Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

Nach der SMS- und GPS-Funktionalität des SIM808 werde ich Ihnen heute die dritte Säule des kleinen Tausendsassas vorstellen. Der kann nämlich auch ganz normal telefonieren. Voraussetzungen sind eine SIM-Karte und die Anwendung der entsprechenden AT-Befehle. Essentiell sind es deren zwei, aber einige weitere machen die Sache dann doch erst so richtig interessant. Die Befehle wird, wie in den vorangegangenen Folgen, wieder der ESP32 an das SIM808 senden. Natürlich brauchen Sie dieses Mal einen Kopfhörer oder einen Minilautsprecher (32 Ohm) und ein Mikrophon. Am besten eignen sich Headsets mit 3,5mm-Klinken-Steckern.

Zum Wählen und zur Ablaufsteuerung brauchen wir eine erweiterte Funktionalität. Dafür verwende ich ein 4x4-Tastenfeld mit den Ziffern 0 bis 9, den Buchstaben A bis D sowie * und #. Dieser Beitrag befasst sich zunächst mit der Abfrage dieser Tastenmatrix, später werden wir telefonieren.

Für das Einlesen der Tasten durchleuchten wir zwei gängige Methoden. Auch das LCD-Keypad kommt wieder zum Einsatz, hauptsächlich das Display. Aber bevor's losgeht, erst einmal herzlich willkommen zum 5. Teil der Blogfolge mit dem Titel

GSM und Telefonie mit MicroPython auf dem ESP32 und dem SIM808

Klar, das Display leistet gute Dienste beim Start des Cellphones (aka Handy) und es ermöglicht eine sichere Eingabe der anzurufenden Nummer auf Sicht, deshalb verwenden wir es wieder. Der Bequemlichkeit wegen habe ich mich für die Eingabe von Telefonnummern für ein Tastenfeld mit 16 Tasten entschieden. Alleine mit den Tasten am LCD-Keypad hätte man Klimmzüge machen müssen, um alle Ziffern eingeben zu können. Allerdings – eine Taste am LCD-Keypad ist nach wie vor wichtig, die verwende ich nämlich wieder als Notbremse in der Steuerungsschleife. Wir hatten das schon in den vorangegangenen Folgen. Notbremse benutzen heißt, punktgenau das laufende Programm beenden, ohne sofortigen Neustart. Alle, bis dahin erstellten Objekte, Variableninhalte und Funktionsdefinitionen bleiben für den manuellen Zugriff über REPL erhalten.

Hardware –moderater Zuwachs

An Hardware kommt, im Vergleich zu Teil 4, ein bisschen dazu, je nach Ihren Vorlieben. Grundsätzlich sind die 4x4-Tastatur und ein Headset in dieser Folge unsere treuen Begleiter. Die Tastatur muss vom ESP32 abgefragt werden. Für den Anschluss gibt es 2 Varianten, für die Abfrage drei. Zwei davon erkläre ich in einem extra Kapitel. Eine dritte Möglichkeit reiße ich dort nur kurz an.

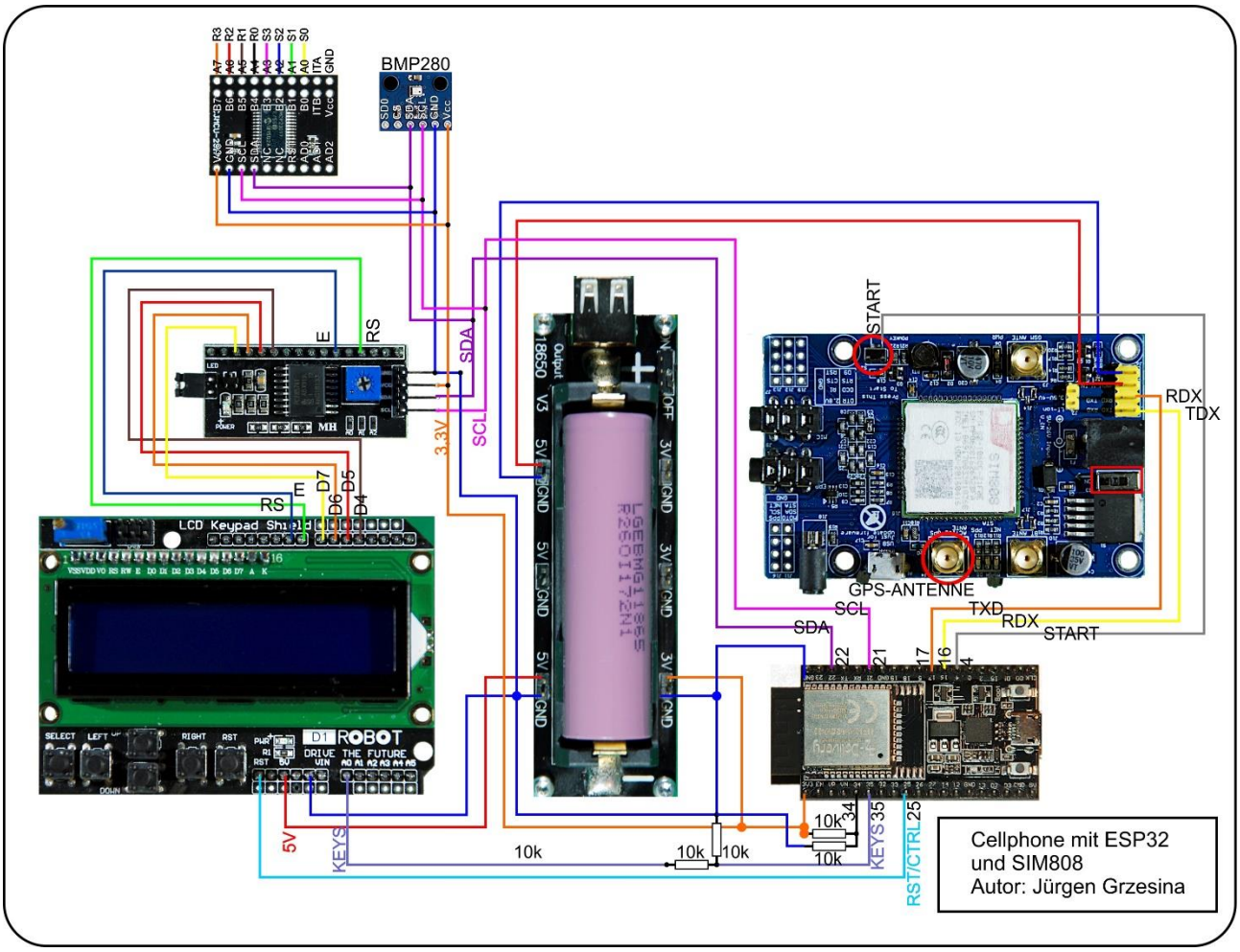
Das Headset wird direkt am SIM808 angesteckt. Sie werden das in einer Abbildung genau orten können.

Hier ist die Liste der Zutaten für die neue Bastelsession. Bis auf die letzten drei Posten haben Sie vermutlich bereits alles, wenn Sie die in den vorangegangenen Folgen schon fleißig mitgebaut haben.

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

Die Schaltung für das Cellphone (engl. für Handy) wird zum großen Teil von Folge 2 übernommen. Neu sind Tastatur und Headset. Weil sich der Anschluss der Tastatur auf zweierlei Weise gestalten lässt, gibt es auch zwei Teilschaltpläne. Natürlich können Sie auch die Bauteile aus vorangegangenen Folgen wieder mit verwenden. Sie entscheiden, welche Teile Sie weglassen, ersetzen oder neu hinzunehmen. Das Projekt ist in jeder Richtung skalierbar. Die programmtechnischen Grundlagen für die Umsetzung des

Cellphones finden Sie in diesem Beitrag. Neben dem BMP280 habe ich einen Schnittstellenbaustein MCP23017 platziert, der zwei 8-Bit-Ports bereitstellt und über den I2C-Bus angesteuert wird. Wir treffen ihn bei der zweiten Variante des Tastaturanschlusses wieder.



Ein besser lesbares Exemplar der Darstellung in DIN A4 bekommen Sie mit dem 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 und Infos zu MicroPython

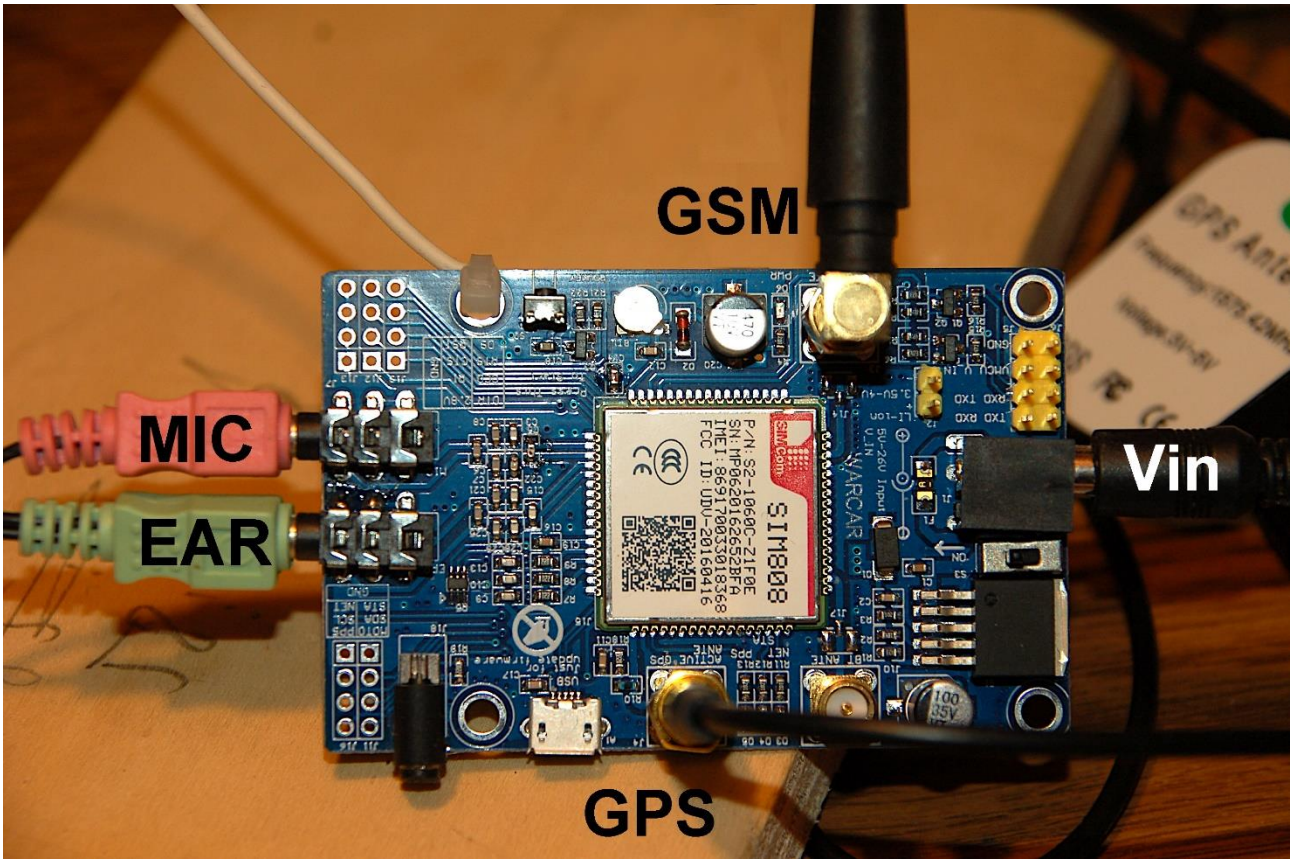
In diesem Projekt wird die Interpretersprache MicroPython benutzt. Der Hauptunterschied zur Arduino-IDE ist, dass Sie die MicroPython-Firmware auf den ESP32 flashen müssen, bevor der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe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

Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Bei der Entwicklung der Software für diesen Blog habe ich von dem direkten Dialog zum ESP32 wieder reichlich Gebrauch gemacht. Das Spektrum reicht von einfachen Tests der Syntax und der Hardware bis zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen. Zu diesem Zweck erstelle ich auch gerne, wie schon bei den vorangegangenen Folgen, kleine Testprogramme. Sie bilden eine Art Macro, weil sie wiederkehrende Befehle zusammenfassen. Zum Test des Ziffernblocks ist es hier das Programm **keypad_p_test.py**, doch dazu später.

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

Das Headset

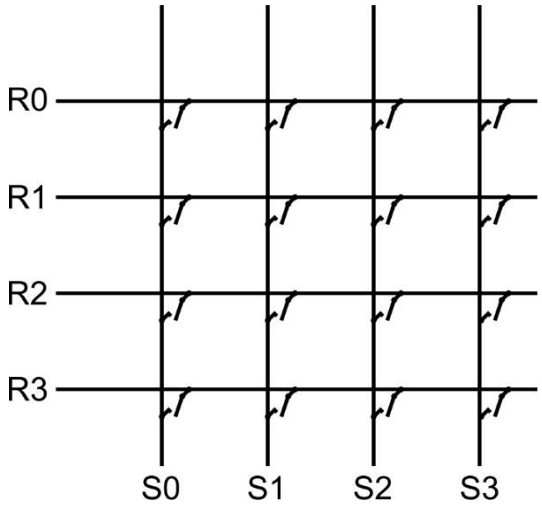
Beginnen wir mit den neuen Bauteilen. Der Anschluss von Kopfhörer und Mikrofon ist unkritisch. Ein normales Headset, wie es für PC oder Laptop üblich ist, entspricht genau unseren Bedingungen. Wie das Headset am SIM808 angeschlossen wird zeige ich schon mal in der Abbildung. Mikrofon- und Kopfhörerstecker sind von der üblichen dreipoligen Ausführung.



Jetzt wollen wir aber erst einmal das Tastenfeld mit seinen 16 Tasten in einer 4x4-Matrix einsatzbereit machen. Schließlich müssen Telefonnummern eingegeben werden können.

Mal was ganz Neues - das 4x4-Tastenfeld

Beim Tastenfeld muss ich weiter ausholen. Wie ist es möglich, 16 Tasten mit nur 8 Leitungen abzufragen? Schauen wir uns einmal die folgende Abbildung an.



Die Tasten sitzen an den Kreuzungspunkten der Zeilen- und Spaltendrähte. Wird eine Taste gedrückt, dann wird eine bestimmte Zeilenleitung mit einer genau definierten Spaltenleitung verbunden. Aufgabe unseres Programms ist es jetzt, herauszufinden, wer mit wem verbunden ist. Dafür gibt es verschiedene Algorithmen. Zwei davon stelle ich Ihnen hier vor. Die Vorgehensweise orientiert sich an der Art der Verdrahtung und der Hardware der vorhandenen Schnittstellen.

Wie muss der Tastenblock an den ESP32 überhaupt angeschlossen werden? Wir können für jede der 8 Leitungen einen GPIO-Pin vorsehen. Unser Controller hat dafür noch genug freie Anschlüsse. Allerdings ist die damit verbundene Decodierung der Tasten durch verschachtelte Schleifen die aufwendigere Softwarelösung, weil eben einzelne Leitungen angesprochen werden müssen. Der ESP32 hat keine Portregister wie die AVR-Controller (Arduino und Co.) mit 5- bis 8-Bit Datenbreite.

Die zweite Anschlussvariante benutzt daher den Porterweiterungsbaustein MCP23017, der an dem I2C-Bus hängt, welcher bereits das Display versorgt. Anhand dieser Schaltung stelle ich die raffiniertere Reversal-Decoderlösung vor. Beide Lösungen bedienen sich im Projekt der Pollingmethode zur Tastaturabfrage. Immer dann, wenn eine Tastenbetätigung erwartet wird, schaut das Programm nach, ob eine Taste gedrückt wird und wartet gegebenenfalls so lange, bis das geschieht.

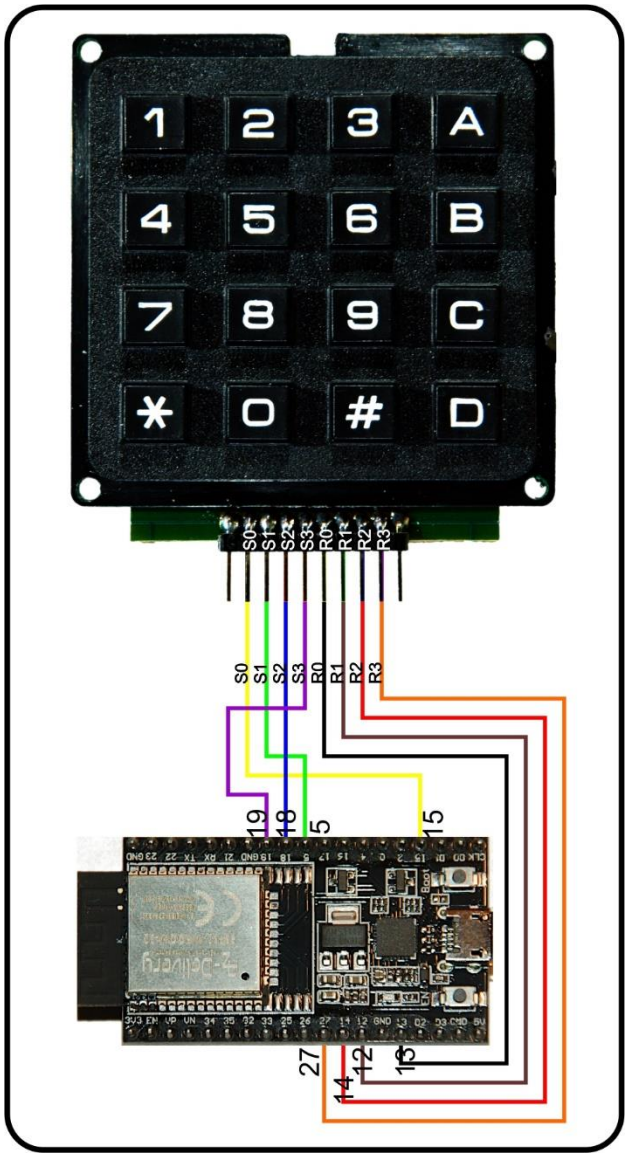
Eine dritte Abfragemöglichkeit bestünde darin, durch zeitgesteuerte Programmunterbrechungen (aka Timer-Interrupts) die Tasten in festen Intervallen abzufragen und die Ergebnisse in einem Pufferspeicher abzulegen. Aus diesem Tastaturpuffer könnte sich das Hauptprogramm dann bei Bedarf die Zeichencodes, die den Tasten entsprechen, abholen. Wir nehmen hier die einfachere Pollingmethode.

Fassen wir das Bisherige zusammen. Wir haben also

- zwei Methoden, die Tasten-Hardware anzuschließen: Einzelleitungen oder parallel
- zwei Methoden, der Tastenabfrage: Polling oder Interruptbetrieb und
- zwei Methoden, die Tasten durch Software zu decodieren: Schleifen oder reversal

Einzelleitungen mit Zeilen- Spaltenabfrage

Weil es beim ESP32 keine "Sammelports", wie bei den AVR-Controllern gibt, wo man mehrere Leitungen zu einem Portregister zusammenfasst, müssen die GPIOs eben einzeln bedient werden. Hardwaremäßig sieht das wie folgt aus. Ich habe die einzelnen Leitungen wie die farbcodierten Jumperkabel eingefärbt, um die Zuordnung eindeutig festzulegen. Das Gesamtschaltschema wird einfach um die folgende Teildarstellung erweitert. Die Pinbelegung und die elektrischen Eigenschaften des Tastenblocks kann man im Datenblatt nachlesen. Die elektrischen Werte sind alle unkritisch, weil die Kontakte ja kaum belastet werden.



Die Tastenabfrage ist in Form der Klasse KEYPAD_P gelöst, die ich in das Modul keypad.py integriert habe. Im gleichen Zug wurde die Tastenabfrage des LCD-Keypads so angepasst, dass sich auch dafür dieselbe API ergibt. Bei allen Klassen gibt es eine Methode key(), die angesprochen wird, wenn das aufrufende Programm eine Tastenbetätigung wünscht.

Aber der Reihe nach. Der Konstruktor der Klasse KEYPAD_P erwartet zwei Listen mit je 4 Pinnummern für die Zeilen und die Spalten des Tastenblocks. Sie werden im Hauptprogramm bestückt, etwa so wie es in der Abbildung vorgegeben ist.

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

Dann rufen Sie damit den Konstruktor auf

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

Hinweis:
Im Handel sind auch Folientastaturen erhältlich. Die sind aber nicht pinkompatibel mit der hier verwendeten Tastatur.

Programmtechnisch steckt Folgendes dahinter:

```
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]
```

Damit die Zuweisung der Pinnummern auf Pin-Objekte in einem Rutsch geht, habe ich mich eines Tricks von MicroPython bedient, der **Comprehension**. In den Listenkontext (eckige Klammern) ist in eine for-Schleife die Anweisung verpackt, wie mit den Nummern in row oder col zu verfahren ist, um daraus ein Pinobjekt zu generieren. Die Instanzvariablen rowPin und colPin sind also Listen von Pinobjekten. In rowPin sind die Objekte Ausgänge, in colPin Eingänge mit aktiviertem Pullup. Letzteres spart externe Widerstände.

Die Lösung mit Listen habe ich deshalb gewählt, damit die Abfrage in Form von Schleifen geschehen kann und nicht für jede der 16 Kombinationen eigener Programmcode erzeugt werden muss. Die Methode **key**() erledigt die Hauptarbeit. Sie stellt die Taste fest und liefert eine Tastennummer zurück, während die Methode **debounceKey**() zusätzlich, falls erforderlich, für das Entprellen der Tasten sorgt und als Ausgabe ebenfalls die Tastennummer als Ausgabe zur Verfügung stellt.

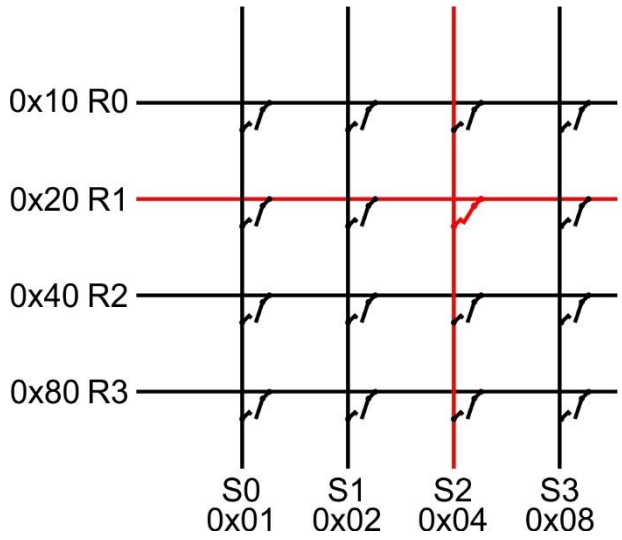
```
    def key(self):
        for i in range(4):
            self.rowPin[i].value(1)
        for j in range(4):  # row setting
```

```
            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
```

Wie arbeitet **key**()?

Die Zeilenpin-Objekte 0..3 werden alle auf 1 gesetzt. (Sie erinnern sich, MicroPython sieht die obere Bereichsgrenze als exclusiv an.) In der Grafik wird die Taste 6 gedrückt. Sie schließt Zeile 1 mit Spalte 2 kurz.



In der folgenden j-Schleife werden die Zeilenleitungen von oben nach unten der Reihe nach einzeln auf GND-Pegel gesetzt. Die eingeschlossene k-Schleife prüft nun, ob eine der Spaltenleitungen auf 0-Potenzial liegt. In diesem Fall bilden wir den rohen Tastencode aus Zeilen- und Spaltennummer. Die Zeilennummer j liefert das höherwertige Nibble des **code**-Bytes, die Spaltennummer das niederwertige Nibble. Dann setzen wir den

Zeilenausgang wieder auf 1 und verlassen mit **return** und der Tastennummer im Gepäck die k-Schleife und die Funktion. Wurde keine Taste gedrückt, laufen beide Schleifen bis zum Ende durch, und die Rückgabe ist der Wert -1. Die rohen Tastencodes werden in diesem Beispiel in positiver Logik ermittelt. Die Bits erhalten ihre Wertigkeit aus den Laufindizes der Schleifen, nicht vom Pegel auf den Leitungen.

Der Tastencode wird als Index des rohen Tastencodes in der Liste keyNumber ermittelt. Die rohen Tastencodes müssen also einfach nur an die Position in der Liste gesetzt werden, deren Index der gewünschten Tastennummer entspricht. Hier erst einmal die Codeliste:

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

Listen werden beginnend mit der 0 indiziert. Die Taste, die in der 3. Zeile in der Spalte 1 liegt, ist die 0. Also muss der Hexcode 0x31 an die Position 0 in der Liste. Alles klar? Nein - nicht so ganz? Gut, noch ein Beispiel. Denken Sie daran, Zeilen, Spalten und Listeneinträge werden ab der 0 hoch gezählt.

Die Taste C soll die Tastennummer 12 liefern. Sie liegt in der 2. Zeile in Spalte 3. Die 0x02 um 4 Bit nach links verschoben (oder mit 16 multipliziert) ergibt als High-Nibble 0x20. Das wird mit 0x03 oderiert. Der rohe Tastencode ist demnach 0x23 und muss an der 12. Position in der Liste stehen. Jetzt vergleichen Sie das mit der keyNumber-Liste – stimmt!

```
KEYPAD_P.keyNumber.index(code)
```

Um den Index eines Listeneintrags feststellen zu können, gibt es die eingebaute Funktion index(), die als Methode des Listenobjekts aufgerufen wird. keyNumber ist eine Klassenvariable und muss daher mit dem Klassenbezeichner KEYPAD_P als Prefix referenziert werden. In code wird der rohe Tastencode übergeben.

Aus dieser Beschreibung können wir schließen, dass die 1 am schnellsten entdeckt wird (0,50ms) und dass der Zustand "keine Taste" am längsten braucht (1,34ms). Werden zwei Tasten oder mehr gleichzeitig gedrückt, dann wird die weiter oben und/oder links liegende Taste decodiert und die andere(n) verworfen, da die Schleifen diese Tasten nicht mehr abfragen. Damit ist auch das Problem der Mehrdeutigkeit gelöst.

Die Methode **debouceKey**() nimmt mit dem Parameter **debounce** eine Ganzzahl, die angibt, wie oft die interne Schleife durchlaufen werden soll. Mechanische Tasten schließen und öffnen nicht exakt zu einem festgelegten Zeitpunkt, sondern der Schaltkontakt flattert jeweils während einer kurze Zeitspanne, was zu chaotischem Öffnen und Schließen des Kontakts führt. Diesen Vorgang nennt man Tastenprellen (aka engl. Bouncing). Um nun sicherzustellen, dass eine Taste wirklich gedrückt ist, muss man die Schaltzustände zu verschiedenen Zeitpunkten einlesen und mit einander vergleichen. Werden in kurzem zeitlichem Abstand die gleichen Zustände festgestellt, dann gilt die Taste als gedrückt. Die Prüfung führt die Methode debounceKey() durch, und der Wert im Parameter debounce gibt an, wie oft diese Prüfung durchgeführt werden soll. Der Defaultwert dafür ist 1.

Wenn der Vergleich positiv verläuft, wird der Tastencode zurückgegeben, andernfalls der Wert -1, der andeutet, dass entweder keine Taste gedrückt wurde oder, dass der Schaltzustand bis zuletzt nicht eindeutig war.

Zum Testen müssen eine Reihe von Modulen ins Flash des ESP32 hochgeladen worden sein. Am besten ist es, wenn Sie alle eingangs genannten Module herunterladen und zum ESP32 schicken. Das sind die Dateien:

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

Die Datei **keypad_p_test.py** öffnen sie bitte in einem Editorfenster. Bevor Sie loslegen, noch ein wichtiger Hinweis. Suchen Sie bitte im Editor nach den folgenden beiden Zeilen.

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

Die erste Zeile ist auskommentiert, weil sie nur für die weiter unten beschriebene zweite Anschlussmethode via I2C-Bus gilt. Hier muss die zweite Zeile ausgeführt werden, darf also kein Kommentarzeichen haben.

Wenn das passt, können Sie das Programm mit F5 starten. Damit sind alle Importe und Declarationen für den Test des Tastenfelds erledigt. Alles für jeden Test erneut von Hand einzugeben, dauert zu lange und ist fehleranfällig.

Das Tastenfeld ist korrekt angeschlossen? Dann los! Geben Sie bitte den folgenden Befehl ein.

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

Wenn Sie keine Taste gedrückt hatten, ist das OK!. Rufen Sie jetzt mit Pfeil nach oben den letzten Befehl noch einmal in die Eingabezeile zurück. Jetzt die Taste 1 drücken und die Eingabezeile mit Enter beenden.

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

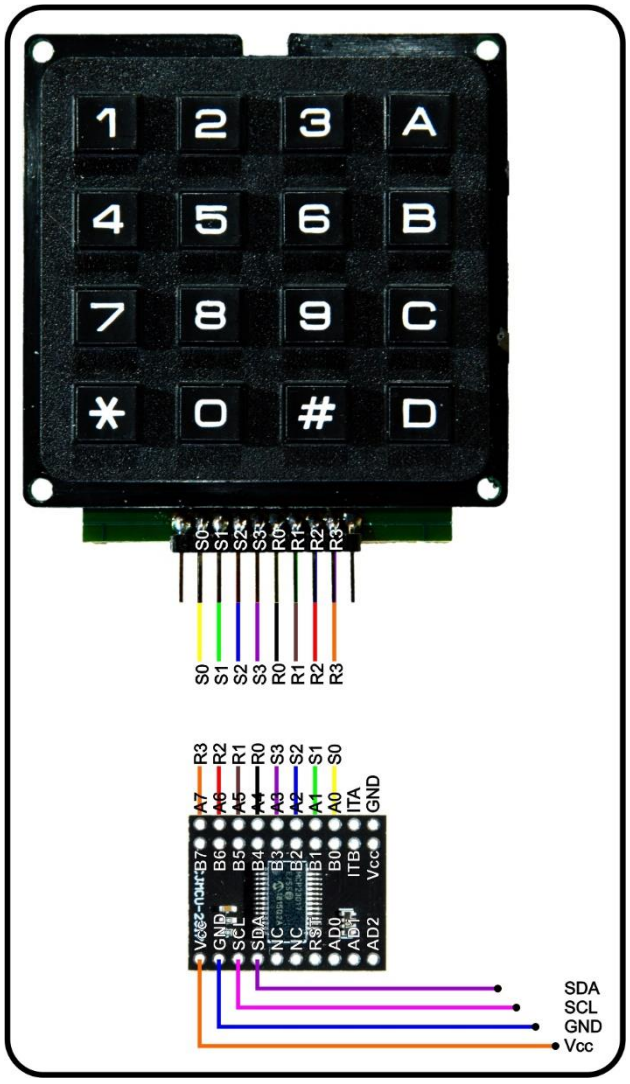
Dasselbe mit der Taste C

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

Prima! Einige höhere, von der Hardware unabhängige Methoden zur Tastenabfrage besprechen wir nach dem folgenden Kapitel. Darin geht es um die effektivere Softwarelösung und den Tastenfeldanschluss via I2C.

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

Der Vorteil dieser Methode liegt eindeutig darin, dass nicht einzelne Leitungen der Reihe nach abgefragt werden müssen. Stattdessen wird einer der 8-Bit-Ports (A) des MCP23017 halbiert. Das obere Nibble steuert die Zeilenleitungen, das untere Nibble die Spaltenleitungen, aber eben nicht einzeln, sondern parallel. Ebenso gut wäre es, den Port A für die Zeilen und B für die Spalten einzusetzen. Für größere Tastenmatrizen ist das unumgänglich.



Angesteuert wird der MCP23017 über den I2C-Bus und damit parallel zum Display und evtl. zum BMP280, falls der noch in der Schaltung ist. Das ist möglich, weil alle Schnittstellen unterschiedliche Hardwareadressen haben. Übrigens, passen Sie auf, dass Sie wirklich einen MCP23017 bekommen. Sein Brüderchen, der MCP23S17, wird über den SPI-Bus angesteuert und ist hier nicht brauchbar.

Für den MCP23017 habe ich ein Modul erstellt. Die Klasse MCP23017 stellt eine ganze Reihe von Konstanten und Methoden bereit, die eine übersichtliche und einfache Programmierung des Bausteins ermöglichen, ohne dass man sich mit den Methoden des I2C-Moduls herumschlagen muss. Die Tastenabfrage ist hiermit in 7 Zeilen passiert. Obwohl keine Schleifen zu durchlaufen sind, ist dieser Algorithmus scheinbar langsamer

wie der oben beschriebene, dauert aber für jede Taste gleich lang (ca. 4ms). Das liegt aber nicht an der Methode an sich, sondern am I2C-Bus, der uns ausbremst. Hätte der ESP32 von vorn herein ansprechbare Parallelports, dann wäre diese Methode eindeutig die schnellere. Die AVR-Processoren mit ihren Parallelports profitieren also von diesem Algorithmus.

Die Klasse KEYPAD_I2C erbt von der Klasse MCP23017 und kann daher direkt auf deren Namensraum zugreifen. Die Liste **keyNumber** der Rohcodes sieht etwas anders aus, wie die bereits bekannte, der Konstruktor auch.

```
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))
```

Der Konstruktor nimmt vom Hauptprogramm als Positionsparameter ein I2CBus-Objekt und als optionalen Parameter die Hardwareadresse des MCP23017. Beide reicht er an die Klasse MCP23017 weiter, die unter dem Alias MCP importiert wurde.

Aber schauen wir uns den Wunderalgorithmus einmal an.

```
    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.

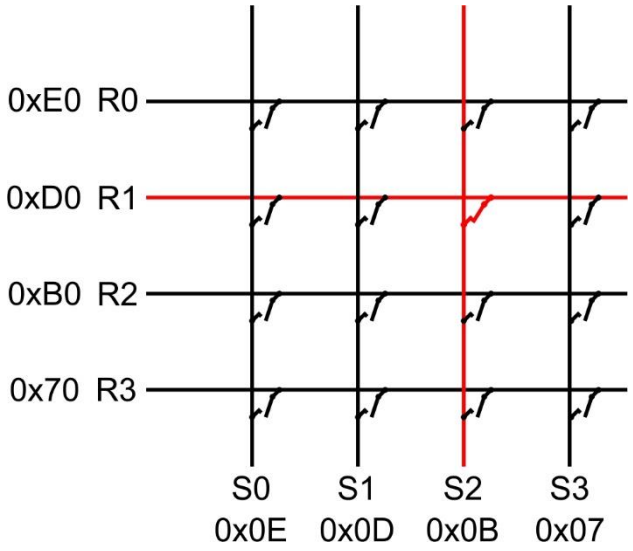
changeIODIRA(0x00,KeyMask,True)
KeyMask hat den Wert 0b00001111. Wir undieren das Datenrichtungsregister A mit 0b00000000, löschen also alle Bits und setzen dann durch Oderieren mit KeyMask das Low-Nibble als Eingänge. True sorgt dafür, dass für diese Eingänge die Pullups eingeschaltet werden. Die vier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



invertIODIRA(True)
Wir invertieren das Datenrichtungsregister A, Eingänge werden zu Ausgängen und umgekehrt. True - die Pullups der Eingangsleitungen werden aktiviert..

setGPIOA(iod)
S2 führt jetzt Low-Potential. Wir lesen GPIOA ein und erhalten den Wert

0b10111101 = 0xDB

Wir treffen hier auf eine negative Logik, das heißt es zählen die Bits, die **nicht** gesetzt sind. Man kann es auch so sehen, für die Spalten, das Low-Nibble, fehlt von der 15 das Bit mit der Wertigkeit 4. Somit bleiben 15 – 4 = 11 = 0x0B übrig. Bei den Zeilen liegt Bit1 auf Low, es fehlt im Nibble also die 2. 15 – 2 = 13. Das High-Nibble erhält den Wert 0xD0. Bitweise oderiert ergibt sich 0xD0 | 0x0B = 0xDB

KEYPAD_I2C.keyNumber.index(iod)
Die Rohwerte stehen wieder an entsprechender Stelle in der Liste keyNumber. Der Index eines Werts liefert die Tastennummer wie im ersten Beispiel. Damit haben beide Methoden denselben Aufruf und liefern das sinngemäß gleiche Ergebnis. Der Weg dorthin ist ein anderer, aber nach außen hin spielt das keine Rolle. Die Methode **debounceKey**() besitzt in beiden Klassen sogar denselben Code. Die beiden Klassen liefern also mit Ausnahme des Konstruktors Instanzen mit derselben API.

Puh! Das war jetzt wirklich weit ausgeholt, aber ich denke, es ist wichtig, die grundlegenden Mechanismen einer Matrixabfrage verstanden zu haben und gegen einander abwägen zu können. Lassen Sie uns noch einen Blick auf die Klasse KEYPAD werfen, welche hardwareübergreifende Methoden für die Benutzung von Tastaturen anbietet.

Der Konstruktor erwartet eine Instanz der Klassen KEYPAD_P oder KEYPAD_I2C. Sogar KEYPAD_LCD – Instanzen werden trotz des geringen Zeichenumfangs akzeptiert. Ein Trick soll nicht unerwähnt bleiben. Die Methode kp.key(), besser deren Referenz, wird der Instanz-Variable self.key zugewiesen. In MicroPython kann man das tun. Das kann helfen, ellenlange Aufrufe von importierten Methoden zu verkürzen. Aus self.k.key() wird self.key(). Sie meinen, das sind doch Peanuts? Na ja, dann stellen Sie sich doch einmal vor, statt k müsste keyPad_I2C_Instanz eingegeben werden und das 20 mal oder öfter.

```
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
```

Die drei Methoden dieser Klasse erfüllen nützliche Zwecke.

Die Methode **waitForKey**(delay) wartet delay Sekunden auf die Betätigung einer Taste. Wird in dieser Zeit eine Taste gedrückt, dann gibt die Methode die Tastennummer zurück, andernfalls -1. Die Methode wartet ewig, falls an delay der Wert 0 übergeben wird.

Je nach dem Definitionsstring **asciiCode** in den KEYPADXXX-Klassen liefert die Methode **asciiKey**() ein normales ASCII-Zeichen statt der Tastennummer zurück. Welche Zeichen das sein sollen, legen Sie mit Hilfe der Zeichenkette in asciiString fest. Die Position des Zeichens im String entspricht der Tastennummer, welche die Methode key() liefert. Nicht druckbare Zeichen, wie \n werden als Hexcode angegeben. Zum Beispiel entspricht \n also \x0D. Meine Definition sieh so aus:

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

Übrigens, wenn Sie mehrere asciiCode-Definitionsstrings anlegen, können Sie eine Tastenmehrfachbelegung etablieren. Allerdings ist dann nicht nur die Beschriftung der Tasten eine Herausforderung.

Die Methode **padInput**() erlaubt schließlich die Eingabe von Ziffernfolgen ergänzt durch "*" und "+", welches durch die Taste "#" eingegeben wird. Taste "A" löscht rückwärts und Taste "D" übernimmt den String zur weiteren Verarbeitung. Wurde dem Konstruktor ein Display-Objekt übergeben, dann erscheinen die Zeichen auch im Display. Ohne Display-Objekt ist Blindflug angesagt.

Drei optionale Parameter steuern die Eingabe. **delay** (default = 0.5s) schützt vor ungewollter Mehrfacheingabe eines Zeichens, weil die key-Methode einfach zu schnell ist, brauchen wir eine Verzögerung. **xp** und **yp** legen die Position im Display fest, ab der die Eingabe dargestellt wird. Default ist die linke obere Cursorposition, entsprechend "home".

An dieser Stelle angekommen, testen wir jetzt die drei Methoden. Falls Sie das noch nicht getan haben, schicken Sie jetzt bitte alle Module ins Flash des ESP32 und öffnen Sie das Programm keypad_p_test.py in einem Editorfenster. Je nach der gewählten Anschlussart der Tastatur muss genau eine der folgenden Zeilen entkommentiert sein.

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

Starten Sie das Programm mit F5. Wenn der REPL-Prompt (>>>) wieder erscheint, sind Sie bereit für die folgenden Tests über die Kommandozeile.

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

Im ersten Fall wurde innerhalb von 3 Sekunden die Taste 5 gedrückt. Beim zweiten Aufruf wurde der Tastendruck versäumt.

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

Jetzt wurde beim ersten Aufruf keine Taste gedrückt und beim zweiten die "*"-Taste. Beachten Sie, dass nicht der Tastencode 14 sondern der ASCII-Code für "*" zurückgegeben wurde.

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

Mit einer Verzögerung von 0,3 Sekunden wird das Tastenfeld abgefragt. Die Eingaben werden ab der Position Spalte 3 und Zeile 1 im Display angezeigt. Stellen Sie die Totzeit nicht zu lang ein, denn sonst werden Tastenbetätigungen übersprungen. Bei zu kurzem Totzeitwert erscheinen Eingaben mehrfach. Um die Eingabe abzuschließen, drücken Sie die Taste D.

Als Ergebnis des vorangegangenen Befehls sollten Sie Ihre Eingabe sowohl auf dem Display als auch im Terminalfenster sehen.

Als die Teleklingel phonte,

treppte ich die Ranne runter und türte gegen die Bums. Wir wollen nicht hoffen, dass das passiert. Aber es wäre schon schön, wenn das Handy uns auf einen Anruf vom SIM808 aufmerksam machen würde. Ich bin sicher, das wird es tun!

Das MicroPython-gps-Modul hat zu diesem Zweck ein paar neue Methoden erhalten. Mit deren Hilfe sind wir in der Lage, weitere AT-Befehle zu nutzen, die das SIM808 zur Verfügung stellt. Ich stelle diesen Ausschnitt aus dem Programmtext kurz vor, dann telefonieren wir, 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()
Wir rufen über den AT-Befehl "AT+CSQ\r\n" die Information über die Feldstärke des Empfangssignals ab und rechnen den Wert in dBm um, den die Methode zurückgibt.

simIsRegistered()
Die Methode prüft über den AT-Befehl " AT+CREG?\r\n" ob das SIM808 in ein Funknetz eingeloggt ist. Wird in der Antwort ein "+CREG:" gefunden, dann teilt uns die folgende Ziffer mit, ob wir in einem heimatlichen Netz (1) oder in einem fremden Netz (5 = roaming) angemeldet sind.

simAlarmSound(nbr=None)
Wie das Handy verfügt das SIM808 über verschiedene Ruftöne mit den Nummern 0 bis 19. Wenn wir eine Nummer aus diesem Bereich an die Methode übergeben, dann wird die entsprechende Melodie eingestellt. Der Aufruf ohne eine Zahl liefert die aktuell eingestellte Nummer zurück

simClock(dt=None)
Wird an dt ein gültiger Date-Time-String übergeben, dann wird die RTC des SIM808 auf diese Werte eingestellt. Ohne Parameterübergabe gibt die Methode die aktuelle Zeit zurück. Ein Date-Time-String hat folgendes Format:

yy/mm/dd,hh:mm:ss±tz

tz ist der zeitliche Versatz der Zeitzone gegenüber UTC in Viertelstunden. Für die Zeitzone MEZ bei Winterzeit (geografische Normalzeit) ergibt sich zum Beispiel folgender String für 15.11.2020, 8:45:32:

20/11/15,08:45:32+04

Im Sommer ist wegen der Zeitumstellung tz=08 zu setzen.

simDialOut(nummer)
Das Beste kommt zum Schluss. Diese Methode nimmt eine Telefonnummer, ruft an und meldet, wenn die Verbindung zustande gekommen ist "OK".

Bei uns übernimmt nun das Programm **cellphone.py** die Steuerung. Es enthält als wesentliche Funktionen:

- Taste B, Nummerneingabe und anrufen (auflegen: Taste D)
- Anruf entgegennehmen Taste A (ablehnen Taste != A,D; auflegen: Taste D)
- eingetroffene SMS-Nachricht stufenweise anzeigen und löschen (automatisch)
- Programmabbruch durch RST-Taste am LCD-Keypad

Sie können natürlich jederzeit den Funktionsumfang erweitern, indem Sie einige der vorgestellten Methoden in die Jobliste aufnehmen. Oder haben Sie Lust, ein paar weitere AT-Befehle in entsprechende eigene Methoden einzubauen? Nach dem Listing, das Sie in die Lage dazu versetzt, habe ich noch einige Empfehlungen. Auf zwei neue AT-Befehle, die im Vorspann verwendet werden, weise ich vorab schon jetzt hin.

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

Der erste weist dem Kopfhörer volle Lautstärke zu und der zweite aktiviert die Anzeige der Rufnummer bei eingehenden Anrufen.

```
# 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% des Programmumfangs nehmen die Vorbereitungen in Anspruch, Importe und Declarationen von Variablen, Objekten und Funktionen. Würden wir all das, was in Modulen verpackt ist, für sich einzeln im Hauptprogramm definieren, wäre die Liste weitaus länger. Die Verwendung von Modulen schafft Übersicht und lesbareren Code. Wenn man es wirklich genauer wissen möchte, lohnt ein Blick hinter die Kulissen in die jeweilige Klassendefinition.

Wie arbeitet die Hauptschleife?
Sobald das Display "SIM808 READY" meldet, wird auch die Zeit aus der RTC angezeigt und unser Programm ist im Rundendauerlauf.

Der UART-Buffer wird ausgelesen. Findet sich darin der String "RING", dann kommt gerade ein Anruf herein. Lassen Sie 2- bis 3-mal klingeln. Jetzt sollte auch die Telefonnummer des Anrufers im Buffer gelandet sein. Das ist dann der Fall, wenn sich der String "+CLIP:" in **buf** findet. Wir parsen den Bufferstring und filtern die Rufnummer heraus, lassen sie im Display anzeigen. 15 Sekunden haben wir Zeit, den Ruf mit der Taste A anzunehmen oder mit einer beliebigen anderen Taste abzulehnen. Wird keine Taste betätigt, erfolgt die Ablehnung des Anrufs nach 15 Sekunden automatisch.

Wurde der Ruf angenommen, kann beliebig lange gesprochen werden. Sie können auflegen, indem Sie die Taste D drücken. Hat die Gegenstation aufgelegt, erkennt das Programm dies anhand des Strings "NO CARRIER" im Bufferstring.

Wurde der Anruf abgelehnt, bekommen beide Seiten eine Ansage vom Provider und eine SMS-Nachricht. Dieser Fall wird spätestens durch das nächste "if" in der Mainloop detektiert. Der UART-Buffer enthält dann die Zeichenkette "+CMTI:". Der Test unmittelbar nach dem abgelehnten Anruf fällt meist negativ aus, weil die Nachricht erst einige Sekunden verzögert eintrifft.

Jede eintreffende SMS-Nachricht macht sich durch den Text "+CMTI:" im Buffer bemerkbar. Indem wir den Bufferinhalt darauf abklopfen, bekommen wir jede ankommende Nachricht am Display präsentiert - nach dem Einlesen und Zerlegen.

Die Funktion **parseForSMS**() liefert uns die Indexnummer der SMS, die wir zum Einlesen benötigen. Einlesen und Zerlegen erledigt die Methode g.gsmReadSMS(nbr), der wir in nbr den Index übergeben. Als Rückmeldung bekommen wir ein Tuple mit den Stringelementen (Status,Phone,Date,Time,Message). Drei Schritte bringen den Inhalt auf das Display, Status und Rufnummer, Datum und Uhrzeit sowie, auf jeweils zwei Zeilen aufgeteilt, die Nachricht selbst. Mit einer beliebigen Taste können sie vorzeitig weiter schalten. Wird keine Taste gedrückt, schaltet das Programm nach 10 Sekunden selbst zur nächsten Stufe weiter.

Ausgehende Anrufe werden mit der Taste B eingeleitet. Sie werden zur Eingabe der Rufnummer aufgefordert. Es können alle Ziffern plus "*" und "+" angegeben werden. Das "+"-Zeichen liegt auf der Taste #. Die Taste A löscht rückwärts, Taste D übernimmt die Eingabe. Ein Semicolon ";" muss angehängt werden, damit die Syntax des AT-Befehls stimmt. Enthält die Eingabe mindestens 4 Zeichen, übernimmt die Funktion **simDialOut**() den Wählvorgang und den Verbindungsaufbau. Natürlich können Sie diese Werte alle an Ihre Bedürfnisse anpassen.

Die Verbindung steht, wenn sich in der Antwort im Buffer ein "OK" findet. Mit der Taste D kann man selber auflegen. Ein "NO CARRIER" findet sich im UART-Buffer, wenn die Gegenstation aufgelegt hat.

Was können Sie tun, um die Endlosschleife geordnet zu verlassen, falls Sie etwas am Programm ändern möchten? Wir hatten anfangs von der Notbremse gesprochen. Aussteigen geht mit der RST-Taste am LCD-Keypad. Der ESP32 meldet den Ausstieg im Terminal und im Display.

Sie möchten, dass der ESP32 autonom mit dem Programm startet? Kein Problem, kopieren Sie den gesamten Programmtext aus der Datei cellphone.py in eine neu angelegte Blankodatei. Speichern Sie diese als boot.py ab und laden Sie boot.py zum ESP32 hoch. Beim nächsten Kaltstart des Controllers wir Ihrem Wunsch entsprochen.

Gut, damit wären wir fast am Ende dieser Folge. Fast, denn ich hatte ja versprochen, noch einige interessante AT-Befehle vorzustellen. Jeden davon können Sie in einen weiteren if-Zweig in der Mainloop zu einem weiteren Menüpunkt machen. Zur Auswahl haben Sie ja noch 14 Tasten auf dem Tastenfeld und 5 auf dem LCD-Keypad.

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="21/05/12, 20:10:04+08"\r\n	AT+CCLK?\r\nAT+CCLK?\r\r\n+CCLK: "21/05/15,15:10:59+08"\r\n\r\nOK\r\n Abfrage der Zeit von der RTC des SIM808 Stellen der RTC **Hinweis**: Der Date-Time-String **MUSS** in **doppelten** Anführungszeichen stehen!

Für den CCLK-Befehl habe ich schon vorgearbeitet. Im Listing finden Sie die Funktion **getDateTime**() womit Sie einen Date-Time-String über den Ziffernblock eingeben können. Die am Beginn der Liste definierte Liste f zeigt, wie man komplexe Strings aufbauen kann. Die Tuples enthalten den Text für einen Eingabeprompt und als zweites Element das Trennzeichen zum nächsten Teilstring. Als Erweiterung ließen sich zum Beispiel auch eine Unter- und Obergrenze für Plausibilitätsprüfungen bei Zahleingaben integrieren.

Im gps-Modul gibt es in der Klasse SIM808 die Methode **simClock**(), die ja auch im Programm benutzt wird. Beide zusammen könnten zu einem Menüpunkt "Stellen der Uhrzeit" werden. Also frisch ans Werk!

Ich wünsche noch viel Spaß beim Einbau der Befehle, beim Telefonieren und SMSen.

PDF in deutsch
PDF in english