Abbildung 1:titel5

Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

After the SMS and GPS functionality of the SIM808, today I will introduce you to the third pillar of the small jack-of-all-trades. That is because it can also make normal calls. Requirements are a SIM card and the use of the appropriate AT commands. There are essentially two of them, but a few more make things really interesting. As in the previous episodes, the ESP32 will send the commands to the SIM808 again. Of course, this time you need headphones or a mini speaker (32 ohms) and a microphone. Headsets with 3.5mm jack plugs are best.

We need an extended functionality for dialing and flow control. For this I use a 4x4 keypad with the digits 0 to 9, the letters A to D as well as * and #. This post deals first with the query of this key matrix, later we will talk on the phone.

We examine two common methods for reading in the keys. The LCD keypad is also used again, mainly the display. But before we start, welcome to the 5th part of the blog episode with the title

GSM and Telefonie with MicroPython on the ESP32 and SIM808

Sure, the display does a good job when starting the cellphone (aka mobile phone) and it allows the number to be called to be entered securely on sight, that's why we're using it again. For convenience, I chose a 16-key keypad for entering phone numbers. Alone with the buttons on the LCD keypad you would have had to do pull-ups to be able to enter all the digits. However - one button on the LCD keypad is still important, because I use it again as an emergency brake in the control loop. We already had that in the previous episodes. Using the emergency brake means terminating the current program precisely without restarting it immediately. All objects, variable contents and function definitions created up to that point are retained for manual access via REPL.

Hardware –moderater Zuwachs

In terms of hardware, compared to part 4, there is a little more, depending on your preferences. Basically, the 4x4 keyboard and a headset are our loyal companions in this episode. The keyboard must be queried by the ESP32. There are 2 variants for the connection, three for the query. I explain two of them in a separate chapter. I will only briefly touch on a third possibility there.

The headset is plugged directly into the SIM808. You will be able to pinpoint this in a picture.

Here is the list of ingredients for the new craft session. With the exception of the last three items, you probably already have everything if you have already worked hard to build the ones in the previous episodes.

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	SIM-Karte (beliebiger Anbieter)
1	4x4 Matrix Keypad Tastatur - 1x Keypad
1	MCP23017 Serielles Interface Modul
1	Headset mit Elektret-Mikrofon

In terms of hardware, compared to part 4, there is a little more, depending on your preferences. Basically, the 4x4 keyboard and a headset are our loyal companions in this episode. The keyboard must be queried by the ESP32. There are 2 variants for the connection, three for the query. I explain two of them in a separate chapter. I will only briefly touch on a third possibility there.

The headset is plugged directly into the SIM808. You will be able to pinpoint this in a picture.

Here is the list of ingredients for the new craft session. With the exception of the last three items, you probably already have everything if you have already worked hard to build the ones in the previous episodes.

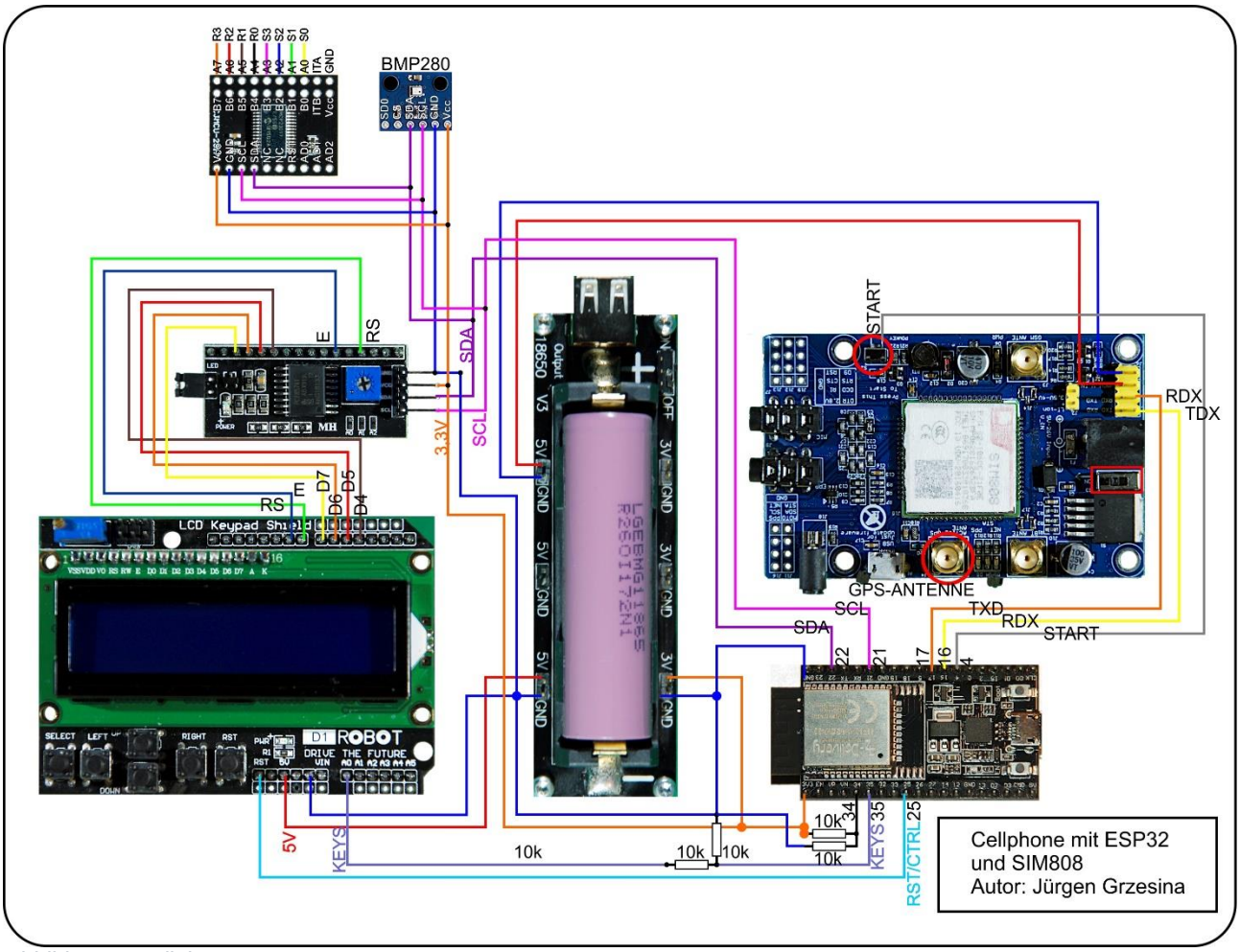

Abbildung 2: cellphone

You can get a more readable copy of the illustration in DIN A4 by downloading the PDF file. Download der PDF-Datei

Die Software

Verwendete Software:
Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:
MicropythonFirmware
Bitte eine Stable Version aussuchen

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

Button Modul
i2cbus-Modul für standardisierten Zugriff auf den Bus
mcp.py hardwarededizierte Methoden für den MCP23017
keypad_p_test.py Testprogramm für das 4x4-Tastenfeld
cellphone.py Hauptprogramm für die Telefonie

Tricks and infos 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 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.

As soon as the firmware is 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. This is exactly what bothers me about the Arduino IDE. When developing the software for this blog, I made ample use of the direct dialog with the ESP32. The spectrum ranges from simple tests of the syntax and hardware to trying out and refining functions and entire program parts. For this purpose, I also like to create small test programs, as in the previous episodes. They form a kind of macro because they combine recurring commands. The keypad_p_test.py program is used here to test the numeric keypad, but more on that later.

Such programs are started from the current editor window in the Thonny IDE using the F5 key. This is faster than clicking the start button or using the Run menu. I also described the installation of Thonny in detail in the first part.

The headset

Let's start with the new components. The connection of headphones and microphone is not critical. A normal headset, as is usual for a PC or laptop, corresponds exactly to our conditions. How the headset is connected to the SIM808 is shown in the illustration. Microphone and headphone plugs are of the usual three-pin design.

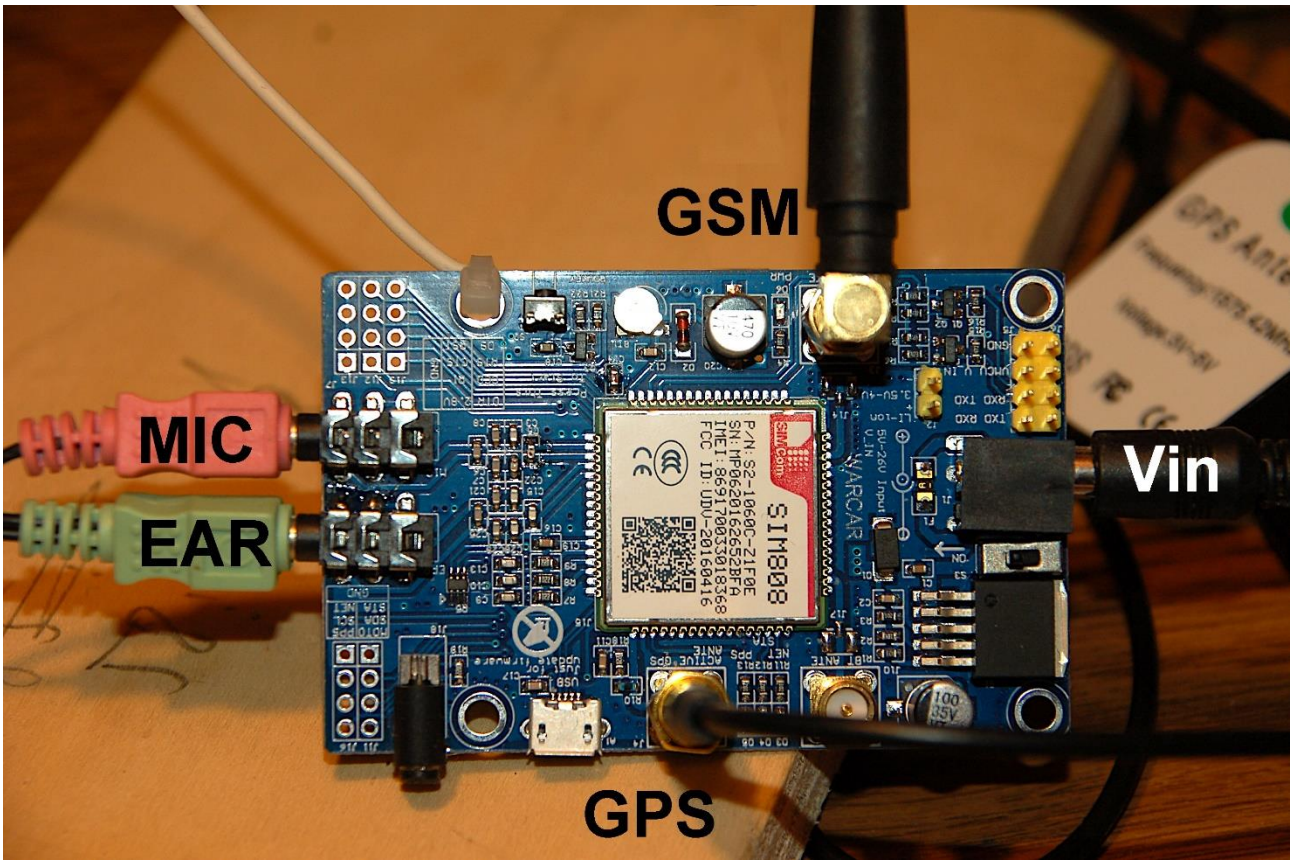

Abbildung 3: Headset_Anschluss

But now we first want to make the keypad with its 16 keys in a 4x4 matrix ready for use. Finally, telephone numbers must be able to be entered.

Something completely new - the 4x4 keypad

With the keypad, I have to go back further. How is it possible to query 16 keys with only 8 lines? Let's take a look at the following illustration.

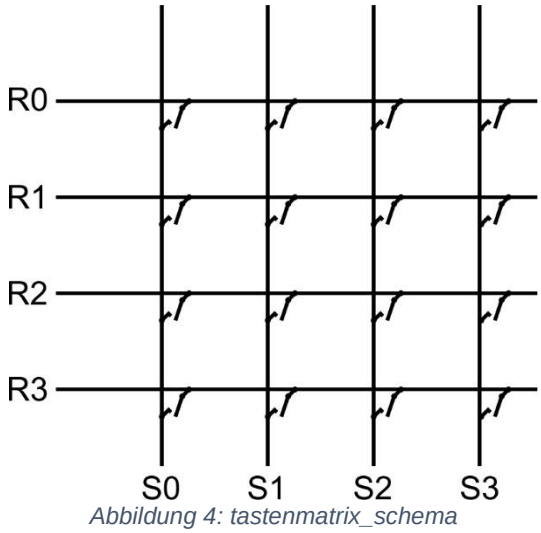

Abbildung 4: tastenmatrix_schema

The buttons are located at the crossing points of the row and column wires. When a key is pressed, a specific row line is connected to a precisely defined column line. The task of our program is now to find out who is connected to whom. There are different algorithms for this. I would like to introduce two of them to you here. The procedure is based on the type of wiring and the hardware of the existing interfaces.

How does the keypad have to be connected to the ESP32? We can provide a GPIO pin for each of the 8 lines. Our controller still has enough free connections for this. However, the associated decoding of the keys through nested loops is the more complex software solution, because individual lines have to be addressed. The ESP32 has no port registers like the AVR controllers (Arduino and Co.) with 5 to 8 bit data width.

The second connection variant therefore uses the port expansion module MCP23017, which is attached to the I2C bus that already supplies the display. Using this circuit, I will introduce the more sophisticated reversal decoder solution. Both solutions use the polling method to query the keyboard in the project. Whenever a key is expected to be pressed, the program checks whether a key has been pressed and, if necessary, waits until this happens.

A third query option would be to query the keys at fixed intervals by means of time-controlled program interruptions (aka timer interrupts) and to store the results in a buffer memory. If necessary, the main program could then fetch the character codes that correspond to the keys from this keyboard buffer. We take the simpler polling method here.

Let's summarize what has been said so far. So we have

- Two methods of connecting the key hardware: single lines or parallel
- two methods of key polling: polling or interrupt mode and
- two methods of decoding the keys by software: looping or reversal

Single lines with row-column query

Because there are no "collective ports" with the ESP32, as with the AVR controllers, where several lines are combined into a port register, the GPIOs have to be operated individually. In terms of hardware, it looks like this. I colored the individual lines like the color-coded jumper cables to clearly determine the assignment. The overall circuit diagram is simply expanded to include the following partial representation. The pin assignment and the electrical properties of the keypad can be found in the data sheet. The electrical values are all uncritical because the contacts are hardly stressed.

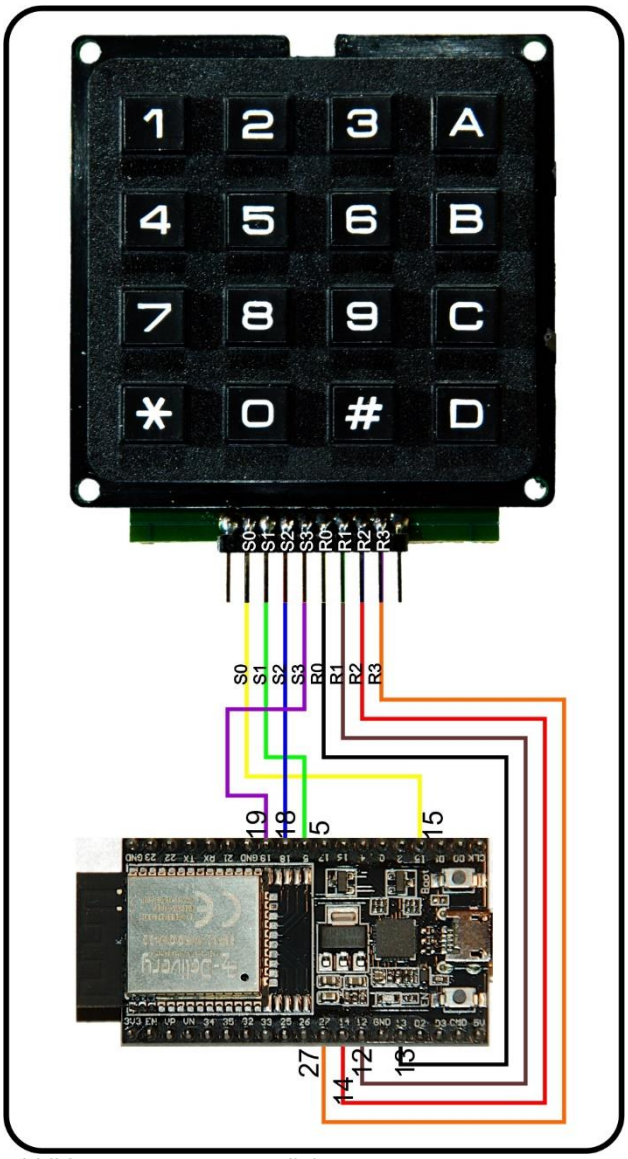

Abbildung 5: tastatur_parallel

The key query is solved in the form of the KEYPAD_P class, which I have integrated into the keypad.py module. At the same time, the key query of the LCD keypad was adapted in such a way that the same API results for it. All classes have a key () method that is addressed when the calling program requests a key to be pressed.

But one after anonther. The constructor of the KEYPAD_P class expects two lists with 4 pin numbers each for the rows and columns of the keypad. They are populated in the main program, roughly as shown in the illustration.

col = [15,5,18,19]
row = [13,12,14,27]

Then use it to call the constructor

from keypad import KEYPAD_P
kp = KEYPAD_P (row, col)

Note:

Membrane keyboards are also available in stores. But they are not pin-compatible with the keyboard used here.

In terms of programming, the following is behind it:

```
class KEYPAD_P: # indexed mit Schleifen
    # Index ist die Tastennummer
    keyNumber=[0x31,0x00,0x01,0x02,
               0x10,0x11,0x12,0x20,
               0x21,0x22,0x03,0x13,
               0x23,0x33,0x30,0x32]

    asciiCode="0123456789\x08\x0b\x0c\x0d*+"

    # row und col sind Listen der Pinnummern)
    # z.B. col=[15,5,18,19] row=[13,12,14,27]
    def __init__(self, row, col):
        self.rowPin=[Pin(i,Pin.OUT) for i in row]
        self.colPin=[Pin(i,Pin.IN,Pin.PULL_UP) for i in col]
```

To ensure that pin numbers can be assigned to pin objects in one go, I used a MicroPython trick, comprehension. In the list context (square brackets) the instruction is packed in a for loop, how to proceed with the numbers in row or col in order to generate a pin object from them. The instance variables rowPin and colPin are lists of pin objects. In rowPin the objects are outputs, in colPin inputs with activated pull-up. The latter saves external resistance.

I chose the solution with lists so that the query can take place in the form of loops and not have to generate separate program code for each of the 16 combinations. The key () method does most of the work. It detects the key and returns a key number, while the debounceKey () method also ensures that the keys are debounced, if necessary, and also provides the key number as an output.

```
    def key(self):
        for i in range(4):
            self.rowPin[i].value(1)
        for j in range(4):  # row setting
            self.rowPin[j].value(0)
            for k in range(4):  # query cols
                w=self.colPin[k].value()
                if w==0:
                    code=j<<4 | k
                    #print(hex(code))
                    self.rowPin[j].value(1)
                    try:
                        return KEYPAD_P.keyNumber.index(code)
                    except:
                        return-1
            self.rowPin[j].value(1)
        return -1

    def debounceKey(self, debounce=1):
```

```
        for i in range(debounce):
            k1=self.key()
            if k1 != -1:
                sleep(0.01)
                k2=self.key()
                if k2 == k1:
                    return k1
        return -1
```

How does key () work?

The line pin objects 0..3 are all set to 1. (Remember, MicroPython regards the upper range limit as exclusive.) In the graphic, key 6 is pressed. It short-circuits row 1 with column 2.

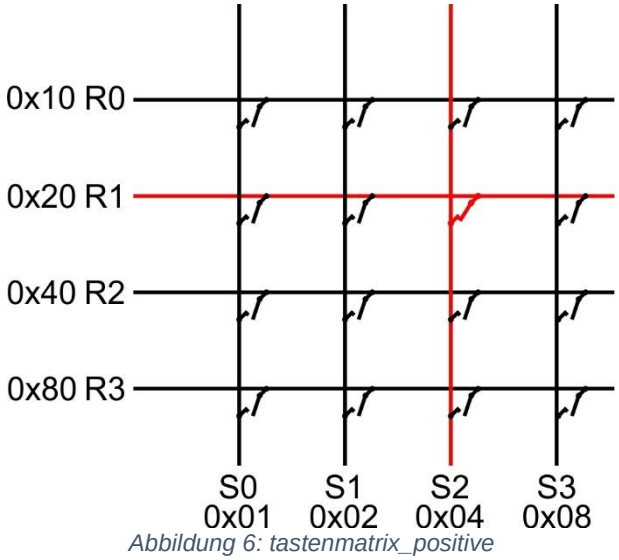

Abbildung 6: tastenmatrix_positive

In the following j-loop, the row lines are set to GND level one after the other from top to bottom. The enclosed k-loop now checks whether one of the column lines is at 0 potential. In this case we build the raw key code from row and column numbers. The line number j supplies the higher-order nibble of the code byte, the column number the lower-order nibble. Then we set the line exit back to 1 and leave the k-loop and the function with return and the key number in the luggage. If no key has been pressed, both loops run through to the end and the return value is -1. In this example, the raw key codes are determined using positive logic. The bits get their significance from the loop indexes, not from the level on the lines.

The key code is determined as the index of the raw key code in the keyNumber list. The raw key codes simply have to be placed at the position in the list whose index corresponds to the desired key number. First of all, here is the code list:

```
keyNumber=[0x31,0x00,0x01,0x02,
           0x10,0x11,0x12,0x20,
           0x21,0x22,0x03,0x13,
           0x23,0x33,0x30,0x32]
```

Lists are indexed starting with 0. The key in the 3rd line in column 1 is 0. So the hex code 0x31 must be in position 0 in the list. All right?

No - not quite? Well, another example. Remember, rows, columns and list entries are counted up from 0 onwards.

The key C should deliver the key number 12. It is in the 2nd line in column 3. Shifted 0x02 4 bits to the left (or multiplied by 16) results in 0x20 as high nibble. This is ored with 0x03. The raw key code is therefore 0x23 and must be in the 12th position in the list. Now compare that with the keyNumber list - that's right!

KEYPAD_P.keyNumber.index (code)

To be able to determine the index of a list entry, there is the built-in function index (), which is called as a method of the list object. keyNumber is a class variable and must therefore be referenced with the class identifier KEYPAD_P as a prefix. The raw key code is passed in code.

From this description we can conclude that the 1 is detected the fastest (0.50 ms) and that the "no key" state takes the longest (1.34 ms). If two or more keys are pressed at the same time, the key further up and / or to the left is decoded and the other key (s) is discarded because the loops no longer query these keys. This also solves the problem of ambiguity.

The method debouceKey () takes an integer with the parameter debounce that specifies how often the internal loop should be run through. Mechanical buttons do not close and open exactly at a specified time, rather the switching contact flutters for a short period of time, which leads to chaotic opening and closing of the contact. This process is called key bouncing. In order to ensure that a key is really pressed, the switching states must be read in at different times and compared with each other. If the same conditions are detected within a short period of time, the key is considered to have been pressed. The check is carried out by the debounceKey () method, and the value in the debounce parameter indicates how often this check should be carried out. The default value for this is 1.

If the comparison is positive, the key code is returned, otherwise the value -1, which indicates that either no key was pressed or that the switching status was not unambiguous until the end.

For testing, a number of modules must have been uploaded to the flash of the ESP32. It is best if you download all the modules mentioned at the beginning and send them to the ESP32. These are the files:

GPS-Modul für SIM808 und GPS6MV2(U-Blocks)
LCD-Standard-Modul
HD44780U-I2C-Erweiterung zum LCD-Modul
Keypad-Modul
Button Modul
i2cbus-Modul für standardisierten Zugriff auf den Bus
mcp.py hardwarededizierte Methoden für den MCP23017
keypad_p_test.pyTesten der Tastatur

Please open the keypad_p_test.py file in an editor window. Before you get started, one more important note. Please search for the following two lines in the editor

```
.#kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
kp=KEYPAD_P(rows,cols) #  HW-objekt mit Parallelanschluss
```

The first line has been commented out because it only applies to the second connection method via I2C bus described below. The second line must be carried out here, i.e. it must not have a comment sign.

If that fits, you can start the program with F5. With this all imports and declarations for the test of the keypad are done. Entering everything again by hand for each test takes too long and is prone to errors.

Is the keypad connected correctly? Let's go! Please enter the following command.

```
>>> kp.key ()
   -1
```

If you haven't pressed a key, that's OK !. Now use the up arrow to call the last command back in the input line. Now press key 1 and exit the input line with Enter.

```
>>> kp.key ()
   1
```

The same with the C key

```
>>> kp.key ()
   12th
```

Great! We will discuss some higher, hardware-independent methods for key interrogation after the following chapter. This is about the more effective software solution and the keypad connection via I2C.

I2C-Anschluss des 4x4-Tastenfelds und die Reversal-Abfrage am Parallelport

The advantage of this method is clearly that it is not necessary to interrogate individual lines one after the other. Instead, one of the 8-bit ports (A) of the MCP23017 is cut in half. The upper nibble controls the row lines, the lower nibble the column lines, but not individually, but in parallel. It would be just as good to use port A for the rows and B for the columns. This is essential for larger key matrices.

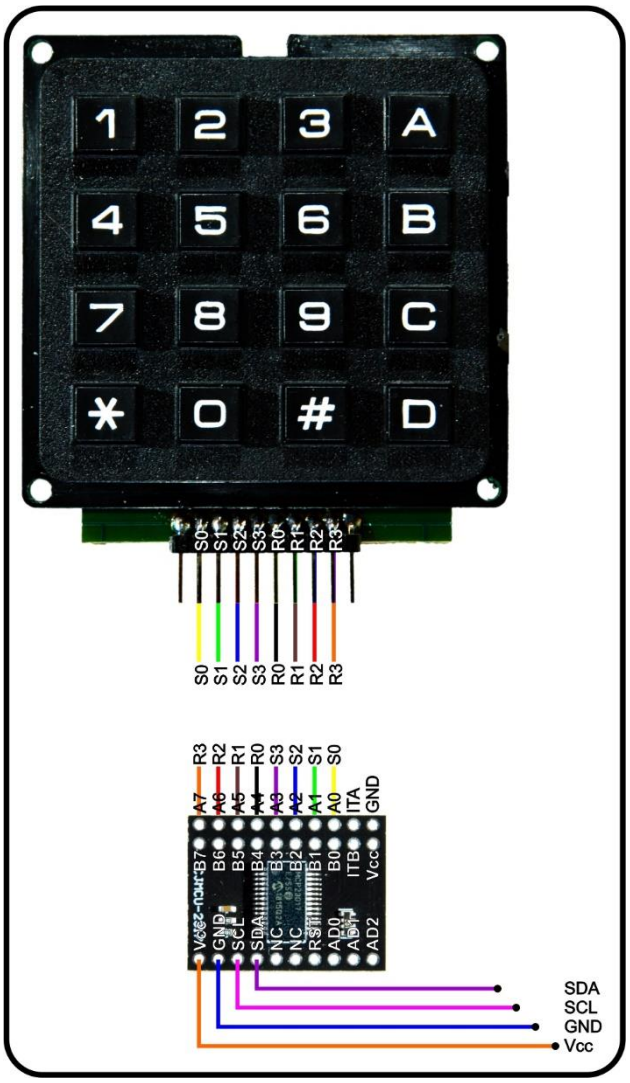

Abbildung 7: tastatur_i2c

The MCP23017 is controlled via the I2C bus and thus parallel to the display and possibly to the BMP280, if it is still in the circuit. This is possible because all interfaces have different hardware addresses. By the way, be careful that you really get an MCP23017. Its little brother, the MCP23S17, is controlled via the SPI bus and cannot be used here.

I created a module for the MCP23017. The MCP23017 class provides a whole range of constants and methods that enable clear and simple programming of the block without having to deal with the methods of the I2C module. The key query is now done in 7 lines. Although there are no loops to run, this algorithm is apparently slower than the one described above, but takes the same length for each key (approx. 4 ms). However, this is not due to the method itself, but to the I2C bus, which slows us down. If the ESP32 had addressable parallel ports from the start, then this method would clearly be the faster. The AVR processors with their parallel ports benefit from this algorithm.

The KEYPAD_I2C class inherits from the MCP23017 class and can therefore access its namespace directly. The keyNumber list of the raw codes looks a little different from the one already known, as does the constructor.

```
class KEYPAD_I2C(MCP): # reversal
    # Keypad  Reihe 0 1 2 3 Spalte 0 1 2 3
```

```
    # MCP GPIOA Pin 4 5 6 7        0 1 2 3
    keyNumber=[0x7D,0xEE,0xED,0xEB,
               0xDE,0xDD,0xDB,0xBE,
               0xBD,0xBB,0xE7,0xD7,
               0xB7,0x77,0x7E,0x7B]

    asciiCode="0123456789\x08\x0b\x0c\x0d*+"

    HWADR=const(0x20)
    KeyMask=const(0b00001111)  # OUT=0 IN=1
    # 0 is OUTPUT - 1 is INPUT
    # Die Zeilen sind zunächst Output

    def __init__(self,i2cbus,hwadr=HWADR):
        self.hwadr=hwadr
        self.i = i2cbus
        super().__init__(i2cbus,hwadr)
        print("Keypad_I2C @", hex(hwadr))
```

The constructor takes an I2CBus object from the main program as a position parameter and the hardware address of the MCP23017 as an optional parameter. He passes both on to the class MCP23017, which was imported under the alias MCP.

But let's take a look at the miracle algorithm.

```
    def key(self):
        self.changeIODIRA(0x00,KeyMask,True)
        self.setGPIOA(KeyMask)
        iod=self.getGPIOA()
        self.invertIODIRA(True)
        self.setGPIOA(iod)
        iod=self.getGPIOA()
        #print(hex(iod),bin(iod))
        try:
            return KEYPAD_I2C.keyNumber.index(iod)
        except ValueError as e:
            return -1
```

Der Code erscheint wesentlich geradliniger und besteht, die Exception-Behandlung nicht mitgerechnet, aus 7 Anweisungen.

changeIODIRA(0x00,KeyMask,True)
KeyMask hat den Wert 0b00001111. Wir undieren das Datenrichtungsregister A mit 0b00000000, löschen also alle Bits und setzen dann durch Oderieren mit KeyMask das Low-Nibble als Eingänge. True sorgt dafür, dass für diese Eingänge die Pullups eingeschaltet werden. Die vier Zeilenleitungen sind jetzt Ausgänge die Spaltenleitungen Eingänge.

setGPIOA(KeyMask)
KeyMask hat immer noch den Wert 0b00001111. Somit werden alle Zeilenleitungen gleichzeitig auf GND-Potential gelegt.

iod=self.getGPIOA()
Die Eingänge von GPIOA werden erfasst. Mit den Pullups ergibt das den Wert

0bxxxx1101

Das High-Nibble interessiert im Moment (noch) nicht.

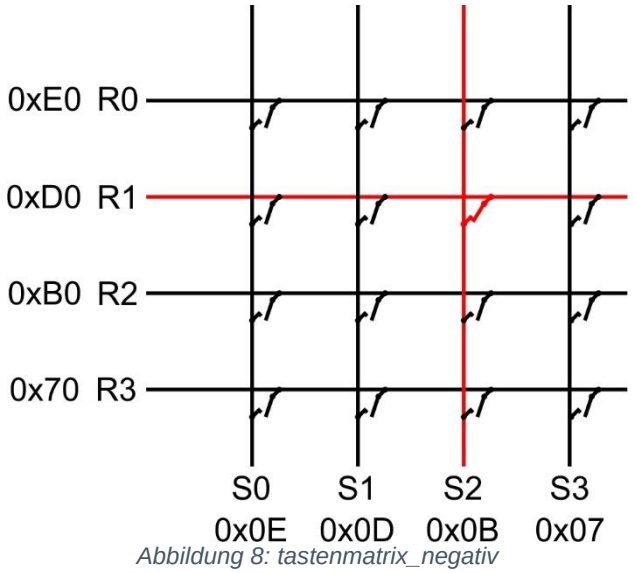

Abbildung 8: tastenmatrix_negativ

invertIODIRA (True)
We invert the data direction register A, inputs become outputs and vice versa. True - the pull-ups of the input lines are activated.

setGPIOA (iod)
S2 now has low potential. We read in GPIOA and get the value

0b10111101 = 0xDB

We encounter a negative logic here, that is, the bits that are not set count. You can also see it like this, for the columns, the low nibble, the bit with the valency 4 is missing from the 15. This leaves 15 - 4 = 11 = 0x0B. Bit1 of the lines is low, so the nibble is missing 2. 15 - 2 = 13. The high nibble is given the value 0xD0. 0xD0 | is obtained bit by bit or divided 0x0B = 0xDB

KEYPAD_I2C.keyNumber.index (iod)
The raw values are again in the corresponding position in the keyNumber list. The index of a value provides the key number as in the first example. This means that both methods have the same call and essentially deliver the same result. The way there is a different one, but it doesn't matter to the outside world. The method debounceKey () even has the same code in both classes. With the exception of the constructor, the two classes provide instances with the same API.

Phew! That was really far out now, but I think it's important to understand the basic mechanisms of a matrix query and to be able to weigh them against each other. Let's take

another look at the KEYPAD class, which offers cross-hardware methods for using keyboards.

The constructor expects an instance of the KEYPAD_P or KEYPAD_I2C classes. Even KEYPAD_LCD instances are accepted despite the small number of characters. One trick should not go unmentioned. The method kp.key (), better its reference, is assigned to the instance variable self.key. You can do that in MicroPython. This can help reduce the length of calls to imported methods. Self.k.key () becomes self.key (). You mean these are peanuts, aren't they? Well, then just imagine, instead of k, keyPad_I2C_Instanz would have to be entered 20 times or more.

```
class KEYPAD:

    # kp ist ein KEYPADXXX-Objekt, wird uebergeben
    def __init__(self,kp,d=None):
        self.k=kp
        self.disp=d
        self.key=kp.key

    def waitForKey(self,timeout=5):
        now=time()
        end=(now+timeout if timeout!=0 else now+10)
        while 1:
            k=self.key()
            if k!=-1:
                return k
            if timeout==0: end=time()+10
            if time()>=end: return -1

    def asciiKey(self):
        n=self.key()
        if 0<=n<=15:
            return self.k.asciiCode[n]
        return -1

    def padInput(self,delay=0.5,xp=0,yp=0):
        s=""
        self.disp.position(xp,yp)
        x=xp
        while 1:
            taste=self.asciiKey()
            if taste != -1:
                if '0'<=taste<='9' or taste=='+' or taste=='*':
                    s+=taste
                    if self.disp:
                        x=self.disp.writeAt(taste,x,yp)
                    else:
                        x+=1
                    sleep(delay)
                elif taste=="\x08":
                    if self.disp and x>xp:
                        self.disp.clearFT(x-1,yp,x)
```

```
                    x=(x-1 if x>xp else xp)
                    s=s[:x-xp]
                    sleep(delay)
                elif taste == "\x0d":
                    return s
```

The three methods in this class serve useful purposes.

The waitForKey (delay) method waits delay seconds for a key to be pressed. If a key is pressed during this time, the method returns the key number, otherwise -1. The method waits forever if the value 0 is passed to delay.

Depending on the definition string asciiCode in the KEYPADXXX classes, the asciiKey () method returns a normal ASCII character instead of the key number. You determine which characters these should be with the help of the character string in asciiString. The position of the character in the string corresponds to the key number that the key () method returns. Non-printable characters like \ n are given as hex codes. For example, \ n corresponds to \ x0D. My definition looks like this:

asciiCode = "0123456789 \ x08 \ x0b \ x0c \ x0d * +"

By the way, if you create several asciiCode definition strings, you can establish multiple key assignments. However, not only the labeling of the keys is a challenge.

Finally, the padInput () method allows digit sequences to be entered, supplemented by "*" and "+", which is entered with the "#" key. Key "A" deletes backwards and key "D" accepts the string for further processing. If a display object was passed to the constructor, the characters also appear in the display. Without a display object, flying blind is the order of the day.

Three optional parameters control the input. delay (default = 0.5s) protects against unintentional multiple entry of a character, because the key method is simply too fast, we need a delay. xp and yp define the position in the display from which the input is shown. The default is the upper left cursor position, corresponding to "home".

At this point, let's test the three methods. If you have not yet done this, please send all modules to the flash of the ESP32 and open the keypad_p_test.py program in an editor window. Depending on the selected connection type of the keyboard, exactly one of the following lines must be uncommented.

# kp = KEYPAD_I2C (ibus, keyHwadr) # Hardware object on the I2C bus
kp = KEYPAD_P (rows, cols) # HW object with parallel connection

Start the program with F5. When the REPL prompt (>>>) appears again, you are ready for the following tests via the command line.

```
>>> k.waitForKey (3)
5
>>> k.waitForKey (3)
-1
>>>
```

In the first case, key 5 was pressed within 3 seconds. The second time the key was not pressed.

```
>>> k.asciiKey ()
-1
>>> k.asciiKey ()
'*'
>>>
```

No key was pressed the first time it was called and the "*" key the second time. Please note that the ASCII code for "*" was not returned but not the key code 14.

```
>>> k.padInput (0.3,3,1)
'1234567'
>>>
```

The keypad is queried with a delay of 0.3 seconds. The entries are shown in the display from position column 3 and line 1. Do not set the dead time too long, otherwise key presses will be skipped. If the dead time value is too short, entries appear several times. To complete the entry, press the D button.

As a result of the previous command, you should see your input both on the display and in the terminal window.

Als die Teleklingel phonte,

treppte ich die Ranne runter und türte gegen die Bums. We don't want to hope that will happen. But it would be nice if the mobile phone would alert us to a call from the SIM808. I am sure it will!

The MicroPython-gps module has received a couple of new methods for this purpose. With their help we are able to use further AT commands made available by the SIM808. I will briefly present this excerpt from the program text, then we will make a call, OK?

Download gps.py:

```
    def simGetSignalStrength(self):
        self.simFlushUART()
        self.simSendCommand("AT+CSQ\r\n")
        self.simWaitForData(500)
        a=self.simReadBuffer(50)[1]
        p=a.find("+CSQ:")
        if p!=-1:
            w=int(a[p+5:a.find(",",p+5)])
            return 2*w-114
        return None

    def simIsRegistered(self):
        self.simFlushUART()
        self.simSendCommand("AT+CREG?\r\n")
        self.simWaitForData(500)
```

```
        a=self.simReadBuffer(50)[1]
        p=a.find("+CREG:")
        if p!=-1:
            w=a.find(",",p+5)+1
            c=int(a[w:w+1]) # 1=home, 5=roaming
            return (c if (c==1 or c==5) else 0)
        return None

    def simAlarmSound(self,nbr=None):
        if nbr==None:
            self.simFlushUART()
            self.simSendCommand("AT+CALS?\r\n")
            self.simWaitForData(500)
            a=self.simReadBuffer(50)[1]
            p=a.find("+CALS:")
            if p!=-1:
                w=a.find(",",p+5)
                c=int(a[p+6:w]) # 1=home, 5=roaming
                return (c if (0<=c<=19) else 0)
            return -1
        elif 0<=nbr<=19:
            return self.simSendCmdChecked\
                   ("AT+CALS={}\r\n".format(nbr),"OK",1)

    def simClock(self,dt=None):
        if dt==None:
            self.simFlushUART()
            sleep(0.3)
            self.simSendCommand('AT+CCLK?\r\n')
            self.simWaitForData(500)
            a=self.simReadBuffer(70)[1]
            p=a.find("+CCLK: \"")
            if p!=-1:
                w=a.find("\"",p+8)
                c=a[p+8:w]
                return c
            return -1
        else:
            return self.simSendCmdChecked\
                   ('AT+CCLK="{}"\r\n'.format(dt),"OK",1)
                   # Obacht geben auf die Anfuehrungszeichen

    def simDialOut(self,number):
        self.simFlushUART()
        self.simSendCommand('ATD {}\r\n'.format(number))
        sleep(1)
        return self.simReadBuffer(500)[1]
```

. simGetSignalStrength ()
We use the AT command "AT + CSQ \ r \ n" to call up the information about the field strength of the received signal and convert the value into dBm that the method returns.

simIsRegistered ()
The method uses the AT command "AT + CREG? \ R \ n" to check whether the SIM808 is logged into a wireless network. If a "+ CREG:" is found in the answer, the following number tells us whether we are logged on to a home network (1) or a foreign network (5 = roaming).

simAlarmSound (nbr = None)
Like the cell phone, the SIM808 has different ring tones with the numbers 0 to 19. If we pass a number from this range to the method, the corresponding melody is set. Calling up without a number returns the currently set number

simClock (dt = None)
If a valid date time string is transferred to dt, the RTC of the SIM808 is set to these values. Without passing parameters, the method returns the current time. A date-time string has the following format:

yy / mm / dd, hh: mm: ss ± tz

tz is the offset of the time zone compared to UTC in quarter hours. For the time zone CET in winter time (geographic standard time), for example, the following string results for November 15, 2020, 8:45:32 am:

20/11 / 15.08: 45: 32 + 04

Due to the time change, tz = 08 must be set in summer.

simDialOut (number)
The best comes last. This method takes a telephone number, calls and reports when the connection is established "OK".

We are now controlled by the cellphone.py program. Its main functions are:

• Key B, enter number and call (hang up: key D)
• Answer call button A (reject button! = A, D; hang up: button D)
• Display and delete incoming SMS messages in stages (automatically)
• Abort the program by pressing the RST button on the LCD keypad

You can of course expand the range of functions at any time by adding some of the presented methods to the job list. Or would you like to incorporate a few more AT commands in your own corresponding methods? After the listing that will enable you to do so, I have a few more recommendations. In advance, I would like to point out two new AT commands that are used in the opening credits.

g.simSendCommand ("ATL9 \ r \ n") # Lautspercher full jug (0..9)
g.simSendCommand ("AT + CLIP = 1 \ r \ n") # Show incoming call number

The first assigns full volume to the headphones and the second activates the display of the phone number for incoming calls

```
# Author:  J. Grzesina
```

```
# Rev.:1.0 - 2021-05-08
#********************** Beginn Bootsequenz *****************
# Dieser Text geht 1:1 an boot.py fuer autonomen Start
#************************ Importgeschaeft ******************
# Hier werden grundlegende Importe erledigt
import os,sys       # System- und Dateianweisungen

import esp          # nervige Systemmeldungen aus
esp.osdebug(None)

import gc           # Platz fuer Variablen schaffen
gc.collect()
#
from machine import I2C,Pin
from i2cbus import I2CBus
from keypad import *
from time import sleep
from lcd import LCD
from hd44780u import HD44780U, PCF8574U
from gps import GPS,SIM808,GSM
from button import BUTTON32,BUTTONS

# ***************** Objekte declarieren ******************
# Pins fuer parallelen Anschluss des 4x4-Pads
# rows=[15,5,18,19]
# cols=[13,12,14,27]

i2c=I2C(-1,scl=Pin(21),sda=Pin(22))
ibus=I2CBus(i2c)

display=LCD(i2c,adr=0x27,cols=16,lines=2) # LCDPad am I2C-Bus

keyHwadr=0x20 # HWADR des Portexpanders fuer das 4x4-Pad
kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
#kp=KEYPAD_P(rows,cols) #  HW-objekt mit Parallelanschluss
k=KEYPAD(kp,d=display) # hardwareunabhaengige Methoden

rstNbr=25
rst=BUTTON32(rstNbr,True,"RST")
ctrl=Pin(rstNbr,Pin.IN,Pin.PULL_UP)
t=BUTTONS() # Methoden fuer Taster bereitstellen

# switch: Pin zum Schalten des SIM808
# key: die LOW-aktive RST-Taste des LCDPads
g=GSM(switch=4,disp=display,key=ctrl)
g.simOn()
g.simGPSDeinit() # GPS-Funktionen ausschalten
sleep(2)
g.simFlushUART() # SIM808-UART-Buffer leeren

g.simSendCommand("ATL9\r\n") # Lautspercher volle Kanne (0..9)
g.simSendCommand("AT+CLIP=1\r\n") # Eingehende Rufnr. zeigen
```

```
g.simFlushUART()
display.clearAll()

def getDateTime(): # Datum/Uhrzeit abfragen/setzen
    f=[("Year:","/"),
       ("Month:","/"),
       ("Day:",","),
       ("Hour:",":"),
       ("Minute:",":"),
       ("Second:","+"),
       ("Offset:",""),
       ]
    display.clearFT(0,1)
    dt=""
    for i in range(7):
        xp=0
        yp=1
        display.clearFT(xp,yp)
        xp=display.writeAt(f[i][0],xp,yp)
        sleep(0.3)
        d=k.padInput(0.5,xp,yp)
        sleep(0.3)
        dt=dt+d+f[i][1]
    display.clearFT(0,1)
    return dt

def parseForSMS(buffer): # Auf eingetroffene SMS testen
    p=buffer.find('+CMTI: "SM"')
    if p!=-1:
        index=int(buffer[p+12:])
        return index
    return -1

#sys.exit()
print("SIM808 READY")
display.writeAt("  SIM808 READY  ",0,0)
ts=g.simClock()
if ts!=-1: display.writeAt(ts[9:-3],3,1)

while 1:
    # Tastenabfrage am Keypad (4x4) und
    # Statusabfrage vom SIM808-UART-Buffer
    buf=g.simReadBuffer(500)[1]
    if buf.find("RING")>=0: # externer Anruf kommt rein
        # Anruf registriert
        print("Eingehender Ruf")
        display.writeAt("INCOMMING CALL",0,0)
        display.writeAt("Accept: >>> A",0,1)
        buf = buf + g.simReadBuffer(500)[1]
        p1=buf.find("+CLIP:") # Eingehende Rufnummer testen
        number=""
        if p1!=-1:
```

```
            p2=buf.find('"',p1+8)
            number=buf[p1+8:p2]
        display.writeAt(number+"    ",0,0)
        g.simFlushUART()
        t=k.waitForKey(15) # 15 Sekunden zum Annehmen
        print("TASTE: ",t)
        if t!=-1:
            t=kp.asciiCode[t]
            if t=='\x08': # Taste A nimmt an
                g.simSendCommand("ATA\r\n")
                while 1:
                    t=k.key()
                    if t==13:
                        g.simSendCommand("ATH\r\n")
                        break # Taste D hat beendet
                    buf=g.simReadBuffer(500)[1]
                    if buf.find("NO CARRIER")!=-1:
                        break # Gegenstation hat aufgelegt
            else: # mit Taste != A abgelehnt
                g.simSendCommand("ATH\r\n")
        else: # mit timeout abgelehnt
            g.simSendCommand("ATH\r\n")
        sleep(1)
        buf=g.simReadBuffer(500)[1]
        if buf.find("+CMTI:")==-1:
            print("CMTI: nicht gefunden")
            print(buf)
        else: # sms abfangen
            print("CMTI: gefunden")
            nbr=parseForSMS(buf)
            if nbr!=-1:
                sms=g.gsmReadSMS(nbr)
                for field in sms: print(field)
                g.gsmDeleteSMS(nbr)
                g.simFlushUART()
        display.clearAll()
        display.writeAt("  SIM808 READY  ",0,0)
        ts=g.simClock()
        if ts!=-1: display.writeAt(ts[9:-3],3,1)

    if buf.find("+CMTI:")!=-1: # liegt eine SMS-Nachricht vor?
        print("SMS eingetroffen")
        nbr=parseForSMS(buf)
        if nbr!=-1:
            sms=g.gsmReadSMS(nbr)
            for field in sms: print(field)
            display.clearAll()
            display.writeAt(sms[0],0,0) # Status
            display.writeAt(sms[1],0,1) # Rufnummer
            k.waitForKey(10)
            display.clearAll()
            sleep(1)
```

```
            display.writeAt(sms[2],0,0) # Datum
            display.writeAt(sms[3],0,1) # Uhrzeit
            # Synchronisation der lokalen Uhrzeit
            k.waitForKey(10)
            display.clearAll()
            sleep(1)
            n=len(sms[4])
            p=0
            while p<n: # Nachricht je zweizeilig anzeigen
                display.clearAll()
                sleep(1)
                display.writeAt(sms[4][p:p+15],0,0)
                display.writeAt(sms[4][p+16:p+31],0,1)
                p+=32
                k.waitForKey(10)
            g.gsmDeleteSMS(nbr)
            display.clearAll()
            display.writeAt("  SIM808 READY  ",0,0)
            ts=g.simClock()
            if ts!=-1: display.writeAt(ts[9:-3],3,1)


    if k.asciiKey()=="\x0b": # Eigenen Anruf einleiten
        display.clearAll()
        display.writeAt("Enter number:",0,0)
        sleep(1)
        nbr=k.padInput(0.5,0,1)+";" # Nummer eingeben
        print("dialing",nbr) # Abbruch durch Taste D
        if len(nbr) > 4:
            buf=g.simDialOut(nbr)
            print("Buffer: ",buf)
            if buf.find("OK") != -1:
                #Verbindung steht
                # warten bis NO CARRIER oder Abbruchtatste D
                while 1:
                    t=k.key()
                    if t==13:
                        g.simSendCommand("ATH\r\n")
                        break # Abbruch durch Taste D
                    buf=g.simReadBuffer(500)[1]
                    if buf.find("NO CARRIER")!=-1:
                        break # Gegenstation hat aufgelegt
        display.clearAll()
        display.writeAt("  SIM808 READY  ",0,0)
        ts=g.simClock()
        if ts!=-1: display.writeAt(ts[9:-3],3,1)

    if ctrl.value()==0: # RST-Taste am LCDPad = Notbremse
        display.clearAll()
        display.writeAt("prog cancelled!!",0,0)
        sys.exit()
    ts=g.simClock()
```

```
    if ts!=-1: display.writeAt(ts[9:-3],3,1)

    # Weitere Aktionen:
    # Uhrzeit setzen/abfragen
    # Alarmmelodie 1 aus 20 einstellen
    # Speaker Lautstaerke einstellen
    # SMS abfragen
```

40% of the program scope is taken up by the preparations, imports and declarations of variables, objects and functions. If we were to individually define everything that is packed in modules in the main program, the list would be much longer. The use of modules creates an overview and more readable code. If you really want to know more, it is worth taking a look behind the scenes at the respective class definition.

How does the main loop work?
As soon as the display reports "SIM808 READY", the time from the RTC is also displayed and our program is continuously running.

The UART buffer is read out. If the string "RING" is found in it, then a call is just coming in. Let the doorbell ring 2 to 3 times. Now the phone number of the caller should also have landed in the buffer. This is the case when the string "+ CLIP:" is found in buf. We parse the buffer string and filter out the phone number and show it on the display. We have 15 seconds to accept the call with key A or reject it with any other key. If no key is pressed, the call is automatically rejected after 15 seconds.

If the call has been accepted, you can speak for as long as you like. You can hang up by pressing the D button. If the other station has hung up, the program recognizes this from the string "NO CARRIER" in the buffer string.

If the call was rejected, both sides receive an announcement from the provider and an SMS message. This case is detected by the next "if" in the main loop at the latest. The UART buffer then contains the character string "+ CMTI:". The test immediately after the rejected call is usually negative because the message is delayed a few seconds.

Every incoming SMS message is indicated by the text "+ CMTI:" in the buffer. By tapping the contents of the buffer on it, we get every incoming message presented on the display - after reading and disassembling.

The parseForSMS () function provides us with the index number of the SMS that we need to read. The g.gsmReadSMS (nbr) method, to which we transfer the index in nbr, is used to read in and decompose. As a response, we get a tuple with the string elements (status, phone, date, time, message). Three steps bring the content to the display, status and phone number, date and time and, divided into two lines, the message itself. You can advance ahead with any key. If no button is pressed, the program automatically switches to the next level after 10 seconds.

Outgoing calls are initiated with the B key. You will be asked to enter the phone number. All digits plus "*" and "+" can be specified. The "+" sign is on the # key. Key A deletes backwards, key D takes over the entry. A semicolon ";" must be appended so that the syntax of the AT command is correct. If the entry contains at least 4 characters, the

simDialOut () function takes over the dialing process and the connection establishment. Of course, you can adjust all of these values to suit your needs.

The connection is established when an "OK" is found in the response in the buffer. With the D key you can hang up yourself. A "NO CARRIER" can be found in the UART buffer when the other station has hung up.

What can you do to get out of the loop in an orderly fashion if you want to change something in the program? At the beginning we talked about the emergency brake. Get off with the RST button on the LCD keypad. The ESP32 reports the exit in the terminal and in the display.

Would you like the ESP32 to start the program autonomously? No problem, copy the entire program text from the cellphone.py file into a newly created blank file. Save this as boot.py and upload boot.py to the ESP32. Your request will be met the next time the controller is cold started.

Well, that brings us almost to the end of this episode. Almost, because I had promised to introduce some interesting AT commands. You can turn each of these into a further menu item in a further if branch in the main loop. You still have 14 buttons on the keypad and 5 on the LCD keypad to choose from.

Befehlsstring	Beschreibung
AT+CBC\r\n	AT+CBC\r\nAT+CBC\r\r\n+CBC: 0,45,3763\r\n\r\nOK\r\n Ladezustand des Akkus testen 0 wird nicht geladen, 45% Kapazität 3,763V Akkuspannung
AT+GSN\r\n	AT+GSN\r\nAT+GSN\r\r\n869170033018368\r\n\r\nOK\r\n IMEI abfragen
AT+IPR?\r\n	AT+IPR?\r\nAT+IPR?\r\r\n+IPR: 0\r\n\r\nOK\r\n Baudrate der seriellen Verbindung abfragen 0: Autobauding Die Baudrate wird durch senden von AT\r\n bei 8,n,1 vom SIM808 automatisch ermittelt und eingestellt. Beim Initialisieren des SIM808 sollte also stets als erstes ein nackter AT-Befehl abgesetzt werden.
AT+CALM?\r\n AT+CALM=1\r\n	AT+CALM?\r\nAT+CALM?\r\r\n+CALM: 0\r\n\r\nOK\r\n 0: Der Rufton ist an 1: Der Rufton ist aus Rufton ausschalten (Silent Mode)
AT+CRSL?\r\n AT+CRSL=33	AT+CRSL?\r\nAT+CRSL?\r\r\n+CRSL: 100\r\n\r\nOK\r\n Ruftonlautstärke anfragen Ruftonlautstärke auf 33 setzen
AT+CLVL?	Lautsprecherlautstärke abfragen
ATV0 ATV1	Klartext-Antwort auf Befehle ausschalten Statt OK oder ERROR kommen Zahlen, hier 0 und 4 Klartext-Antwort ein
AT+CCLK?\r\n	AT+CCLK?\r\nAT+CCLK?\r\r\n+CCLK: "21/05/15,15:10:59+08"\r\n\r\nOK\r\n

AT+CCLK="21/05/12, 20:10:04+08"\r\n	Abfrage der Zeit von der RTC des SIM808 Stellen der RTC **Hinweis**: Der Date-Time-String **MUSS** in **doppelten** Anführungszeichen stehen!

I have already worked on the CCLK command. In the listing you will find the getDateTime () function with which you can enter a date-time string using the numeric keypad. The list f defined at the beginning of the list shows how complex strings can be built up. The tuples contain the text for an input prompt and, as a second element, the separator for the next substring. As an extension, for example, a lower and an upper limit for plausibility checks for numerical entries can be integrated.

In the gps module there is the simClock () method in the SIM808 class, which is also used in the program. Both together could become a menu item "Setting the time". So fresh to work!

I hope you enjoy installing the commands, making calls and texting.

PDF in deutsch
PDF in english