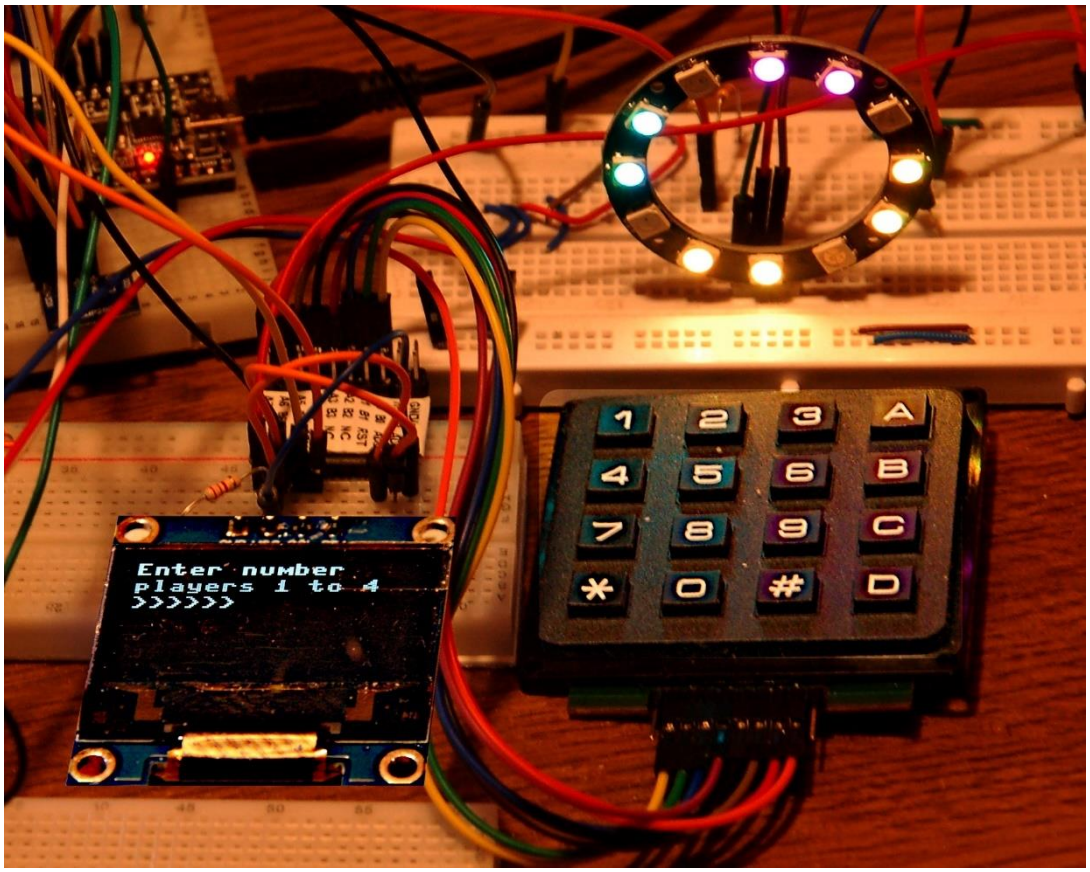


Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

Im ersten Beitrag zu dieser Reihe stellte ich ein Spiel vor, bei dem es um das Erraten von vier Farben geht. Die Magerausstattung kam mit einem zweizeiligen LCD-Keypad aus, bei dem die Anzeige und ein paar Tasten auf einem Board vereinigt waren. Mehrere Spieler verlangen mehr Platz in der Anzeigeeinheit. Bei der Auswahl für eine mehrzeilige Darstellung entschied ich mich aus Platzgründen für ein 6-zeiliges OLED-Display. Nur musste jetzt auch für die Tastatur ein Ersatz her. Meine Wahl fiel auf eine 4x4-Tastaturmatrix mit zehn Ziffern- und sechs Sondertasten. Für diesen Tastenblock hatte ich bereits bei einem anderen Projekt ein MicroPython-Modul erstellt.

Das Ziel des Spiels ist es nach wie vor, vier Farben mit möglichst wenig Durchgängen zu erraten. Da ist Strategie gefragt. Anders als beim Vorgänger ist diese Version allerdings für mehrere Spieler ausgelegt, deren Anzahl beim Start angegeben werden muss. Die maximale Spieleranzahl hängt nur vom Arbeitsspeicher des ESP32 ab und darf gerne durch TRYAL AND ERROR und "Jugend forscht" herausgefunden werden. Das Display bewältigt in dieser Ausbaustufe bis zu vier Spieler.

Der Hardwareeinsatz ähnelt sehr stark dem in den Blogs zur Anwendung von SMS und Telefonie mit dem ESP32. Wir brauchen aber keine Funkverbindungen. Auch Sensoren werden keine benötigt. Seien Sie gespannt auf den Einsatz von OLED-Display und Tastaturmatrix. Damit willkommen zu

Ring Master 2 + Codenumber
Spiele mit dem ESP32 in MicroPython

Das LCD-Keypad aus anderen Blogfolgen wird also hier durch ein OLED-Display ersetzt. Es bietet sechs Zeilen zu je 16 Zeichen und ist darüber hinaus grafikfähig mit 128 mal 64 einfarbigen Pixeln. Die Ansteuerung geschieht wie beim LCD-Keypad seriell über den I2C-Bus. Durch vorausschauende Programmierung des Treibermoduls bietet die Klasse oled.OLED dieselbe API wie die Klasse lcd.LCD. Man muss sich daher nicht an neue Befehle gewöhnen oder das Programm umschreiben, wenn das Display getauscht wird.

Die Steuerung des Spiels passiert in dieser Folge über einen 16-er-Tastaturblock. Die dahintersteckende Funktionsweise werde ich weiter unten genau besprechen.

Weil die "Notbremse" auf dem LCD-Keypad nicht mehr zur Verfügung steht, wurde die Taste A des Tastenblocks an bestimmten Stellen im Programm dafür hergenommen. Natürlich kann man auch eine einzelne normale Taste für diese Funktion hinzufügen. So eine Notbremse erfüllt bei der Programmentwicklung eine sehr nützliche Aufgabe in Strukturen wie der Hauptschleife (aka Mainloop).

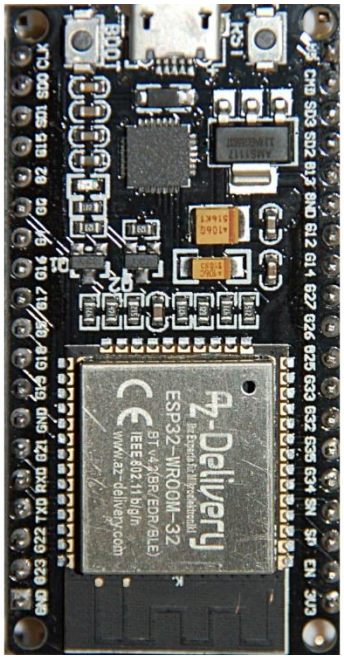
Alle, bis zum Abbruch erstellten Objekte, Variableninhalte und Funktionsdefinitionen, bleiben für den manuellen Zugriff über REPL, die MicroPython-Kommandozeile, erhalten. Auf diese Weise lassen sich zum Beispiel Funktionen und Programmteile testen, ohne jedes Mal vorher einen ganzen Rattenschwanz an Imports und Declarationen etc. neu eingeben zu müssen. Dass über die REPL-Kommandozeile solche Tests einfach durchgeführt werden können, ist ein entscheidender Vorteil der MicroPython-Umgebung gegenüber der Arduino-IDE.

Hardware

Für "Ring Master" wird ein MicroPython-Programm erstellt. Das heißt wir brauchen einen MicroPython-fähigen Controller. Die Wahl fiel auf einen ESP32, denn es soll kein großer Bildschirm wie beim Raspi, sondern nur ein OLED-Display angesteuert werden und ein Neopixel-Ring. Der ESP8266-12F scheidet wegen zu wenig RAM-Speicher und GPIO-anschlüssen aus, ihm fehlen gut 1200 Bytes. Das Display wird über einen I2C-Anschluss bedient. Der für das LCD-Keypad verwendete Seriell-Parallel-Umsetzer entfällt, weil das OLED-Display selbst über einen I2C-Adapter verfügt. Für den Neopixelring gibt es in der MicroPython-Firmware ein bereits eingebautes Modul, das die Programmierung kinderleicht macht. Zur Funktion des Rings folgen weiter unten einige Anmerkungen. Seine Stromaufnahme liegt bei ca. 20mA.

1	ESP32 NodeMCU Module WLAN WiFi Development Board mit CP2102 oder ähnlich
1	LCD1602 Display Keypad Shield HD44780 1602 Modul mit 2x16 Zeichen
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 oder ähnlich

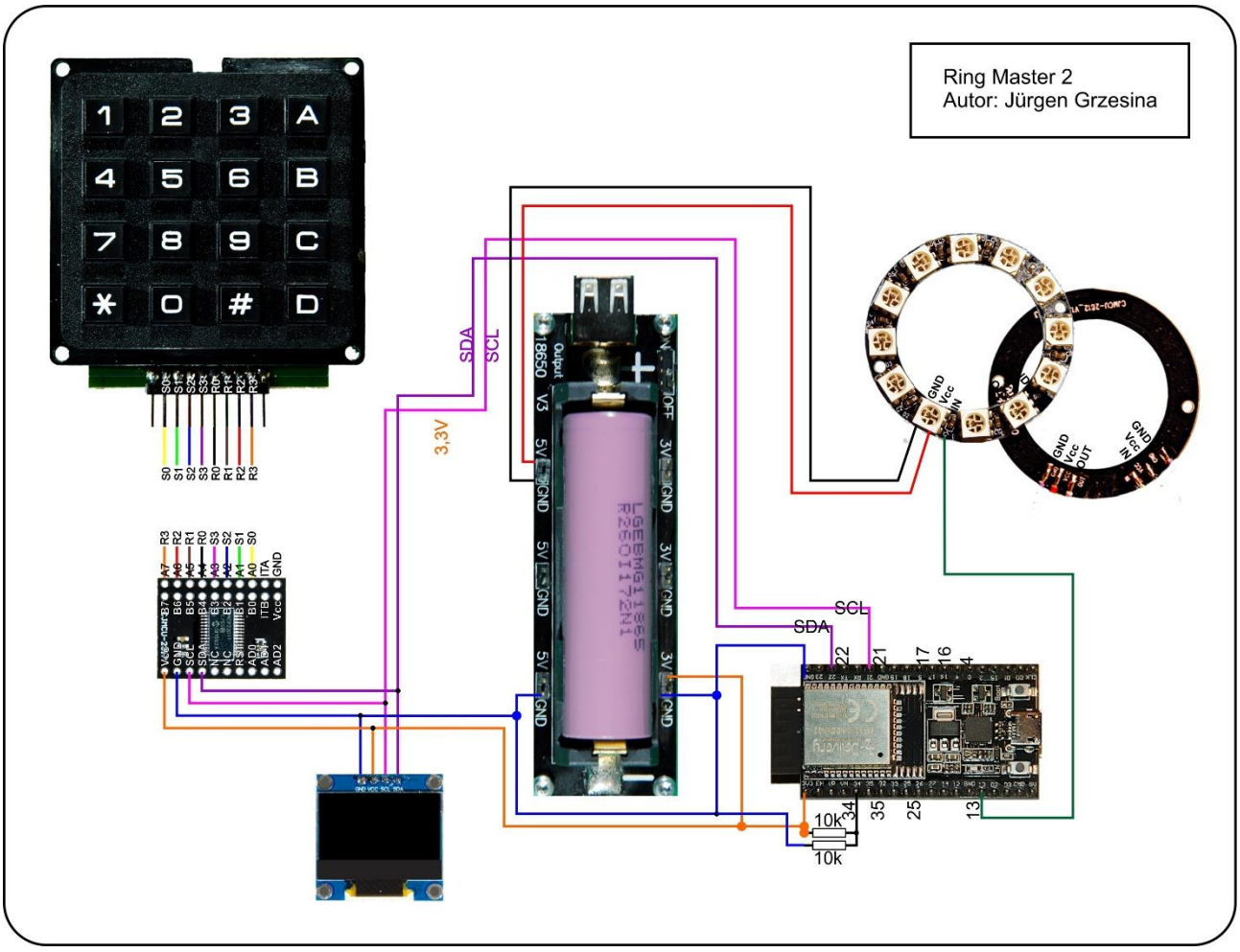
Die Schaltung für "Ring Master 2" wird zum großen Teil aus der fünften Folge des GPS-Blogs "Telefonie" http://www.grzesina.de/az/gps/teil2/gps_mcp_teil2_ger.pdfübernommen. Falls Sie statt des Batteriehalters und des Li-Akkus ein 5V-Netzteil verwenden wollen, müssen Sie die 5V an den Pin 20, Vin, des ESP32 legen. Der 3,3V-Pin des ESP32 versorgt dann den I2C-Parallelwandler für die Tastatur mit.

		CLK	1		20	Vin		
		SD0	2		21	CMD		
		SD1	3		22	SD3		
TP3	ADC2-3	G15	4		23	SD2		
TP2	ADC2-2	G2	5		24	G13	ADC2-4	TP4
TP4	ADC2-1	G0	6		25	GND		
TP0	ADC2-0	G4	7		26	G12	ADC2-5	TP5
		G16	8		27	G14	ADC2-6	TP6
		G17	9		28	G27	ADC2-7	TP7
		G5	10		29	G26	ADC2-9	
		G18	11		30	G25	ADC2-8	
		G19	12		31	G33	ADC1-5	TP8
		GND	13		32	G32	ADC1-4	TP9
		G21	14		33	G35	ADC1-7	
		RXD	15		34	G34	ADC1-6	
		TXD	16		35	SN/G39	ADC1-3	
		G22	17		36	SP/G36	ADC1-0	
		G23	18		37	EN/RST		
		GND	19		38	3V3		

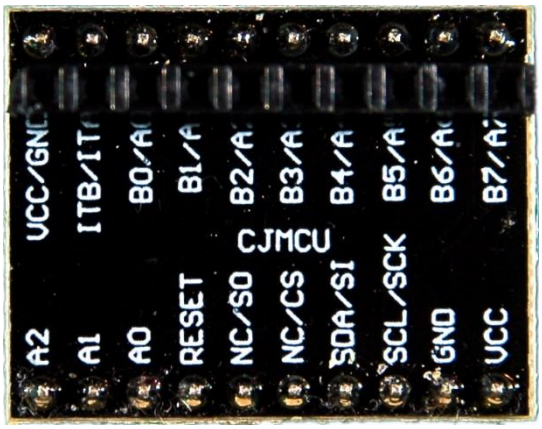
*

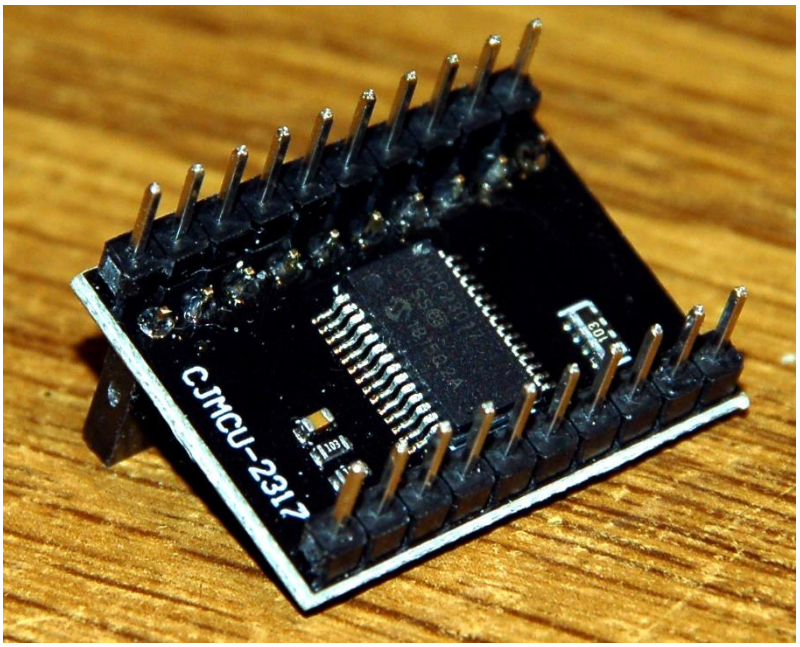
Die Versorgung aus einem 4,5V-Block aus Alkalizellen ist ebenfalls brauchbar allerdings sollten Sie dann für den Neopixelring eine eigene 3,3V-Versorgung daraus ableiten. Der 3,3V-Ausgang des ESP32 schafft das nicht alles. Als Baustein bietet sich ein AMS1117 3,3V Stromversorgungsmodul für Raspberry Pi an. Für Versorgungsspannungen über 5V muss 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 zeigt das Schaltschema. Ein besser lesbares Exemplar in DIN A4 können Sie als PDF-Datei downloaden.



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 Seite des Boards nach oben, was die Verdrahtung deutlich erleichtert.





Auf dem LED-Ring sind 12 Neopixel-LEDs vom Typ WS2812B verbaut. Die Spannungsversorgung erfolgt parallel. Die Datenleitung führt seriell von einer LED-Einheit zur nächsten und stellt eine besondere Art von Bus dar. Jede Einheit enthält eine RGB-LED und einen Controller, der auf die erste ankommende 24-Bit-Folge der Farbinformation reagiert. Die Signale, mit derselben Periodendauer aber unterschiedlichem Duty Cycle, werden von einem Microcontroller, wie dem ESP32 erzeugt. Je Neopixel-Einheit werden 24 Bit generiert (jeweils 8 für grün, rot und blau). Die Periodendauer für ein Bit ist 1,25µs +/-0,150µs, die Übertragungsfrequenz beträgt somit ca. 800kHz. Für eine 1 liegt die Leitung 0,8µs auf HIGH und 0,45µs auf LOW, eine 0 wird durch 0,4µs HIGH und 0,85µs LOW codiert. Die ersten ankommenden 24 Bits verarbeitet jede WS2812B-Einheit selbst, ohne sie weiterzugeben. Alle nun folgenden werden verstärkt und an die nächste Einheit weitergereicht. Die Signalfolge vom Microcontroller wird also von LED zu LED um 24 Bit kürzer. Anders als bei einem üblichen Datenbus erhalten die WS2812B-Einheiten die Daten aber nicht gleichzeitig, sondern zeitversetzt um jeweils die Dauer von 24Bit mal 1,25µs/Bit = 30µs.

Ein Framebuffer im RAM des ESP32 speichert die Farbwerte (3 x 256 = 16,7 Mio.) zwischen, und der Befehl NeoPixel.write() schickt die Informationen über den "Bus", der an einem GPIO-Ausgang hängt (bei uns GPIO13). Mehrere Ringe kann man genau so wie einzelne LEDs cascadieren, indem man den Eingang des nächsten Rings mit dem Ausgang des Vorgängers verbindet. Die Anschlüsse erfolgen rückseitig, am besten mittels dünner Litzen. Um die Augen zu schonen, verwende ich als Helligkeitsstufe maximal 32. Die Gesamtstromaufnahme des Rings beläuft sich dadurch im Mittel auf weniger als 20mA. Die Komponenten für die Mischfarben ermittelt man am einfachsten experimentell über REPL. Die Helligkeit der einzelnen Teil-LEDs einer Einheit ist recht unterschiedlich. Die RGB-Farbcodes in den Tupels werden also bei den Mischfarben selten den gleichen Wert haben.

```
>>> from neopixel import NeoPixel
>>> neoPin=Pin(13)
>>> neoCnt=12
>>> np=NeoPixel(neoPin,neoCnt)
>>> np[0]=(32,16,0)
>>> np.write()
```

Zum Abgleich werden die beiden letzten Befehle mit anderem RGB-Code wiederholt, bis die Farbwiedergabe passt. Die hier angegebenen Werte erzeugen gelb als Mischfarbe von rot und grün.

Bei voller Leuchtkraft saugen die LED-Einheiten 50mA pro Stück, was eine gute Konstantspannungsquelle und eine Kühlung des Rings erforderlich macht.

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
ringmaster2.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.

Tricks und Infos zu MicroPython

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, bevor 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 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.

Die im Programm verwendeten Datenstrukturen für die Farbverwaltung habe ich in dem vorangegangenen Beitrag Ringmaster 1 ausführlich erklärt. Wie die Abfrage der **Tastaturmatrix** mit Hilfe des Moduls keypad.py arbeitet, ist hier für zwei verschiedene Ansätze dargestellt. Neben den Hinweisen zu den Anschlussmöglichkeiten finden Sie dort auch die genaue Beschreibung der im Modul enthaltenen Klassen.

Nur ganz kurz zum OLED-Display. Die Klasse OLED stellt dieselbe API zur Verfügung wie die Klasse LCD und erlaubt Zeilenangaben über 0 und 1 hinaus, je nach eingesetzter Hardware. Somit kann ein OLED-Display jeder Zeit ein LCD ersetzen. Umgekehrt geht das nur reibungsfrei, wenn höchstens die Zeilen 0 und 1 im Programm verwendet werden und keine Grafikfunktionen benutzt werden.

Damit das Programm ringmaster2.py ausgeführt werden kann, müssen alle oben aufgelisteten Module in den Flashspeicher des ESP32 hochgeladen werden. Das sind die Dateien ssd1306.py, i2cbus.py, oled.py, keypad.py und mcp.py. Wenn das erledigt ist, können wir das Programm ringmaster2.py im Editorfenster mit F5 starten – sofern die Hardware zusammengebaut und der ESP32 am PC angeschlossen ist.

Hier folgt das Listing des Programms ringmaster2.py.

```
# ringmaster2.py
# Author: Juergen Grzesina
# Revision: 1.1
#    Beseitigung Score-Bug
#    Adaption an OLED und Matrixtastatur
# Stand: 11.06.2021
# *********************************************************
# Importgeschaeft
# *********************************************************
import os,sys       # System- und Dateianweisungen

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

import gc           # Platz fuer Variablen schaffen
gc.collect()
#
from machine import Pin, I2C
from neopixel import NeoPixel
from keypad import KEYPAD_I2C, KEYPAD
from i2cbus import I2CBus
from time import sleep, time, ticks_ms
#from lcd import LCD
#from hd44780u import HD44780U, PCF8574U_I2C
from oled import OLED
#from button import BUTTON32,BUTTONS
#
# ***************** Objekte declarieren ******************

i2c=I2C(-1,scl=Pin(21),sda=Pin(22),freq=400000)
ibus=I2CBus(i2c)

#disp=LCD(i2c,adr=0x27,cols=16,lines=2) # LCDPad am I2C-Bus
disp=OLED(i2c,128,64) # LCDPad am I2C-Bus

keyHwadr=0x20 # HWADR des Portexpanders fuer das 4x4-Pad
kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
```

```
# Pins fuer parallelen Anschluss des 4x4-Pads
#cols=(15,5,18,19)
#rows=(13,12,14,27)
#kp=KEYPAD_P(rows,cols) #  HW-objekt mit Parallel-Anschluss
#kp=KEYPAD_LCD(pin=35) # LCD-Keypad-Tastatur an ADC35
k=KEYPAD(kp,d=disp) # hardwareunabhaengige Methoden

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

neoPin=Pin(13)
neoCnt=12
np=NeoPixel(neoPin,neoCnt)
# Farbsortierung
r=1; g=0; b=2
palette= { # (g,r,b)
    "red":(32,0,0), # rot
    "green":(0,16,0), # gruen
    "blue":(0,0,16), # blau
    "yellow":(32,16,0), # gelb
    "magenta":(16,0,8),
    "cyan":(0,16,8),
    "white":(12,12,12),
    "black":(0,0,0)
    }

color=[ # (g,r,b)
    "red",
    "green",
    "blue",
    "yellow",
    "magenta",
    "cyan",
    "white",
    "black",
    ]
colors=len(color)
red=0; green=1; blue=2; yellow=3
magenta=4; cyan=5; white=6; black=7

kringel=[7 for i in range(neoCnt)]
ready=False

gameState=[7,7,7,7]
myState=gameState[:]
positions=len(gameState)
numberOfTrials=0

def lightKringel():
    for i in range(neoCnt):
```

```
        np[i]=palette[color[kringel[i]]]
    np.write()

def clearKringel():
    global kringel
    kringel=[7 for i in range(neoCnt)]
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
    sleep(0.03)

def clearRing():
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
    sleep(0.03)

def rainbowKringel(colList,cnt=3,delay=0.3):
    colors=len(colList)
    cols=colList[:]
    cols.extend([7 for i in range(colors,neoCnt)])
    colors=len(cols)
    global kringel
    global ready
    ready = False
    clearKringel()
    for m in range(colors):
        for n in range(m+1):
            kringel[m-n]=cols[n]
        lightKringel()
        sleep(delay)
    for m in range (cnt-1):
        for k in range(neoCnt):
            h11=kringel[neoCnt-1]
            for n in range(neoCnt-1):
                kringel[(neoCnt-1)-n]=kringel[(neoCnt-1)-n-1]
            kringel[0]=h11
            lightKringel()
            sleep(delay)
    ready=True

def dimKringel(delay=0.1,stufen=8,down=True):
    global ready
    ready=False
    for h in range(stufen+1):
        for i in range(neoCnt):
            r,g,b=palette[color[kringel[i]]]
            if down:
                col=(r*(stufen-h))//stufen
                rn=(col if h<stufen else 0)
                col=(g*(stufen-h))//stufen
                gn=(col if h<stufen else 0)
```

```
                col=(b*(stufen-h))//stufen
                bn=(col if h<stufen else 0)
            else:
                col=(r*(h))//stufen
                rn=(col if h<stufen else r)
                col=(g*(h))//stufen
                gn=(col if h<stufen else g)
                col=(b*(h))//stufen
                bn=(col if h<stufen else b)
            np[i]=(rn,gn,bn)
        np.write()
        sleep(delay)
    ready=True

def blinkKringel(on=0.3,off=0.7,cnt=1):
    c=cnt
    for i in range (c):
        lightKringel()
        sleep(on)
        clearRing()
        sleep(off)

def randomKringel():
    global kringel
    kringel=[int(i)%7 for i in os.urandom(neoCnt)]
    lightKringel()

def showStatus(stat):
    for i in range(len(stat)):
        kringel[i*3]=stat[i]
    lightKringel()

def initGame():
    clearKringel()
    edge=[int(i)%6 for i in os.urandom(4)]
    #print(edge) # uncomment for testing
    for i in range(positions): # uncomment for testing
        #np[i*3]=palette[color[edge[i]]]
        print(color[edge[i]],end="*")
        pass
    #np.write() # uncomment for testing
    #print("")
    rainbowKringel([red,yellow,green,cyan,blue,magenta],\
                   cnt=2,delay=0.03)
    dimKringel(stufen=8)
    clearKringel()
    return edge # goes to gameState

def startGame():
    # Keyblock:
    # Taste  Funktion
    # *      Position back  (n+9)%12 (n+3)%4
```

```
    # #      Position next  (n+3)%12 (n+1)%4
    # A      Abbruch
    # D      OK, set myStatus
    state=[7,7,7,7]
    clearKringel()
    getColorStatus(state)
    return state

def compareToSolution(mystat):
    global kringel
    global numberOfTrials
    numberOfTrials+=1
    reply=True
    for i in range(positions):
        kringel[3*i+1]=7
        if mystat[i]==gameState[i]:
            kringel[3*i+1]=gameState[i]
            reply=reply & True
        elif mystat[i] in gameState:
            kringel[3*i+1]=6
            reply=False
        else:
            reply=False
    lightKringel()
    return reply

def getColorStatus(myStat,delay=0.3):
    ms=myStat
    showStatus(ms)
    i=0
    w=ms[i]
    np[neoCnt-1]=palette[color[white]]
    np.write()
    disp.clearAll()
    disp.writeAt("up=2, down=0  {}".format(numberOfTrials),0,0)
    while 1:
        disp.writeAt("Position {}".format(i),0,1)
        ch=k.asciiKey()
        if ch != "\xFF":
            if ch=="*":
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[black]]
                i=(i+3)%positions # 1 Position zurueck
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[white]]
                np.write()
                w=ms[i]
                sleep(delay)
            elif ch=="+":
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[black]]
                i=(i+1)%positions # 1 Position vor
```

```
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[white]]
                np.write()
                w=ms[i]
                sleep(delay)
            elif ch=="\x0d":
                rp=(i*3+(neoCnt-1))%neoCnt
                np[rp]=palette[color[black]]
                np.write()
                disp.clearAll()
                return ms
            elif ch=="2":
                w=ms[i]
                w=(w+1)%(colors-2) # mod (colors-2) Addition
                np[i*3]=palette[color[w]]
                ms[i]=w
                np.write()
                sleep(delay)
            elif ch=="0":
                w=ms[i]
                w=(w+colors-3)%(colors-2) # mod (colors-2) Subtr.
                np[i*3]=palette[color[w]]
                ms[i]=w
                np.write()
                sleep(delay)
            elif ch=="\x08":
                print("Game Over")
                disp.clearAll()
                disp.writeAt("  GAME OVER",0,0)
                clearKringel()
                sleep(delay)
                sys.exit()
        if ctrl.value()==0:
            print("Game Over")
            disp.clearAll()
            disp.writeAt("  GAME OVER",0,0)
            sys.exit()

def play(mystat):
    ms=mystat
    showStatus(ms)
    if compareToSolution(ms):
        disp.clearAll()
        return
    print("Start:",numberOfTrials,ms)
    while 1:
        ms=getColorStatus(ms)
        showStatus(ms)
        if compareToSolution(ms):
            disp.writeAt("TRIALS: {}".format(numberOfTrials),6,0)
            sleep(1)
            disp.clearAll()
```

```
            return
        sleep(0.5)

# ********************************************************
# ******************** Hauptschleife *********************
# ********************************************************
disp.clearAll()
disp.writeAt("RINGMASTER 2",0,0)
disp.writeAt("WELCOME",0,1)
sleep(3)
disp.clearAll()
disp.writeAt("Enter number of",0,0,False)
disp.writeAt("players 1 to 4",0,1,False)
x=disp.writeAt(">>>>>> ",0,2)
nbrOfPlayers=int(k.padInput(xp=x,yp=2))
totalScore=[0 for i in range(nbrOfPlayers)]
games=[0 for i in range(nbrOfPlayers)]
player=nbrOfPlayers
playerIDs=""
for i in range(nbrOfPlayers):
    playerIDs=playerIDs+str(i)+","
playerIDs=playerIDs[:-1]+":"
while True:
    while player >= nbrOfPlayers:
        try:
            disp.clearAll()
            disp.writeAt("Enter Player",0,0,False)
            disp.writeAt("number:",0,1,False)
            disp.writeAt(playerIDs,0,2,False)
            x=disp.writeAt(">>>>>> ",0,3)
            sleep(1)
            player=int(k.padInput(xp=x, yp=3))
        except:
            player=nbrOfPlayers
    gameState=initGame()
    clearKringel()
    numberOfTrials=0
    disp.writeAt("Start now!",0,4)
    sleep(1)
    myState=startGame() #[1,0,3,1]
    play(myState)
    totalScore[player]=totalScore[player]+numberOfTrials
    games[player]+=1
    disp.clearAll()
    disp.writeAt("Player {}".format(player),0,0)
    disp.writeAt("Rounds: {}".format(games[player]),0,1)
    disp.writeAt("Total score {}".format(totalScore[player]),0,2)
    taste=k.waitForKey(0,ascii=True)
    if taste=="\x08":
        print("Game Over")
        disp.clearAll()
        disp.writeAt("  GAME OVER",0,0)
```

```
        sys.exit()
    sleep(0.8)
    player=nbrOfPlayers
    disp.clearAll()
    disp.writeAt("LOW-SCORE",0,0)
    score={i:totalScore[i] for i in range(len(totalScore))}
    sl=sorted(score.items(), key=lambda x: x[1])
    for i in range(nbrOfPlayers):
        disp.writeAt("Player{}:{};".format(sl[i][0],\
                     sl[i][1],0,1+i))
    taste=k.waitForKey(0,ascii=True)
    if taste=="\x08":
        print("Game Over")
        disp.clearAll()
        disp.writeAt("  GAME OVER",0,0)
        sys.exit()
    sleep(0.8)
```

Das Hauptprogramm fällt durch die Verlagerung der Teilaufgaben auf die diversen Funktionen und Module relativ überschaubar aus. Die Erweiterung auf mehrere Spieler machte natürlich diverse neue Programmzeilen notwendig. Für jeden Spieler wird jetzt ein Punktekonto geführt, und die Spiele werden personenbezogen gezählt. Die Erzeugung dieser Listen erfolgt nach dem Programmstart dynamisch je nach Anzahl der Mitspieler mit Hilfe einer List-Comprehension. MicroPython macht möglich, was der Arduino-IDE versagt bleibt. Nach jedem Spiel wird die Punkteliste aufsteigend sortiert. Wer die niedrigste Anzahl von Versuchen brauchte, hat die Nase vorn.

Nun ist es aber nicht damit getan, einfach die Punkteliste zu sortieren, denn es muss ja auch die Nummer des Spielers korrekt zugeordnet werden. Das passiert in den folgenden beiden Zeilen und verdient deshalb eine besondere Beachtung und Erklärung.

```
    score={i:totalScore[i] for i in range(len(totalScore))}
    sl=sorted(score.items(), key=lambda x: x[1])
```

Mit Hilfe einer Dict-Comprehension wird aus der personenbezogenen Punkteliste und dem Index automatisch ein Dictionary aufgebaut. Der Schlüsselbegriff für jeden Eintrag ist die Spielernummer, welcher der Punktestand nach dem ":" zugeordnet wird. Die sorted-Funktion erzeugt daraus eine sortierte Liste mit dem Punktestand als Sortierkriterium. score.items() liefert hierfür eine Liste aus Tupeln mit dem Index als erstem Wert und dem Punktestand als zweitem Wert. Die lambda-Funktion greift je ein Tupel x heraus und gibt als Sortierkriterium den Punktestand x[1]zurück. Das Ergebnis der sorted-Funktion ist die sortierte Liste der Tupel. Nehmen wir ein Zahlenbeispiel zu Hilfe.

```
>>> score={0:23,1:12,2:6,3:19}

>>> score.items()
dict_items([(0, 23), (1, 12), (2, 6), (3, 19)])

>>> sl=sorted(score.items(), key=lambda x: x[1])
>>> sl
[(2, 6), (1, 12), (3, 19), (0, 23)]
```

Der Rest des Programms enthält keine großen Geheimnisse. Die Funktionen haben im Vergleich zu Ringmaster 1 keine Änderung erfahren. Neu ist die Verwaltung von mehreren Spielern, die der Reihe nach zum Zug kommen. Die Anzahl der Spieler wird eingangs festgelegt, dann startet nach der Eingabe der Teilnehmernummer das Spiel.

Zunächst erzeugt **initGame**()ein neues 4-Tupel an Farben, die zu erraten sind. Alle Spielfarben marschieren ein und tanzen 3 Reigen. **startGame**() fordert zum ersten Tanz auf, will sagen zur ersten Farbwahl. Die weiße LED kennzeichnet die Eingabeposition. Das ist stets die nächste LED im Uhrzeigersinn daneben. Mit den Tasten 2 und 0 blättern Sie die Farbskala durch, mit * und # steuern Sie die nächste LED-Position OST, NORD, WEST, SÜD oder umgekehrt an. Die Auswahl wird mit D übernommen.

Mit der Funktion play() treten Sie in die heiße Spielphase ein. Nach der Überprüfung der ersten Farbwahl, die wohl in den meisten Fällen keinen sofortigen Volltreffer melden wird, werden wir zu einer weiteren Auswahl aufgefordert. Stellt die Überprüfung die Übereinstimmung der Farbfolge des **gameState** mit **myState** fest, haben wir die Farben alle richtig geortet - Volltreffer. Jede richtig erratene Farbe wird durch das Einschalten des gleichen Farbtons auf der im Uhrzeigersinn folgenden LED angezeigt. Ist die von uns gewählte Farbe in der Lösung enthalten, aber in einer anderen Himmelsrichtung zu finden, dann wird uns das durch die Farbe weiß mitgeteilt. Mit jedem D wird die Anzahl der Versuche um 1 erhöht. Dieser Wert wird im Display separat angezeigt.

Nach der Feststellung der Übereinstimmung für alle Positionen kehrt das Programm aus der Funktion play() zurück. Der Inhalt der globalen Variable numberOfTrials, die Anzahl an Versuchen, wird zu totalScore addiert. Dieser Wert und die Anzahl an Spielrunden erscheinen in der Anzeige. Nach dem Drücken einer (fast) beliebigen Taste wird die Wertungsliste angezeigt. Wieder wartet Ringmaster 2 auf eine Tastenbetätigung und startet eine neue Spielrunde. Es sei denn die Taste A wurde gedrückt, sie beendet das Programm an dieser Stelle.

Ach, ich vergaß die Erwähnung einiger Programmzeilen. In der Funktion **initGame**() gibt es eine for-Schleife, die einzig und allein zum Schummeln dient.

```
    for i in range(positions):
        #np[i*3]=palette[color[edge[i]]]
        #print(color[edge[i]],end="*")
        pass
    #np.write()
```

Während der Testphase verraten die auskommentierten Zeilen den geheimen Zahlencode für die Farbenvorlage. Danach sollten sie auskommentiert werden, sonst hat der Spaß am Spiel schnell ein Loch.

Codenumber – Erraten Sie die geheime Zahl

Ein weiteres einfaches Spiel braucht für die Eingabe von Lösungsversuchen ebenfalls wenigstens einer Zehnertastatur. Es geht um das erraten einer Geheimzahl, die das System "gewürfelt" hat. Das Spiel an sich ist vom Programm her, das sich in ein paar Zeilen hinschreiben lässt, nicht anspruchsvoll. Interessant ist aber die Antwort des ESP32, die in Form eines "magischen Auges" erfolgt. Über das Display kann die letzte Eingabe verfolgt werden, während der Neopixelring die Tendenz des Vorgangs erkennen lässt. Vom Pixel mit der Nummer 0 aus zeigen die Bögen in der linken und rechten Ringhälfte an, ob der Spieler unter oder über der zu ratenden Zahl liegt und wie nahe er dem Ziel gekommen ist. Jede LED entspricht rund 16% der Strecke zwischen der Codezahl und entweder der 0 oder der Obergrenze. Und damit es nicht zu schnell langweilig wird, ändert der ESP32 bei jedem Durchlauf neben der Codenummer auch die Obergrenze. Wie Ringmaster 2 ist Codenumber ebenfalls für mehrere Spieler ausgelegt.

Am Schaltungsaufbau zu Ringmaster 2 gibt es keine Änderungen. Auch die Programmstruktur ist vergleichbar. Die hauptsächlichen Änderungen passieren in den Funktionen initGame(), startGame(), play() und compareToSolution(). So sich an der Grundidee des Spiels nichts ändert, kann man die vorliegende Programmstruktur beibehalten und durch Anpassen der vier eben genannten Funktionen und evtl. durch Austauschen von Hardware dem Spiel ein neues Gesicht geben.

Hier kommt das Listing von Codenumber.

```
# codenumber.py
# Author: Jürgen Grzesina
# Revision: 1.0
# Stand: 07.06.2021
# ***********************************************************
# Importgeschaeft
# ***********************************************************
import os,sys       # System- und Dateianweisungen

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

import gc           # Platz fuer Variablen schaffen
gc.collect()
#
from machine import Pin, I2C
from neopixel import NeoPixel
from keypad import KEYPAD_I2C, KEYPAD
from i2cbus import I2CBus
from time import sleep, time, ticks_ms
#from lcd import LCD
#from hd44780u import HD44780U, PCF8574U_I2C
from oled import OLED
#from button import BUTTON32,BUTTONS
#
# ***************** Objekte declarieren ******************
```

```
i2c=I2C(-1,scl=Pin(21),sda=Pin(22),freq=400000)
ibus=I2CBus(i2c)

#disp=LCD(i2c,adr=0x27,cols=16,lines=2) # LCDPad am I2C-Bus
disp=OLED(i2c)
keyHwadr=0x20 # HWADR des Portexpanders fuer das 4x4-Pad
kp=KEYPAD_I2C(ibus,keyHwadr) #  Hardware Objekt am I2C-Bus
# Pins fuer parallelen Anschluss des 4x4-Pads
#cols=(15,5,18,19)
#rows=(13,12,14,27)
#kp=KEYPAD_P(rows,cols) #  HW-objekt mit Parallel-Anschluss
#kp=KEYPAD_LCD(pin=35) # LCD-Keypad-Tastatur an ADC35
k=KEYPAD(kp,d=disp) # hardwareunabhaengige Methoden

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

neoPin=Pin(13)
neoCnt=12
np=NeoPixel(neoPin,neoCnt)
# Farbsortierung
r=1; g=0; b=2
palette= { # (g,r,b)
    "red":(32,0,0), # rot
    "green":(0,16,0), # gruen
    "blue":(0,0,16), # blau
    "yellow":(32,16,0), # gelb
    "magenta":(16,0,8),
    "cyan":(0,16,8),
    "white":(12,12,12),
    "black":(0,0,0)
    }

color=[ # (g,r,b)
    "red",
    "green",
    "blue",
    "yellow",
    "magenta",
    "cyan",
    "white",
    "black",
    ]
colors=len(color)
red=0; green=1; blue=2; yellow=3
magenta=4; cyan=5; white=6; black=7

kringel=[7 for i in range(neoCnt)]

codeNumber=0
```

```
myNumber=0
numberOfTrials=0

def lightKringel():
    for i in range(neoCnt):
        np[i]=palette[color[kringel[i]]]
    np.write()

def clearKringel():
    global kringel
    kringel=[7 for i in range(neoCnt)]
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
    sleep(0.03)

def clearRing():
    for i in range(neoCnt):
        np[i]=(0,0,0)
    np.write()
    sleep(0.03)

def rainbowKringel(colList,cnt=3,delay=0.3):
    colors=len(colList)
    cols=colList[:]
    cols.extend([7 for i in range(colors,neoCnt)])
    colors=len(cols)
    global kringel
    global ready
    ready = False
    clearKringel()
    for m in range(colors):
        for n in range(m+1):
            kringel[m-n]=cols[n]
        lightKringel()
        sleep(delay)
    for m in range (cnt-1):
        for k in range(neoCnt):
            h11=kringel[neoCnt-1]
            for n in range(neoCnt-1):
                kringel[(neoCnt-1)-n]=kringel[(neoCnt-1)-n-1]
            kringel[0]=h11
            lightKringel()
            sleep(delay)
    ready=True

def faecherKringel(colList, percent, delay=0.1,hemi=3, dim=False):
    global kringel
    colors=len(colList)
    cols=colList[:]
    last=colList[-1]
    if colors < 7:
```

```
        cols.extend([last for i in range(colors,7)])
    bis=int(percent/98*6)
    clearKringel()
    for i in range(bis+1):
        kringel[0]=cols[i]
        for j in range(1,i+1):
            if hemi&1:
                kringel[j]=cols[i-j]
            if hemi&2:
                kringel[(12-j)%12]=cols[i-j]
        #np.write()
        lightKringel()
        sleep(delay)
    sleep(1)
    if percent==100:
        blinkKringel(on=0.1, off=0.2,cnt=5,remain=True)
    if dim:
        dimKringel(delay=0.05,stufen=16)

def staryNightKringel(delay=5, duration=300):
    global position
    global currentColor
    verteilung=[0 for i in range(neoCnt)]
    laufzeit=ticks_ms()
    ende=laufzeit+delay*1000
    while laufzeit<ende:
        pos=(os.urandom(1)[0])%neoCnt
        position=pos
        verteilung[pos]+=1
        col=(os.urandom(1)[0])%colors-1
        currentColor=col
        np[pos]=palette[color[col]]
        np.write()
        leuchtZeit=ticks_ms()
        ausZeit=leuchtZeit+duration
        while leuchtZeit<ausZeit:
            leuchtZeit=ticks_ms()
        np[pos]=palette["black"]
        np.write()
        laufzeit=ticks_ms()
        sleep(0.05)
    print(verteilung)

def dimKringel(delay=0.1,stufen=8,down=True):
    global ready
    ready=False
    for h in range(stufen+1):
        for i in range(neoCnt):
            r,g,b=palette[color[kringel[i]]]
            if down:
                col=(r*(stufen-h))//stufen
                rn=(col if h<stufen else 0)
```

```
                col=(g*(stufen-h))//stufen
                gn=(col if h<stufen else 0)
                col=(b*(stufen-h))//stufen
                bn=(col if h<stufen else 0)
            else:
                col=(r*(h))//stufen
                rn=(col if h<stufen else r)
                col=(g*(h))//stufen
                gn=(col if h<stufen else g)
                col=(b*(h))//stufen
                bn=(col if h<stufen else b)
            np[i]=(rn,gn,bn)
        np.write()
        sleep(delay)
    ready=True

def blinkKringel(on=0.3,off=0.7,cnt=1,remain=False):
    c=cnt
    for i in range (c):
        lightKringel()
        sleep(on)
        clearRing()
        sleep(off)
        if remain:
            lightKringel()

def randomKringel():
    global kringel
    kringel=[int(i)%7 for i in os.urandom(neoCnt)]
    lightKringel()

def showStatus(stat):
    for i in range(len(stat)):
        kringel[i*3]=stat[i]
    lightKringel()

def initGame():
    clearKringel()
    bf=os.urandom(4)
    cnbr=bf[1]<<8 | bf[0]
    obergrenze=bf[3]<<8 | bf[2]
    if cnbr>obergrenze: cnbr,obergrenze=obergrenze,cnbr
    print(cnbr,obergrenze)
    for i in range(3):
        rainbowKringel([red,yellow,green,cyan,blue,magenta,\
                        red,yellow,green,cyan,blue,magenta],\
                       cnt=1,delay=0.05*i)
    dimKringel(stufen=8)
    clearKringel()
    return (cnbr,obergrenze) # goes to codeNumber

def startGame():
```

```
    state=0
    clearKringel()
    disp.clearAll()
    disp.writeAt("RATE DIE ZAHL",0,0)
    state=int(k.padInput(yp=1))
    return state  # goes to myNumber

def compareToSolution(mynum):
    global numberOfTrials
    clearKringel()
    numberOfTrials+=1
    if mynum==codeNumber:
        faecherKringel([0,3,1,2,5],100,dim=True)
        return True
    else:
        reply=False
    if mynum>codeNumber:
        prozente=int((obergrenze-mynum)/(obergrenze-
codeNumber)*100)
        faecherKringel([0,3,1,2,5],prozente, hemi=2)
    else:
        prozente=int(mynum/codeNumber*100)
        faecherKringel([0,3,1,2,5],prozente, hemi=1)
    return reply

def play(mynum):
    ms=mynum
    if compareToSolution(ms):
        disp.clearAll()
        return
    disp.clearAll()
    x=disp.writeAt("NEW TRY:",0,1)
    while 1:
        disp.clearAll()
        disp.writeAt("LAST:{}  T{}".format(ms,numberOfTrials),0,0)
        try:
            ms=int(k.padInput(xp=x,yp=1))
            vergleich=compareToSolution(ms)
        except:
            vergleich=False
        if vergleich:
            disp.writeAt("TRIALS: {}".format(numberOfTrials),6,1)
            sleep(1)
            disp.clearAll()
            return
        sleep(0.5)
    pass

# ******************************************************
# ******************** Hauptschleife *******************
# ******************************************************
disp.clearAll()
```

```
disp.writeAt("WELCOME TO",0,0)
disp.writeAt("CODE NUMBER",0,1)
sleep(2)
disp.clearAll()
disp.writeAt("Enter Number",0,0)
x=disp.writeAt("of Players ",0,1)
try:
    nbrOfPlayers=int(k.padInput(xp=x,yp=1))
except:
    nbrOfPlayers=1
totalScore=[0 for i in range(nbrOfPlayers)]
games=[0 for i in range(nbrOfPlayers)]
player=nbrOfPlayers
playerIDs=""
for i in range(nbrOfPlayers):
    playerIDs=playerIDs+str(i)+","
playerIDs=playerIDs[:-1]+":"
while True:
    if nbrOfPlayers>1:
        while player >= nbrOfPlayers:
            try:
                disp.clearAll()
                disp.writeAt("Enter Player",0,0)
                x=disp.writeAt("Number {}".format(playerIDs),0,1)
                sleep(1)
                player=int(k.padInput(xp=x, yp=1))
            except:
                player=nbrOfPlayers
    else:
        player=0
    codeNumber,obergrenze=initGame()
    clearKringel()
    numberOfTrials=0
    disp.writeAt("Start now!",0,1)
    sleep(1)
    myNumber=startGame()
    play(myNumber)
    totalScore[player]=totalScore[player]+numberOfTrials
    games[player]+=1
    disp.clearAll()
    disp.writeAt("Playr{}
Rounds:{}".format(player,games[player]),0,0)
    disp.writeAt("Total score {}".format(totalScore[player]),0,1)
    taste=k.waitForKey(0,ascii=True)
    if taste=="\x08":
        print("Game Over")
        disp.clearAll()
        disp.writeAt("  GAME OVER",0,0)
        sys.exit()
    sleep(0.8)
    player=nbrOfPlayers
    disp.clearAll()
```

```
    disp.writeAt("LOW-SCORE",0,0)
    score={i:totalScore[i] for i in range(len(totalScore))}
    sl=sorted(score.items(), key=lambda x: x[1])
    scores=""
    for i in range(nbrOfPlayers):
        scores=scores+"P{}:{};".format(sl[i][0],sl[i][1])
    disp.writeAt(scores,0,1)
    taste=k.waitForKey(0,ascii=True)
    if taste=="\x08":
        print("Game Over")
        disp.clearAll()
        disp.writeAt("  GAME OVER",0,0)
        sys.exit()
    sleep(0.8)
```

Die Funktion getColor() werden Sie bei Codenumber vergeblich suchen, die habe ich nämlich entfernt, weil sie nur für Ringmaster eine Bedeutung hatte. Auf eine ganz unscheinbare Stelle in der Funktion initGame() will ich Sie noch hinweisen.

```
>>> bf=os.urandom(4)
>>> cnbr=bf[1]<<8 | bf[0]
>>> obergrenze=bf[3]<<8 | bf[2]
>>> if cnbr>obergrenze: cnbr,obergrenze=obergrenze,cnbr
```

bf bekommt von urandom ein Bytesobjekt zugewiesen. Aus jeweils 2 Bytes wird eine Integerwert im Bereich zwischen 0 und 65535 incl. berechnet. Und jetzt kommt's, falls cnbr größer als obergrenze sein sollte, muss getauscht werden.

```
cnbr,obergrenze=obergrenze,cnbr
```

Jetzt probieren Sie das einmal mit der Arduino-IDE, ganz zu schweigen davon, dass das interaktiv gar nicht möglich ist. Zum Testen brauchen Sie mindestens ein Programm. - Ich liebe MicroPython, weil es solche Dinge ganz einfach macht!

Vorschau:
Der Park an Funktionen zur Steuerung des Neopixelrings ist inzwischen so weit angewachsen, dass es sich lohnt, daraus eine Klasse zu bauen. Genau das machen wir in der nächsten Folge. Als Anwendung programmieren wir ein Modell für einen von den Typen, die in den Cassinos von Las Vegas den Leuten das Geld aus der Tasche ziehen. Die Rede ist von "einarmigen Banditen".

Bis dann, viel Vergnügen beim Basteln, Programmieren und Spielen!

Nützliche Links:
PDF in deutsch
PDF in english
Wie arbeitet die Abfrage einer Tastaturmatrix?
Farben-Raten mit Ringmaster1
Thonny – Installation und Einführung