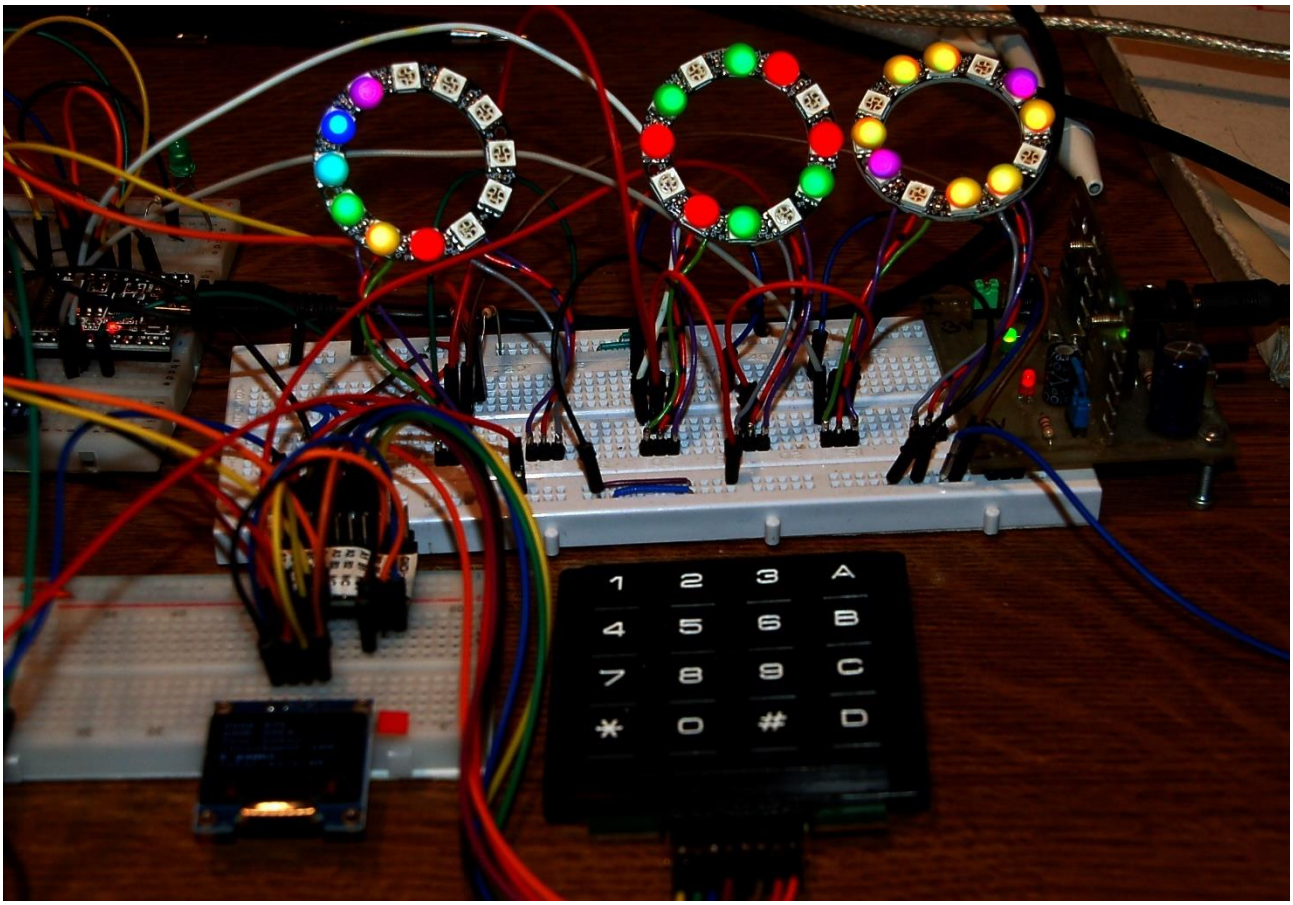

Abbildung 1: Bandit-Aufbau

Diesen Beitrag gibt es auch als:
PDF in deutsch

This episode is also available as:
PDF in english

Bei den Spielen, die ich bisher vorgestellt habe, ging es um Strategie und Überlegung. Natürlich kann man auch bis zur Erschöpfung einfach nur herumprobieren. In diesem Spiel regiert, je nach Programmierung mehr oder weniger, Meister Zufall. Trotzdem ähnelt der Hardwareeinsatz wieder stark dem in den Blogs zu Ringmaster 1 und 2. Damit willkommen bei der Vorstellung des Spiels

Bandit –
Spiele mit dem ESP32 in MicroPython

Das OLED-Display aus der vorangegangenen Blogfolge Ringmaster 2 kommt wieder zum Einsatz, auch die 4x4 Tastenmatrix. Der erste Neopixelring bekommt Zuwachs durch zwei Brüder. Mehr Ausgabeinformationen machten das 6-zeilige OLED-Display erforderlich. Das 16-er Tastenfeld ersetzt im Feldeinsatz des Spielaufbaus die Zehnertastatur des PC.

Worum geht es denn nun in diesem Spiel? Vielleicht kennen Sie die Art von Spielautomaten, wie sie in den Casinos von Las Vegas oder auch in deutschen Spielhöllen stehen? Gemeint sind die Dinger mit drei oder mehr Walzen, die sich mittels eines Hebels in Bewegung setzen lassen und danach entweder selbst auslaufen oder

durch Tasten gestoppt werden können. Wir wollen heute das Modell für so einen "einarmigen Banditen", wie man die Dinger auch nennt, mit dem ESP32 unter MicroPython modellieren. Das Programm wird in einer Endlosschleife laufen. Damit man diese während der Entwicklungszeit gezielt verlassen kann, habe ich, wie üblich, eine Notbremse eingebaut.

Notbremse benutzen heißt, punktgenau das laufende Programm beenden, ohne sofortigen Neustart, der durch die RST-Taste am ESP32 erfolgen würde, wenn das Programm unter dem Namen boot.py gestartet wurde. Genau Letzteres ist notwendig, wenn das Programm autonom, also ohne angeschlossenem PC, starten soll. In der Testphase ist ein Abbruch oft nur durch eine Notfalltaste möglich.

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 interaktiv testen, ohne jedes Mal vorher einen ganzen Rattenschwanz an Imports und Declarationen etc. neu eingeben zu müssen, das erledigt für uns das zuvor gestartete Programm. Dass über die REPL-Kommandozeile solche Tests einfach durchgeführt werden können, ist ein entscheidender Vorteil der MicroPython-Umgebung.

Hardware

Für "Bandit" wird also 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. Der ESP8266-12F scheidet wegen zu wenig RAM-Speicher aus, ihm fehlen gut 1200 Bytes. Aber nicht nur das, für diesen Einsatz hat der ESP8266 auch entschieden zu wenige GPIO-Anschlüsse aufzuweisen.

Als Tastatur kommt ein 4x4-Matrix-Tastenfeld zur Anwendung. Das OLED-Display wird über den I2C-Bus bedient, der auch bereits den Tastaturanschluss versorgt. Für den Neopixelring gibt es in der MicroPython-Firmware ein eingebautes Modul, das die Programmierung kinderleicht macht. Zur Arbeitsweise 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	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 ein TTP224 4-Kanal Digitaler Berührungssensor Kapazitiver Touch Modul

Die Schaltung für "Bandit" wird im Prinzip aus der Episode "Ringmaster 2" http://www.grzesina.de/az/gps/teil2/gps_mcp_teil2_ger.pdfübernommen. Für die "Spielwalzen" werden drei Neopixelringe benötigt. Das Rollen wird durch acht verschiedene LED-Muster simuliert.

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 der Tastatur mit, kann aber die drei Neopixelringe nicht füttern. Um deren Strombedarf bereitzustellen, bedarf es einer eigenen 3,3V-Spannungsquelle, die Sie mittels eines Reglermoduls aus den 5V ableiten können. Dieser Regler ist in der obigen Teileliste nicht aufgeführt.

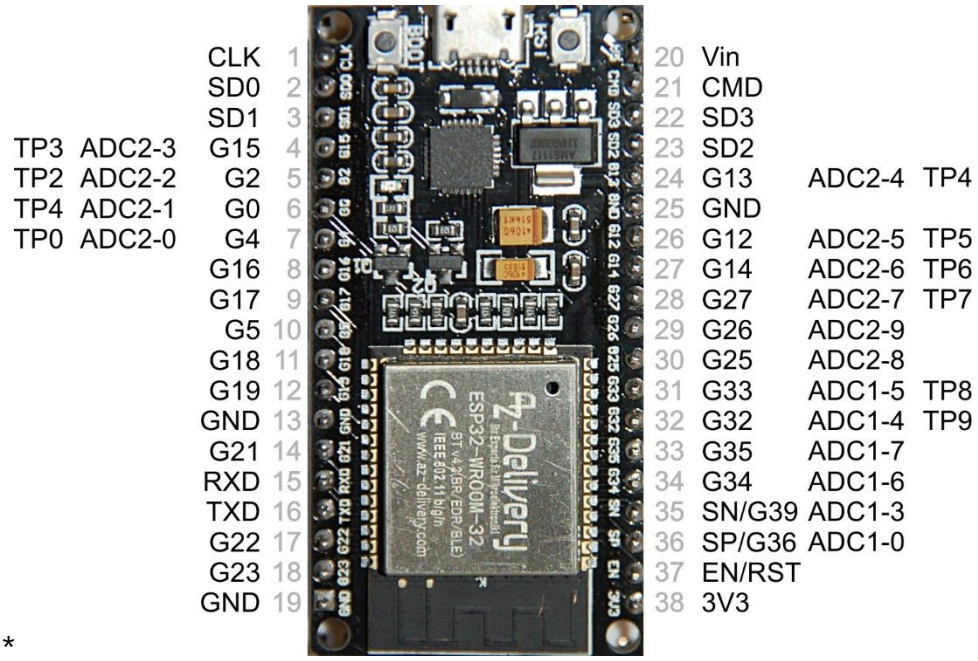

Abbildung 2: ESP32-DEVKITC_V3_Pinout

Die Versorgung aus einem 4,5V-Block aus Alkalizellen wäre zwar für das Controllerboard ausreichend, aber das Display gibt sich damit leider nicht zufrieden. Für Versorgungsspannungen über 5V muss ein extra 5V-Regler verwendet werden, denn der Neopixelring darf nicht mehr als 5,3V abbekommen.

Tipp:
Zum Experimentieren eignen sich alte PC-Netzteile sehr gut, weil sie neben 5V auch 3,3V und 12V zur Verfügung stellen.

Die folgende Abbildung zeigt das Schaltschema. Ein besser lesbares Exemplar in DIN A4 können Sie als PDF-Datei downloaden. Die drei Taster können evtl. durch ein kapazitives Touchmodul TTP224 4-Kanal Digitaler Berührungssensor mit 4 Pads ersetzt werden, das jedoch nur für schmale Finger geeignet ist.

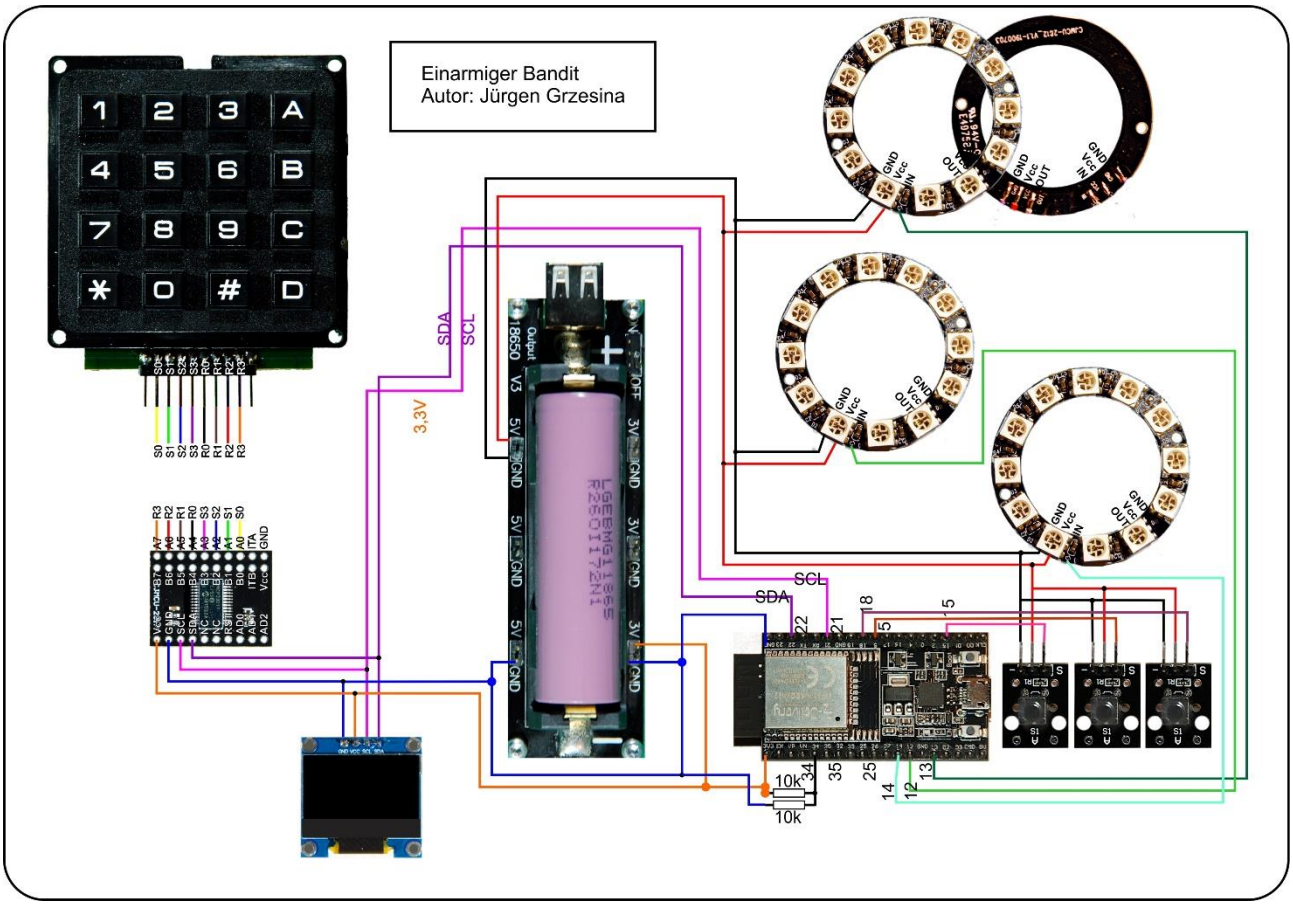

Abbildung 3: Bandit_Schaltung

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 werden von einem Microcontroller wie dem ESP32 erzeugt. Je Neopixel-Einheit werden 24 Bit generiert (jeweils 8 für grün, rot und blau). Die Dauer 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, alle nun folgenden Bits 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 nicht gleichzeitig, sondern zeitversetzt um jeweils 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). Die Farbreihenfolge beim MicroPython-Modul ist rot-grün-blau, die Reihenfolge auf dem Bus ist grün – rot – blau. Mehrere Ringe kann man genau so wie einzelne LEDs cascadieren, indem man den Eingang des nächsten Rings mit dem Ausgang des Vorgängers verbindet. Wir werden die drei Ringe aber an drei verschiedenen GPIOs betreiben, weil das durch die bereits bestehenden Funktionen von der Adressierung her einfacher ist.

Die Anschlüsse am Ring erfolgen rückwärtig, 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 selbst experimentell über REPL. Die Helligkeit der einzelnen Farbkanäle ist recht unterschiedlich. Die Codes in den Tupels für Mischfarben werden also 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".

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

Abbildung 4: LED-Ring_vorn

Die Software

Verwendete Software:
Fürs Flashen und die Programmierung des ESP32:
Thonny oder
µPyCraft

Verwendete Firmware:
MicropythonFirmware
Bitte eine Stable-Version aussuchen

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

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, damit der Controller MicroPython-Anweisungen versteht. Sie können dazu Thonny, µPyCraft oder esptool.py benutzen. Für Thonny habe ich den Vorgang hier beschrieben.

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

Manuell gestartet werden Programme aus dem aktuellen Editorfenster in der Thonny-IDE über die Taste F5. Das geht schneller als der Mausklick auf den Startbutton oder über das Menü **Run**. Lediglich die im Programm verwendeten Module müssen sich im Flash des ESP32 befinden.

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

Die im Programm verwendeten Datenstrukturen für die Farbverwaltung habe ich in dem 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.

Damit das Programm bandido.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, mring.py und mcp.py. Wenn das erledigt ist, können wir das Programm bandido.py im Editorfenster mit F5 starten – sofern die Hardware zusammengebaut und der ESP32 am PC angeschlossen ist.

Hier das Listing des Programms.

```
# 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)
```

Das Hauptprogramm (while-Schleife) umfasst trotz umfangreicher Ausgabeaktivitäten nicht einmal 100 Programmzeilen. Das liegt daran, dass verschiedene Module sich um die vielfältigen Rahmenjobs wie Displaysteuerung, Tastenabfrage, I2C-Busbedienung und, mit mring.py, um die Darstellung von animierten Mustern auf den Neopixelringen kümmern. Das letztere Modul habe ich erstellt, weil sich mittlerweile eine nette Sammlung von Steuerfunktionen ergeben hat. Funktionen, die ausschließlich mit dem Neopixelring zu tun haben, wurden in der Klasse MAGIC_RING als Methoden vereint. Im Einzelnen sind das:

```
    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,stufen=8,down=True):
    def boostKringel(self,faktor=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):
```

Jede Instanz dieser Klasse enthält ferner einen kompletten Satz an Farbdefinitionen und ein Array namens kringel, das die Farbnummern aller Pixel des Rings vorhält. Damit ist es möglich, die Daten für jeden Ring einzeln anzupassen. Andererseits kann man durch Angabe einer höheren Pixelzahl bei der Instanzierung Ringe auch problemlos cascadieren. Natürlich muss dann die Auswahl über die Adressierung der Pixel eines speziellen Rings durch den Programmierer erfolgen. Am besten, Sie probieren die einzelnen Methoden selbst einmal aus, um die gebotenen Möglichkeiten auszuloten.

Die Spieleinsätze und Gewinne werden aufsaldiert und bei Spielbeendigung in einer Datei gespeichert, damit die Daten beim nächsten Einschalten wieder zur Verfügung stehen. Wie das geht zeigen die Funktionen writeAsset() und readAsset().

Wie ist der Spielablauf oder wie arbeitet das Programm?

Der Begrüßung folgt das Löschen der Neopixelringe. Nach 2 Sekunden versucht das Programm, die letzten Spielstände einzulesen, für das Maschinenkonto und für das Spielerkonto. Gibt es die Dateien noch nicht oder passiert beim Zugriff ein Fehler, dann bekommt die Maschine, also der ESP32, ein Guthaben von 100 Euro. Der Spieler startet bei 0. Ein paar Variablen werden initialisiert und dann befinden wir uns bereits in der Hauptschleife.

Falls das Guthaben des Spielers weniger als 20 Cent beträgt, wird er aufgefordert einen beliebigen Betrag einzuzahlen. Das geschieht über das Tastenfeld. Die Eingabe wird mit "D" abgeschlossen. Wird statt einer Zahl ein "#" eingegeben, führt das zur Beendigung des Spiels. Die Kontostände werden gespeichert. Dasselbe passiert beim nächsten Schritt, wenn man, statt den Hebel des "Einarmigen" zu ziehen, sprich die Taste "*" zu drücken, die Taste "#" erwischt.

Nach dem Start der Spielrunde laufen die "Räder" mit ihren 8 Symbolen 240 : 8 = 30 Runden und kommen spätestens danach zum Stehen. Für den Ablauf gibt es zwei Modi, einen rein zufallsgesteuerten und einen, bei dem der Musterindex beim ersten "Rad" um 1, beim zweiten "Rad" um 3 und beim 3. "Rad" um 5 erhöht wird. Bei beiden Verfahren wird der Zielindex stets durch Modulo-Berechnung auf den Bereich 0 bis 7 getrimmt.

Dazu zwei Beispiele:

beliebiger Ring
Zufallszahl etwa 237
Index = 237 % 8 = **5**, weil 237 : 8 = 29 **Rest 5**

Ring 2:
letzter Index: 4
(4 + 3) % 8 = 7
Ring 3:
letzter Index: 6
(6+5) % 8 = 11 % 8 = 3

Während der Laufzeit der "Räder" kann man mit der jeweiligen Taste den Lauf stoppen. Durch eine if-Bedingung wird getestet, ob das "Rad" schon angehalten" wurde. wenn ja, wurde auch der Mustercode festgehalten. Eine zweite if-Bedingung testet andernfalls, ob die entsprechende Stopptaste gedrückt ist und setzt gegebenenfalls das stopped-Flag. Die

Spielrunde ist zu Ende, wenn entweder alle "Räder" angehalten wurden oder die maximale Laufzeit erreicht wurde.

Wie wahrscheinlich ist die Übereinstimmung der Muster auf 2 oder gar drei "Rädern"? Beim Stoppen mit den Tasten kann man keine Voraussage treffen. Bei der Verwendung von os.urandom() ergibt sich bei einigen hundert Würfen eine fast exakte Gleichverteilung der Werte. Was nicht heißt, dass nicht doch im Bereich 0 bis 7 hin und wieder zwei oder gar drei gleiche Werte geworfen werden können. Eine theoretische Trefferquote von ca. 30% sagt für 240 Ziehungen das folgende kleine Programm voraus.

```
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))
```

Beim Einsatz der Ringaddition modulo 8 können Sie durch die Summanden und die Anzahl von Durchläufen sowie den Startwerten für die Mustercodes einen gewissen Einfluss auf die Gewinnchancen ausüben. In folgenden Fall ergeben sich bei 1000 Runden theoretisch insgesamt 250 Drillinge. Wenn Sie das Spiel, ohne einen Ring zu stoppen, auslaufen lassen, werden Sie unter Umständen dennoch keine Treffer erzielen, wenn das Spiel nach 240 Updates der Indices automatisch stoppt. Die folgenden kleinen Programme können Ihnen dazu einiges über die Zusammenhänge verraten. Verstellen Sie einfach die verschiedenen Parameter und schauen Sie sich die gesamte Ausgabe im Terminalfenster an.

```
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))
```

Diese Erkenntnisse können Sie zum Erweitern der Einstellungen für das Spiel verwenden. Die jetzt im Spiel eingesetzte Strategie entspricht folgendem Testprogramm und liefert keine einzige Übereinstimmung bei 1000 Ziehungen.

```
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))
```

Damit die Sache noch interessanter wird, können wir außer den Übereinstimmungen auch noch eine Gewichtung der Rundenergebnisse für den Gewinn vornehmen, wie es im Programm geschehen ist. Hier wird festgelegt, in welchem Bereich eine Übereinstimmung welchen Gewinn beschert.

Eines wird in jedem Fall deutlich, Gewinner ist unter dem Strich stets der Spielautomat, so wie im wirklichen Leben halt auch. Darüber geben das Display und das Terminalfenster nach Beendigung des Spiels Auskunft. Gut, dass es bei uns nur um virtuelle Kohle geht, da kann man nix verlieren – nur Spaß haben. Und ganz nebenbei lernen Sie die Programmierung mit MicroPython.

Viel Vergnügen beim Basteln, Programmieren und Spielen!

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 Muster-Treiber für NeopixelRinge