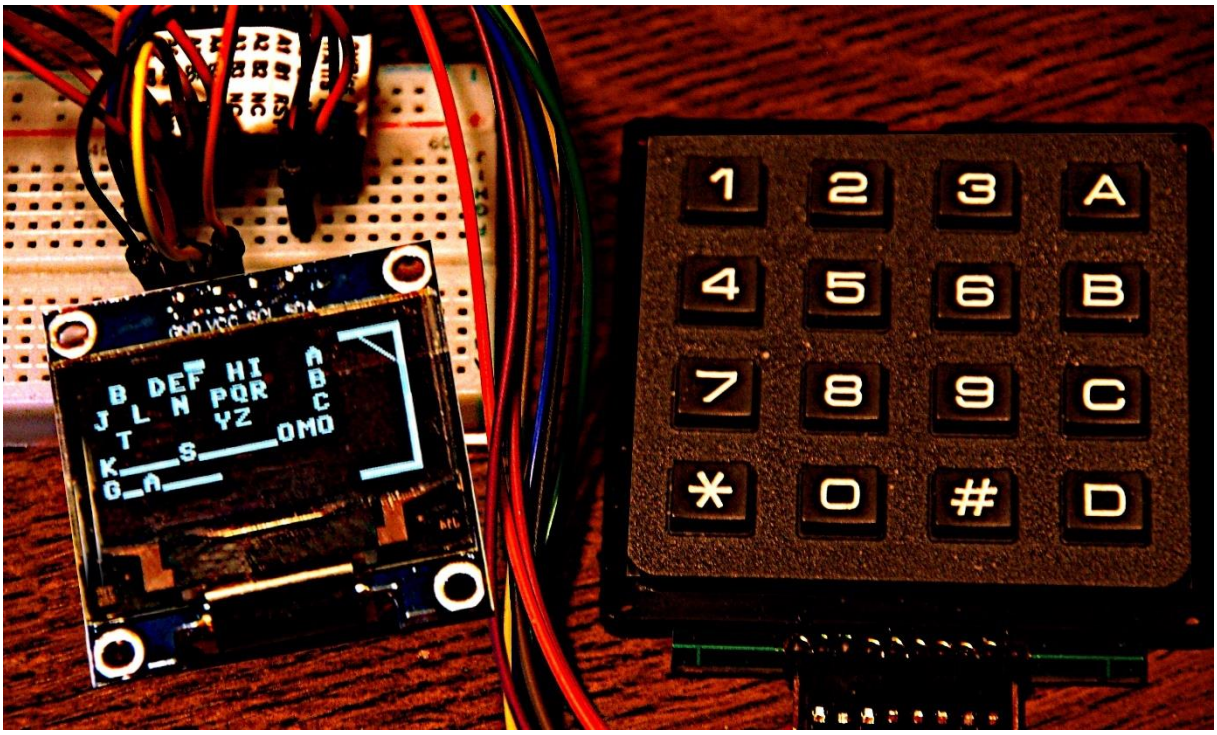


Abbildung 1: Hangman auf dem ESP32

Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

In den vorangegangenen Episoden (Ringmaster2 und Bandit) zum Thema "Spiele am ESP32 mit MicroPython" hatten Neopixelringe die Hauptrolle inne. Bei der heutigen Folge wäre deren Einsatz zur bunten Ergebnisanzeige zwar auch denkbar, aber im Vordergrund steht eindeutig das OLED-Display mit seiner Text- und Grafikdarstellung. Letztere lässt das LCD-Keypad als Möglichkeit der Anzeige schon einmal außen vor. Die 5 Tasten würden ausreichen, aber niemand setzt ein LCD-Keypad nur der Tasten wegen. Deshalb verwende ich wieder das 4x4-Tastenfeld. Und wenn bei Bandit Zufall und Glück den Spielausgang beeinflusst haben, dann ist jetzt die Kombinationsgabe der Schlüssel zum Erfolg. Herzlich willkommen, lassen Sie uns gemeinsam ein bekanntes Spiel programmieren, das sich auch als Vokabeltrainer gut eignet, gemeint ist

Hangman

Spiele am ESP32 mit MicroPython

Bei der Beschreibung des Moduls keypad.py hatte ich bereits angedeutet, dass mit dem 4x4-Tastenfeld in Verbindung mit drei ASCII-Tabellen die Eingabe des gesamten Alphabets möglich wäre. Während der Planung zu Hangman dachte ich genau an den Einsatz dieser Lösung, um Buchstaben an den ESP32 zu liefern.

Allein, das rein mechanische Problem, eine ordentliche Beschriftung für die Tastatur zu erstellen, ließ mich jedes Mal wieder davon Abstand nehmen.

Nun, bei Hangman müssen ja nicht ellenlange Texte eingegeben werden, es reichen ja einzelne Buchstaben. Das führte letztlich zur Programmierung einer Bildschirmtastatur auf dem OLED-Display. Und weil diese Anzeige keinen Cursor besitzt, wurde auch ein solcher programmtechnisch hinzugefügt. Wie das alles funktioniert und noch einiges mehr, erfahren Sie in diesem Beitrag. Ein Teil der Hardware aus den letzten Folgen kommt wieder zum Einsatz.

1	ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder ähnlich
1	0,96 Zoll OLED I2C Display 128 x 64 Pixel - 1x OLED
1	4x4 Matrix Keypad Tastatur - 1x Keypad
1	MCP23017 Serielles Interface Modul
1	Battery Expansion Shield 18650 V3 inkl. USB Kabel
1	Li-Akku Typ 18650
1	LED Ring 5V RGB WS2812B 12-Bit 37mm als Option

Da wir recht umfangreiche Wortlisten bearbeiten wollen, was logischerweise sehr speicherplatzintensiv ist, wurde erneut ein ESP32 in Dienst gestellt. Weil das OLED-Display über I2C angesprochen wird, habe ich auch die Tastatur wieder auf diese Weise verbunden. Nur wenn Sie es etwas bunter mögen, können Sie zur Anzeige der Annäherung an die Lösung einen Neopixelring verwenden. Der ist aber nicht zwingend nötig, sondern nur nettes Beiwerk, könnte aber darüber informieren, wie viele Zwölftel der gesamten Buchstabenmenge bereits gefunden sind.

Das Spiel

Das Spielfeld habe ich in Libre-Office-Calc entworfen. Mit viel Phantasie kann man das Aussehen der späteren Grafik erahnen.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	A	B	C	D	E	F	G	H	I		A		-	-	-	|
2	J	K	L	M	N	O	P	Q	R		B			|	\	|
3	S	T	U	V	W	X	Y	Z			C			O		|
4	_	_	_	_	_	_	_	_	_	_	_	_	/	|	\	|
5	_	_	_	_	_	_	_	_	_	_	_	_	/		\	|
6	_	_	_	_	_	_	_	_	_	_	_	_		-	-	-

Abbildung 2: Spielfeld

Schauen wir uns doch gleich einmal die OLED-Tastatur an. Nach Auswahl einer der 9 Spalten übernehmen wir eines der drei Zeichen durch drücken der Taste A, B oder

C auf dem 4x4-Tastenfeld. Das entsprechende Zeichen wird dann für etwa eine Sekunde in Spalte J angezeigt. Wenn der Buchstabe im Lösungswort vorkommt, erscheint er überall dort statt dem Unterstrich. War es ein Fehlgriff, wird ein Stück des Galgens aufgebaut. Das kennen wir alle aus der Schulzeit, wo wir in drögen Geographie- oder Deutschstunden Hangman hinter dem Rücken des Vordermanns auf Papier gespielt haben.

Zu Beginn jeder Spielrunde bekommen wir 12 Punkte auf unser Konto, sie entsprechen den 12 Teilen des Galgenmännchens. Mit jedem falschen Buchstaben wird nicht nur das Bild vervollständigt, sondern auch ein Punkt abgezogen. Es kommt also nicht nur darauf an, irgendwann alle Buchstaben des Lösungsworts gefunden zu haben, sondern mit möglichst wenig falschen Buchstaben, denn am Ende werden die übrig gebliebenen Punkte zum Gesamtspielstand addiert. Letzterer wird angezeigt, bis es mit einem Tastendruck in die nächste Runde geht.

Erneut wird eine Zufallszahl ermittelt und, mit dieser als Index, ein neues Wort aus der Wortliste geholt. Die Wortliste wohnt als Datei im Flash des ESP32, bis sie beim Kaltstart ins RAM geladen wird. Die Wörter dürfen bis zu 36 Zeichen haben, wobei Umlaute und scharfes "s" als Doppellaut notiert werden, da sie im Display-Alphabet nicht vorkommen. Da der Flashspeicher nahezu unerschöpflich ist, könnte man dort auch mehrere derartige Dateien, zum Beispiel zu verschiedenen Themenbereichen, unterbringen. Damit während einer Sitzung Wörter nicht mehrfach geladen werden, führt hangman.py darüber Buch. Erst wenn die ganze Liste (60 Begriffe) durch ist, kommen Wiederholungen zwangsweise vor. Die Wortlisten werden mit einem beliebigen Editor erstellt und als Textdatei im Workspace abgespeichert. Mit Thonny werden sie von dort auf den ESP32 übertragen.

So, jetzt wissen Sie, was Sie erwartet. Klingt alles ganz einfach, aber ich bin sicher, dass Sie viel lieber wissen wollen, wie das in MicroPython umgesetzt wird. Dem kann entsprochen werden, zumal es dieses Mal um ein Feature der Sprache Python ganz allgemein gehen wird. Wir beschäftigen uns gleich im nächsten Kapitel damit, dort besprechen wir auch die anderen interessanten Stellen des Programms hangman.py. Danach stelle ich die benötigte Software vor. Abschließend geht es um Hardwareaufbau und Inbetriebnahme.

Tricks und Infos zu MicroPython

Sie kennen Funktionen und haben sie auch schon vielfach benutzt, um Code, der mehrfach in einem Programm genutzt wird, an einer Stelle zu konzentrieren. Das spart Speicherplatz und erhöht die Lesbarkeit eines Programms.

Sie wissen auch schon, dass Objekte in einer Funktion **lokal** sind, sofern sie nicht durch **global** als solche im Funktionskörper declariert werden. Lokale Objekte sind nur im Scope der Funktion referenzierbar. Wenn MicroPython die Funktion verlassen hat, kann es sich an die lokalen Variablen nicht mehr erinnern. Ähnlich verhält es sich mit Objekten, die als Parameter an die Funktion übergeben wurden.

Nun kommt es vor, dass innerhalb einer Funktion weitere Funktionen definiert werden, wie im nächsten Beispiel.

```
# alltest.py
def outer(x):
    n=3
    def mal():
        produkt=n*x
        return produkt
    return x+produkt
```

Start mit F5 macht dem globalen Namensraum (aka Scope) die Funktion outer() samt Inhalt bekannt. Die Syntax ist korrekt, der Aufruf liefert aber eine Fehlermeldung.

```
>>> outer(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in outer
NameError: name 'produkt' isn't defined
```

Klar, **outer**(x) kennt **produkt** nicht, denn die Variable ist lokal zur inneren Funktion **mal**(). Der globale Scope kennt natürlich weder **x**, noch **n** und schon gar nicht **produkt**.

```
>>> x
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'x' isn't defined
```

```
>>> produkt
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'produkt' isn't defined
```

Eine analoge Fehlermeldung bekommen wir für n.

Ändern wir die letzte Zeile, um das zu erhalten, was wir wohl haben wollten.

```
def outer(x):
    n=3
    def mal():
        produkt=n*x
        return produkt
    return x+mal()
```

F5 und outer(7) liefern wie erwartet 28. Die innere Funktion **mal**() kann also auf das außerhalb definierte n und auch auf das als Parameter übergebene x lesend zugreifen. Anders sieht es aus, wenn wir innerhalb von mal() den wert von n verändern wollen.

```
# alltest.py
def outer(x):
    n=3
    def mal():
        n=n*x
        return produkt
    return x+mal()
```

F5 und outer(7):

```
>>> outer(7)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 7, in outer
  File "<stdin>", line 5, in mal
NameError: local variable referenced before assignment
```

n wird jetzt von MicroPython durch die linke Seite der Zuweisung als lokal zu **mal**() aufgefasst. Andererseits wird es im Produkt n*x aber referenziert, bevor man n einen Wert zugewiesen hat. Aber auch n=8*x liefert an dieser Stelle nicht das Ersehnte, denn nach dem Verlassen von mal() ist n immer noch 3 statt 56.

```
def outer(x):
    n=3
    def mal():
        n=8*x
        produkt=n*x
        print("innen",n)
        return produkt
    ergebnis=x+mal()
    print("außen",n)
    return ergebnis
```

F5 und outer(7):

```
innen 56
außen 3
399
```

Das alles ändert aber auch nichts an der Tatsache, dass nach dem Verlassen von outer() keiner der Inhalte der Funktionen mehr bekannt, geschweige denn wieder herstellbar ist.

Ich habe oben das Schlüsselwort **global** erwähnt, das es ermöglicht, einer Funktion den übergeordneten Scope einer Variablen mitzuteilen. Dadurch wird es möglich, aus einer Funktion heraus, deren Wert nachhaltig zu verändern. Während sich global auf die oberste Ebene bezieht, gibt es bei verschachtelten Funktionen etwas Vergleichbares durch das Schlüsselwort **nonlocal**.

```
def outer(x):
    n=3
    def mal():
        nonlocal n
        print("vorher",n)
        n=8*x
        produkt=n*x
        print("innen",n)
        return produkt
    ergebnis=x+mal()
    print("außen",n)
    return ergebnis
```

F5 und outer(7):

```
>>> outer(7)
vorher 3
innen 56
außen 56
399
```

Wie Sie sehen, ist es jetzt möglich, der außerhalb von mal() liegenden Variablen n, aus der Funktion mal() heraus, einen neuen Wert zuzuweisen.

Jetzt setzen wir noch eins drauf und sorgen dafür, dass sich die innere Funktion mal() auch nach dem Verlassen von outer() noch aufrufen lässt. Mehr noch, wir sorgen zusätzlich dafür, dass sich die innere Funktion auch an die zu ihr nichtlokale Variable n und sogar an den Parameter x der äußeren Funktion outer() erinnert. Dieses "Geheimnis" von MicroPython heißt Closure (sprich: "Klouscher"). Was steckt dahinter?

Funktionen können in MicroPython als Werte behandelt werden. Das heißt man kann eine Funktion einer Variablen zuweisen, sie als Funktionsparameter übergeben oder als Ergebnis einer Funktion zurückgeben. Bitte beachten Sie, ich spreche hier nicht von einem Funktionswert, sondern von der Funktion selbst. Untersuchen wir dazu das nächste Beispiel.

```
def outer(x):
    n=0
    def mal():
        nonlocal n
        print("vorher",n)
        n=n+1
        produkt=n*x
        print("nachher",n)
        return produkt
    return mal
```

F5 und h=outer(2)

```
>>> h=outer(2)
>>> h()
vorher 0
nachher 1
2
>>> h()
vorher 1
nachher 2
4
>>> h()
vorher 2
nachher 3
6
```

Haben Sie bemerkt, dass outer() diesmal keinen Funktionswert zurückgibt wie bei den früheren Beispielen? Die äußere Funktion gibt die innere Funktion **mal** zurück, die wir der Variablen h zuweisen. Hiermit ist die Funktion **mal** aufrufbar, auch wenn wir den Scope von outer bereits verlassen haben. Und noch zwei Dinge sind absolut bemerkenswert. h() alias **mal**() erinnert sich bei jedem Aufruf sowohl noch an den Parameter 2, den wir in x an outer() übergeben haben, als auch an die außerhalb von **mal** gelegene Variable n. Viel mehr noch, es ist sogar möglich, mit jedem Aufruf den Wert von n zu erhöhen.

Auf ähnliche Weise können Sie das 5-er-Einmaleins erzeugen

```
>>> k=outer(5)
>>> k()
vorher 0
nachher 1
5
>>> k()
vorher 1
nachher 2
10
>>> k()
vorher 2
nachher 3
15
```

Die innere Funktion ist eine Closure. Das sagt uns auch der Aufruf von h oder k ohne die Funktionsklammern.

```
>>> h
<closure>
```

Allgemein ist das eine Funktion, die von einer anderen umgeben ist und einen erweiterten Namensraum besitzt, in den die nichtlokalen Variablen der umgebenden Funktion eingeschlossen sind, sofern sie in der inneren Funktion vorkommen. Eine lokale Variable wie **produkt** im Scope der inneren Funktion nennt man gebunden (aka bound variable), eine Funktion, die nur gebundene Variablen enthält ist ein geschlossener Ausdruck (aka closed term). Freie Variablen (aka free variables),

nichtlokale Variablen also, sind solche, die außerhalb einer Funktion definiert sind. Eine Funktion, die freie Variablen enthält ist ein offener Term (aka open term). Als Closure bezeichnet man nun eine Funktion, die ihre nichtlokalen Variablen quasi einfängt und damit den offenen Term zu einem geschlossenen macht, daher kommt der Name Closure.

Was können wir jetzt mit Closures machen? Nun wir haben ein Strichmännchen am Galgen zu zeichnen. Das besteht aus 12 Linienelementen. Die Befehle dazu habe ich als Strings in den Elementen der Liste **galgen** abgelegt, die sich zusammen mit einer Zählvariablen n, beide als freie Objekte, in der Funktion **initGalgen**() befinden. Im Scope von initGalgen() liegt auch die innere Funktion **nextPart**(). Für nextPart sind **galgen** und **n** nichtlokal, also freie Variablen. Mit der Rückgabe der Funktion nextpart durch initGalgen() wird die innere Funktion zu einer Closure, die sich bei wiederholten Aufrufen an die eingefangenen **galgen** und **n** erinnert. Weil überdies n als **nonlocal** extra ausgewiesen ist, kann diese freie Variable durch nextPart() auch noch nach außen wirkend verändert werden. Damit realisieren wir eine automatische Zählfunktion. Wenn wir jetzt die Rückgabe von initGalgen an die Variable **zeichnen** zuweisen, können wir vollautomatisch mit jedem Aufruf von zeichnen() den nächsten Strich unseres Bildes zeichnen, ohne uns um eine Reihenfolge kümmern zu müssen. Ferner teilt uns zeichnen() durch den Rückgabewert False mit, dass die Runde abgelaufen ist, was wollen wir mehr?

Anmerkung:
Das Erinnerungsvermögen an den Wert einer lokalen Variablen kann in C und der Arduino-IDE dadurch erreicht werden, dass man die Variable in der Funktion als **static** delcariert.

```
def initGalgen():
    n=0
    galgen=[
        "d.fill_rect(d.width-25,d.height-3,24,3,1)", # Boden
        "d.fill_rect(d.width-3,d.height//2,3,d.height//2,1)",
        "d.fill_rect(d.width-3,0,3,d.height//2,1)", # Mast
        "d.fill_rect(d.width-25,0,24,3,1)", # Balken
        "d.line(d.width-18,2,d.width-2,18,1)", # Stuetze
        "d.fill_rect(d.width-22,0,3,15,1)", # Strick
        "d.writeAt('O',13,1,False)",  # Kopf
        "d.fill_rect(d.width-22,19,3,20,1)", # Body
        "d.line(d.width-22,18,d.width-26,33,1)", # Arm links
        "d.line(d.width-19,18,d.width-14,33,1)", # Arm rechts
        "d.line(d.width-22,37,d.width-26,50,1)", # Bein links
        "d.line(d.width-19,37,d.width-14,50,1)",]# Bein rechts
    def nextPart():
        nonlocal n
        exec(galgen[n])
        d.show()
        n+=1
        if n<=11:
            return True
        else:
            return False
    return nextPart
```

```
>>> zeichnen = initGalgen()
>>> zeichnen()
>>> zeichnen)() …
```

Der Befehl exec(galgen[n]) ist, für sich allein gesehen, schon ein Bonbon, das man auf der Zunge zergehen lassen muss. Ermöglicht er doch die Ausführung von MicroPython-Code, den man als String vorhält. Wieder genau richtig für unser Vorhaben. Allerdings hat diese Anweisung auch ein erhöhtes Gefahrenpotential, dann nämlich, wenn der String von einem Benutzer durch einen Input-Befehl oder gar über ein Web-Portal eingegeben werden kann.

Ebenso als Closure verpackt ist das Erzeugen der Suchwortliste und der Wortauswahl aus derselben, inclusive Wiederholungsschutz. Letzterer wird, solange noch neue Wörter zur Verfügung stehen, durch die Liste **numbers** gewährleistet, indem der aktuell gewürfelte Index für die Wortliste an numbers angehängt wird. Indices in dieser Liste werden, so lange wie möglich, bis zum nächsten Kaltstart nicht ein zweites Mal verwendet. Aufbau und Funktion von initSuperWord() können Sie jetzt sehr wahrscheinlich bereits selbst erklären. Aber wie sieht es mit der Erklärung der folgenden Anweisung aus?

```
>>> initSuperWord()()
0
'Ouvertuere'
```

Besprechen wir jetzt den Rest des Programms, drei weitere Funktionen und das recht kurze Hauptprogramm warten noch darauf.

Die Funktion **showString**() bringt den gesuchten Begriff in seinen Entwicklungsstufen während der Spielrunde aufs Display. Zu dessen Ansteuerung dient die Klasse OLED zu der es einige Neuerungen gegenüber der Vorgängerversion zu vermelden gibt. Es gibt eine neue Instanzvariable yOffset, die bei den Methoden writeAt() und clearFT() die Wiedergabe oder Löschung der Textzeilen um bis zu 4 Pixelpositionen nach unten verschiebt. Das schafft für einen Cursorstrich am oberen Displayrand Platz. Dieser Cursor markiert die Auswahlspalte für Zeichen aus dem Bildschirm-Alphabet.

```
cursor(x,h,y=0,cset=True,show=True)
```

Die Parameterliste umfasst die Textspaltennummer und die Strichstärke h (1..4). cset =True setzt den Corsor, False löscht ihn und show=True schreibt sofort ins Display.

Die Methode blinkDisplay(cnt,off=0.2,on=0.8) tut, was ihr Name sagt. Mit cnt gibt man die Anzahl der Blinkvorgänge an.

Um zu verstehen, warum die Ausgabe des Lösungsworts so kompliziert aussieht, ist ein kurzer Abstecher ins Hauptprogramm empfehlenswert. Das Suchwort s wird als String aus der Datei "superwords.txt" in eine Liste eingelesen. Strings sind immutabel, das bedeutet, dass einzelne Zeichen eines Strings nicht einfach so verändert werden können, wie Elemente einer Liste.

```
>>> t="test"
>>> t[2]="x"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object doesn't support item assignment
```

Zwar gibt es eine Stringmethode replace(), aber die arbeitet nicht positionsbestimmt, sondern global über den String. Man kann also nicht einzelne Zeichen nur an einer ganz bestimmten Stelle ersetzen.

Wir müssen aber genau das tun. Anstatt nun den String an den gewünschten Stellen aufzutrennen, den Unterstrich durch das Zeichen zu ersetzen und das Ganze wieder zu einem neuen String zusammenzusetzen, gehen wir einen speicherplatzsparenderen Weg. Der Lösungsbegriff ist zunächst ein String mit der gleichen Anzahl "_" wie Zeichen im Suchwort sind. Diese "_" müssen nun an allen Stellen durch den ausgewählten Buchstaben ersetzt werden, an denen dieser im Suchwort vorkommt. Daher bauen wir den Lösungsbegriff in ein Bytearray um, dessen Elemente ganz gezielt verändert werden können, auch an mehreren Stellen.

```
s=getSuperWord()
print("Superword=",s)
s=s.upper()
U="_"*len(s)
Ub=bytearray(U.encode("utf8"))
showString(Ub)
```

showString() nimmt also nicht den Lösungsstring, sondern das dazu äquivalente Bytearray Ub. Aus dem String U wird auf dem Umweg über das bytes-Objekt U.encode("utf8") das Bytearray Ub, das die ASCII-Codes der Zeichen enthält. Für die Ausgabe müssen diese Codes natürlich wieder in ASCII-Zeichen zurückübersetzt werden. Das macht die Funktion chr(). Das Verfahren ist aufwendig, aber praktisch.

Außerdem muss der Lösungsbegriff auf bis zu drei Zeilen umgebrochen werden. Das geschieht in den beiden for-Schleifen nach der Ganzzahldivision seiner Zeichenanzahl und der Restberechnung modulo der Zeilenlänge.

In der Funktion **getChar**() stellen wir den Cursor mit Hilfe der Tasten "*" und "#" in die Spalte, aus der eines der drei Zeichen geholt werden soll. Wird dann eine der Tasten "A", "B" oder "C" gedrückt, pickt die Funktion das Zeichen aus dem Zeilenstring des Alphabets und gibt es zurück. Die Funktion eval() steht uns dabei wieder trickreich zur Seite.

```
A="ABCDEFGHI"
B="JKLMNOPQR"
C="STUVWXYZ "
……
                z=eval(taste)[cursorPos]
                d.writeAt(z,len(A)+1,ord(taste)-65)
                d.writeAt(" ",cursorPos,ord(taste)-65)
```

In **taste** steht der Kennbuchstabe der Zeile, A, B oder C. **eval**() macht daraus die Variable A, B oder C und **cursorPos** zeigt auf die Position im String. Das Zeichen wird vor der Zeilenkennung ausgegeben. Die Zeilennummer ergibt sich trickreich aus dem ASCII-Code der Zeilenauswahl. Die dritte Codezeile nimmt das Zeichen, der besseren Übersich wegen, aus der Alphabetauswahl heraus.

Die Taste "D" beendet das gesamte Spiel. Mit dem Parameter delay bremsen wir die Cursorbewegung gezielt aus, das erleichtert das Positionieren.

Die Funktion **checkChar**(c="") nimmt das von uns ausgewählte Zeichen und schaut nach, ob es im Suchwort-String vorkommt. Ist das der Fall, suchen wir dort nach den Positionen, ersetzen an diesen Stellen aber im Bytearray **Ub** den "_" durch den ASCII-Code des Zeichens und geben True zurück. Das bedeutet, wir dürfen weiterraten. Damit Ub innerhalb der Funktion verändert werden kann, muss das Bytearray als global angemeldet werden.

Haben wir falsch getippt, dann wird uns ein Punkt abgezogen und der nächste Strich in die Zeichnung eingetragen. Unsere Closure **nextPart**() in **initGallows**() die wir an die Variable **gallows** zugewiesen haben, erledigt das zuverlässig. Wenn noch Striche übrig sind, gibt **gallows**() True zurück, sonst False.

```
gallows=initGalgen()
……
return gallows()
```

Mit der Closure gallows() erkennen Sie jetzt sicher den Vorteil, den uns die Funktion verschafft. Es wird gezeichnet und der Stand aktualisiert, alles automatisch mit dem stets gleichen Aufruf gallows().

Der Rest des Hauptprogramms ist schnell erklärt, neben je rund 20 trivialen Zeilen Vor- und Abspann, die sich weitgehend selbst erklären, bleibt nicht mehr viel übrig. Die Spielrunde läuft so lange, bis kein "_" mehr im Lösungs-Array übrig oder der letzte Strich des Gehängten gezeichnet ist.

```
    while ("_" in Ub) and ergebnis==True:
        zeichen=getChar()
        sleep(1)
        ergebnis=checkChar(zeichen)
```

Ach ja – die folgende Zeile in der äußeren while-Schleife entfernen Sie besser, wenn der ESP32 am PC hängt, denn der print-Befehl gibt das Suchwort im Terminal aus.

```
    print("Superword=",s)
```

Hier kommt das Programm hangman.py am Stück.

```
# File: hangman.py
# Author: Jürgen Grzesina
```

```
# Rev. 1.0
# Stand 16.06.2021
# ****************************************************
# Importgeschaeft
# ****************************************************
import os,sys       # System- und Dateianweisungen

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

import gc           # Platz fuer Variablen schaffen
gc.collect()
#
from oled import OLED
from ssd1306 import SSD1306_I2C
from machine import Pin,I2C
from keypad import KEYPAD_I2C,KEYPAD
from i2cbus import I2CBus
from mring import MAGIC_RING
from time import sleep


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

d=OLED(i2c,128,64)
keyHwadr=0x20 # HWADR des Portexpanders fuer das 4x4-Pad
kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
k=KEYPAD(kp,d=d) # hardwareunabhaengige Methoden

mr=MAGIC_RING(neoPin=12,neoCnt=12)

d.yOffset=4
A="ABCDEFGHI"
B="JKLMNOPQR"
C="STUVWXYZ "

s="JOGILOEWSOBERFLASCHENELFVEREIN"
U="_"*len(s)
sb=bytearray(s.encode("utf8"))
Ub=bytearray(U.encode("utf8"))
alphaLength=12

# *************** Funktionen declarieren ******************
def showString(q=Ub):  # q ist ein Bytearray mit den
Stringcodes
    laenge=len(q)
    zeilen=laenge//alphaLength
    rest=laenge%alphaLength
    for line in range(zeilen):
        for pos in range(alphaLength):
```

```
d.writeAt(chr(q[line*alphaLength+pos]),pos,line+3,False)
    for pos in range(rest):

d.writeAt(chr(q[zeilen*alphaLength+pos]),pos,zeilen+3,False)
    d.show()

def getChar(delay=0.3):
    global cursorPos
    d.cursor(cursorPos,3,y=0,cset=True,show=True)
    d.clearFT(len(A)+1,0,len(A)+1,2)
    while 1:
        d.cursor(cursorPos,3,y=0,cset=True,show=True)
        taste=k.waitForKey(timeout=0,ASCII=True)
        sleep(delay)
        if taste=="*":
            d.cursor(cursorPos,3,y=0,cset=False,show=True)
            cursorPos=(cursorPos-1)%9
        if taste=="+":
            d.cursor(cursorPos,3,y=0,cset=False,show=True)
            cursorPos=(cursorPos+1)%9
        if taste in "ABC":
            z=eval(taste)[cursorPos]
            d.writeAt(z,len(A)+1,ord(taste)-65)
            d.writeAt(" ",cursorPos,ord(taste)-65)
            return z
        if taste=="\x0d":
            d.clearAll()
            d.writeAt("GAME OVER",3,2)
            sys.exit()

def checkChar(c=""):
    global Ub
    global punkte
    pos=[]
    if c in s:
        p=-1
        while 1:
            p=s.find(c,p+1)
            if p!=-1:
                Ub[p]=ord(c) # gefunden und ersetzt
            else:
                break # fertig mit c
        showString(q=Ub) # anzeige updaten
        return True
    else:
        punkte-=1
        return gallows()

def initGalgen():
    n=0
    galgen=[
```

```
        "d.fill_rect(d.width-25,d.height-3,24,3,1)", # Boden
        "d.fill_rect(d.width-3,d.height//2,3,d.height//2,1)",
        "d.fill_rect(d.width-3,0,3,d.height//2,1)", # Mast
        "d.fill_rect(d.width-25,0,24,3,1)", # Balken
        "d.line(d.width-18,2,d.width-2,18,1)", # Stuetze
        "d.fill_rect(d.width-22,0,3,15,1)", # Strick
        "d.writeAt('O',13,1,False)",  # Kopf
        "d.fill_rect(d.width-22,19,3,20,1)", # Body
        "d.line(d.width-22,18,d.width-26,33,1)", # Arm links
        "d.line(d.width-19,18,d.width-14,33,1)", # Arm rechts
        "d.line(d.width-22,37,d.width-26,50,1)", # Bein links
        "d.line(d.width-19,37,d.width-14,50,1)", # Bein rechts
        ]
    def nextPart():
        nonlocal n
        exec(galgen[n])
        d.show()
        n+=1
        if n<=11:
            return True
        else:
            return False
    return nextPart

def initSuperWord():
    L=[]
    f=open("superwords.txt","r")
    for w in f:
        w = w.strip(" \r\n\t")
        L.append(w)
    f.close()
    numbers=[]
    #print(L)
    def findOne():
        nonlocal numbers
        guess = os.urandom(3)[2]%len(L)
        print(len(numbers))
        while guess in numbers and len(numbers)<len(L):
            guess = os.urandom(3)[2]%len(L)
        if len(numbers)<len(L):
            numbers.append(guess)
        return L[guess]
    return findOne

# *************** Hauptprogramm *************************
showString(Ub)
games=0
gesamt=0
getSuperWord=initSuperWord()
while 1:
    punkte=12
    d.clearAll()
```

```
    d.writeAt(A+"  A",0,0,False)
    d.writeAt(B+"  B",0,1,False)
    d.writeAt(C+"  C",0,2,True)
    cursorPos=0
    gallows=initGalgen()
    s=getSuperWord()
    print("Superword=",s)
    s=s.upper()
    U="_"*len(s)
    Ub=bytearray(U.encode("utf8"))
    showString(Ub)
    games+=1
    ergebnis=True
    while ("_" in Ub) and ergebnis==True:
        zeichen=getChar()
        print(zeichen)
        sleep(1)
        ergebnis=checkChar(zeichen)
    if ergebnis == True:
        gesamt+=punkte
        d.blinkDisplay(3,0.5,0.5)
        d.clearFT(0,0,12,2)
        d.writeAt("GEWONNEN!!!",0,0,False)
        d.writeAt("PUNKTE {}".format(punkte),0,1,False)
    else:
        d.clearFT(0,0,12,2)
        d.writeAt("NEW GAME -",0,0,False)
        d.writeAt("NEW LUCK!!",0,1,False)
        showString(q=sb)
    d.writeAt("TOTAL {}".format(gesamt),0,2)
    taste=k.waitForKey(0,ASCII=True)
    if taste=="\x0d":
        print("Game over")
        d.clearAll()
        d.writeAt("GAME OVER",3,2)
        break
```

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
keypad.py Modul für Tastenfeld-Unterstützung
mcp.py Modul für Porterweiterungsbaustein MCP23017
i2cbus.py zum Austausch verschiedener Datentypen
oled.py die API zur Ansteuerung des OLED-Moduls
ssd1306.py der Hardwaretreiber für das Display
mring.py Muster-Treiber für Neopixel-Ringe
hangman.py Hauptprogramm

Zur Installation von Thonny finden Sie hier eine ausführliche Anleitung. Darin gibt es auch eine Beschreibung wie die MicropythonFirmware auf den ESP32 gebrannt wird.

MicroPython ist eine Interpretersprache. Der Hauptunterschied zur Arduino-IDE, wo Sie stets und ausschließlich ganze Programme flashen, ist der, dass Sie die MicroPython-Firmware nur einmal zu Beginn auf den ESP32 flashen müssen, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

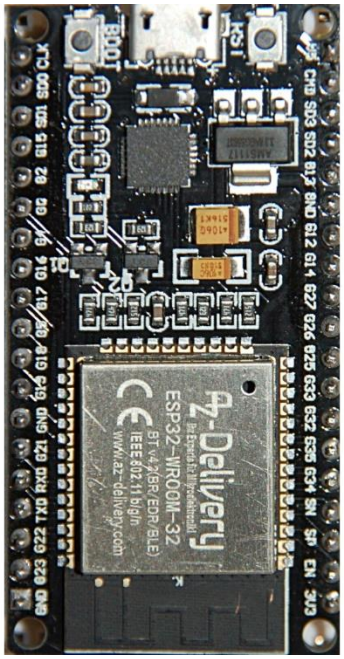
Sobald die Firmware geflasht ist, können Sie sich zwanglos mit Ihrem Controller im Zwiegespräch unterhalten, einzelne Befehle testen und sofort die Antwort sehen, ohne vorher ein ganzes Programm compilieren und übertragen zu müssen. Genau das stört mich nämlich an der Arduino-IDE. Man spart einfach enorm Zeit, wenn man einfache Tests der Syntax und der Hardware bis hin zum Ausprobieren und Verfeinern von Funktionen und ganzen Programmteilen, über die Kommandozeile vorab prüfen kann, bevor man ein Programm daraus strickt. Zu diesem Zweck erstelle ich auch gerne immer wieder kleine Testprogramme. Als eine Art Macro fassen sie wiederkehrende Befehle zusammen. Aus solchen Programmfragmenten entwickeln sich dann mitunter ganze Anwendungen. Soll das Programm autonom mit dem Einschalten des Controllers starten, kopieren Sie den Programmtext in eine neu angelegte Blankodatei. Speichern Sie diese Datei unter boot.py im Workspace ab und laden Sie sie zum ESP32/ESP8266 hoch. Beim nächsten Reset oder Einschalten startet das Programm automatisch.

Manuell gestartet werden 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**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

Sollten Sie den Controller später wieder zusammen mit der Arduino-IDE verwenden wollen, flashen Sie das Programm einfach in gewohnter Weise. Allerdings hat der ESP32/ESP8266 dann vergessen, dass er jemals MicroPython gesprochen hat. Umgekehrt kann jeder Espressif-Chip, der ein compiliertes Programm aus der Arduino-IDE oder die AT-Firmware oder LUA enthält, problemlos mit der MicroPython-Firmware versehen werden. Der Vorgang ist immer wie hier beschrieben durchzuführen.

Schaltungsaufbau

Die Schaltung für "Ring Master 2" kann ohne Änderung auch für Hangman dienen. Falls Sie statt des Batteriehalters und des Li-Akkus ein 5V-Netzteil verwenden wollen, müssen Sie die 5V an den Pin 20, Vin, des ESP32 legen. Der 3,3V-Pin des ESP32 versorgt dann den I2C-Parallelwandler für die Tastatur mit.



*

Abbildung 3: ESP32-DEVKITC_V3_Pinout

Die Versorgung aus einem 4,5V-Block aus Alkalizellen ist ebenfalls brauchbar. Der Anschluss geht auch an Pin 20 des ESP32. Falls Sie einen Neopixelring mit einplanen wollen, sollten Sie allerdings für denselben eine eigene 3,3V-Versorgung daraus ableiten. Der 3,3V-Ausgang des ESP32 schafft das nicht alles. Als Baustein bietet sich ein AMS1117 3,3V Stromversorgungsmodul für Raspberry Pi an. Für Versorgungsspannungen über 5V muss dann auch ein extra 5V-Regler verwendet werden, denn der Neopixelring darf nicht mehr als 5,3V abbekommen. Zum Experimentieren eignen sich übrigens alte PC-Netzteile sehr gut, weil sie neben 5V auch 3,3V und 12V zur Verfügung stellen. Damit können auch hungrige Stromfresser zufriedengestellt werden.

Die folgende Abbildung 4 zeigt das Schaltschema. Ein besser lesbares Exemplar in DIN A4 können Sie als PDF-Datei downloaden.

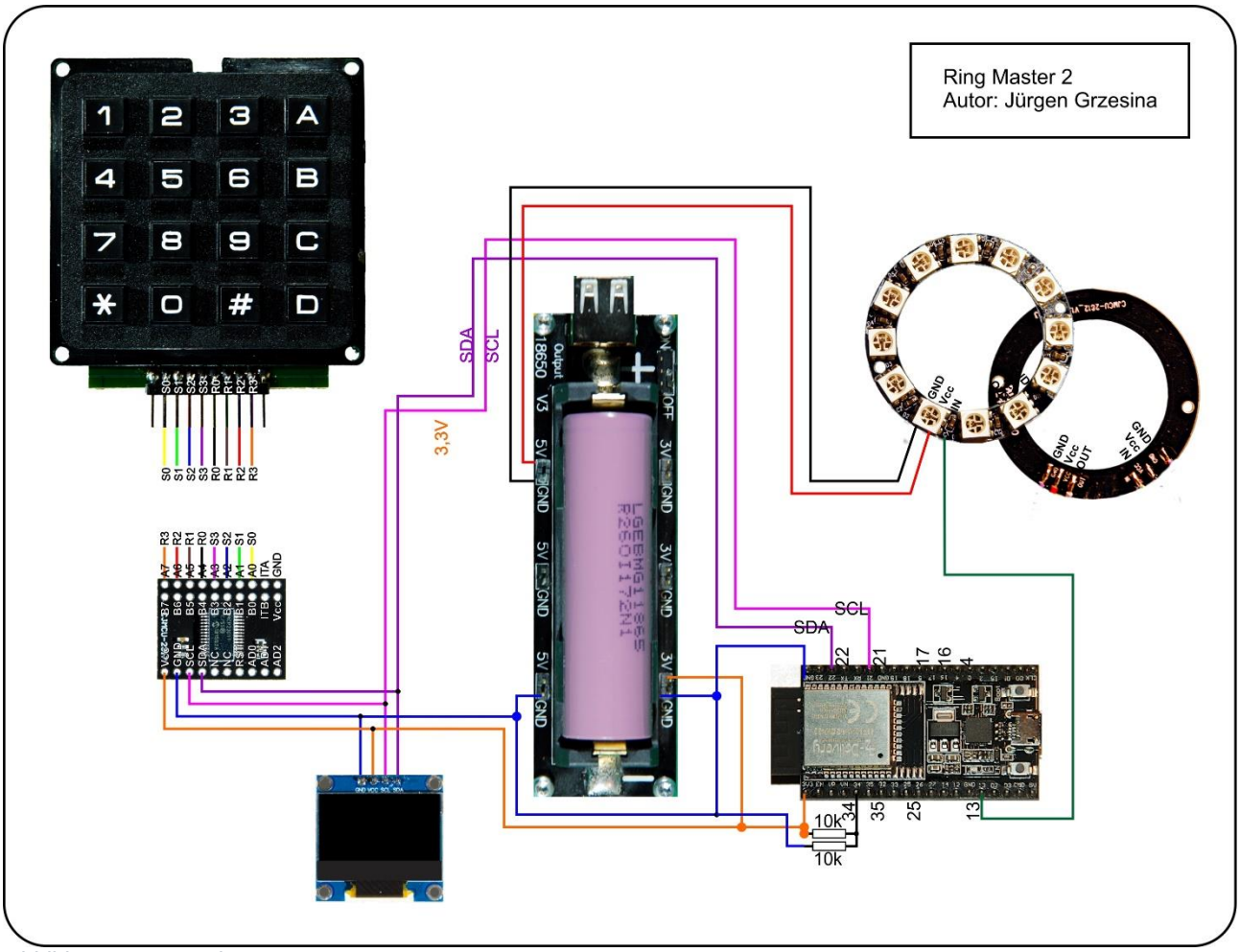


Abbildung 4: Masterring2

Die Tastatur wird so angeschlossen, dass gleichfarbige (oder gleichnamige) Leitungen mit denen vom MCP23017 zusammentreffen. Damit man das mit Jumperkabeln erledigen kann, muss die Tastaturplatine mit einer 8-poligen (gewinkelten) Stiftleiste versehen werden. Die beiden äußersten Lötpins bleiben unbeschaltet. Auch das Modul mit dem MCP23017 bekommt zwei Steckerleisten und zwar gehen die beiden äußeren Reihen mit den Stiften nach oben in Richtung Bauteilseite, die innere Reihe bekommt eine Stift- oder Buchsenleiste nach unten. Wird die Platine jetzt in ein Breadboard gesteckt, dann zeigt die beschriftete Unterseite des Boards nach oben, was die Verdrahtung deutlich erleichtert.

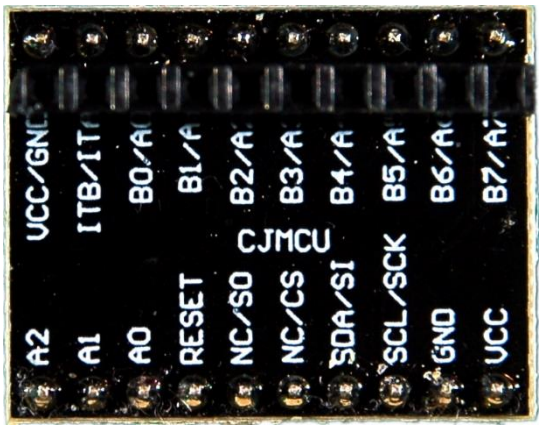


Abbildung 5: MCP23017_Unterseite

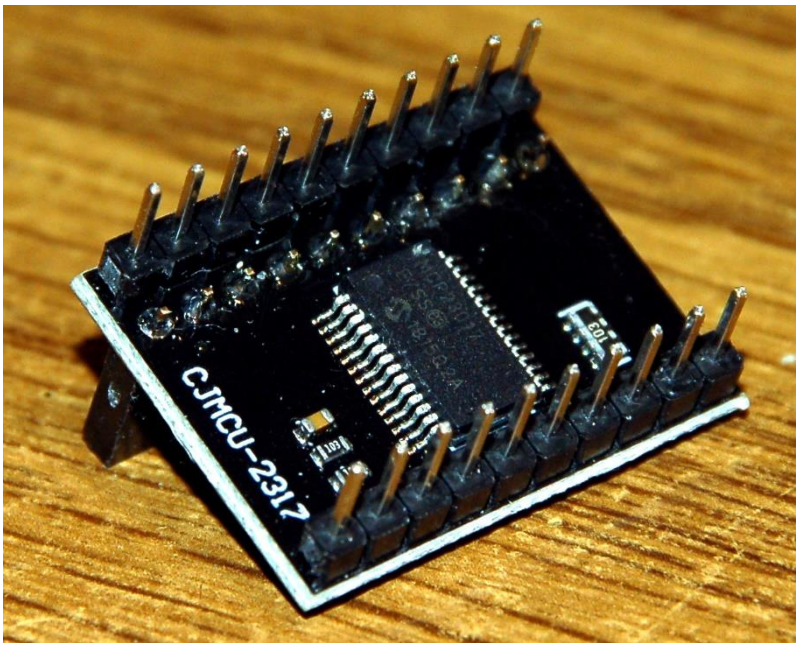


Abbildung 6: MCP23017_Bestückungsseite

Für den autonomen Betrieb ohne PC ist nur noch ein Schritt nötig. Wenn wir nämlich den Programmtext von hangman.py in die Datei boot.py verfrachten und diese wieder zum ESP32 hochschicken, startet der Controller nach einem Reset automatisch das Spiel.

Viel Vergnügen beim Bauen, Programmieren und Spielen.

Links zum Thema Spiele:

PDF in deutsch
PDF in english

Ringmaster2 Farbenraten mit Neopixelring
bandido.py der einarmige Bandit