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

Bernd Albrecht's contributions on the topic of "Easter egg hunt with GPS" inspired me to check whether the project could also be done with a micropython. Well, of course it is when you have the tools you need. And I'll tell you something else right at the beginning: the program is very short and clear - thanks to two classes that I built specifically for the SIM808 and the keypad. I would like to spare you your own search for sources, because in this article I will tell you which tools you need, where you can get them and how they are used. So welcome to the first part of

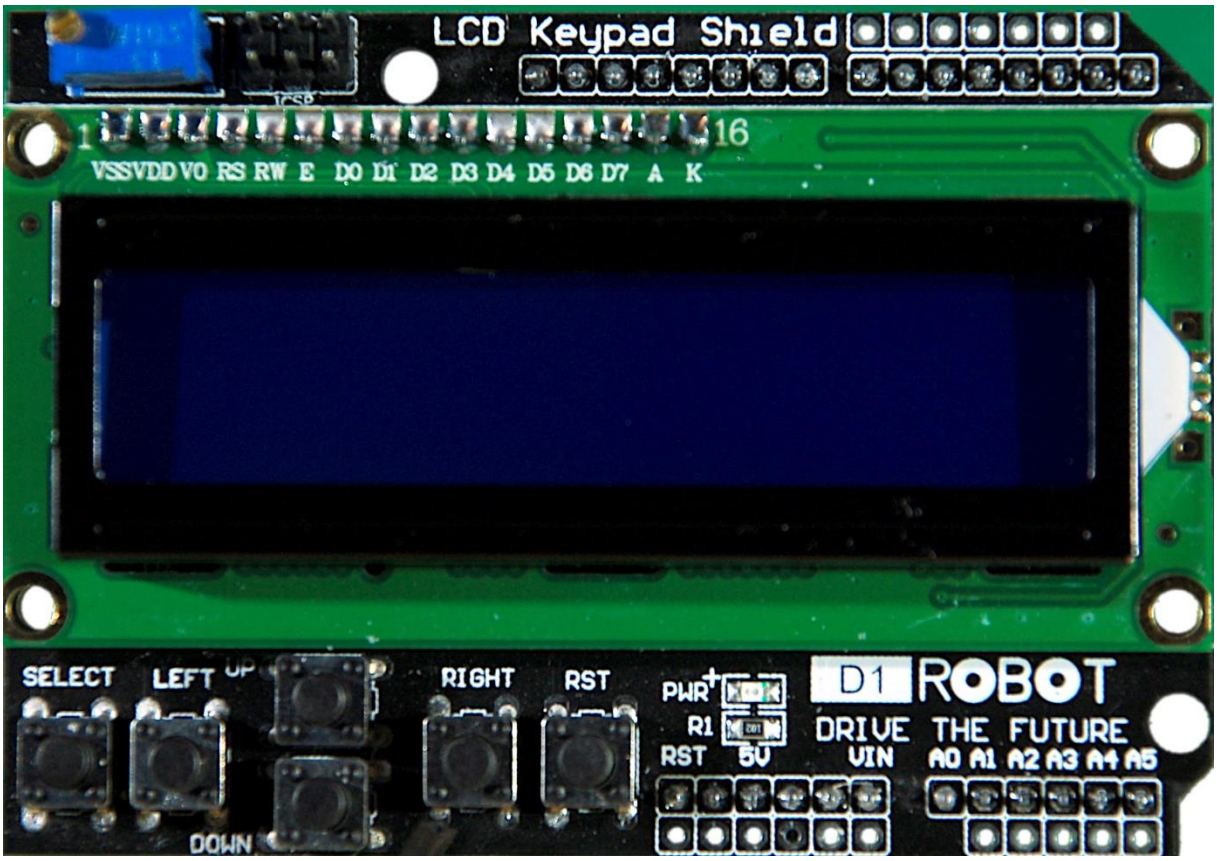
GPS with MicroPython on the ESP32

A couple of hardware considerations

If you have used the GPS to help the Easter Bunny hide the eggs, you already have most of the hardware you need. Which changes I made to the list and why, you will find out immediately afterwards.

1	ESP32 Dev Kit C V4 unverlötet or similar
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

The choice of the controller fell clearly on the ESP32, on the one hand because of the possibility of establishing MicroPython as firmware there, which is not possible on the Arduino. An ESP8266 is also ruled out because there is no second serial hardware interface available. And Raspi would clearly be oversized. The software solutions, like on the Arduino, did not work reliably. In addition, the in-out lines on the ESP8266 are not sufficient for my entire project.

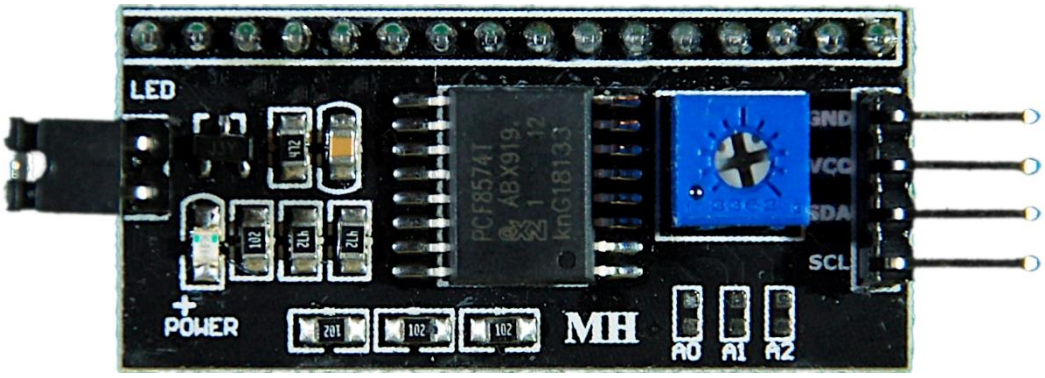


In addition to the 16x2 representation, the display with keypad offers a total of 6 keys, five of which have a control function in the current project. These buttons supply the voltage from the nodes of a resistor cascade to an ADC input (GPIO35). The levels are decoded by the ESP32 and assigned to various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

The 6th key, RST, does not follow this pattern. If the keypad is used as a shield on the Arduino, the button sets the RST input of the AT-Mega328 (with 10kΩ pull-up resistor) to GND potential and thus triggers a cold start. In order to be able to use this button on the ESP32 for other purposes, we have to give it a pull-up resistor against 3.3V (not 5V!). Then you can connect any input of the ESP32 to the RST output of

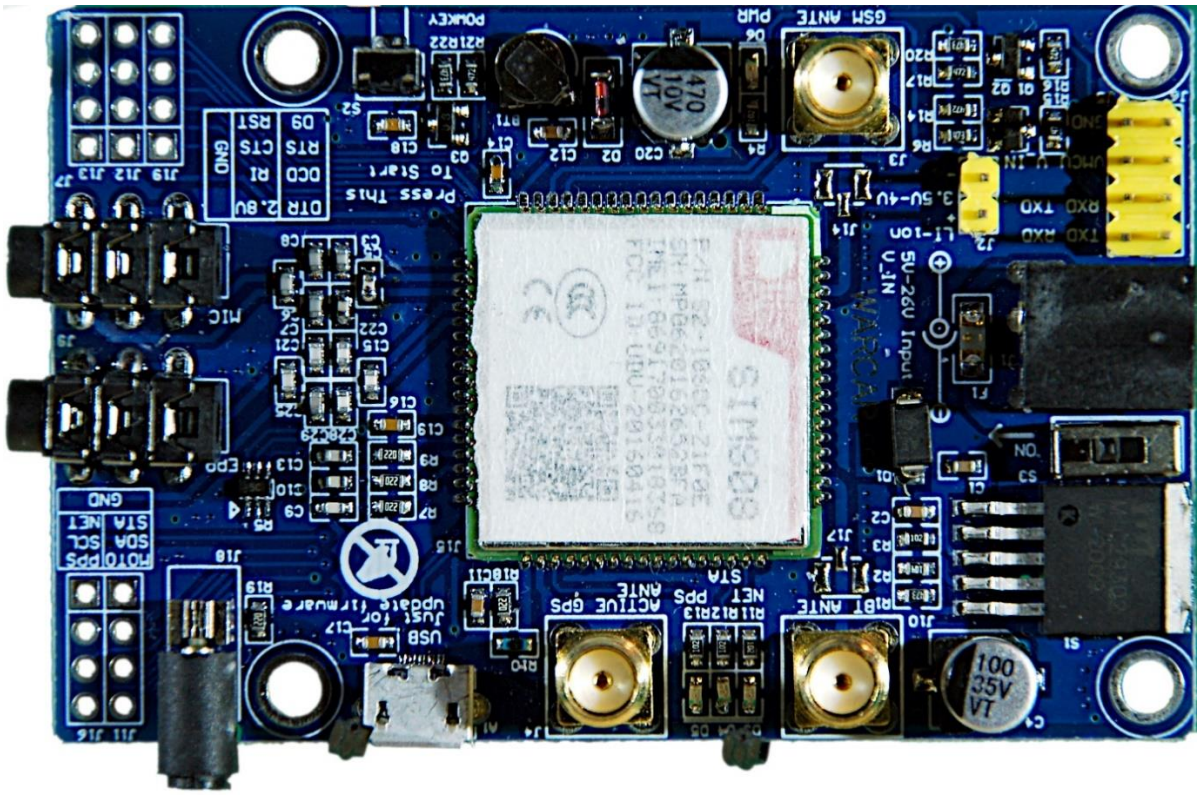
the keypad board and query this key with the controller. I chose the GPIO25 for this purpose. In terms of circuitry, it would have been easier to use an OLED display instead of the keypad with LDC, but I didn't want to do without the buttons that I would otherwise have had to replace otherwise. Because somehow you have to be able to control the construction when you are out in the middle of nowhere.

With the LCD, however, I have another problem. The display needs a supply voltage of 5V to work. This means that the inputs of the display, 4 times data, RS and E, are connected to 5V via pull-ups. But the ESP32 does not tolerate this at its outputs. The voltage level at the pins of the controller should not exceed 3.3V. The approach using voltage dividers was too time-consuming for me. A ULN2803 as a level converter would also have required more cables. Since there was already a MicroPython module for an I2C LCD in my tool box, I decided to buy an I2C adapter for the LCD.



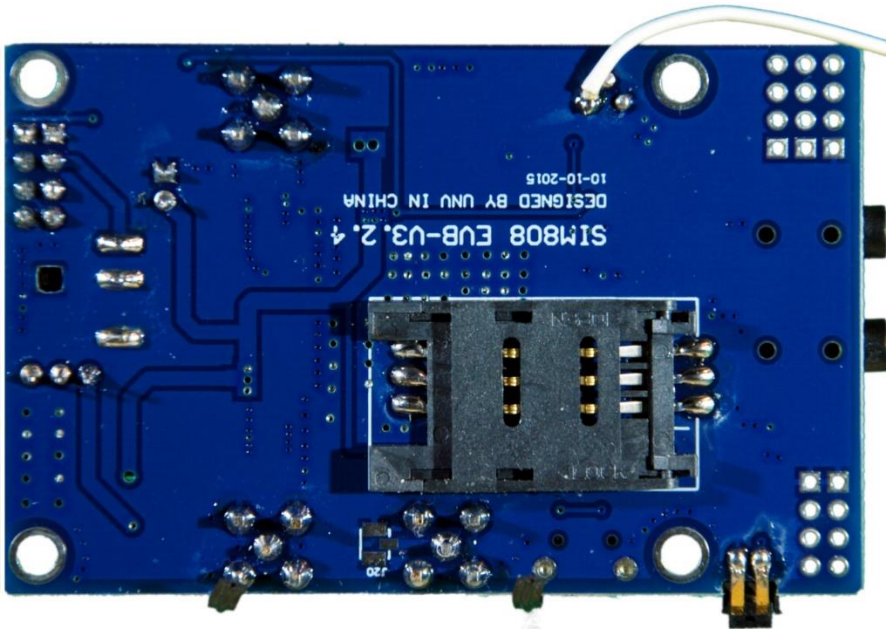
Its inputs are 3.3V-compatible and the outputs can handle the 5V from the display. To connect to the ESP32, only 2 lines, SCL (21) and SDA (22) are now required instead of the 6 direct connections. The I2C bus is also available to other sensors, such as a BMP / BME280, which is intended as an extension of the circuit.

For the power supply I use a battery holder for an 18650 lithium cell because the two operating voltages 3.3V and 5V are already available via plug-in pins. The SIM808 could even be supplied directly from the Li cell. In this case, however, a piece of wire would have to be soldered to the soldering points of the bracket on the circuit board. However, I prefer the 5V connections that can be routed directly to the SIM808 as well as the LCD keypad. 3.3V go to the I2C adapter and the ESP32.



There remains the SIM808. It is normally supplied with voltages of 5..12V via the pipe socket (2.1x5.5), but it also offers a connection for 3.5 to 4.2V for a lithium cell. A connection next to one of the GND pins is available on the yellow pin header for external 5V. On the larger yellow pin base on the top left is GND and exactly below + 5V Vin.

It also makes sense to solder a cable to the pins of the power button, which is then connected to a pin (here GPIO4) of the ESP32. The controller can automatically start the SIM808 when it boots. The button then no longer needs to be operated.



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

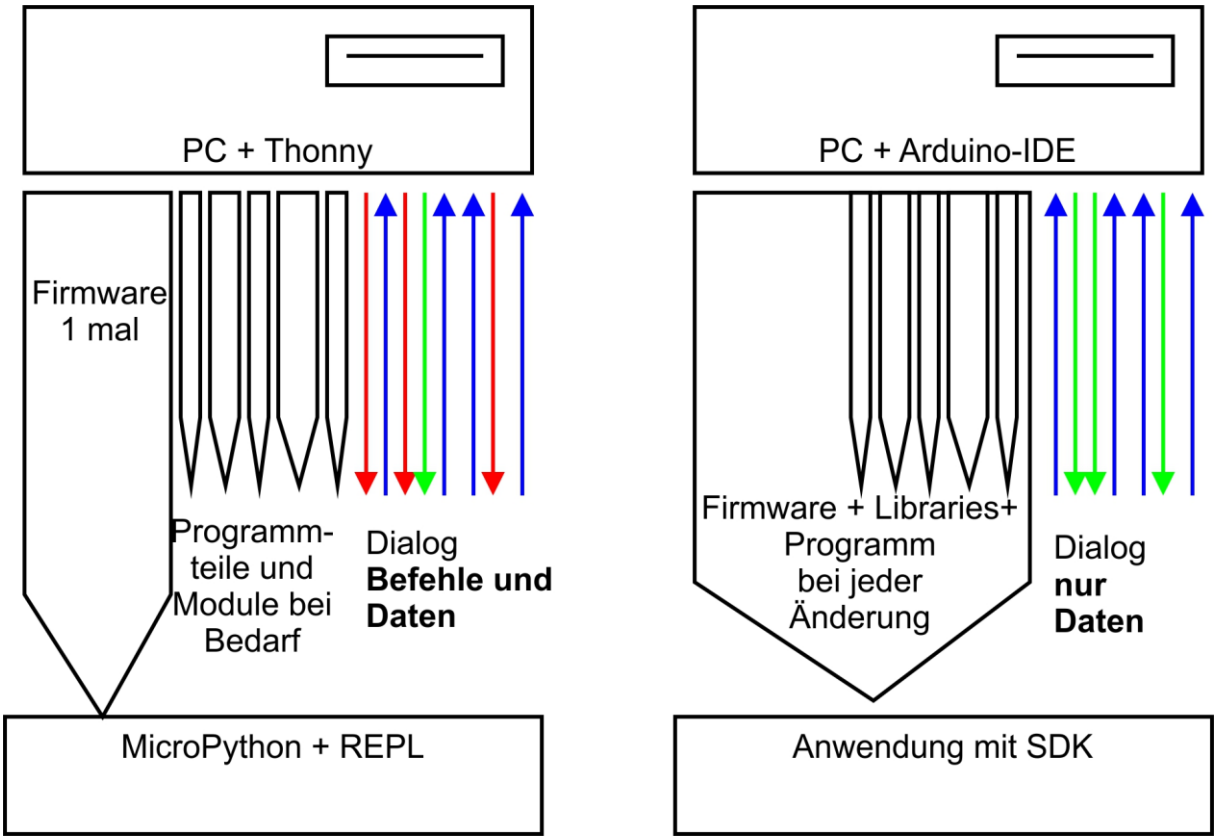
Used software:
For flashing and programming the ESP:
Thonny oder
µPyCraft
MicropythonFirmware

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

A few thoughts on 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 onto the ESP32 before the controller understands MicroPython instructions.

The graphic shows this difference, but also a second, perhaps even more important.



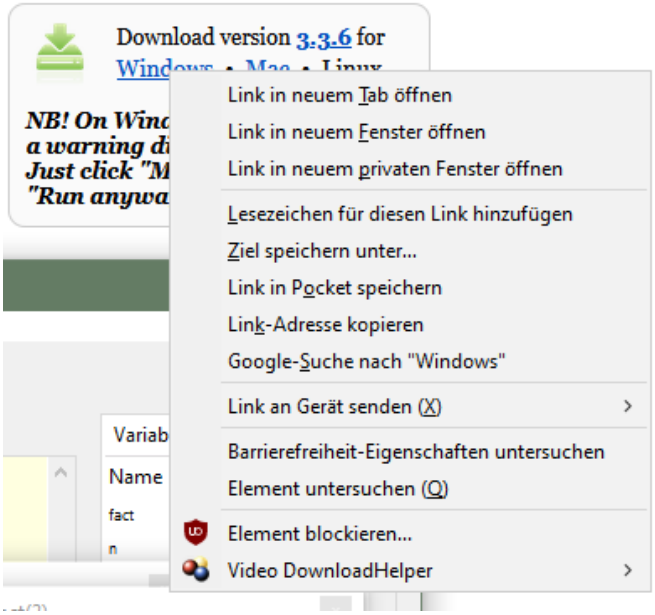
After the firmware has been flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without having to compile an entire program beforehand. Changes to program parts are updated individually under MicroPython. I don't really need to mention that this method is

faster. The following section tells you which steps are necessary to program in this convenient way.

The development environment - example: Thonny

Thonny is the counterpart to the Arduino IDE under MicroPython. In Thonny, a program editor and a terminal as well as other interesting development tools are combined in one interface. You have the working directory on the PC, the file system on the ESP32, your programs in the editor, the terminal console and, for example, the object inspector in one window.

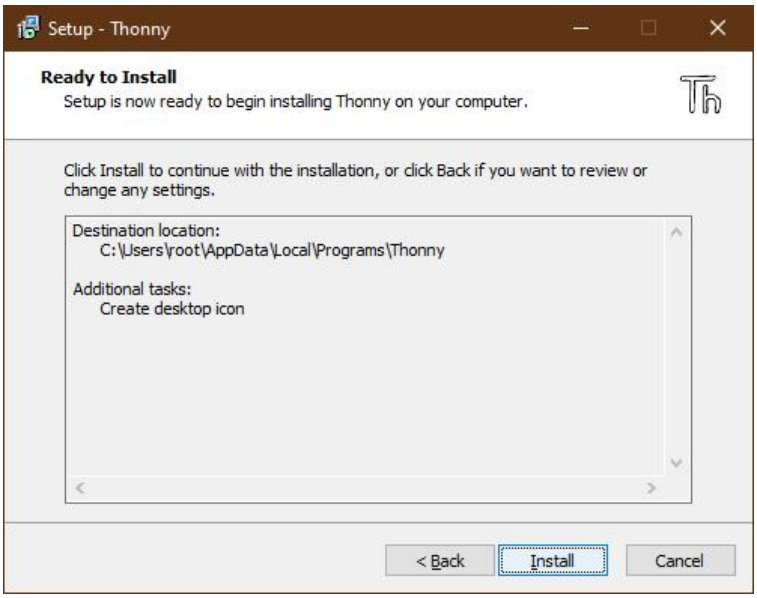
The resource for Thonny is the file thonny-3.3.x.exe, the latest version of which can be downloaded directly from the product page. There you can also get an initial overview of the features of the program.



Right-click on Windows and save target as to download the file to any directory of your choice. Alternatively, you can also follow this direct link.
In addition to the IDE itself, the Thonny bundle also includes Python 3.7 for Windows and esptool.py. Python 3.7 (or higher) is the basis for Thonny and esptool.py. Both programs are written in Python and therefore require the Python runtime environment. esptool.py also serves as a tool in the Arduino IDE to transfer software to the ESP32 (and other controllers).

Now start the installation of Thonny by double-clicking on your downloaded file if you only want to use the software for yourself. If Thonny & Co. is to be available to all users, you must run the exe file as an administrator. In this case, right click on the file entry in Explorer and select Run as administrator.

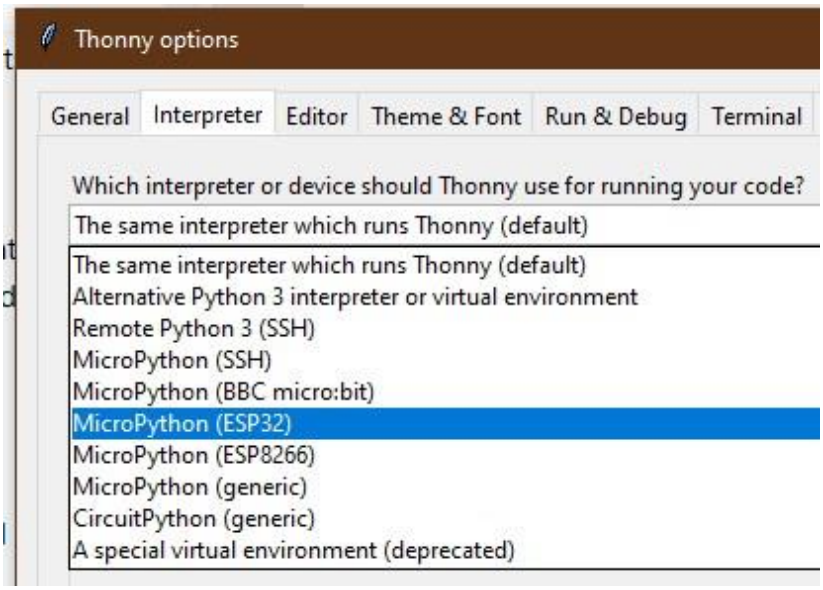
Most likely, Windows Defender (or your antivirus software) will answer you. Click on more information and, in the window that opens, click on Run anyway. Now just follow the user guidance with Next.



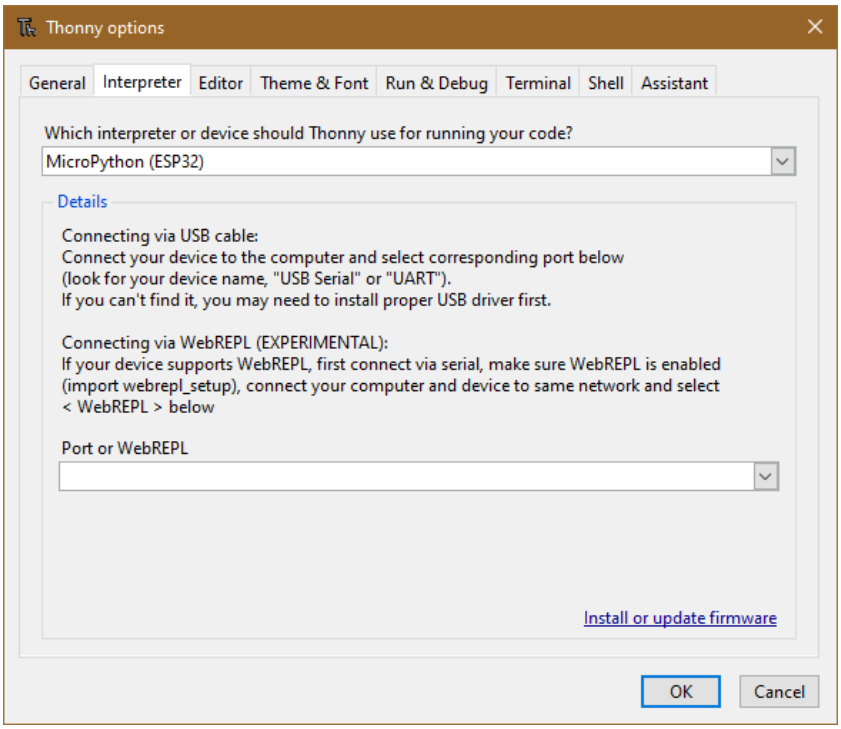
Click on Install to start the installation process.

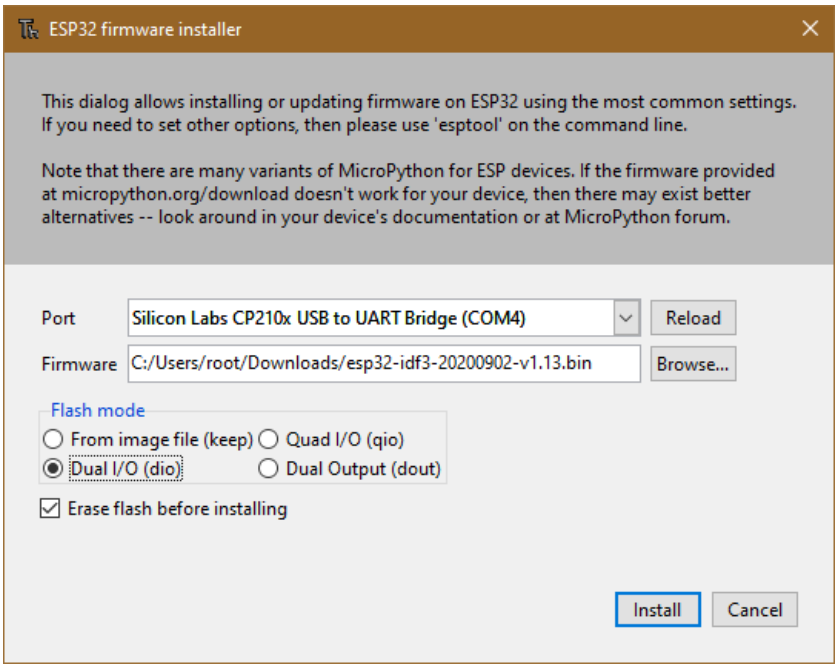
When you start the program for the first time, you specify the language, then the editor window is displayed together with the terminal area.

As the first action, set the type of controller used. With Run - Select Interpreter… you land in the options. For this project, please set Micropython (ESP32).



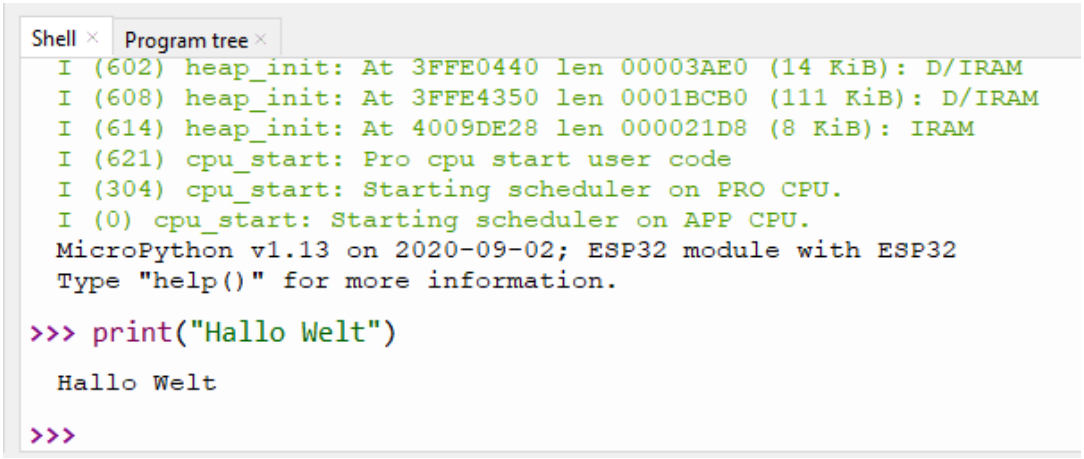
Now download the Micropython firmware for the ESP32 and save this file in a directory of your choice. The bin file must first be transferred to the ESP32. This also happens to Thonny. Call up Thonny Options again with Run - Select Interpreter…. At the bottom right click on Install or update Firmware.





Select the serial port to the ESP32 and the downloaded firmware file. Start the process with Install. After a short time, the MicroPython firmware is on the controller and you can send the first commands to the controller via REPL, the MicroPython command line. For example, enter the following command in the Terminal window.

print ("Hello world")



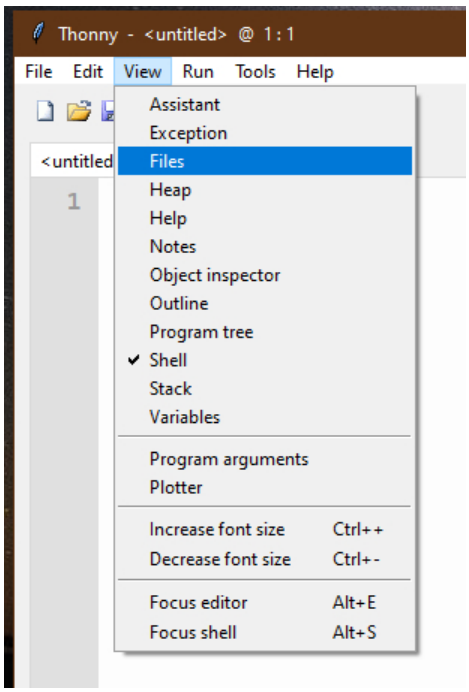
In contrast to the Arduino IDE, you can send individual commands to the ESP32 and, if it is MicroPython instructions, it will respond well. On the other hand, if you send a text that the MicroPython interpreter cannot understand, it will alert you to this with an error message.

>>> print "hello again"

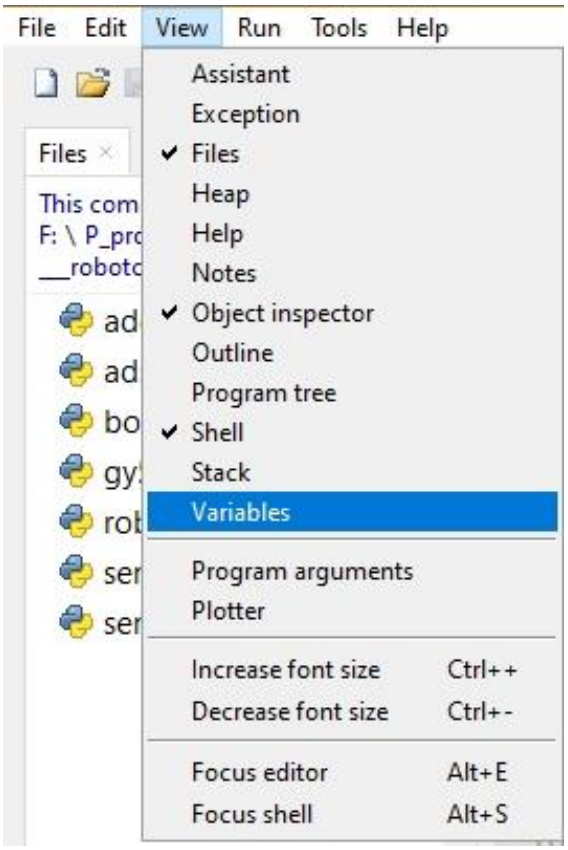
SyntaxError: invalid syntax
Traceback (most recent call last):
  File "<stdin>", line 1
SyntaxError: invalid syntax

To work, however, the overview of the workspace and the device directory is still missing. The workspace is a directory on the PC in which all files important for a project are located. In Thonny his name is This Computer. The device directory is the counterpart on the ESP32. In Thonny it is called MicroPython device. You can display it as follows.

Click on View and then click on Files



Now both areas, the workspace at the top and the device directory at the bottom, are displayed. You can display additional tools via the View menu.



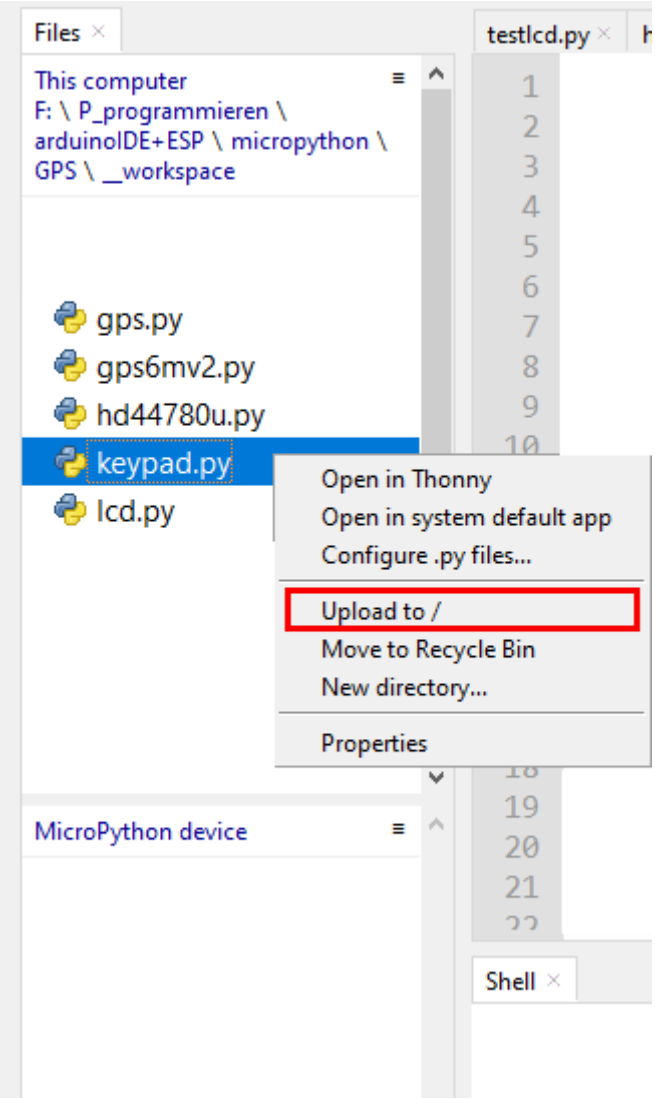
We enter our programs in the editor area. For a new program, open an editor window by clicking the New button or using the key sequence Ctrl + N.

In the Arduino IDE, libraries are recompiled each time the program is compiled and integrated into the program text. In MicroPython you only have to upload finished modules, they correspond to the libraries of the Arduino IDE, to the flash of the ESP32 once at the beginning. I will show this with an example.

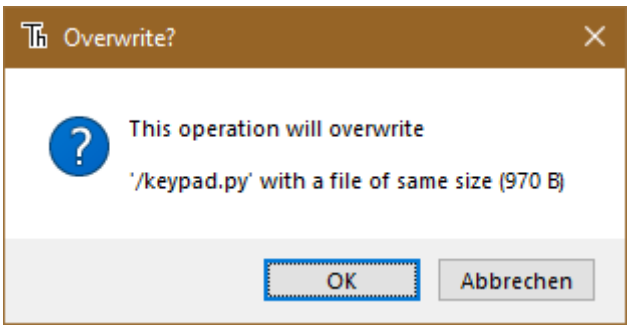
Create a project folder on your computer in any directory in Explorer. In this directory you create a folder with the name workspace. All further actions start in this directory and all programs and program parts will live there.

The KEYPAD class is required in the project. The text for this is in the keypad.py file. The best thing to do is to load all modules into your workspace right away. If you

have not already done so, start Thonny and navigate to your working directory in the "This Computer" window. The downloaded files should now appear in the workspace. A right click opens the context menu and the process is started by clicking on Upload to /.

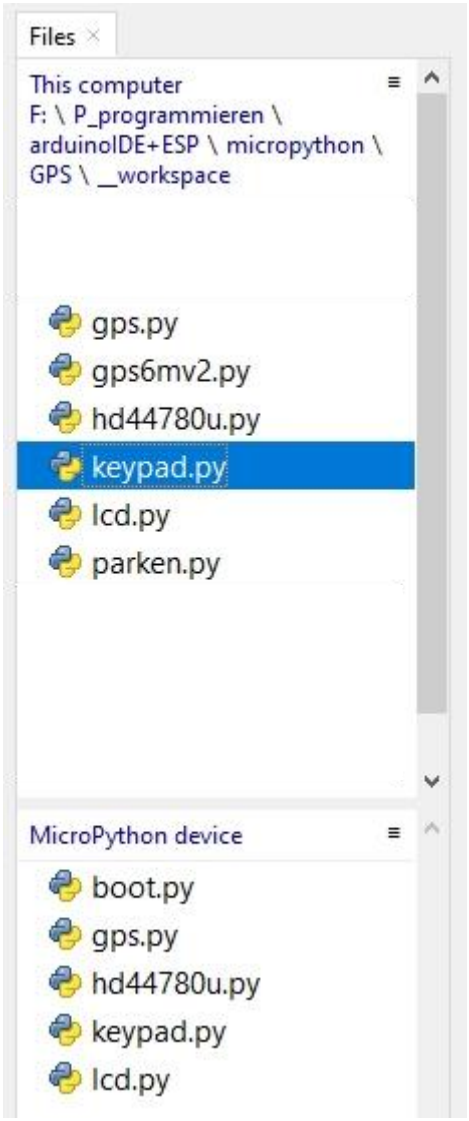


If you have changed something on a module, it must be uploaded again, but only this one. Then answer the security question about overwriting with OK.

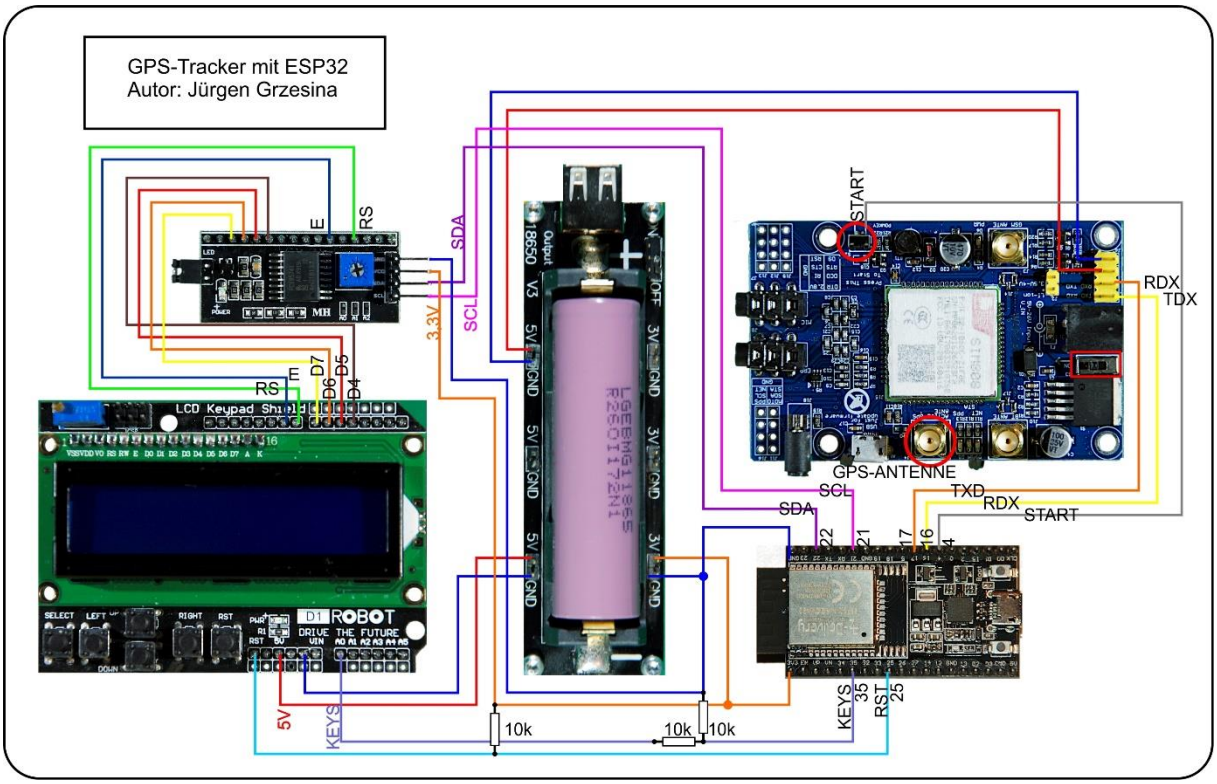


After uploading the first 4 modules it looks like this. The boot.py file in the device directory is automatically created when the firmware is flashed. At the end, when everything has been tested, we will copy the content of our program into this file. The

ESP32 will then run the program autonomously each time it is started. A connection to the PC is then no longer necessary.

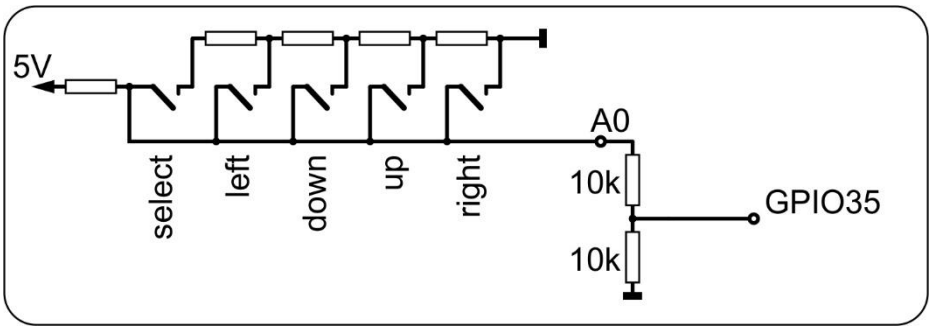


Before we can test the modules, however, the hardware must first be connected. Please use the following circuit diagram as a guide.



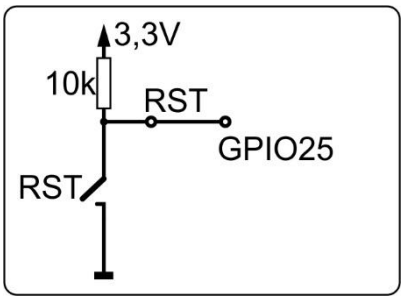
You can get a more readable copy by downloading the PDF file.

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. The figure shows a possible connection of the cascade.



The buttons SELECT, LEFT, UP, DOWN and RIGHT are connected to a resistor cascade, which in turn is connected to 5V at the hot end. The buttons apply the respective level to connection A0 of the keypad, which therefore carries 5V when idle. 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 that the RST button on the ESP32 can also be used, we connect a 10kΩ resistor from the RST output to + 3.3V. We connect the RST output to the digital input GPIO25.



If you solder a wire to the hot connection of the start button on the underside of the SIM808 board as in the following figure, you can switch on the module from the ESP32 with a pulse of one second duration. This can be done while booting, as I did in my program. In the idle state, the line is at 3.3V, so the pulse must be against GND. The module is switched off again with a pulse of more than 3 seconds.

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**.**

```
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
```

The key values fluctuate on the one hand due to measurement errors of the ADC (aka analog-digital converter) on the other hand due to differences in the supply voltage. therefore the constructor of future keypad objects will calibrate 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 according to the above scheme 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.

When creating classes, all objects that will become part of the later class instance, i.e. the object derived from the class, are given the prefix (aka prefix) self. The name of the object will later be used instead of self. You will soon see the connection when we test the KEYPAD class.

First, let's take a look at the key () method. In order to reduce the spread of the ADC, I have the mean of 5 individual values averaged. 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. If not, the next area is checked. If none of the ranges matched, apparently no key was pressed and the value 5 was returned.

Areas are also necessary as a throughput quantity for for-loops. Therefore you have to know that for a range all values greater than or equal to the first to the second count exclusively. So:

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

Now let's test the KEYPAD class.

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

If you didn't press a key, the last command will return one

5

RIGHT delivers 0, UP a 1, DOWN a 2 and LEFT and SELECT 3 and 4. If these assignments do not result consistently, 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.

Let's get to the display. I have already explained why I chose the LCD above. On the other hand, I also like to use OLED displays, they are smaller and also allow simple graphic representations. So that a program can cope with both LCDs and OLEDs without changes, I have developed a module that uses the same user interface (aka API). This is followed by further modules that take care of the specific commands on the one hand and the control of the hardware on the other. The latter is usually of no interest to the user and, as a rule, it is possible to manage without device-specific commands. Only the type of hardware and the type of data transfer are important. However, it is always the case that the entire command structure 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.

```
>>> 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: 2x4
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.

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 for the display do not have to be changed, only the constructor call needs to be adapted.

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 circuit diagram).

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.

```
"""
#### 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
```

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. If necessary, it can also be used for other purposes there, for example for a BMP280 module for measuring air pressure and temperature.

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, 0x20, 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).

If you want to know the whole truth, please examine the contents of the ht44780u.py file. This is not absolutely necessary for the function of the current project, but I know that you want to know how the classes really work, so what is behind them. I'll just tell you that the HD44780U class has more to offer than the LCD class - turn cursor on and off, display on | out how does an LCD driver module work? Aroused interest? Sure, of course!

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 GMS 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.

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 is called for the GPS object, 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 and 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 three 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 the conversation 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. The same procedure takes place in connection with the AT firmware of the ESP8266 modules. However, the scope of commands with the SIM808 is significantly larger. But don't worry, two of the AT commands are basically sufficient for our project. They are combined in the methods init808 () and deinit808 ().

```
    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 by AT commands, but by its own syntax.

The listing follows to study the gps module in more detail.

```
"""
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)
```

The application program itself is quite manageable thanks to the different classes. If you pack it in the boot.py file and upload it to the ESP32, it will start 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 main GPS program

```
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)
```

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 indicated at the beginning, the use of the ESP32 opens up various applications for additional sensors. In the second part of this article I put a BMP / BME280 into service in addition to the SIM808. This creates a tracking module that can provide weather data as well as geodata.

More download links:

Beitrag als PDF
Episode as PDF