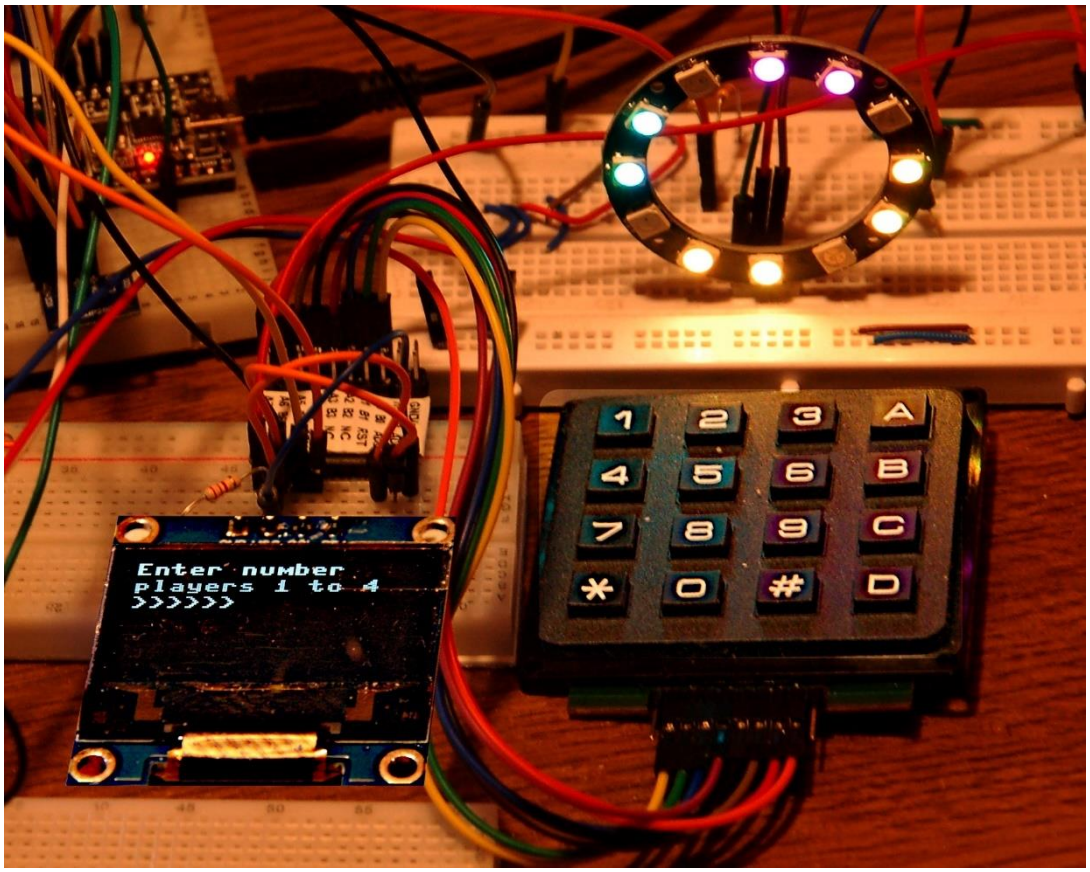


Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

In the first post in this series, I introduced a game that involves guessing four colors. The lean equipment got by with a two-line LCD keypad, in which the display and a few buttons were combined on one board. Several players require more space in the display unit. When choosing a multi-line display, I opted for a 6-line OLED display for reasons of space. But now a replacement also had to be found for the keyboard. My choice fell on a 4x4 keyboard matrix with ten digit and six special keys. I had already created a MicroPython module for this keypad in another project.

The object of the game is still to guess four colors with as few passes as possible. Strategy is required. Unlike its predecessor, this version is designed for several players, the number of which must be specified at the start. The maximum number of players depends only on the memory of the ESP32 and can be found out with TRYAL AND ERROR and "Jugend forscht". The display can handle up to four players in this expansion stage.

The use of hardware is very similar to that in the blogs about the use of SMS and telephony with the ESP32. But we don't need any radio links. Sensors are also not required. Look forward to the use of the OLED display and keyboard matrix. So welcome to

Ring Master 2 + Codenumber
Games with the ESP32 in MicroPython

The LCD keypad from other blog episodes is replaced here by an OLED display. It offers six lines of 16 characters each and is also capable of graphics with 128 by 64 monochrome pixels. As with the LCD keypad, it is controlled serially via the I2C bus. By proactively programming the driver module, the oled.OLED class offers the same API as the lcd.LCD class. You don't have to get used to new commands or rewrite the program when the display is swapped.

In this episode, the game is controlled using a 16-key keypad. I will discuss the underlying functionality in detail below.

Because the "emergency brake" is no longer available on the LCD keypad, key A on the keypad was used at certain points in the program. Of course, you can also add a single normal key for this function. Such an emergency brake fulfills a very useful task in program development in structures such as the main loop.

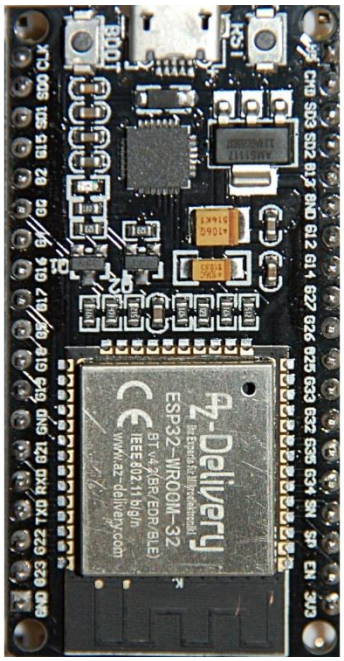
All objects, variable contents and function definitions created up to the point of cancellation are retained for manual access via REPL, the MicroPython command line. In this way, for example, functions and program parts can be tested without having to re-enter a whole series of imports and declarations, etc. each time. The fact that such tests can be carried out easily via the REPL command line is a decisive advantage of the MicroPython environment compared to the Arduino IDE.

Hardware

A MicroPython program is created for "Ring Master". That means we need a MicroPython-capable controller. The choice fell on an ESP32, because it should not be a large screen like the Raspi, but only an OLED display and a neopixel ring. The ESP8266-12F is eliminated because of insufficient RAM memory and GPIO connections, it lacks a good 1200 bytes. The display is operated via an I2C connection. The serial-parallel converter used for the LCD keypad is not required because the OLED display itself has an I2C adapter. There is a built-in module in the MicroPython firmware for the neopixel ring, which makes programming child's play. A few comments on the function of the ring follow below. Its current consumption is around 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

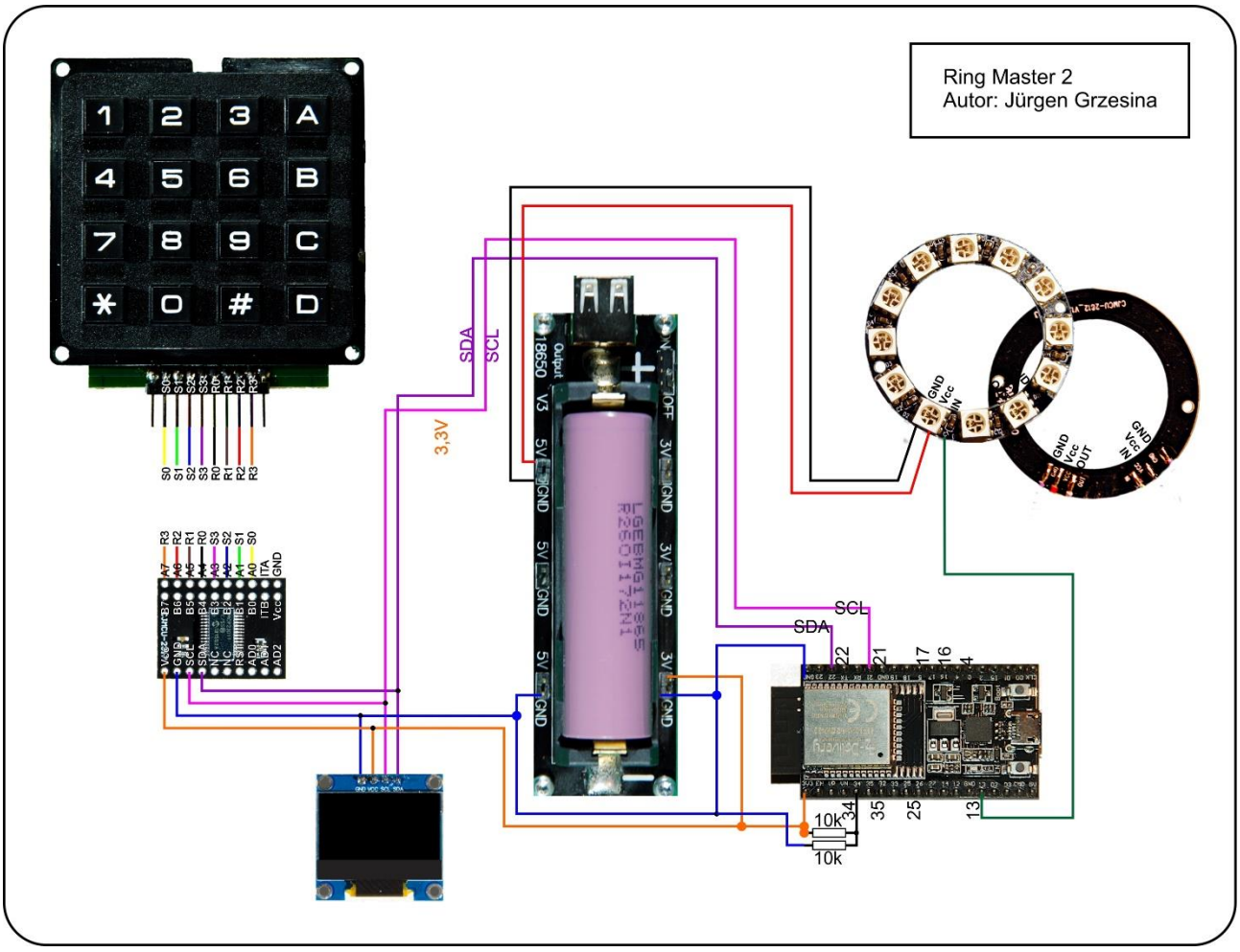
The circuit for "Ring Master 2" is largely taken from the fifth episode of the GPS blog "Telephony". If you want to use a 5V power supply instead of the battery holder and the Li battery, you have to connect the 5V to pin 20, Vin, of the ESP32. The 3.3V pin of the ESP32 then supplies the I2C parallel converter for the keyboard



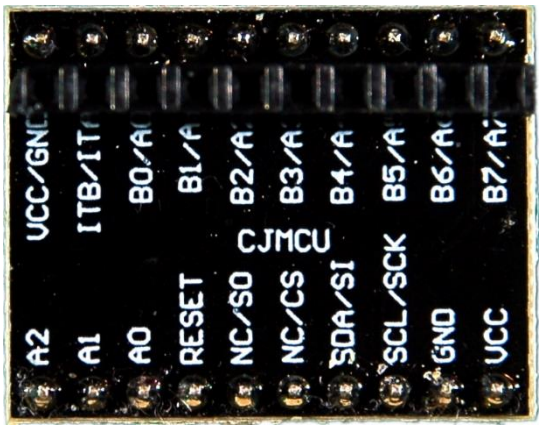
*

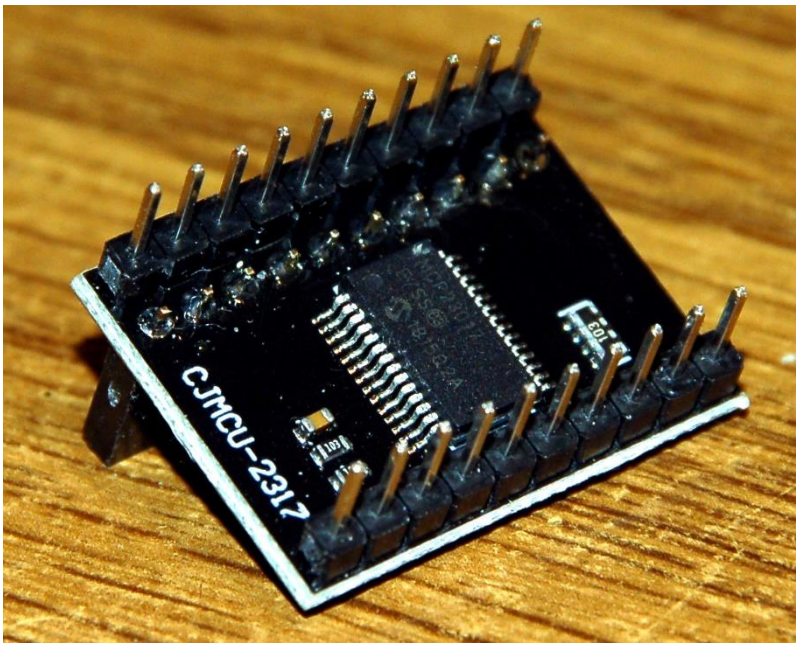
The supply from a 4.5V block of alkaline cells can also be used, but you should then derive your own 3.3V supply for the neopixel ring. The 3.3V output of the ESP32 doesn't do all of that. An AMS1117 3.3V power supply module for Raspberry Pi can be used as a component. An extra 5V regulator must be used for supply voltages above 5V, because the neopixel ring must not receive more than 5.3V. By the way, old PC power supplies are very suitable for experimenting because they provide 3.3V and 12V in addition to 5V. This means that even hungry power guzzlers can be satisfied.

The following figure shows the circuit diagram. You can download a more readable copy in DIN A4 as a PDF file.



The keyboard is connected in such a way that lines of the same color (or lines of the same name) meet those of the MCP23017. So that you can do this with jumper cables, the keyboard board must be provided with an 8-pin (angled) pin header. The two outermost solder pins remain unconnected. The module with the MCP23017 also has two connector strips, namely the two outer rows with the pins up towards the component side, the inner row has a pin or socket strip downwards. If the circuit board is now plugged into a breadboard, the labeled side of the board points upwards, which makes wiring much easier.





12 Neopixel LEDs of type WS2812B are installed on the LED ring. Power is supplied in parallel. The data line runs serially from one LED unit to the next and represents a special type of bus. Each unit contains an RGB LED and a controller that reacts to the first incoming 24-bit sequence of color information. The signals, with the same period but different duty cycles, are generated by a microcontroller such as the ESP32. 24 bits are generated for each neopixel unit (8 each for green, red and blue). The period for one bit is 1.25µs +/- 0.150µs, the transmission frequency is thus approx. 800kHz. For a 1 the line is 0.8µs on HIGH and 0.45µs on LOW, a 0 is coded with 0.4µs HIGH and 0.85µs LOW. The first incoming 24 bits are processed by each WS2812B unit without passing them on. All those who follow are reinforced and passed on to the next unit. The signal sequence from the microcontroller is therefore 24 bits shorter from LED to LED. In contrast to a conventional data bus, the WS2812B units do not receive the data at the same time, but with a time delay of 24 bits times 1.25 µs / bit = 30 µs.

A frame buffer in the RAM of the ESP32 temporarily stores the color values (3 x 256 = 16.7 million), and the NeoPixel.write () command sends the information over the "bus" that is attached to a GPIO output (in our case GPIO13). Several rings can be cascaded just like individual LEDs by connecting the input of the next ring to the output of the previous one. The connections are made on the back, ideally using thin strands. To protect the eyes, I use a maximum brightness level of 32. The total current consumption of the ring is less than 20mA on average. The easiest way to determine the components for the mixed colors is by experiment using REPL. The brightness of the individual sub-LEDs of a unit is quite different. The RGB color codes in the tuples will therefore rarely have the same value for the mixed colors.

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

For comparison, the last two commands are repeated with a different RGB code until the color rendering is correct. The values given here produce yellow as a mixed color of red and green.

At full luminosity, the LED units suck 50mA each, which requires a good constant voltage source and cooling of the ring.

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

You can find detailed instructions for installing Thonny here. There is also a description of how the Micropython firmware is burned to the ESP32.

Tricks and Infos on MicroPython

MicroPython is an interpreter language. The main difference to the Arduino IDE, where you always and exclusively flash entire programs, is that you only have to flash the MicroPython firmware once at the beginning on the ESP32 before the controller understands MicroPython instructions. You can use Thonny, µPyCraft or esptool.py for this. I have described the process for Thonny here.

As soon as the firmware is flashed, you can have a casual conversation with your controller, test individual commands and immediately see the answer without first having to compile and transfer an entire program. This is exactly what bothers me about the Arduino IDE. You simply save an enormous amount of time if you can do simple tests of the syntax and hardware through to trying out and refining functions and entire program parts via the command line before you knit a program out of it. For this purpose I also like to create small test programs over and over again. As a kind of macro, they combine recurring commands. From such program fragments, entire applications can develop. If the program is to start autonomously when the controller is switched on, copy the program text into a newly created blank file. Save this file as boot.py in the workspace and upload it to the ESP32. The program starts automatically the next time it is reset or switched on.

Programs are started manually from the current editor window in the Thonny IDE using the F5 key. This is faster than clicking the start button or using the Run menu. Only the modules used in the program must be in the flash of the ESP32.

If you want to use the controller again later together with the Arduino IDE, simply flash the program in the usual way. However, the ESP32 / ESP8266 then forgot that it ever spoke MicroPython. Conversely, any Espressif chip that contains a compiled program from the Arduino IDE or the AT firmware or LUA can easily be provided with the MicroPython firmware. The process is always as described here.

I explained the data structures used in the program for color management in detail in the previous article, Ringmaster 1. How the query of the keyboard matrix works with the help of the keypad.py module is shown here for two different approaches. In addition to the information on the connection options, you will also find a detailed description of the classes contained in the module.

Just briefly about the OLED display. The OLED class provides the same API as the LCD class and allows line specifications beyond 0 and 1, depending on the hardware used. This means that an OLED display can replace an LCD at any time. The reverse is only possible if lines 0 and 1 are used in the program and no graphic functions are used.

So that the ringmaster2.py program can be executed, all modules listed above must be uploaded to the ESP32's flash memory. These are the files ssd1306.py, i2cbus.py, oled.py, keypad.py and mcp.py. When this is done, we can start the program ringmaster2.py in the editor window with F5 - provided the hardware is assembled and the ESP32 is connected to the PC.

Here follows the listing of the program 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)
```

The main program is relatively manageable due to the relocation of the subtasks to the various functions and modules. The expansion to several players naturally made various new lines of program necessary. A points account is now kept for each player and the games are counted individually. These lists are generated dynamically after the program has started, depending on the number of players with the help of a list comprehension. MicroPython makes possible what the Arduino IDE fails to do. After each game, the list of points is sorted in ascending order. Whoever needed the lowest number of attempts is ahead of the game.

But it is not enough to simply sort the list of points, because the player's number has to be assigned correctly. This happens in the following two lines and therefore deserves special attention and explanation.

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

With the help of a dict comprehension, a dictionary is automatically built from the personal point list and the index. The key concept for each entry is the player number to which the score is assigned after the ":". The sorted function uses this to generate a sorted list with the score as a sorting criterion. score.items () returns a list of tuples with the index as the first value and the score as the second value. The lambda function picks out one tuple x and returns the score x [1] as a sorting criterion. The result of the sorted function is the sorted list of tuples. Let us use a numerical example as an aid.

```
>>> 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)]
```

The rest of the program doesn't hold any great secrets. The functions have not changed in comparison to Ringmaster 1. What is new is the management of several players who take turns in turn. The number of players is determined at the beginning, then the game starts after entering the participant number.

InitGame () first creates a new 4-tuple of colors that are to be guessed. All colors of the game march in and dance 3 rounds. startGame () prompts for the first dance, that is to say, for the first choice of color. The white LED indicates the input position. This is always the next LED next to it in a clockwise direction. Use the 2 and 0 buttons to scroll through the color scale, and use * and # to move to the next LED position EAST, NORTH, WEST, SOUTH or vice versa. The selection is accepted with D.

With the play () function you enter the hot phase of the game. After checking the first color choice, which in most cases will probably not report an immediate hit, we are asked to make another selection. If the check determines that the color sequence of the gameState matches myState, we have all located the colors correctly - a direct hit. Each correctly guessed color is indicated by switching on the same color on the LED following clockwise. If the color we have chosen is contained in the solution, but can be found in a different direction, then this is communicated to us by the color white. Each D increases the number of attempts by 1. This value is shown separately in the display.

After finding the agreement for all positions, the program returns from the play () function. The content of the global variable numberOfTrials, the number of attempts, is added to totalScore. This value and the number of game rounds appear in the display. After pressing (almost) any key, the rating list is displayed. Ringmaster 2 waits again for a button to be pressed and starts a new round of the game. Unless key A has been pressed, it terminates the program at this point.

Oh, I forgot to mention a few lines of the program. There is a for loop in the initGame () function, which is only used for cheating.

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

During the test phase, the commented out lines reveal the secret numerical code for the color template. After that, they should be commented out, otherwise the fun of the game quickly becomes dull.

Codenumber – Guess the secret number

Another simple game also needs at least a numeric keypad for entering solution attempts. It's about guessing a secret number that the system "rolled". The game itself is not demanding in terms of the program, which can be written down in a few lines. What is interesting, however, is the response from the ESP32, which takes the form of a "magic eye". The last entry can be followed on the display, while the neopixel ring shows the tendency of the process. From the pixel with the number 0, the arcs in the left and right halves of the ring show whether the player is below or above the number to be guessed and how close he has come to the goal. Each LED corresponds to around 16% of the distance between the code number and either the 0 or the upper limit. And so that it doesn't get boring too quickly, the ESP32 changes both the code number and the upper limit with each run. Like Ringmaster 2, Codenumber is also designed for multiple players.

There are no changes to the circuit structure for Ringmaster 2. The program structure is also comparable. The main changes happen in the functions initGame (), startGame (), play () and compareToSolution (). If nothing changes in the basic idea of the game, you can keep the existing program structure and give the game a new face by adapting the four functions just mentioned and possibly by exchanging hardware.

Here comes this 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)
```

You will look in vain for the getColor () function in Codenumber, because I removed it because it was only relevant for ring masters. I would like to draw your attention to a very inconspicuous place in the initGame () function.

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

urandom assigns a byte object to bf. An integer value in the range between 0 and 65535 incl. Is calculated from 2 bytes each. And now it comes, if cnbr should be greater than the upper limit, it has to be exchanged.

cnbr, upper limit = upper limit, cnbr

Now try that with the Arduino IDE, not to mention that it is not possible to do it interactively. You need at least one program to test it. - I love MicroPython because it makes things like this very easy!

Preview:
The park of functions for controlling the neopixel ring has grown so much that it is worth building a class out of it. That's exactly what we'll do in the next episode. As an application, we're programming a model for one of the guys who pull the money out of people's pockets in the Cassinos in Las Vegas. We are talking about "one-armed bandits".

Until then, have fun tinkering, programming and playing!

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