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

Imagine you are on the road with the GPS tracker and are surprised by a storm. Wouldn't it be better if the device could also record air pressure and temperature in addition to the position? What for? Well, because the air pressure usually drops rapidly before a thunderstorm. And when you know this value, you will have time to seek safe shelter. In order to be able to record air pressure values, I have given my GPS tracker a BMP280. It also measures the temperature. Both quantities are recorded with astonishing accuracy. In this article, you can find out what other news about the device and the program is also available. You are very welcome to the second part of

GPS with MicroPython on the ESP32

Hardware - only very little growth

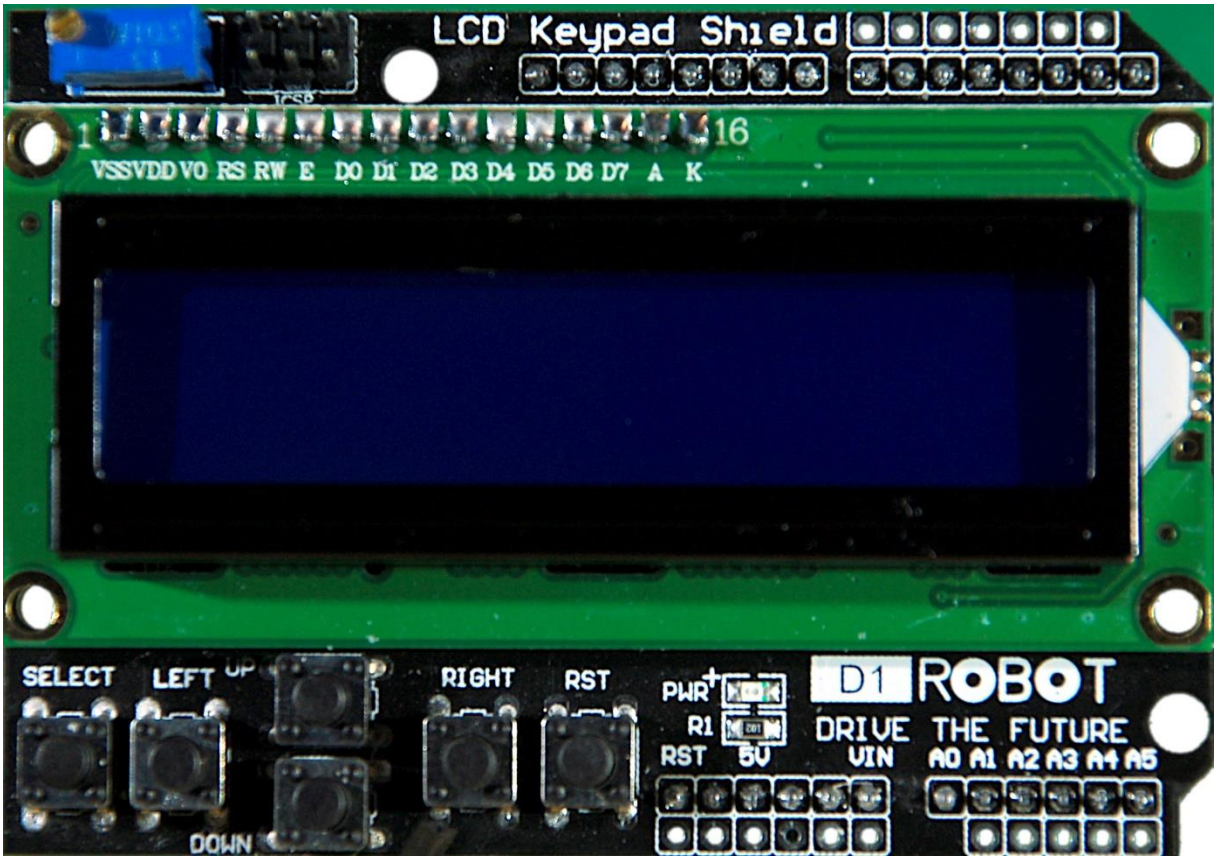
In the list below you will find all the parts for the project. OK, most of it was already used in Part 1. Of course we use these components again. A new addition is a BMP280 module that I put at the end of the list. It must also be mentioned that the 10kΩ resistor at the RST connection of the keypad board has been replaced by the pull-up resistor of the GPIO25. This resistor and another 10kΩ now serve as a voltage divider for measuring the battery voltage on the ESP32. Since there are still

free analog inputs on the controller, you could also add a 5V monitor. A buzzer that screams when the voltages fall below a minimum level is also conceivable. But that is part of the free routine and not a duty. The selection of the ESP32 controller still gives you a lot of freedom when expanding the project. Let your imagination run wild!

So here is the list of ingredients, the recipe comes right after that.

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

Unfortunately, I still haven't found an easy way to temporarily turn off the display backlight to save battery capacity. The switching transistor can only be reached by bending the display up. The base connection of the transistor would then have to be separated from the 5V supply voltage and led out to a pin.

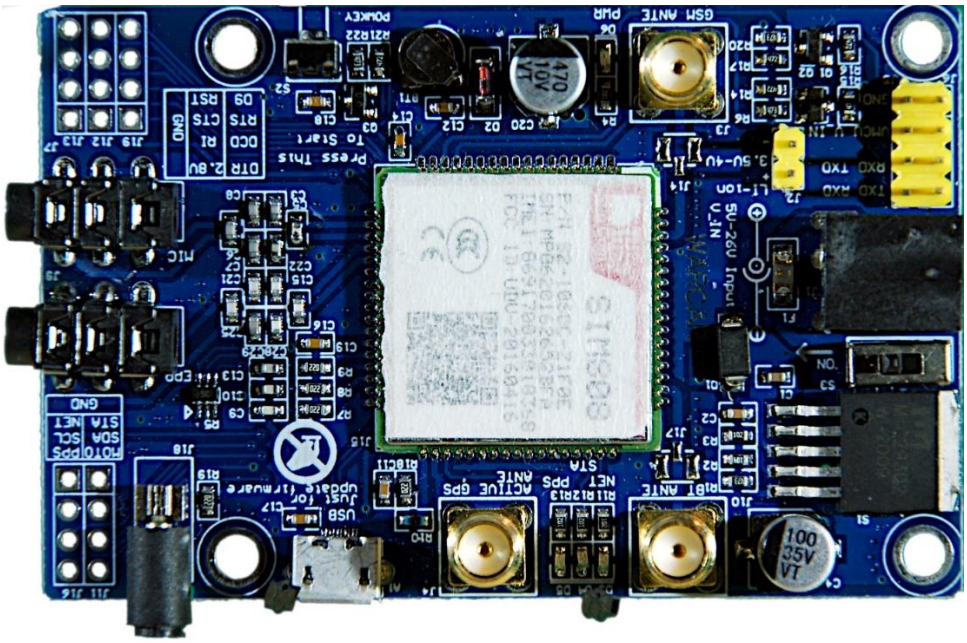


In addition to the 16x2 representation, the display with keypad offers a total of 6 buttons, all of which have a control function in the current project. Five of these buttons supply the voltage from the nodes of a resistor cascade to an ADC input (GPIO35) of the ESP32. The levels are decoded by the ESP32 and assigned to various actions. You will learn more about this later.

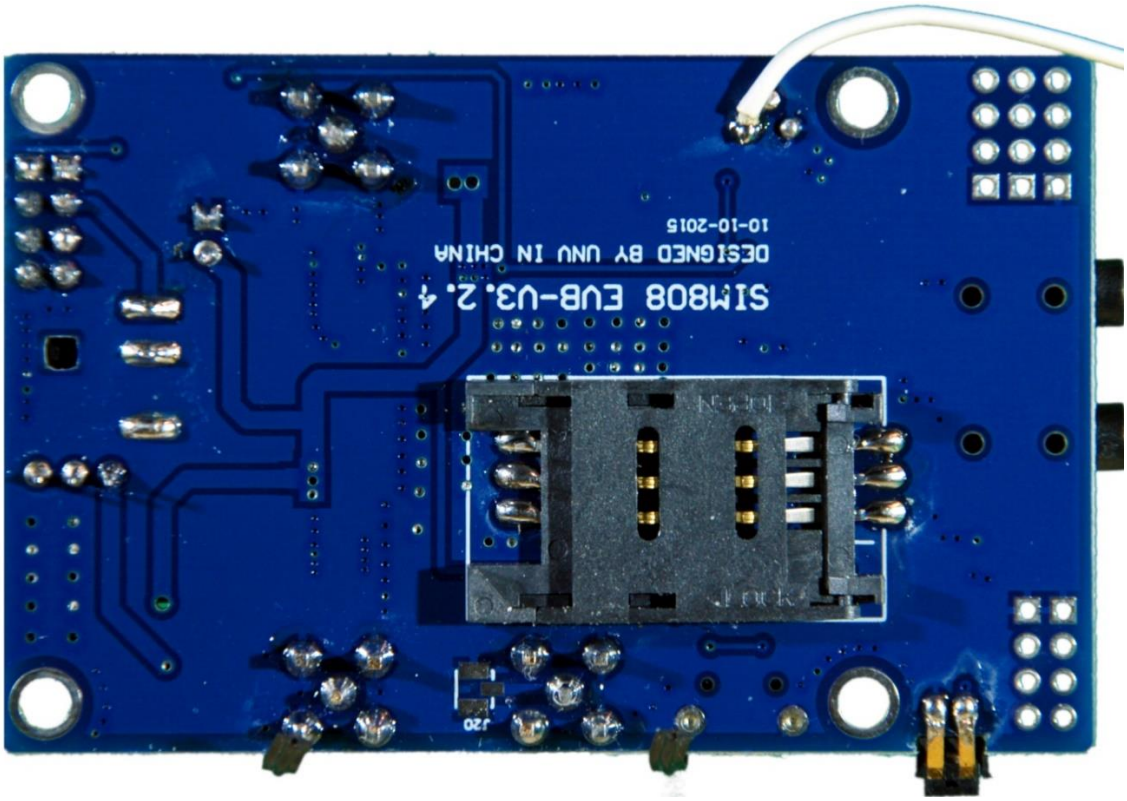
The 6th key, RST, does not follow this pattern. In order to be able to use this button on the ESP32 for other purposes, I selected the RST connection of the keypad board to the digital input GPIO25. In the first part, a pull-up resistor of 10kΩ was used here. I removed this resistor and activated the internal pull-up of the input in the ESP32. How this button is used to expand the functionality of the circuit is explained below in the program discussion.

The serial-parallel converter module for the LCD has got a little brother in the form of the BMP280, which is also connected to the I2C bus.

Like the ESP32, both components are operated with 3.3V. This means that the quiescent level of the bus lines SCL and SDA is also 3.3V and there is no potential risk.



The wiring of the SIM808 has not changed from the first part. There's not much to say about that either. Just four lines are required for the connection to the ESP32, GND, TXD, RXD and the line from the start button of the SIM808 to the GPIO4 pin of the ESP32. The next figure shows the connection of this line to the SIM808 board.



After the hardware, let's take a quick look at the software required. Here is the list.

Used software:
For flashing and programming the ESP:
Thonny oder
µPyCraft
MicropythonFirmware

MicroPython-Module und Programme
GPS-Modul für SIM808 und GPS6MV2(U-Blocks)
LCD-Standard-Modul
HD44780U-I2C-Expansion for LCD-Modul
Keypad-Modul
Button Modul
BMP208-Modul
i2cbus-Modul for standardized access to the bus
Das Hauptprogramm rambler.py
testkeypad.py to test the key decoding

Tricks and information about MicroPython

The interpreter language MicroPython is used in this project. The main difference to the Arduino IDE is that you have to flash the MicroPython firmware on the ESP32 before the controller understands MicroPython instructions. You can use Thonny, µPyCraft or esptool.py for this. For Thonny, I described the process in the first part of the blog on this topic.

After the firmware has been flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without first having to compile and transfer an entire program. I made ample use of it when developing the software for this blog. The spectrum ranges from simple tests of the syntax to trying out and refining functions and entire program parts. For this purpose, I have created a test program for each of the various modules in which tedious import and configuration tasks are summarized. The rambler.py program emerged from one of them.

Such programs are started from the current editor window in the Thonny IDE using the F5 key, which is faster than clicking the start button. I already described the installation of Thonny in detail in the first part.

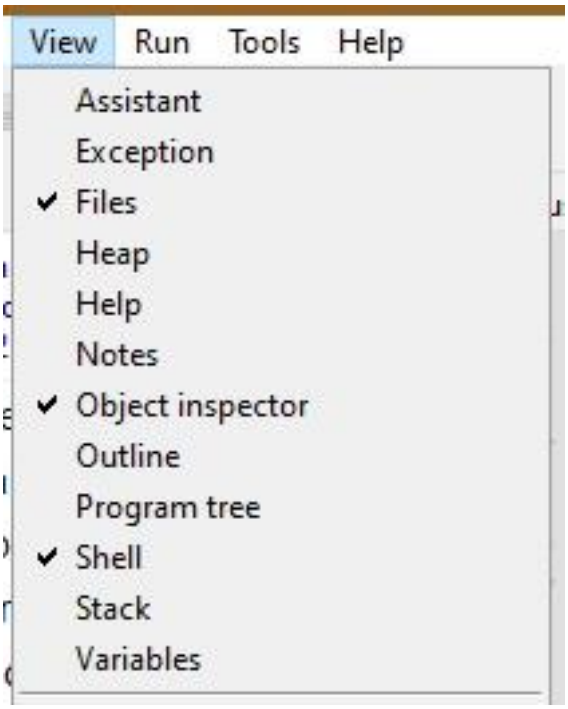
A few other shortcuts help you edit the programs. Python lives from the structuring of the program text through indentations. It can happen that you have to move entire passages one (or more) steps in or out. So that each line does not have to be treated individually, mark the entire block from the first column and press the tab key to indent, Shift (superscript) and tab key to move out. This is quick and helps to avoid mistakes. The indentation depth at Thonny is 4. Spaces are used for indenting, no tabs.

I also found commenting out and uncommenting several lines at the same time very helpful. In test programs, I usually combine several sequences for different purposes. So that everything is not always carried out, I simply comment out lines that are currently not supposed to be processed. Commenting out is done by putting a "#" at the beginning of the line. This can quickly become annoying for several lines. Then I mark the block of lines and press Alt + 3 (not F3!). To remove the comment marks, I press Alt + 4.

To see the attributes of objects, you can use the dir () command. The dir (ADC) command shows you the attributes and methods of the ADC class, which of course you have to import beforehand.

```
>>> 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']
>>>
```

This can be done more clearly via the Object inspector window. If it is not yet displayed, you can open it via the View menu.



The object inspector has two folders. Attributes is the more interesting one. It represents the properties and property values of objects. The illustration shows this for the ADC class. In the terminal window, type ADC and press Enter.

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

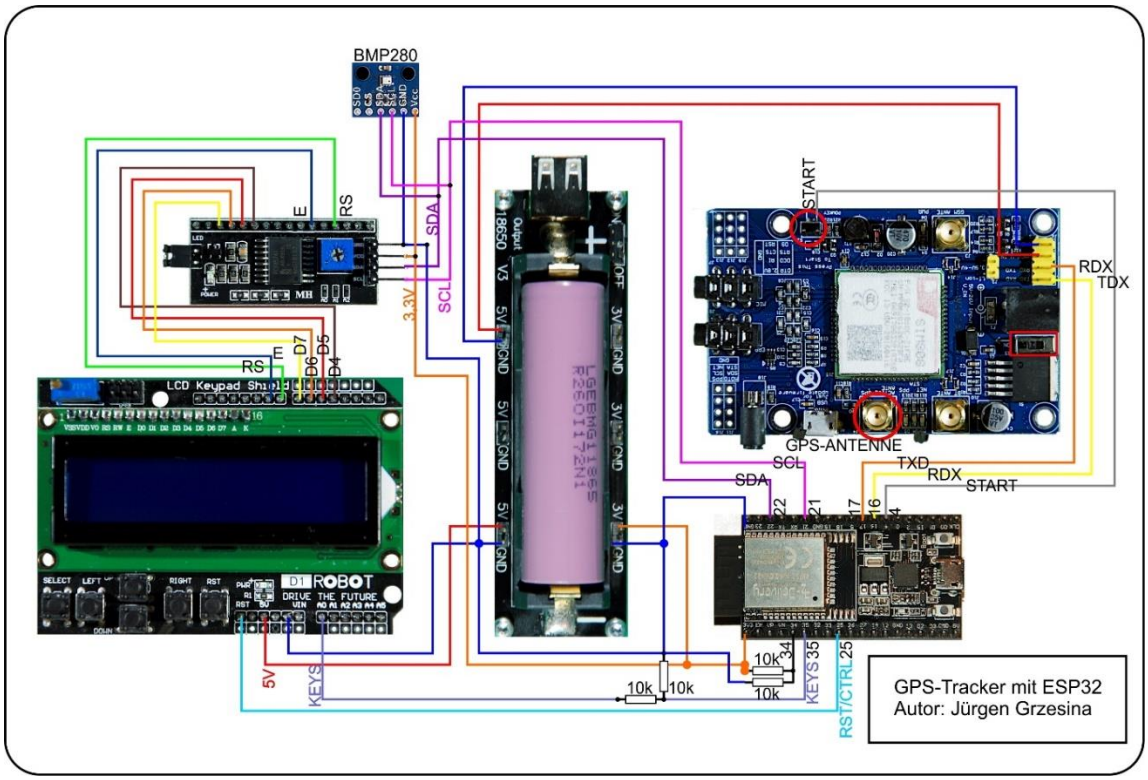
Object inspector

type
@
0x3f4
1729
8

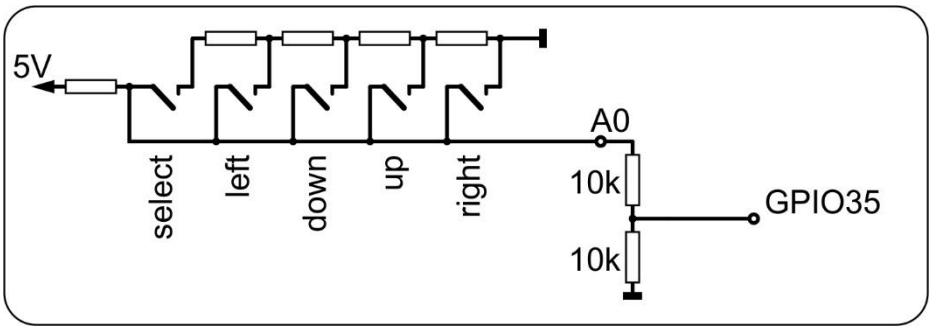
Data | Attributes

Name	Value
ATTN_0DB	0
ATTN_11DB	3
ATTN_2_5DB	1
ATTN_6DB	2
WIDTH_10BIT	1
WIDTH_11BIT	2
WIDTH_12BIT	3
WIDTH_9BIT	0
atten	<function>
read	<function>
read_u16	<function>
width	<bound_m

Before we look at the program modules, the BMP280 must first be connected and the voltage measurement of the 3.3V line must be prepared. Please use the following circuit diagram as a guide. You get a more legible copy with the Download of the PDF-File .



A few remarks on the circuit should clarify its function. The LCD keypad is supplied with 5V, I mentioned that above. The ESP32 works with a maximum of 3.3V for the supply voltage and for the GPIO pins. Therefore the 5V level at the inputs and outputs of the LCD and the keypad must be reduced to ESP32-compatible values. This is done by the voltage divider from the two 10kΩ resistors.



The figure shows a possible circuit of the resistor cascade with the SELECT, LEFT, UP, DOWN and RIGHT buttons. The first resistor on the hot end is at 5V. The buttons switch the respective level through to connection A0 of the keypad, which thus carries approx. 5V when idling. We halve the voltage applied here using a voltage divider made up of two 10kΩ resistors to a tolerable 2.5V. We connect the center tap to the analog input GPIO35 of the ESP32. We are making further adjustments in the KEYPAD class.

So, the hardware is ready, let's test the buttons on the LCD keypad. Let's start first on foot with REPL, the MicroPython command line, in the terminal area. We import the ADC class and the pin class from the integrated module machine, create an ADC object on GPIO35 and set its properties to 12-bit width (0… 4095) and maximum measuring range, ADC.ATTEN_11DB, with which the ESP32 has a maximum Can detect a voltage of approx. 3.3V.

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

The result of the last command should return a value around 2500. Repeat the read command, but first press one of the keys. The following values came to light for me:

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

For an easier query of the keys in the main program, I built a module around these values. It contains the KEYPAD class, which in turn has two methods, the constructor, that is the __init __ () method and the key () method. __init __ () processes basic things for every class, creates instance variables (aka attributes),

defines interfaces and thus creates the environment for an object from the blueprint of this class when called. An ADC object is defined here and a first measurement is initiated. Calling the instance method key () from the class definition is not yet possible at this point because the range list keyRange has not yet been defined. Therefore, the first measurement to crank the ADC cannot be made by calling the key method.

Then we determine the calibration factor k and build up the list with the areas that on the one hand follow the guide values of the keys and on the other hand leave a margin open, taking the calibration factor into account. The almost limitless combination of objects in MicroPython enables such clear programming. The formatted text output including numerical variables closes the constructor.

The key () method provides an integer from 0 up to and including 4 as a return value, which corresponds to one of the keys. To make it easier to deal with, I've defined the Right, Up, Down, Left, and Select class attributes as constants that can be used in place of the numbers.

```
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
```

This is how the constructor works

The key values fluctuate on the one hand due to measurement errors of the ADC (aka analog-digital converter) and on the other hand due to differences in the supply voltage. The constructor of future keypad objects therefore calibrates them when they are called. With the factor k, the limit values of the key recognition are adapted to the idle value without pressing a key. After my first measurements, I set the limit values with the help of a small test program (testkeypad.py) so that the areas do not overlap. I have grouped these range objects (in MicroPython everything is one object) in the keyRange list object. The individual fields are addressed by a list pointer, called an index.

The class as the blueprint of an object (aka instance) defines the ingredients for this object (aka instance). Every object that is supposed to belong to the instance itself is given the prefix self. In this context, Self represents the name of the object that is later derived from the class. For functions that are referred to as methods in this consensus in object-oriented programming, this self is the first parameter in the parameter list. It can also be the only parameter, but it cannot be omitted. Instance methods, like all functions, are declared using the def keyword. If an instance method is called within the class definition, the name of the method must also be preceded by a self. The prefix self is always separated from the subsequent name by a period. The constructor in the form of the __init __ () method creates the instance variables, their initial values and other objects when called. From the outside, the constructor bears the name of the class. You can see these relationships well from the KEYPAD class definition and the test program.

How does the key () method work? In order to reduce the spread of the ADC, I have the mean of 5 individual measurements taken. This is done by the first for loop. In the following for loop, I check whether the mean value is in the range that is addressed by the index. If so, the index is returned as a function value, it encodes the selection. If the ADC value is not in the addressed area, the next area from keyRange is checked. If none of the ranges matched, then apparently no key was pressed and in this case the value 5 was returned.

In addition to other sources, areas are also required as an index pool for for loops, as in the case of the key () method. Therefore you have to know that all integer values from the first inclusive to the second exclusive always count for a range. Of course, there are various other options for defining areas, more on that another time. To introduce a few examples:

It is
range (0.5) = range (5) = 0,1,2,3,4
range (23.24) = 23

And that is also possible

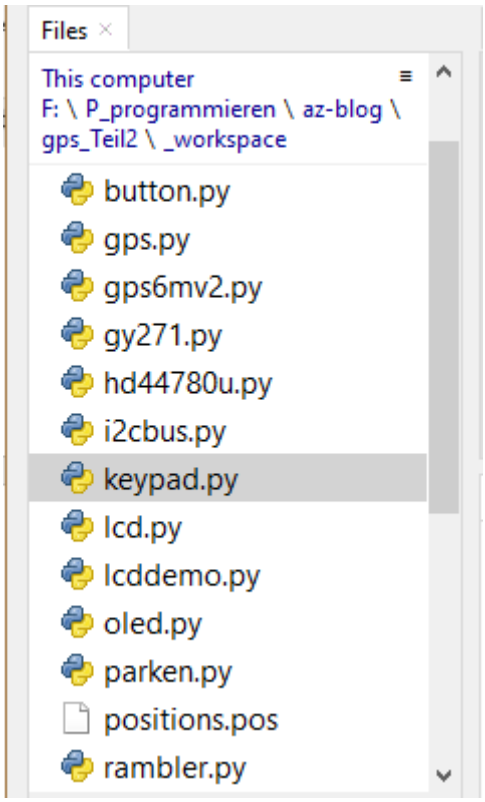
```
>>> List = [0,1,2,3,4,5,6]
>>> for i in range (4):
    print (list [i])
```

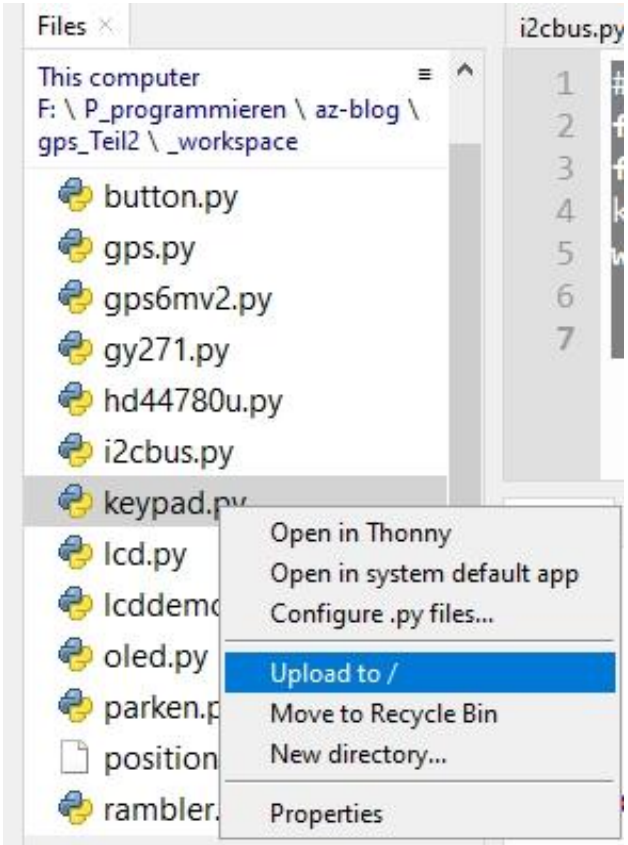
```
0
1
2
3
>>> for i in range (3,6):
    print (list [i])

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

1
3
5
>>>
```

Now let's test the KEYPAD class. We use the test program testkeypad.py mentioned above for this. Here you can see the very short listing.

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

Bring keypad.py and testkeypad.py into Thonny's working directory and copy keypad.py to the ESP32.





Open testkeypad.py in an editor window and start the program with the F5 key

If you don't hold down a key, a column of five will now appear in the terminal

5
5
5…

Press the keys one after the other and check the output. Cancel the program with Ctrl + C when the cursor is in the terminal window.

k is the name of the KEYPAD instance in the test program, it replaces the keyword self. The keyRange list is a property (aka attribute) of the instance k. You can recognize this in the class definition by the prefix self. In the case of object k, the name k replaces this self. The keyRange list can therefore be called up (aka referenced) as follows.

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

If the number assignments to the keys are incorrect, RIGHT delivers 0, UP a 1, DOWN a 2 and LEFT and SELECT 3 and 4, then this is most likely due to incorrectly set limit values in the constructor. You then have to redefine the key values and set the limit values sensibly. After the KEYPAD constructor has generated the ADC object a, you can use this to query the ADC values of the keys. You are instructing the ADC to measure the voltage at input GPIO35 and output it as an integer. With the setting a.width (ADC.WIDTH_12BIT) you get numerical values between 0 and 4095 inclusive.

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

Let's get to the display. In addition to the display, the keypad has 6 buttons. That was the reason for choosing this module. On the other hand, I also like to use OLED displays; they are smaller and also allow simple graphical representations. So that a program can cope with both LCDs and OLEDs without changes, I have developed a module for both cases that uses the same user interface (aka API). This is followed by further modules that take care of hardware-specific commands on the one hand and communication with the hardware on the other. Both are usually of no interest to the user and are done in the background by libraries or modules, which are often referred to as device drivers.

As a rule, you can usually get by without these device-specific commands. With the modules at hand, however, it is always the case that the entire command structure across all classes is ultimately available to the user. This is made possible by inheriting classes. This means that all essential commands, represented by the methods of the classes, are in the same namespace. The LCD class inherits from PCF8574U and this in turn inherits from HD44780U. Ultimately, all methods from each of the three classes are available if you only import the class LCD in the following way. The constructors of the father (PCF8574U) and grandfather (HD44780U) of the class LCD report in sequence when the object d of this class is instantiated.

```
>>> 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 the HD44780U class
Size: 16x2
Constructor of PCF8574U class
this is the constructor of the LCD class
Size: 16x2
>>> d
<LCD object at 3ffe9540>
>>> you (d)
```

The last command gives you an extensive list of methods and attributes, the names of which you can find in one of the three class definitions. You can get a nicer list in the Object inspector.

The same commands are superficially available for the LCD class as for the OLED class. Methods that do not make sense in one of the classes are secured in such a way that nothing happens, so there is no program termination or error message. For the present project, this means that you could always direct the display to an OLED display if you could find another solution for the button control. The method calls to the display do not have to be changed, only the constructor call needs to be adapted and of course you have to import the OLED classes

Next, you can study and test the lcd.py module. I use an adapter with the PCF8574 chip for the wiring, which converts I2C signals from the ESP32 to the parallel output to the LCD. You control the 2 + 4 (+2) control lines of the LCD module via 2 bus lines. And that including level conversion from 3.3V I2C of the ESP32 to 5V LCD input on the keypad. The connection distribution is as follows (see also schematic).

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

According to the use of the upper data nibble on the chip, the class PCF8574U is imported. U stands for 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
```

The important methods from this module are the constructor and clearAll () as well as writeAt ().

In the constructor __init __ (), which is called by LCD () in the program, we pass an I2C object that must have already been defined in the main program. In the current project, the bus is also used for a BMP280 module for measuring air pressure and temperature. I will come to this module later.

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

Then the hardware address of the PCF8574 is transferred if it differs from the default address 0x27. This is followed by the number of columns and rows. It looks like this in the main program.

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

The writeAt () method takes the string to be output as the first parameter, followed by the column and row position in the display. Please note that both counts start from zero.

The method

clearAll ()

clears the entire display and places the cursor at position 0,0 (= top left corner = home position).

Well, the most important thing for a GPS application is: How do I address the GPS services of the SIM808?

Three stages lead to success. The first stage is purely manual, you have to slide the small slide switch right next to the pipe socket for the power supply in the direction of the SIM808 chip. A red LED lights up next to the GSM antenna socket.

A little further to the left is the start button. If you press this for approx. 1 second, two more LEDs light up between the other two antenna sockets, the right one flashes. The active GPS antenna should already be connected to the left screw socket. It is best to place them near a window.

So that you don't have to open the housing of your GPS receiver every time to start the SIM808, I recommend that you do the same and solder a cable to the hot connection of the start button. Viewed from above, it is the right one if the pipe socket also points to the right. You can now start the SIM808 by defining a GPIO pin of the ESP32 as an output and switching from high to low and back to high for one second. I intended pin 4 for this.

When the constructor for the GPS object is called, the number of the pin is transferred together with the display object as a parameter.

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

If no display object (d) is passed, there is also no output on the LCD or OLED. There is no error message, but the key control works. Almost all important results are also output in the terminal window.

The GPS class does most of the work. As mentioned, the constructor expects a display object that must be defined in the calling program or must already be known. A serial channel to the SIM808 is opened at 9600 baud, 8,0,1, then the instance variables are set up to record the GPS data.

The waitForLine () method does what its name says, it waits for a NMEA sentence from the SIM808. The type of NMEA sentence that is expected is given as a parameter. If the record is complete and free of errors, it is returned to the calling program. In the current version of the program, $ GPRMC and $ GPGGA records can be received. They contain all relevant data such as validity, date, time, geographical latitude (latitude, from the equator to the poles in degrees) and longitude (longitude from the zero meridian +/- 180 °) as well as height above sea level in meters.

The decodeLine () method takes the received record and tries to decode it. This method contains a local function that converts the angle specifications into the formats degrees, minutes, seconds and fractions, degrees and fractions, or degrees, minutes and fractions, according to the specification of the mode attribute.

The method printData () outputs a data record in the terminal window. showData () returns the result to the display. Because only a two-line display is used, the display must be divided into several sections. The keys on the keypad take control.

You are probably most interested in the methods of class SIM808, because this is how the GPS data is only processed in the ESP32.

Basically, the data exchange between ESP32 and SIM808 takes place via a serial data connection with 9600 baud, 8,0,1. This means that 9600 bits are transmitted per second, whereby a data frame (aka frame) consists of 8 data bits, 0 parity bits and one stop bit. The start bit is mandatory and is not mentioned in this list. A data frame thus comprises 10 bits, the LSB (aka Least Significant Bit) is transmitted first after the start bit (0). The stop bit (1) completes the transmission. At the TTL level, a 1 corresponds to the 3.3V level, the 0 to the GND level.

Because the UART0 interface is reserved for REPL, a second interface must be available for communication with the SIM808. The ESP32 provides such a UART2. The connections for RXD (reception) and TXD (transmission) can even be freely selected. For full duplex operation (send and receive simultaneously) the RXD and TXD connections from the ESP32 to the SIM808 must be crossed. You can understand this on the circuit diagram. The default values on the ESP32 are RXD = 16 and TXD = 17. The connection is organized by the gps.GPS class.

This begins when the SIM808 is switched on. If you followed my recommendation and soldered a cable to the power button, you can now switch on the SIM808 with the following command, provided that this cable is connected to pin 4 of the ESP32.

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

Commands to the SIM808 are transmitted in AT format. There is a huge variety of commands that can be looked up in a PDF file. But don't worry, a few commands are basically enough for our project. They are combined in the init808 () and deinit808 () methods.

```
    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 switches on the power supply to the GPS module and AT + CGNSTST = 1 activates the transmission of the NMEA sentences to the ESP32 via the serial interface UART2. The controller receives the information from the SIM808 and provides it in the manner described above via the terminal and LCD.

In addition to the hardware control of the SIM808, the gps.py module also contains the necessary commands for the smaller GPS system GPS6MV2 with the Neo 6M chip from UBLOX. This module is not controlled via AT commands but via its own syntax.

The listing now follows to study the gps module in more detail. It contains a whole range of functions that are only used in the third part of the blog, which deals with

sending and receiving SMS messages. As for the first part, you do not need a SIM card for the article you are currently reading; you will only need one in the third part of the blog series.

The SIM808 class takes care of the hardware control and the data transfer to the ESP32. The GPS class contains methods for decoding the NMEA sentences from the SIM808, for displaying them on the display and for calculating the course.

```
"""
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
*************************
```

The application program has increased in scope compared to the first part. This is because the number of functions has increased from 4 to 11. If you pack it in the boot.py file and upload it to the ESP32, it starts autonomously without the need for the USB connection to the PC. So you are independent in the field. The display takes place via the LCD and control via the keypad buttons. Here is the listing of the extended main GPS program 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)
```

The main program is usually just a pretty package. The real work is done by the methods in the classes that are built around the hardware. You will also recognize layers in the programs in this article, the structure of which is repeated and follows the following scheme.

Hardware - communication driver - basic functions - API - main program

It is the same with the last module that deals with the BMP280. It includes all functions that are necessary to address the register structure of the BME280. This allows the operating mode to be set and measurement results to be queried.

In order for the transport to and from the ESP32 to work, an I2C bus driver is switched on, which translates the internal MicroPython bus commands of the ESP32 into standardized commands and, if possible, adjusts the numerical and text variables. This class I2CBus works in the background and is imported and initialized by the class BME280. By inheritance, the BME280 class automatically also has the I2CBus namespace. In principle, the user of the BME280 class does not need to know anything about this. It is sufficient to know the API of the BME280 class. And here is their 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)
```

A few more comments on the operation of the main program.
• It may take a few minutes for the SIM808 to provide usable results.
• The program starts with the display of degrees, minutes and fractions.
• After pressing the button, please wait until the display is cleared, then release the button. The next display is in the new mode.

Keys:
RST at the start input of waypoints as target specification
right degrees and decimals
Up degrees, minutes, seconds and fractions
Down degrees, minutes and fractions
Left date and time
Select altitude and time
RST + Right Save current position as a waypoint
RST + Up Retrieve waypoints and course and distance display
RST + Down Course and distance from the last saved waypoint
RST + Left air pressure and temperature
RST + Select voltage on the 3.3V line
After the RST functions, the display returns to the previous normal display.

Now you only need to stow all the equipment in a box and then the mail goes off to the site. And who knows, maybe after the test you will pack everything in an elegant housing.

As already indicated, the next article deals with the use of the SIM808 to transmit geodata and other information via SMS. This creates a tracking module or a stationary measuring module that can transmit data via the cellular network via time control or a call trigger. So it is finally easy to determine where the son's man is curving around with Papa's posh carriage or where the runaway dog can be caught or ...

Have fun implementing the project!

More download links:
PDF in deutsch
PDF in english