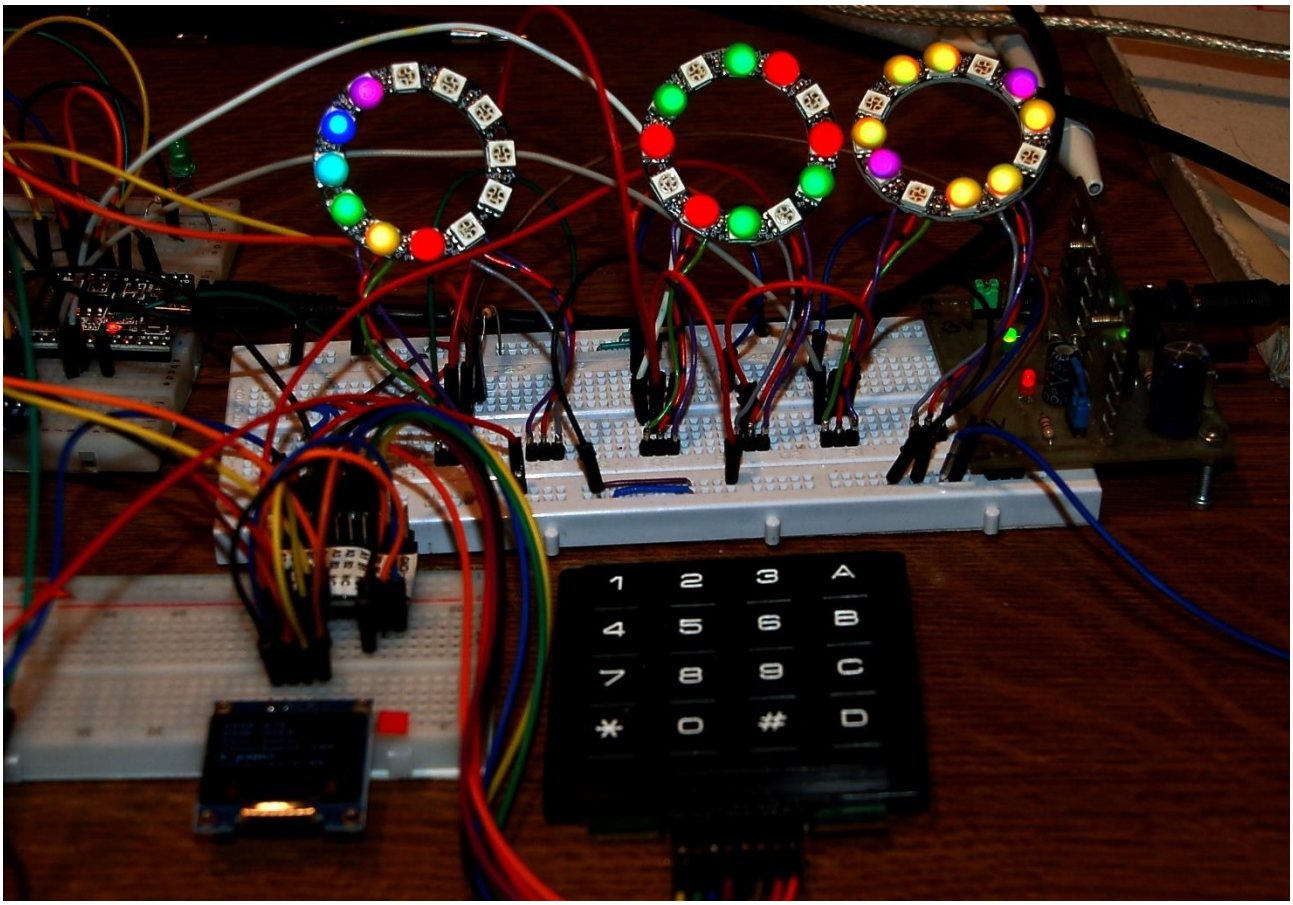


*Abbildung 1: Bandit-Aufbau*

Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

The games I've featured so far have been about strategy and reasoning. Of course, you can just try it out until you are exhausted. Master chance reigns in this game, depending on the programming more or less. Nevertheless, the use of hardware is again very similar to that in the blogs for Ringmaster 1 and 2. So welcome to the presentation of the game

# Bandit –
# Games on the ESP32 in MicroPython

The OLED display from the previous blog series Ringmaster 2 is used again, as is the 4x4 key matrix. The first neopixel ring has two brothers. The 6-line OLED display required more output information. The 16-key field replaces the numeric keyboard of the PC in the field of the game setup.

What is this game about now? Perhaps you know the kind of slot machines as they are in the casinos of Las Vegas or in German gambling dens? What are meant are things with three or more rollers that can be set in motion by means of a lever and then either run down themselves or can be stopped by pressing a button. Today we want to model the model for such a "one-armed bandit", as they are called, with the ESP32 under

MicroPython. The program will run in an endless loop. As usual, I built in an emergency brake so that you can specifically leave it during the development phase.

Using the emergency brake means terminating the running program precisely without an immediate restart, which would be done by pressing the RST button on the ESP32 if the program was started under the name boot.py. Exactly the latter is necessary if the program is to start autonomously, i.e. without a connected PC. In the test phase, cancellation is often only possible with an emergency button.

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 interactively without having to re-enter a whole series of imports and declarations, etc. each time, the previously started program does that for us. The fact that such tests can be carried out easily via the REPL command line is a decisive advantage of the MicroPython environment.

# Hardware

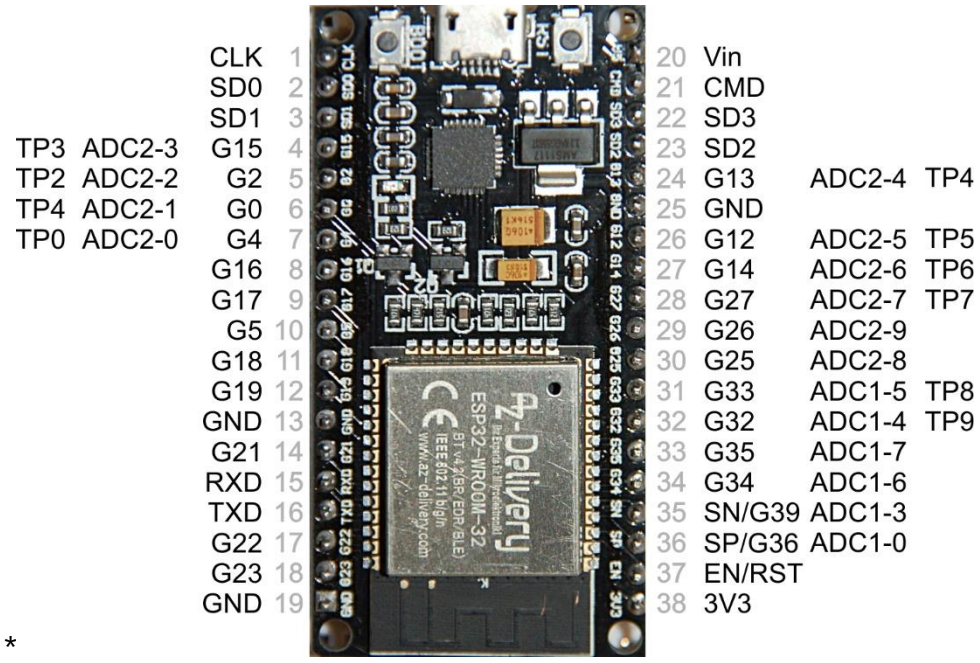
A MicroPython program is created for "Bandit". That means we need a MicroPython-capable controller. The choice fell on an ESP32, because it should not be a large screen like the Raspi, but only an OLED display. The ESP8266-12F was eliminated due to insufficient RAM memory, it lacks a good 1200 bytes. But not only that, the ESP8266 decided to have too few GPIO connections for this use.

A 4x4 matrix keypad is used as the keyboard. The OLED display is operated via the I2C bus, which also supplies the keyboard connection. There is a built-in module in the MicroPython firmware for the neopixel ring, which makes programming child's play. Below are a few comments on how the ring works. Its current consumption is around 20mA.

| | |
|---|---|
| 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 |
| 3 | LED Ring 5V RGB WS2812B 12-Bit 37mm oder ähnlich |
| 3 | KY-004 Taster Modul Sensor Taste Kopf Schalter oder<br>ein TTP224 4-Kanal Digitaler Berührungssensor Kapazitiver Touch Modul |

The circuit for "Bandit" is basically taken from the episode "Ringmaster 2". Three neopixel rings are required for the "play rollers". Rolling is simulated by eight different LED patterns.

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 keyboard's I2C parallel converter, but cannot feed the three neopixel rings. In order to provide their power requirements, a separate 3.3V voltage source is required, which you can derive from the 5V using a controller module. This regulator is not included in the parts list above




*

*Abbildung 2: ESP32-DEVKITC_V3_Pinout*

The supply from a 4.5V block of alkaline cells would be sufficient for the controller board, but unfortunately the display is not satisfied with that. An extra 5V regulator must be used for supply voltages above 5V, because the neopixel ring must not receive more than 5.3V.

Tip:
Old PC power supplies are very suitable for experimenting because they provide 3.3V and 12V in addition to 5V.

The following figure shows the circuit diagram. You can download a more readable copy in DIN A4 as a PDF file. The three buttons can possibly be replaced by a capacitive touch module TTP224 4-channel digital touch sensor with 4 pads, which is only suitable for narrow fingers

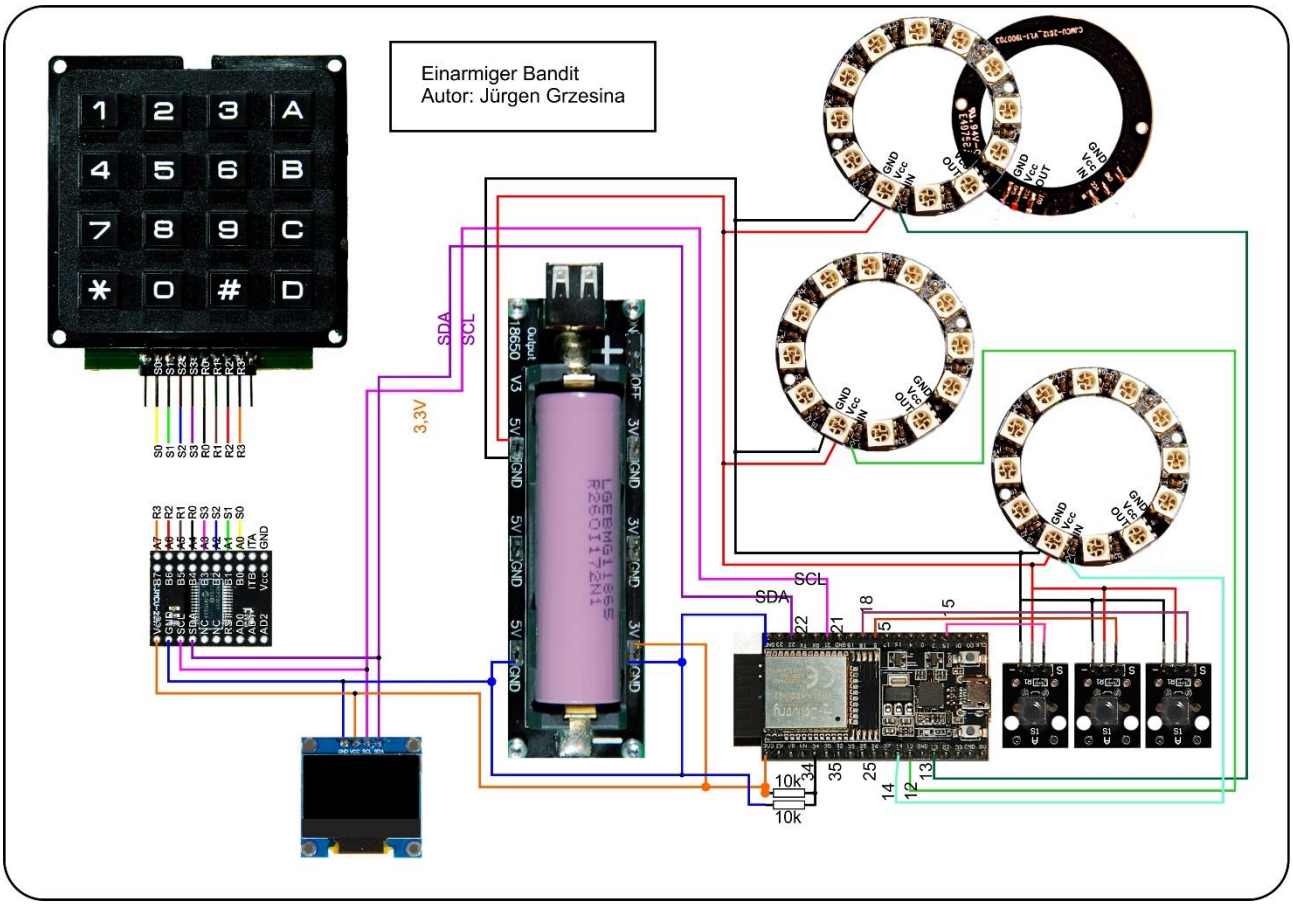


*Abbildung 3: Bandit_Schaltung*

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 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 duration 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, all of the following bits are amplified 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 simultaneously, 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 ). The sequence of colors on the MicroPython module is red-green-blue, the sequence on the bus is green - red - blue. Several rings can be cascaded just like individual LEDs by connecting the input of the next ring to the output of the previous one. However, we will operate the three rings on three different GPIOs because the addressing functions that already exist make this easier.

The connections on the ring are made at the rear, preferably using thin strands. In order 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 yourself is by experiment using REPL. The brightness of the individual color channels is quite different. The codes in the tuples for mixed colors will therefore rarely have the same value.

```
>>> 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".

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


*Abbildung 4: LED-Ring_vorn*

# The Software

**Used Software:**
For flashing and programming the ESP32:
Thonny oder
µPyCraft

**Used Firmware:**
MicropythonFirmware

**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
bandido.py Hauptprogramm

# 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 on the ESP32 at the beginning so that 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 under boot.py in the workspace and upload it to the ESP32 / ESP8266. 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 later want to use the controller together with the Arduino IDE again, 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 must always be carried out as described here.

I explained the data structures used in the program for color management in detail in the Ringmaster 1 article. 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.

In order for the bandido.py program to 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, mring.py and mcp.py. When that is done, we can start the program bandido.py in the editor window with F5 - provided the hardware is assembled and the ESP32 is connected to the PC.

Here is the listing of the program

```
# bandido.py
# Author: Juergen Grzesina
# Revision: 1.0
# Stand: 08.06.2021
# ********************************************************
# 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 mring import MAGIC_RING
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 ******************
# Pins fuer parallelen Anschluss des 4x4-Pads

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

```
leftPinNbr=13
midPinNbr=12
rightPinNbr=14
leftStopNbr=15
midStopNbr=5
rightStopNbr=18
leftStop=Pin(leftStopNbr,Pin.IN,Pin.PULL_UP) #,Pin.PULL_UP
midStop=Pin(midStopNbr,Pin.IN,Pin.PULL_UP)
rightStop=Pin(rightStopNbr,Pin.IN,Pin.PULL_UP)
lPos=0; mPos=0; rPos=0

neoCnt=12
left=MAGIC_RING(leftPinNbr,neoCnt)
mid=MAGIC_RING(midPinNbr,neoCnt)
right=MAGIC_RING(rightPinNbr,neoCnt)
l_stoped,m_stoped,r_stoped=False,False,False

games=0

palette= {
    "red":(32,0,0),
    "green":(0,16,0),
    "blue":(0,0,16),
    "yellow":(32,16,0),
    "magenta":(16,0,8),
    "cyan":(0,16,8),
    "white":(12,12,12),
    "black":(0,0,0)
    }

color=[
    "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

muster=[
    [blue,yellow,black,blue,yellow,black,blue,yellow,\
     black,blue,yellow,black],
    [green,red,black,red,green,black,green,red,\
     black,red,green,black],
    [yellow,black,magenta,yellow,black,yellow,yellow,black,\
     magenta,yellow,black,yellow],
```

```
    [red,red,red,black,black,black,blue,blue,blue,\
     blue,black,black,black],
    [red,black,yellow,black,green,black,red,black,\
     yellow,black,green,black],
    [red,yellow,green,blue,black,black,red,yellow,\
     green,blue,black,black],
    [red,yellow,green,cyan,blue,magenta,black,black,\
     black,black,black,black],
    [red,yellow,green,cyan,blue,magenta,red,yellow,\
     green,cyan,blue,magenta],
]
anzahlMuster=len(muster)

# ******************************************************
# *************** Function Definitions *****************
# ******************************************************
def clearKringels(*positionen):
    for p in positionen: eval(p +".clearKringel()")

def initGame():
    clearKringels("left","mid","right")
    mid.rainbowKringel([red,yellow,green,cyan,blue,magenta, \
                        red,yellow,green,cyan,blue,magenta],\
                        cnt=2,delay=0.1)
    mid.dimKringel(delay=0.1,stufen=8,down=True)
    mid.clearKringel()
    disp.clearAll()

def writeAsset(name,asset):
    try:
        file=open(name,"w")
        file.write(str(asset)+"\n")
        file.close()
    except:
        disp.clearAll()
        disp.writeAt(name+" asset",0,1)
        disp.writeAt("not written!",0,2)
        k.waitForTaste()

def readAsset(name):
    try:
        file=open(name,"r")
        asset=int(file.readline())
        file.close()
        return asset
    except:
        if name=="machine.txt":
            return 1000
        else:
            return 0

def tellState():
```

```
    disp.clearAll()
    disp.writeAt("GOOD BYE",0,0,False)
    disp.writeAt("GAME OVER",0,1,False)
    disp.writeAt("Your total",0,2,False)
    disp.writeAt("investment:{}".format(totalInvestment)\
                 ,0,3,False)
    disp.writeAt("{} game(s)".format(totalGames),0,4,False)
    disp.writeAt("Total gain:{}".format(totalGain),0,5)
    print("Machine gained:",machineGain)
    print("Machine Asset:",machineAsset)

# ******************************************************
# ********************* Main Loop **********************
# ******************************************************
disp.clearAll()
disp.writeAt("THE ONEARMED",0,0)
disp.writeAt("BANDIT",0,1)
disp.writeAt("WELLCOME",0,2)
clearKringels("left","mid","right")
sleep(2)
myAsset=readAsset("myAsset.txt")
machineAsset=readAsset("machine.txt")
print("Maschinenkapital",machineAsset)
delay=0.03
random=True
totalInvestment=0
totalGain=0
totalGames=0
machineGain=0

while 1:
    # Guthaben pruefen
    if myAsset<20:
        disp.clearAll()
        disp.writeAt("ENTER YOUR",0,0,False)
        disp.writeAt("INVESTMENT",0,1,False)
        disp.writeAt("(20C per Game)",0,2,False)
        disp.writeAt("TO ABORT PRESS",0,3,False)
        disp.writeAt("KEYS # D",0,4,False)
        x=disp.writeAt("ENTER>>> ",0,5)
        amount=k.padInput(xp=x,yp=5)
        if amount=="+":
            writeAsset("machine.txt",machineAsset)
            writeAsset("myAsset.txt",myAsset)
            tellState()
            sys.exit()
        myAsset+=int(amount)
        totalInvestment+=int(amount)
    disp.clearAll()
    disp.writeAt("Your asset:{}".format(myAsset),0,0)
    disp.writeAt("TO START GAME",0,1)
    disp.writeAt("PRESS * KEY",0,2)
```

```
    disp.writeAt("TO ABORT PRESS # KEY",0,3)
    taste=k.waitForKey(timeout=0, ASCII=True)
    if taste=="+":
        writeAsset("machine.txt",machineAsset)
        writeAsset("myAsset.txt",myAsset)
        tellState()
        print("Machine gained",machineGain)
        sys.exit()
    #
    myAsset-=20
    machineAsset+=20
    machineGain+=20
    l_stoped,m_stoped,r_stoped=False,False,False
    lauf=0
    maxLauf=240
    totalGames+=1
    while not(l_stoped and m_stoped and r_stoped) and \
              lauf<=maxLauf:
        for i in range (len(muster)):
            if random:
                l,m,r=os.urandom(3)
                l,m,r=l%anzahlMuster,m%anzahlMuster,r%anzahlMuster
            else:
                l,m,r=i,(i+3)%anzahlMuster,(i+5)%anzahlMuster
            if not l_stoped:
                left.kringel=muster[l]
                left.lightKringel()
                lPos=l
                if leftStop.value()==0:
                    l_stoped=True
            if not m_stoped:
                mid.kringel=muster[m]
                mid.lightKringel()
                mPos=m
                if midStop.value()==0:
                    m_stoped=True
            if not r_stoped:
                right.kringel=muster[r]
                right.lightKringel()
                rPos=r
                if rightStop.value()==0:
                    r_stoped=True
            sleep(delay)
            lauf+=1
    # Gewinnermittlung
    gewinn=0
    if mPos==lPos or mPos==rPos or lPos==rPos:
        if mPos<=3 or lPos<=3:
            gewinn=10
        elif mPos<=6 or lPos<=6:
            gewinn=20
        else:
```

```
            gewinn=30
    if mPos==lPos and mPos==rPos and lPos==rPos:
        if mPos<=5 or lPos<=5:
            gewinn=50
        else:
            gewinn=100
    if gewinn > 0:
        machineAsset-=gewinn
        machineGain-=gewinn
        myAsset+=gewinn
        totalGain+=gewinn
        disp.clearAll()
        disp.writeAt("YOUR GAIN:{}".format(gewinn),0,0)
    else:
        disp.clearAll()
        disp.writeAt("SORRY",0,1)
        disp.writeAt("MORE LUCK",0,2)
        disp.writeAt("NEXT TIME!!!",0,3)
    sleep(3)
```

The main program (while loop) does not even comprise 100 program lines despite extensive output activities. This is because different modules take care of the various frame jobs such as display control, key query, I2C bus operation and, with mring.py, the display of animated patterns on the neopixel rings. I created the latter module because there is now a nice collection of control functions. Functions that only have to do with the neopixel ring have been combined as methods in the MAGIC_RING class. In detail these are:

```
def lightKringel (self):
def clearKringel (self):
def clearRing (self):
def rainbowKringel (self, colList, cnt = 3, delay = 0.3):
def faecherKringel (self, colList, percent, delay = 0.1, hemi = 3, \
            dim = False, updown = 1):
def starryNightKringel (self, delay = 5, duration = 100):
def dimKringel (self, delay = 0.1, steps = 8, down = True):
def boostKringel (self, factor = 1.3):
def blinkKringel (self, on = 0.3, off = 0.7, cnt = 1, remain = False):
def randomKringel (self):
def ringGraphKringel (self, colList, size):
def showStatus (self, stat):
```

Each instance of this class also contains a complete set of color definitions and an array called kringel, which holds the color numbers of all pixels in the ring. This makes it possible to adapt the data for each ring individually. On the other hand, rings can also be cascaded without problems by specifying a higher number of pixels when instantiating. Of course, the selection must then be made by the programmer by addressing the pixels of a special ring. The best thing to do is to try out the individual methods yourself to explore the possibilities.

The stakes and winnings are added up and saved in a file when the game is over so that the data is available the next time you switch on the game. The functions writeAsset () and readAsset () show how this works.

## How is the game play or how does the program work?

The greeting is followed by the deletion of the neopixel rings. After 2 seconds the program tries to read in the last scores for the machine account and for the player account. If the files do not yet exist or if an error occurs when accessing them, the machine, i.e. the ESP32, receives a credit of 100 euros. The player starts at 0. A few variables are initialized and then we are already in the main loop.

If the player's balance is less than 20 cents, he will be asked to deposit any amount. This is done using the keypad. The entry is completed with "D". If a "#" is entered instead of a number, the game ends. The account balances are saved. The same thing happens with the next step if, instead of pulling the lever of the "one-armed man", i.e. pressing the "*" key, you catch the "#" key.

After the start of the game round, the "wheels" with their 8 symbols run 240: 8 = 30 rounds and then come to a stop at the latest. There are two modes for the process, one purely randomly controlled and one in which the pattern index is increased by 1 for the first "wheel", by 3 for the second "wheel" and by 5 for the third "wheel". With both methods, the target index is always trimmed to the range 0 to 7 by modulo calculation.

Here are two examples:

any ring
Random number about 237
Index = 237% 8 = 5, because 237: 8 = 29 remainder 5

Ring 2:
last index: 4
(4 + 3)% 8 = 7
Ring 3:
last index: 6
(6 + 5)% 8 = 11% 8 = 3

While the "wheels" are running, you can stop the run with the respective button. The The game round is over when either all "wheels" have stopped or the maximum running time has been reached.

How likely is the pattern to match on two or even three "wheels"? No prediction can be made when stopping with the keys. When using os.urandom (), the values are almost exactly evenly distributed for a few hundred throws. Which does not mean that two or even three identical values cannot be thrown in the range 0 to 7 every now and then. A theoretical hit rate of approx. 30% predicts the following small program for 240 drawings.

```
import os,sys

i1,i2,i3=2,4,6
a1,a2,a3=1,3,6
```

```
s=0
runden=240
games=100
for m in range(games):
    for n in range(runden):
        w=os.urandom(3)
        i1=(w[0])%8
        i2=(w[1])%8
        i3=(w[2])%8
        if (i1 == i2 or i2==i3) and n==(runden - 1):
            print (m,i1,i2,i3)
            s+=1
print (s, "{}%".format(s/games*100))
```

When using the ring addition modulo 8, you can exert a certain influence on the chances of winning through the summands and the number of runs as well as the starting values for the sample codes. In the following case, 1000 rounds theoretically result in a total of 250 triplets. If you let the game run down without stopping a ring, you may still not get any hits if the game automatically stops after 240 updates of the indices. The following small programs can tell you a lot about the connections. Just adjust the various parameters and look at the entire output in the terminal window.

```
i1,i2,i3=4,4,4
a1,a2,a3=1,3,4
s=0
runden=240
for n in range(runden):
    i1=(i1+a1)%8
    i2=(i2+a2)%8
    i3=(i3+a3)%8
    if i1 == i2 or i2==i3:
        print (n,i1,i2,i3)
        s+=1
print (s, "{}%".format(s/runden*100))
```

You can use these insights to expand settings for the game. The strategy now used in the game corresponds to the following test program and does not provide a single match in 1000 draws.

```
i1,i2,i3=0,0,0
a1,a2,a3=1,3,5
s=0
runden=1000
for n in range(runden):
    i1=(n+a1)%8
    i2=(n+a2)%8
    i3=(n+a3)%8
    if i1 == i2 or i2==i3:
        print (n,i1,i2,i3)
        s+=1
print (s, "{}%".format(s/runden*100))
```

To make things even more interesting, in addition to the matches, we can also weight the round results for the profit, as was done in the program. Here it is determined in which area a match brings which profit.

One thing becomes clear in any case, the bottom line is that the slot machine is always the winner, just like in real life. The display and the terminal window provide information about this after the game has ended. It's good that we are all about virtual cash, there is nothing to lose - just have fun. And by the way, you will learn programming with MicroPython.

Have fun tinkering, programming and playing!

Links zur Blogreihe:
PDF in deutsch
PDF in english
Wie arbeitet die Abfrage einer Tastaturmatrix?
Thonny – Installation und Einführung
Farben-Raten mit Ringmaster1
Ringmaster2 und Codenumber
mring.py driver for Neopixel rings