Pico-Bibliothek

LedRing

Der LED-Ring besteht aus einer ringförmig angeordneten Reihe von NeoPixels. NeoPixels, auch WS2812-LEDs genannt, sind seriell angeschlossene Vollfarb-LEDs, die einzeln adressierbar sind und deren Rot-, Grün- und Blauanteil zwischen 0 und 255 eingestellt werden kann.

RGB-Farbmischer: https://informatik.schule.de/rgb/RGB_farbmischer.html

Anlegen des LedRing-Objekts:

```
from LedRing import LedRing

NUM_LEDS = 24
ledRing = LedRing(NUM_LEDS)
```

Setzen der Farbe einer LED:

```
ledRing[0] = (255, 0, 0) # set to red, full brightness
ledRing[1] = (0, 128, 0) # set to green, half brightness
ledRing[2] = (0, 0, 64)  # set to blue, quarter brightness
```

Setzen der Farbe aller LEDs:

```
ledRing.fill(0, 0, 0) # set all LEDs to black
```

Die gesetzte Farbe aller LEDs auf dem LED-Ring anzeigen:

```
ledRing.write()
```

Die gesetzten Farben der LEDs werden erst dann auf dem LED-Ring dargestellt, nachdem die `write()`-Methode ausgeführt wurde.

Auslesen der Farbe einer LED:

```
r, g, b = ledRing[0]     # read color of first LED
```

Auslesen der Anzahl der LEDs im Ring:

```
n = ledRing.n
```

Beispielprogramm:

```
from LedRing import LedRing

NUM_LEDS = 24
ledRing = LedRing(NUM_LEDS)

BLACK   = (0, 0, 0)
RED     = (255, 0, 0)
GREEN   = (0, 255, 0)
BLUE    = (0, 0, 255)
YELLOW  = (255, 255, 0)
MAGENTA = (255, 0, 255)
CYAN    = (0, 255, 255)
WHITE   = (255, 255, 255)

ledRing.fill(BLACK)     # set all LEDs to black
ledRing[0] = RED        # set first LED to red
ledRing[1] = GREEN      # set second LED to green
ledRing[2] = BLUE       # set third LED to blue

ledRing.write()         # show previously set colors on LED ring
```

PicoWlan

Anlegen des WLAN-Objekts:

```
from PicoWlan import PicoWlan
import secrets

wlan = PicoWlan(secrets.WIFI_SSID, secrets.WIFI_PASSWORD,
secrets.WIFI_COUNTRY)
```

Die erforderliche Zugangsdaten werden in einer Datei namens **„secrets.py“** abgelegt:

secrets.py

```
#### create file "secrets.py" containing:
#### WIFI_SSID     = "my_ssid"
#### WIFI_PASSWORD = "my_password"
#### WIFI_COUNTRY  = "my_country" e.g. "EN", "DE", ...
```

```
WIFI_SSID     = "my_ssid"
WIFI_PASSWORD = "my_password"
WIFI_COUNTRY  = "my_country"
```

Starten des WLANs:

```
wlan.run()
```

Timer

Anlegen des Timer-Objekts:

```
from Timer import CyclicTimerMs

MILLISECS = 500 # number of milliseconds
cyclicTimer = CyclicTimerMs(MILLISECS )
```

Starten des Timers:

```
cyclicTimer.start()
```

Prüfen, ob die eingestellte Zeit verstrichen ist:

```
if (cyclicTimer.overrun() == True:
    print("Overrun of timer")
```

Beispielprogramm:

```
from Timer import CyclicTimerMs

MILLISECS = 1000 # number of milliseconds

cyclicTimer = CyclicTimerMs(MILLISECS )
cyclicTimer.start()

while True:
    while not cyclicTimer.overrun():
        time.sleep(0.01)
```

```
    print("Next second")
```

TimeApiIo

Die TimeApiIo.Klasse ermöglicht das Holen der aktuellen Zeitinformation von der von https://timeapi.io/ bereitgestellten API. Hierfür ist ein Zugang zum Internet erforderlich (z.B. über die Klasse `PicoWlan`).

Weitere Infos gibt es hier: https://timeapi.io/swagger/index.html

Anlegen des TimeApiIo-Objekts:

```
from TimeApiIo import TimeApiIo

timeApiIo = TimeApiIo()
```

Setzen der Zeitzone:

```
timeApiIo.setTimeZone("Europe/Berlin")
```

Lesen der aktuellen Zeitinformation:

```
t = timeApiIo.getCurrentTime()
```

Die Methode liefert ein `Dictionary`-Objekt mit folgender Struktur zurück:

```
{
  "year": 2023,
  "month": 12,
  "day": 4,
  "hour": 12,
  "minute": 21,
  "seconds": 54,
  "milliSeconds": 652,
  "dateTime": "2023-12-04T12:21:54.6520906",
  "date": "12/04/2023",
  "time": "12:21",
  "timeZone": "Europe/Berlin",
  "dayOfWeek": "Monday",
  "dstActive": false
}
```

Beispielprogramm:

```
import TimeApiIo from TimeApiIo

timeApiIo = TimeApiIo()
timeApiIo.setTimeZone("Europe/Berlin")

data = timeApiIo.getCurrentTime()
print(data) # Aktuelle Zeitinformation ausgeben
```

Button

Anlegen des Button-Objekts:

```
from Button import Button

pin = 14                    # GPIO14
pressedLevel = Button.LOW  # Taster schaltet gegen GND

button = Button(pin, pressedLevel, bounceTimeMs=200, longTimeMs=5000)
```

Die Parameter sind:

- `pin` gibt den Anschlusspin des Taster an.
- `pressedLevel` definiert, welcher Pegel bei gedrückter Taste an dem Pin anliegt. Erlaubte Werte sind:
  - `Button.LOW`
  - `Button.HIGH`
- `bounceTimeMs` ist ein optionaler Parameter und gibt die Prellzeit des Taster in Millisekunden an. Der voreingestellte Wert liegt bei `100`
- `longTimeMs` ist ein optionaler Parameter, der zur Erkennung zwischen kurzem und langem Tastendruck dient. Dr Wert wird in Millisekunden angegeben. Wir kein Wert mitgegeben, erfolgt keine Unterscheidung zwischen langen und kurzem Tastendruck.

Lesen eines Events

```
event = button.event()
```

Die Methode liefert folgende Ergeignisse zurück:

- `Button.Event.NONE` kein Ereignis
- `Button.Event.PRESSED` Taste wurde gerade gedrückt
- `Button.Event.RELEASED` Taste wurde gerade losgelassen
- `Button.Event.LONG_PRESS_DETECTED` Taste ist gedrückt, Zeitintervalls abgelaufen (nur wenn `longTimeMs` gesetzt wurde)

- `Button.Event.LONG_PRESS_RELEASED` Taste wurde nach Aublauf des Zeitintervalls losgelassen (nur wenn `longTimeMs` gesetzt wurde)

Potentiometer

Anlegen des Potentiometer-Objekts:

```
from Potentiometer import Potentiometer

pin = 28
potentiometer = Potentiometer(pin)
```

Die Parameter sind:

- `pin` gibt den Anschlusspins des Potentiometers an.

Lesen einer Potentiometerstellung

```
# Auslesen der Potentiometerstellung
value = potentiometer.read()
```

Die Methode liefert Werte zwischen `0.0` und `1.0` zurück.

Beispielprogramm:

```
from Potentiometer import Potentiometer
from time import sleep_ms

potentiometer = Potentiometer(28)

# To read from the pin, set up a loop:
while True:
    print(potentiometer.read())
    sleep_ms(200)
```

DipSwitch

Anlegen des DipSwich-Objekts:

```
from DipSwitch import DipSwitch
```

```
pins = [ 19, 20, 21, 22 ]
onLevel = DipSwitch.LOW
dipSwitch = DipSwitch(pins, onLevel)
```

Die Parameter sind:

- `pins` gibt den Anschlusspins der Schalter an.
- `pressedLevel` definiert, welcher Pegel bei eingeschaltetem Schalter an dem jeweiligen Pin anliegt. Erlaubte Werte sind:
  - `DipSwitch.LOW`
  - `DipSwitch.HIGH`

Lesen einer Schalterstellung

```
# Auslesen des ersten Schalters
value = dipSwitch[0]
```

Die Methode liefert folgende Werte zurück:

- `True` Schalter ist eingeschaltet
- `False` Schalter ist ausgeschaltet

Anzahl der Schalter ermitteln

```
n = dipSwitch.size()
```

Die Methode liefert die Anzahl der Schalter des DIP-Schalters zurück.

Beispielprogramm:

```
from DipSwitch import DipSwitch

DIP_PINS = [ 19, 20, 21, 22 ]
dipSwitch = DipSwitch(DIP_PINS, DipSwitch.LOW)

for i in range(dipSwitch.size()):
    print ("DIP-Switch "+ str(i) + ": " + str(dipSwitch[i]))
```

Buzzer

Die Buzzer-Klasse ermöglicht die Ansteuerung eines Summers.

Anlegen des Buzzer-Objekts:

```
from Buzzer import Buzzer

pin = 16
buzzer = Buzzer(pin)
```

Die Parameter sind:

- `pin` gibt den Anschlusspin des Summers an.

Ausgabe eines Tons:

```
frequency = 1000
milliseconds = 200
buzzer.beep(frequency, milliseconds)
```

Die Parameter sind:

- `frequency` gibt die Frequenz des Tons an. Die Angabe kann auch über vordefinierte Notenwerte erfolgen. Diese sind:
  - `Buzzer.C`
  - `Buzzer.DES`
  - `Buzzer.D`
  - `Buzzer.ES`
  - `Buzzer.E`
  - `Buzzer.F`
  - `Buzzer.GES`
  - `Buzzer.G`
  - `Buzzer.AS`
  - `Buzzer.A`
  - `Buzzer.B`
  - `Buzzer.H`
- `milliseconds` bestimmt die Dauer des Tons in Millisekunden. Die Angabe kann auch über vordefinierte Werte erfolgen. Vordefinierte Notenlängen sind:
  - `Buzzer.WHOLE`
  - `Buzzer.HALF`
  - `Buzzer.QUARTER`
  - `Buzzer.EIGHTH`

Die nächste Oktave erhält man durch Multiplikation der Frequenz `frequency` mit dem Wert 2 (z.B.: `Buzzer.C * 2`), die vorherige durch Division durch den Wert 2 (z.B.: `Buzzer.A / 2`).

Dieses Vorgehen kann man kaskadieren. Das bedeutet, dass die Frequenzen der zweithöheren Oktave durch eine weitere Multiplikation mit dem Wert 2 ermittelt werden können, also z.B. `Buzzer.C * 2 * 2` bzw. `Buzzer.C * 4`. Analog dazu lassen sich natürlich auch die Frequenzen der vorherigen Oktaven durch mehrfache Division

ermitteln.

Beispielprogramm:

Das folgende Beispielprogramm spielt das Kinderlied „Alle meine Entchen“ auf dem Summer ab.

```
from Buzzer import Buzzer

pin = 16
buzzer = Buzzer(pin)

buzzer.beep(Buzzer.C, Buzzer.QUARTER)
buzzer.beep(Buzzer.D, Buzzer.QUARTER)
buzzer.beep(Buzzer.E, Buzzer.QUARTER)
buzzer.beep(Buzzer.F, Buzzer.QUARTER)
buzzer.beep(Buzzer.G, Buzzer.HALF)
buzzer.beep(Buzzer.G, Buzzer.HALF)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.G, Buzzer.WHOLE)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.A, Buzzer.QUARTER)
buzzer.beep(Buzzer.G, Buzzer.WHOLE)
buzzer.beep(Buzzer.F, Buzzer.QUARTER)
buzzer.beep(Buzzer.F, Buzzer.QUARTER)
buzzer.beep(Buzzer.F, Buzzer.QUARTER)
buzzer.beep(Buzzer.F, Buzzer.QUARTER)
buzzer.beep(Buzzer.E, Buzzer.HALF)
buzzer.beep(Buzzer.E, Buzzer.HALF)
buzzer.beep(Buzzer.G, Buzzer.QUARTER)
buzzer.beep(Buzzer.G, Buzzer.QUARTER)
buzzer.beep(Buzzer.G, Buzzer.QUARTER)
buzzer.beep(Buzzer.G, Buzzer.QUARTER)
buzzer.beep(Buzzer.C, Buzzer.WHOLE)
```

RGB

tbd.

---

mymachine

Das Modul `mymachine` stellt Objekte zur Verfügung, die spezifische Funktionen im Zusammenhang mit der Hardware bereitstellen. Die Funktionen ermöglichen den Zugriff auf und die Steuerung von Hardware-Blöcken des Systems.

```
from mymachine import rightButton, leftButton    # Zwei Objekte der Klasse Button
from mymachine import dipSwitch                  # Objekt der Klasse DipSwitch
from mymachine import potentiometer              # Objekt der Klasse Potentiometer
from mymachine import ledRing                    # Objekt der Klasse LedRing
from mymachine import buzzer                     # Objekt der Klasse Buzzer
```