist, können Sie sich aber zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die

Antwort sehen, ohne vorher ein ganzes Programm compilieren zu müssen. Änderungen an Programmteilen werden unter MicroPython einzeln aktualisiert. Dass diese Methode schneller ist, brauche ich eigentlich nicht extra erwähnen. Welche Schritte nötig sind, um auf diese komfortable Art zu programmieren, sagt Ihnen der folgende Abschnitt.

Die Entwicklungsumgebung – Beispiel: Thonny

Thonny ist unter MicroPython das Gegenstück zur Arduino-IDE. In Thonny sind ein Programmeditor und ein Terminal sowie weitere interessante Entwicklungstools in einer Oberfläche vereint. So haben sie das Arbeitsverzeichnis auf dem PC, das Dateisystem auf dem ESP32, Ihre Programme im Editor, die Terminalconsole und zum Beispiel den Object inspector in einem Fenster übersichtlich im Zugriff.

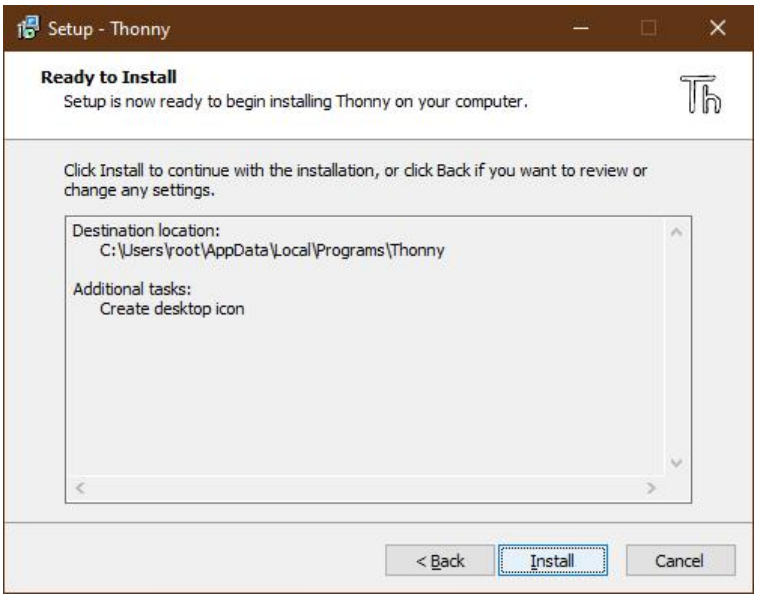
Die Ressource zu Thonny ist die Datei thonny-3.3.x.exe , deren neuste Version direkt von der Produktseite heruntergeladen werden kann. Dort kann man sich auch einen ersten Überblick über die Eigenschaften des Programms verschaffen.



Mit Rechtsklick auf **Windows** und **Ziel speichern unter** laden Sie die Datei in ein beliebiges Verzeichnis Ihrer Wahl herunter. Alternativ können Sie auch diesem Direktlink folgen.
Im Bundle von **Thonny** sind neben der IDE selbst auch **Python 3.7** für Windows und **esptool.py** enthalten. Python 3.7 (oder höher) ist die Grundlage für Thonny und esptool.py. Beide Programme sind in Python geschrieben und benötigen daher die Python-Laufzeitumgebung. **esptool.py** dient unter anderem auch in der Arduino-IDE als Werkzeug, um Software auf den ESP32 (und andere Controller) zu transferieren.

Starten Sie jetzt die Installation von Thonny durch Doppelklick auf ihre heruntergeladene Datei, wenn Sie die Software nur für sich selbst nutzen möchten. Wenn Thonny & Co. allen Usern zur Verfügung stehen soll, müssen Sie die exe-Datei als Administrator ausführen. In diesem Fall klicken Sie rechts auf den Dateieintrag im Explorer und wählen **Als Administrator ausführen**.

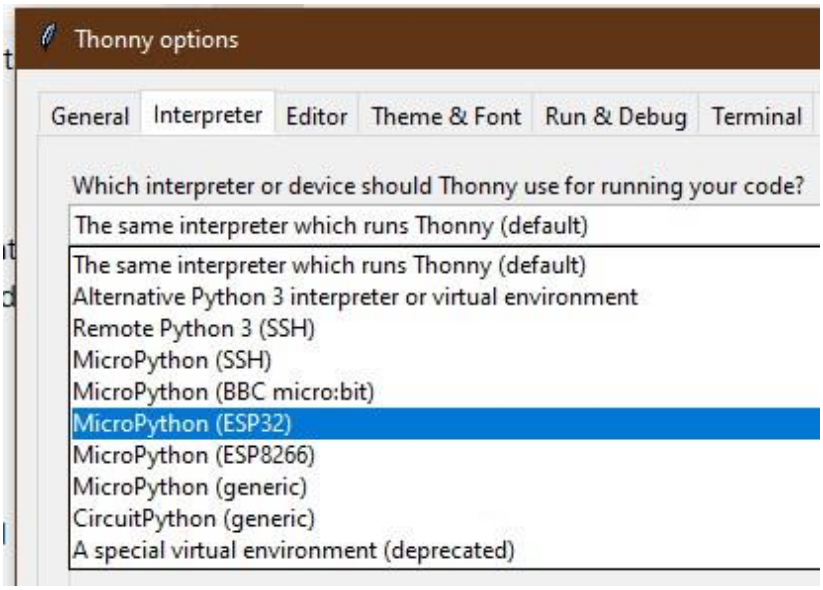
Sehr wahrscheinlich meldet sich der Windows Defender (oder Ihre Antivirensoftware). Klicken Sie auf **weitere Informationen** und im folgenden Fenster auf **Trotzdem ausführen**. Folgen Sie jetzt einfach der Benutzerführung mit **Next**.



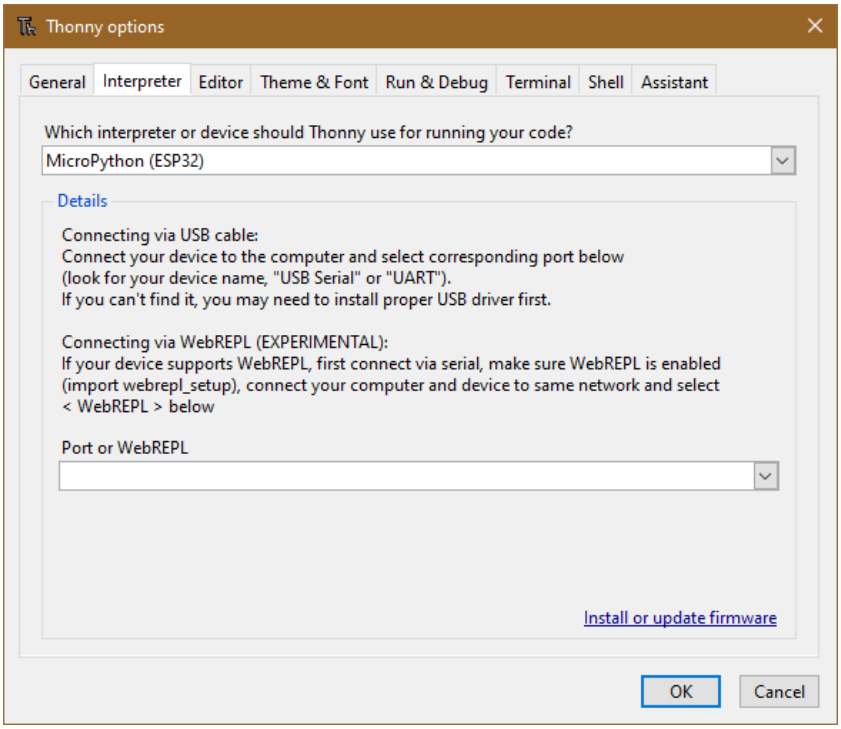
Mit Klick auf **Install** startet der Installationsprozess.

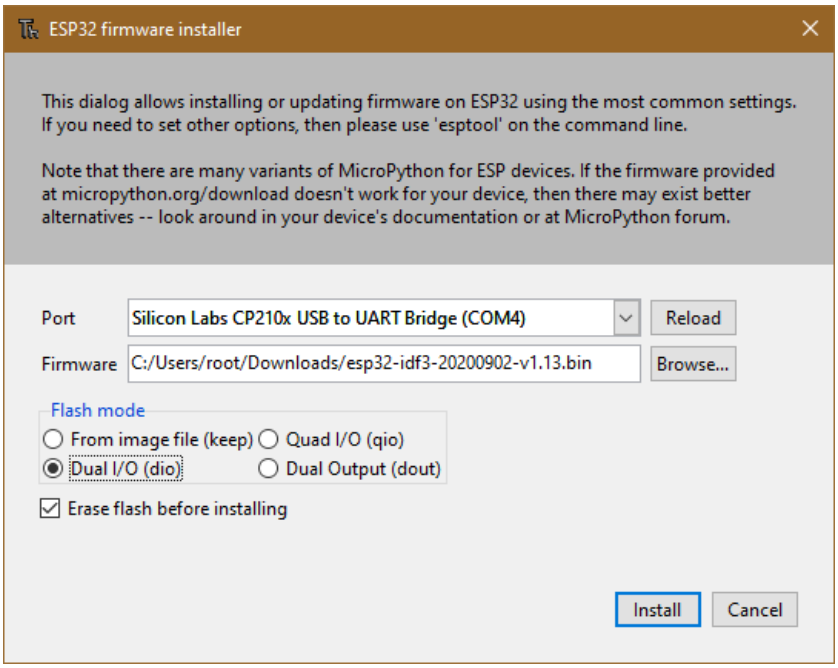
Beim ersten Start geben Sie die Sprache an, dann wird das Editorfenster zusammen mit dem Terminalbereich angezeigt.

Stellen Sie als erste Aktion den verwendeten Controllertyp ein. Mit **Run – Select Interpreter …** landen Sie in den Optionen. Für dieses Projekt stellen Sie bitte Micropython(ESP32) ein.



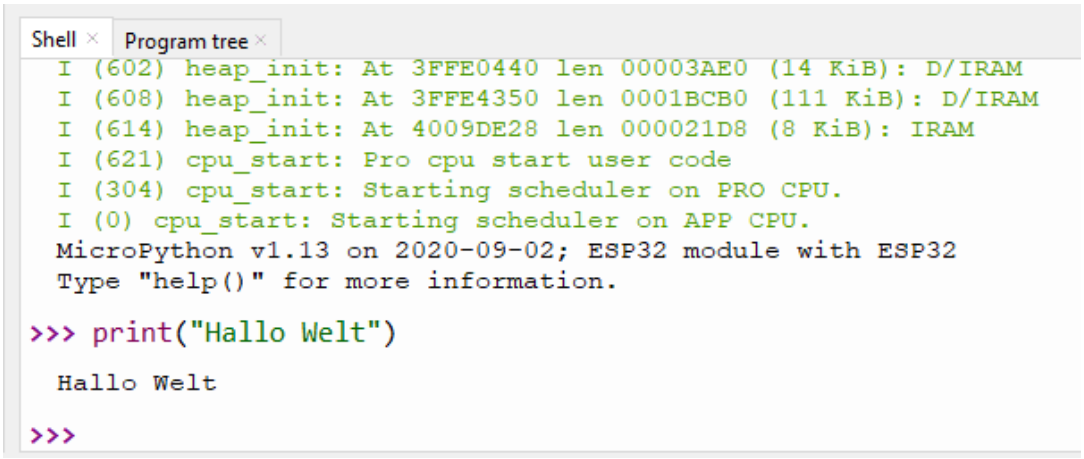
Laden Sie jetzt die [Firmware Micropython für den ESP32] herunter und speichern Sie diese Datei in einem Verzeichnis Ihrer Wahl. Die bin-Datei muss als erstes auf den ESP32 transferiert werden. Das geschieht auch mit Thonny. Rufen Sie wieder mit **Run – Select Interpreter … Thonny Options** auf. Rechts unten klicken Sie auf **Install or update Firmware**.





Wählen Sie den seriellen Port zum ESP32 und die heruntergeladene Firmwaredatei aus. Mit **Install** starten Sie den Prozess. Nach kurzer Zeit befindet sich die MicroPython-Firmware auf dem Controller und Sie können die ersten Befehle über REPL, die MicroPython-Kommandozeile, an den Controller senden. Geben Sie im Terminalfenster zum Beispiel folgenden Befehl ein.

print("Hallo Welt")



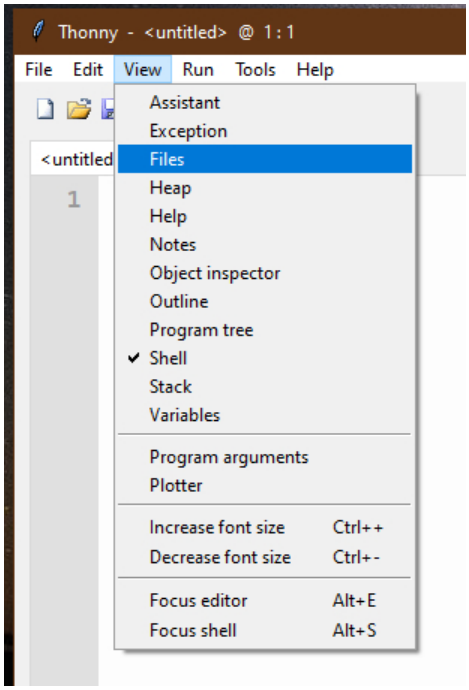
Anders als in der Arduino-IDE können Sie einzelne Befehle an den ESP32 senden und er wird, so es MicroPython-Anweisungen sind, brav antworten. Senden Sie dagegen einen für den MicroPython-Interpreter unverständlichen Text, wird er sie mit einer Fehlermeldung darauf aufmerksam machen.

>>> print"hallo nochmal"

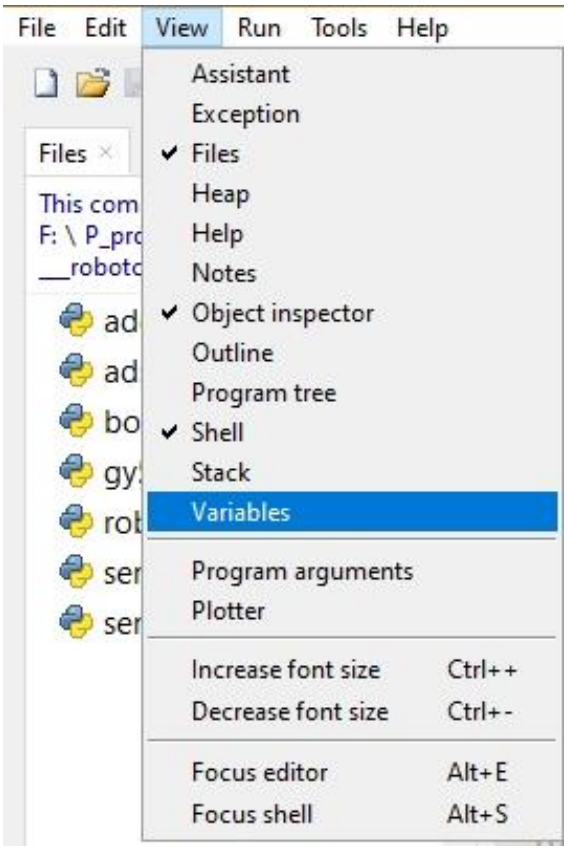
SyntaxError: invalid syntax
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: invalid syntax

Zum Arbeiten fehlt jetzt aber noch die Übersicht über den Workspace und das Device Directory. Der Workspace ist ein Verzeichnis auf dem PC, in dem sich alle für ein Projekt wichtigen Dateien befinden. In Thonny ist sein Name **This Computer**. Das Device Directory ist dazu das Gegenstück auf dem ESP32. In Thonny heißt es **MicroPython device.** Sie bringen es folgendermaßen zur Anzeige.

Klicken Sie auf **View** und dann auf **Files**



Jetzt werden beide Bereiche, oben der Workspace und unten das Device Directory, angezeigt. Weitere Tools blenden Sie über das Menü **View** ein.



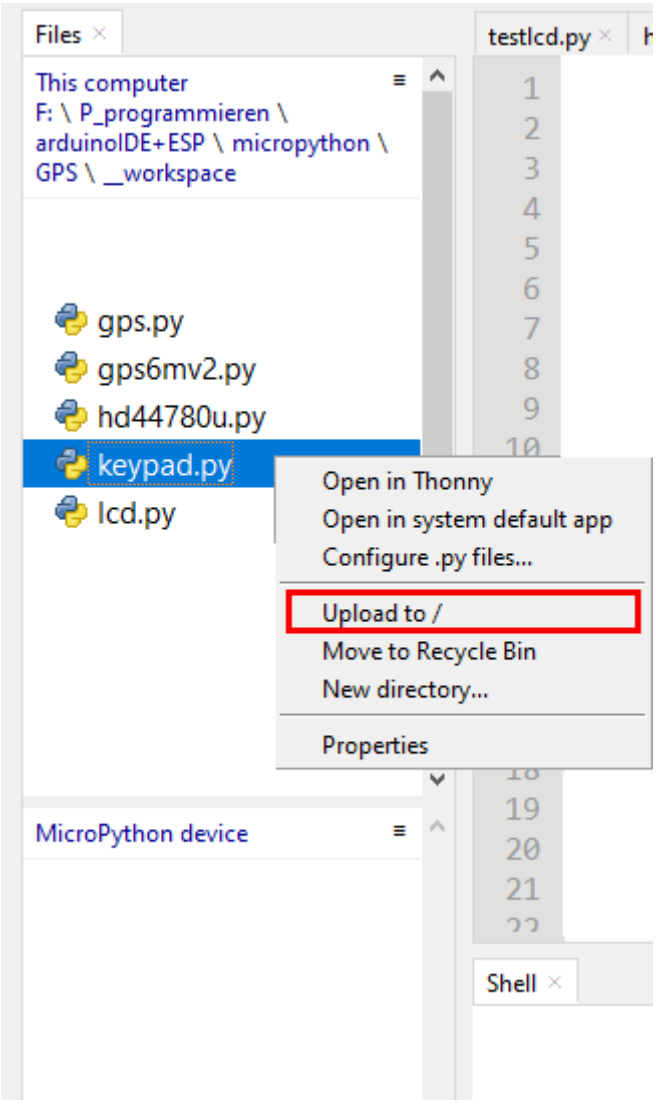
Unsere Programme geben wir im Editorbereich ein. Für ein neues Programm öffnen Sie ein Editorfenster durch Klick auf die Schaltfläche **New** oder durch Tastenfolge **Strg+N**.

In der Arduino-IDE werden Libraries bei jeder Übersetzung des Programms neu übersetzt und in den Programmtext eingebunden. In MicroPython müssen Sie fertige Module, sie entsprechen den Libraries der Arduino-IDE, nur einmal am Beginn in den Flash des ESP32 hochladen. Ich zeige das an einem Beispiel.

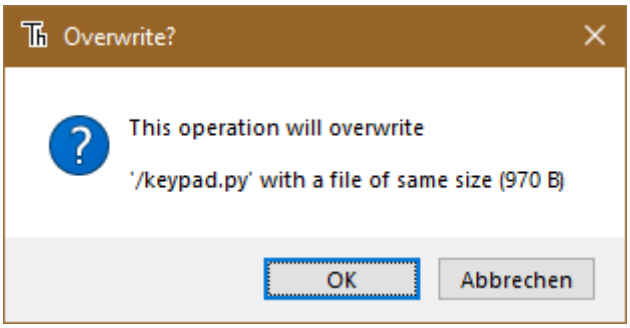
Legen Sie im Explorer in einem beliebigen Verzeichnis einen Projektordner auf Ihrem Rechner an. In diesem Verzeichnis erzeugen Sie einen Ordner mit dem Namen

workspace. Alle weiteren Aktionen starten in diesem Verzeichnis und alle Programme und Programmteile werden dort wohnen.

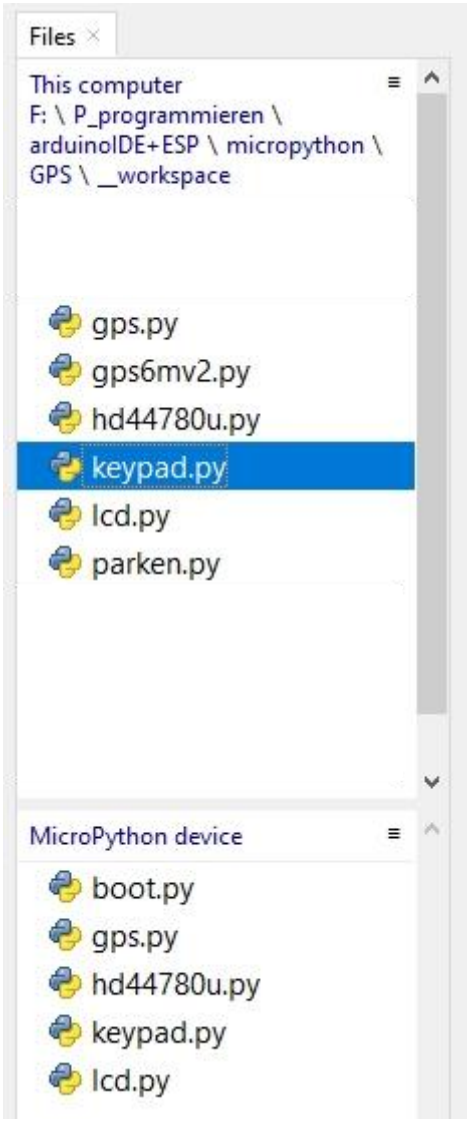
Im Projekt wird die Klasse KEYPAD benötigt. Der Text dazu steht in der Datei keypad.py. Laden Sie am besten gleich alle Module in Ihren workspace. Starten Sie jetzt, falls noch nicht geschehen, Thonny und navigieren Sie Fenster "This Computer" zu Ihrem Arbeitsverzeichnis. Im workspace sollten jetzt die heruntergeladenen Dateien erscheinen. Ein Rechtsklick öffnet das Kontextmenü, und mit Klick auf **Upload to /** wird der Vorgang gestartet.



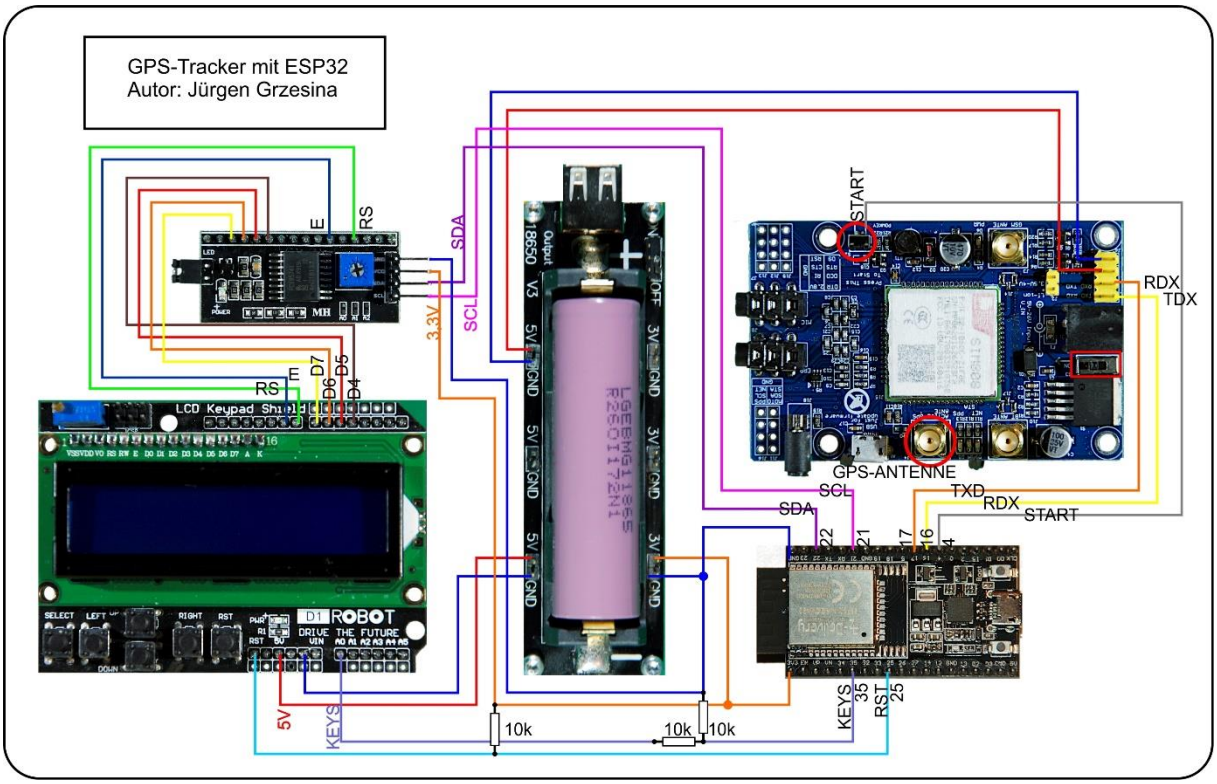
Haben Sie an einem Modul etwas geändert, muss dieses, aber auch nur dieses, erneut hochgeladen werden. Die Sicherheitsabfrage zum Überschreiben beantworten Sie dann mit **OK**.



Nach dem Hochladen der ersten 4 Module sieht das dann so aus. Die Datei boot.py im Device Directory wird beim Flashen der Firmware automatisch angelegt. In diese Datei werden wir am Schluss, wenn alles getestet ist, den Inhalt unseres Programms kopieren. Danach wird der ESP32 bei jedem Start das Programm autonom ausführen. Eine Verbindung zum PC ist dann nicht mehr nötig.

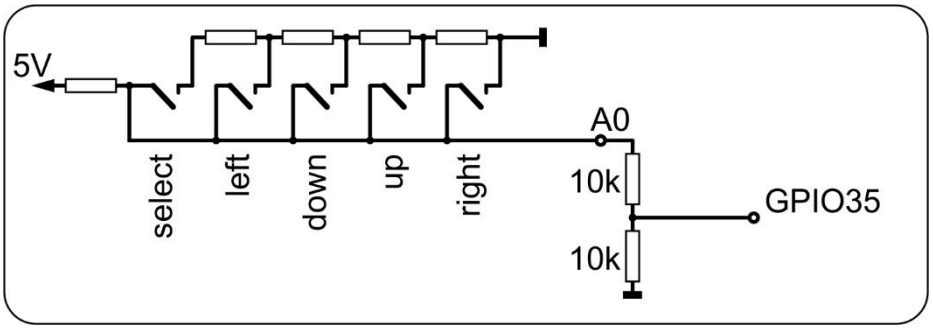


Bevor wir die Module testen können, muss aber erst noch die Hardware angeschlossen werden. Orientieren Sie sich dazu bitte an folgendem Schaltschema.



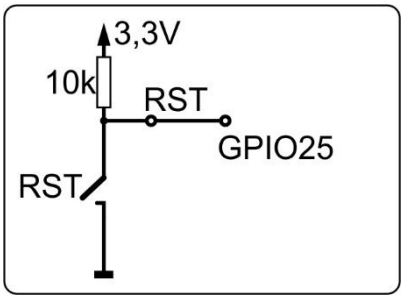
Ein besser lesbares Exemplar bekommen Sie mit dem Download der PDF-Datei .

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. Die Abbildung zeigt eine mögliche Schaltung der Kaskade.



Die Tasten SELECT, LEFT, UP, DOWN und RIGHT liegen an einer Widerstandskaskade, die ihrerseits am heißen Ende an 5V anliegt. Die Taster legen den jeweiligen Pegel an den Anschluss A0 des Keypads, der somit im Leerlauf 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.

Damit auch die RST-Taste am ESP32 nutzbar wird, legen wir einen 10kΩ-Widerstand vom RST-Ausgang aus an +3,3V. Den RST-Ausgang verbinden wir mit dem Digitaleingang GPIO25.



Wenn Sie wie in der folgenden Abbildung einen Draht an den heißen Anschluss der Starttaste an der Unterseite der SIM808-Platine löten, können Sie das Modul über einen Puls von einer Sekunde Dauer vom ESP32 aus einschalten. Das kann beim Booten geschehen, wie ich es in meinem Programm gemacht habe. Die Leitung lieg im Ruhezustand auf 3,3V, der Puls muss also gegen GND erfolgen. Mit einem Puls von mehr als 3 Sekunden Dauer wird das Modul wieder ausgeschaltet.

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().**

```
from machine import ADC,Pin

class KEYPAD:

        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
        #self.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)),
                       range(int(100*k),int(300*k)),
                       range(int(440*k),int(850*k)),
                       range(int(900*k),int(1300*k)),
                       range(int(1450*k),int(2000*k)),
                       ]
        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    from machine import ADC,Pin
```

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 Calibrierung vor. Mit dem Faktor k werden die Grenzwerte der Tastenerkennung an den Leerlaufwert ohne Tastendruck angepasst. Die Grenzwerte habe ich nach meinen ersten Messungen nach obigem Schema 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.

Bei der Erstellung von Klassen werden alle Objekte, die Teil der späteren Klasseninstanz, also des von der Klasse abgeleiteten Objekts, werden, mit dem Prefix (aka Vorsilbe) **self** versehen. Anstelle von self wird später der Name des Objekts verwendet. Sie werden gleich den Zusammenhang erkennen, wenn wir die Klasse KEYPAD testen.

Werfen wir zuvor noch einen Blick auf die Methode **key**(). Ich lasse mir, um Streuungen des ADC zu verringern, den Mittelwert von 5 einzelnen Werten bilden. 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. Falls nicht, wird der nächste Bereich geprüft. Hat keiner der Bereiche entsprochen, wurde offenbar keine Taste gedrückt und dafür der Wert 5 zurückgegeben.

Bereiche sind unter anderem auch als Durchlaufmenge für for-Schleifen notwendig. Deshalb muss man wissen, dass zu einem Bereich stets alle Werte größer oder gleich dem ersten bis zum zweiten ausschließlich zählen. Also gilt:

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

Testen wir jetzt die Klasse KEYPAD.

```
>>> from keypad import KEYPAD.
>>> k=KEYPAD(35)
>>> k.key()
```

Wenn Sie keine Taste gedrückt hatten, liefert der letzte Befehl eine

5

RIGHT liefert 0, UP eine 1, DOWN eine 2 und LEFT und SELECT 3 und 4. Sollten sich diese Zuordnungen nicht konsequent ergeben, dann liegt das sehr wahrscheinlich an falsch gelegten Grenzwerten im Constructor. Sie müssen dann die Tastenwerte neu bestimmen und die Grenzwerte sinnvoll festlegen.

Kommen wir zum Display. Warum ich mich für das LCD entschieden habe, habe ich oben schon erläutert. 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 jeweils ein Modul entwickelt, das dieselbe Bedieneroberfläche (aka API) verwendet. Dahinter stehen dann weitere Module, die sich um die spezifischen Befehle einerseits und um die Ansteuerung der Hardware andererseits kümmern. Das Letztere ist für den Anwender meist uninteressant und in der Regel kommt man auch ohne gerätespezifische Befehle aus. Wichtig sind nur die Art der Hardware sowie die Art der Datenübertragung. Es ist aber stets so, dass dennoch die gesamte Befehlsstruktur 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.

```
>>> 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:2x4
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.

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.

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**.

```
"""
#### File: 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
    self.adr=adr
    super().__init__(i2c,adr,cols,lines)
    self.columns = cols
    self.rows = lines
    self.name="LCD"
    self.clear()
    print("this is the constructor of LCD class")
    print("Size:{}x{}".format(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 **Constructor** 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 Bedarfsfall kann das dort auch anderweitig verwendet werden, zum Beispiel für ein BMP280-Modul zur Messung von Luftdruck und Temperatur.

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,0x20,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).

Wenn Sie die ganze Wahrheit erfahren möchten, bitte ich Sie, den Inhalt der Datei ht44780u.py näher unter die Lupe zu nehmen. Für die Funktion des aktuellen Projekts ist das zwar nicht zwingend erforderlich, aber ich weiß, dass Sie wissen wollen, wie die Klassen wirklich arbeiten, was also dahintersteckt. Ich verrate nur so viel, dass die Klasse HD44780U mehr zu bieten hat als die Klasse LCD – Cursor ein- und ausschalten, Display an | aus, wie funktioniert ein LCD-Treibermodul? Interesse geweckt? Na klar!

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 GMS-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.

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 Constructors 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 und 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 drei 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 Konversation 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. Das gleiche Prozedere findet im Zusammenhang mit der AT-Firmware der ESP8266-Module statt. Allerdings ist der Befehlsumfang beim SIM808 bedeutend größer. Aber keine Sorge, für unser

Projekt reichen im Prinzip zwei von den AT-Befehlen. 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 hier das Listing.

```
"""
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.
GPS6MV2 und SIM808 beziehen sich auf die entsprechende
Hardware.
"""
from machine import UART,I2C,Pin
import sys
from time import sleep

class GPS:
    #
    gDeg=const(0)
    gFdeg=const(1)
    gMin=const(1)
    gFmin=const(2)
    gSec=const(2)
    gFsec=const(3)
    gHemi=const(4)

    def __init__(self,disp=None):  # display mit OLED-API
        self.u=UART(2,9600)  # Mit standardPins rx=16, tx=17
```

```
        # u=UART(2,9600,tx=19,rx=18)  # mit alternativen Pins
        self.display=disp
        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
        print("GPS initialized")

    def decodeLine(self,zeile):
        latitude=["","","","","N"]
        longitude=["","","","","E"]

        def formatAngle(angle):  # Eingabe ist Deg:Min:Fmin
            minute=int(angle[1])     # min als int
            minFrac=float("0."+angle[2]) # minfrac als float
            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.Longitude=formatAngle(longitude)
        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):
        line=""
        c=""
        while 1:
            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:
                            return line
                        else:
                            line=""
                    else:
                        if c != "\r":
                            line +=c

    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)
                pass
            print(msg)


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

    def SIMOn(self):
        self.switch.off()
        sleep(1)
        self.switch.on()

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

    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")

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

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


class GPS6MV2(GPS):
    # Befehlscodes setzen
    GPGLLcmd=const(0x01)
    GPGSVcmd=const(0x03)
    GPGSAcmd=const(0x02)
    GPVTGcmd=const(0x05)
    GPRMCcmd=const(0x04)

    def __init__(self,delay=1,disp=None):
        super().__init__(disp)
        self.display=disp
        self.delay=delay  # GPS sendet im delay Sekunden
Abstand
        period=delay*1000

SetPeriod=bytearray([0x06,0x08,0x06,0x00,period&0xFF,(period>>
8)&0xFF,0x01,0x00,0x01,0x00])
        self.sendCommand(SetPeriod)
        self.sendScanOff(bytes([GPGLLcmd]))
        self.sendScanOff(bytes([GPGSVcmd]))
        self.sendScanOff(bytes([GPGSAcmd]))
        self.sendScanOff(bytes([GPVTGcmd]))
        self.sendScanOn(bytes([GPRMCcmd]))
        print("GPS6MV2 initialized")

    def sendCommand(self,comnd):  # comnd ist ein bytearray
        self.u.write(b'\xB5\x62')
        a=0; b=0
        for i in range(len(comnd)):
            c=comnd[i]
            a+=c  # Fletcher Algorithmus
            b+=a
            self.u.write(bytes([c]))
        self.u.write(bytes([a&0xff]))
        self.u.write(bytes([b&0xff]))

    def sendScanOff(item):  # item ist ein bytes-objekt
        shutoff=b'\x06\x01\x03\x00\xF0'+item+b'\x00'
        self.sendCommand(shutoff)

    def sendScanOn(item):
```

```
        turnon=b'\x06\x01\x03\x00\xF0'+item+b'\x01'
        self.sendCommand(turnon)
```

Das Anwendungsprogramm selbst ist dank der verschiedenen Klassen recht überschaubar. 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 GPS-Haupt-Programms

```
from time import sleep
from gps import GPS,SIM808
from lcd import LCD
from keypad import KEYPAD
from machine import ADC, Pin, I2C
i2c=I2C(-1,Pin(21),Pin(22))

#LUA-Pins     D0 D1 D2 D3 D4 D5 D6 D7 D8
#ESP8266 Pins 16  5  4  0  2 14 12 13 15

k=KEYPAD(35)
d=LCD(i2c,0x20,cols=16,lines=2)
d.printAt("SIM808-GPS",0,0)
d.printAt("GPS booting",0,1)
sleep(1)
g=SIM808(4,d)
g.init808()
g.SIMOn()
sleep(10)
g.mode="DDF"
while 1:
    rmc=g.waitForLine("$GPRMC")
    try:
        g.decodeLine(rmc)
        if g.Valid == "A":
            try:
                gga=g.waitForLine("$GPGGA")
                g.decodeLine(gga)
                g.printData()
                g.showData()
            except:
                g.showError("Invalid GGA-set!")
    except:
        g.showError("Invalid RMC-set!")
    wahl=k.key()
    if 0<=wahl<=2:
        d.clear()
        g.Mode=g.AngleModes[wahl]
        g.DMode="pos"
    if wahl==3:
        d.clear()
```

```
        g.DMode="time"
    if wahl==4:
        d.clear()
        g.DMode="height"
    sleep(0.1)
```

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 anfangs angedeutet, eröffnen sich mit dem Einsatz des ESP32 diverse Anwendungen von weiteren Sensoren. Im zweiten Teil dieses Beitrags stelle ich einen BMP/BME280 zusätzlich zum SIM808 in Dienst. Damit entsteht ein Trackingmodul, das neben den Geodaten auch noch Wetterdaten zur Verfügung stellen kann.

Weitere Downloadlinks:

Beitrag als PDF
Episode as PDF