Interrupts

Was ist ein Interrupt?

Interrupt bedeutet unterbrechen. Genau das macht ein Interrupt. Er unterbricht das laufende Programm und führt stattdessen eine Interrupt-Service-Routine aus. Anschließend wird das zuvor unterbrochene Programm fortgesetzt.
Interrupts werden in der Regel von einer Hardware ausgelöst. Z.B. Taster, Timer oder ADC.

Interrupts

Um einen Interrupt zu implementieren sind 2 Dinge erforderlich:

- Eine **Interrupt-Service-Routine** (ISR) auch **Callback-Funktion** (Cb) oder **Interrupt Handler** genannt, die den Interrupt bearbeitet.
- Eine **Interruptquelle**, die mit der ISR verbunden wird.

Micropython kennt 2 Interruptquellen:

- **Pin-Interrupts**
- **Timer-Interrupts**

Interrupts

Micropython stellt für die **Pin** Interrupts bereit. Dazu dient die Methode **Pin.irq()**.

```
>>> from m5import import *

>>> from machine import Pin

>>> dir(Pin)
['__class__', '__name__', 'value',
'__bases__', '__dict__', 'IN',
'IRQ_FALLING', 'IRQ_RISING',
'OPEN_DRAIN', 'OUT', 'PULL_DOWN',
'PULL_HOLD', 'PULL_UP', 'WAKE_HIGH',
'WAKE_LOW', 'init', 'irq', 'off',
'on']
```

Interrupts

Wir müssen 3 Dinge tun um einen Interrupt zu implementieren:

1. Zuerst muss der Pin als Eingang konfiguriert werden.
2. Dann sollte der Handler definiert werden.
3. Und schließlich kann der Interrupt eingerichtet werden.

```
# Pin von Button A konfigurieren
from m5import import *
from machine import Pin

int_pin = Pin(37, Pin.IN, Pin.PULL_UP)


def int_handler():
    pass


int_pin.irq(trigger=Pin.IRQ_FALLING,
    handler=int_handler)
```

Interrupts

PinMap

RED LED & IR Transmitter & BUTTON A & BUTTON B

ESP32	GPIO10	GPIO9	GPIO37	GPIO39	GPIO2
RED LED	LED Pin				
IR Transmitter		Transmitter Pin			
BUTTON A			Button Pin		
BUTTON B				Button Pin	
Buzzer					Buzzer Pin

Interrupts

```
from m5import import *
from machine import Pin
from time import sleep_ms

lcd.clear()

main_loop_zahl = 0
interrupt_zahl = 0

main_text_box =
  M5TextBox(20, 10, str(main_loop_zahl), lcd.FONT_DejaVu24, 0xFFFFFF, rotate=0)
interrupt_text_box =
  M5TextBox(120, 10, str(interrupt_zahl), lcd.FONT_DejaVu24,0xFFFFFF, rotate=0)
```

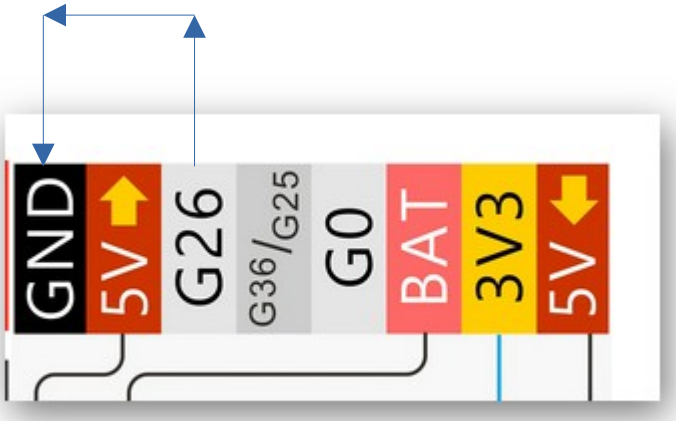
Interrupts

```
int_pin = Pin(37, Pin.IN, Pin.PULL_UP)  # Pin von Taster A mit Pullup-R

def int_handler(int_info):        # muss einen Parameter haben
    global interrupt_zahl
    global interrupt_text_box
    interrupt_zahl += 1
    interrupt_text_box.setText(str(interrupt_zahl))
    interrupt_text_box.show()
    print('Interrupt wurde ausgelöst!')
    return

int_pin.irq(trigger=Pin.IRQ_FALLING, handler=int_handler)

while True:
    main_loop_zahl += 1
    main_text_box.setText(str(main_loop_zahl))
    main_text_box.show()
    sleep_ms(1000)
```

Interrupts

Wir wollen jetzt den an den Interrupt-Handler übergebenen Wert näher untersuchen.

Dazu schaffen wir eine zweite Interruptquelle mit GPIO26.



Interrupts

```
# int_test_006.py

from m5import import *
from machine import Pin
from time import sleep_ms

lcd.clear()

interrupt_data = 0

interrupt_data_text_box = M5TextBox(60, 30, str(interrupt_data),
lcd.FONT_DejaVu24,0xFFFFFF, rotate=0)

int_pin37 = Pin(37, Pin.IN, Pin.PULL_UP)  # Pin von Taster A
int_pin26 = Pin(26, Pin.IN, Pin.PULL_UP)  # Pin GPIO26
```

Interrupts

```
def int_handler(int_data):        # muss einen Parameter haben
    global interrupt_data
    print(int_data)
    print(type(int_data))
    interrupt_data = int_data
    interrupt_data_text_box.setText(str(interrupt_data))
    interrupt_data_text_box.show()
    if int_data == Pin(37):
        print('Interrupt 37 wurde ausgelöst!')
    elif int_data == Pin(26):
        print('Interrupt 26 wurde ausgelöst!')
    else:
        print('Fehler!')
    return
```

Interrupts

```
int_pin37.irq(trigger=Pin.IRQ_FALLING, handler=int_handler)
int_pin26.irq(trigger=Pin.IRQ_FALLING, handler=int_handler)

while True:
    pass
```

Interrupts

Bei dem an den Interrupthandler übergebenen Wert **int_info** handelt es sich um das Objekt des Pins an dem der Interrupt eingegangen ist.
Ich hatte aufgrund der Darstellung im Display angenommen es wäre ein String.
Wenn dieser Wert ausgewertet werden soll dann so:

```
if int_info == Pin(37):
```

Also keine Anführungszeichen! Es ist der Name einer Instanz der Klasse Pin.

Interrupts

Interrupts können ein- und ausgeschaltet werden.

int_state speichert den Interrupt Status.

Das ist z.B. wichtig, wenn eine Programmsequenz nicht unterbrochen werden darf.

Funktioniert nicht in der REPL.

```
import machine

int_state = machine.disable_irq()

machine.enable_irq(int_state)

count = 0

def cb(): # An interrupt callback
    count +=1

def main():
    # Code to set up the interrupt
    # callback omitted
    while True:
        count += 1
```

Interrupts

Einschränkungen für Interrupt-Handler.

Interrupt Handler können keinen Speicher benutzen. Deshalb ist ihre Anwendung eingeschränkt. Alles was zusätzlichen Speicher benötigt kann in einer ISR nicht ausgeführt werden. Z.B:

- Es können keine Fließkommazahlen benutzt werden.
- Listen und Dictionaries können nicht erweitert werden.
- Es können nicht mehrere Werte zurückgegeben werden.
- Texte können nicht bearbeitet werden.
- Es können keine Fehlermeldungen erzeugt werden.

Interrupts

Interrupt Handler schreiben.

ISR’s sollten immer **so kurz wie möglich** gehalten werden. Nur Aktivitäten sie sofort ausgeführt werden müssen (z.B. Notstop) sollten in der ISR ausgeführt werden. Alles, was etwas warten kann sollte erst in der Hauptschleife bearbeitet werden. Dort gelten auch die eben dargestellten Einschränkungen nicht. Dazu ist in der ISR **ein Flag zu setzen** und in der Hauptschleife **abzufragen**.

Interrupts

Probleme bei Interrupts.

Interrupts arbeiten auf Maschinencode Ebene. Deshalb kann es passieren, das ein Pythonbefehl mitten in seiner Ausführung unterbrochen wird und erst die ISR abgearbeitet wird bevor der Pythonbefehl zu Ende ausgeführt wird. Das kann zu Fehlern führen wenn die ISR etwas verändert, das vom Hauptprogramm bearbeitet wird.
Dadurch können Fehler entstehen, die nur sporadisch auftreten und sehr schwer zu finden sind.

Interrupts

ENDE

Interrupts

Timerinterrupts

Timerinterrupt im Micropython von micropython.org

Timer sind sehr komplexe Gebilde die viele Anwendungsmöglichen bieten. Im Micropython ist nur eine grundlegende Funktion implementiert. Man kann mit ihnen nach einer definierten Zeitdauer einen Interrupt auslösen. Dieser Interrupt kann einmalig (one_shot) sein oder wiederholt (periodic) werden bis der Timer abgeschaltet wird.

Interrupts

Um einen **Timerinterrupt** verwenden zu können muss

- eine **ISR** geschrieben werden
- eine **Timer-Instanz** erzeugt werden
- der **Timer initialisiert** werden

Interrupts

```
>>> from m5import import *

>>> timer1 = machine.Timer(1)   # Einen Timer erzeugen

>>> flag = False                # Eine Flagge zur Signalisierung
>>> flag                        # des Interrupts einrichten
False

>>> def timer1_isr(x):          # Die ISR als Funktion mit 1 Para schreiben
        global flag             # Die ISR kann keinen Speicher allozieren
        flag = True             # Deshalb muss flag vorher erzeugt werden

>>> timer1.init(mode=machine.Timer.ONE_SHOT, period=1000, callback=timer1_isr)

>>> flag                        # Kontrolle, ob ISR funktioniert hat
True                            # Hat funtioniert :)
>>>
```

Interrupts

```
# Timermodes: Timer.ONE_SHOT   Timer.PERIODIC
# Timerzeit im ms

# Timer stoppen: Timer.deinit()
# Die Timerinstanz bleibt aber weiterhin erhalten:
>>> from m5import import *
>>> timer1 = machine.Timer(1)
>>> id(timer1)
1073499024
>>> machine.Timer.deinit(timer1)
>>> id(timer1)
1073499024
>>> del(timer1)
>>> id(timer1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'timer1' isn't defined
```

Interrupts

```
# Timermodes: Timer.ONE_SHOT   Timer.PERIODIC
# Timerzeit im ms

# Timer stoppen: Timer.deinit()   - wird nicht gelöscht

# Interrupts freigeben    machine.enable_irq(state)
# Interrupts sperren      machine.disable_irq()     - Thonny stürzt ab.

# Anwendung – funktioniert wenn hochgeladen:

irq_state = machine.disable_irq()
print(irq_state)
machine.enable_irq(irq_state)

# Ausgabe der Konsole:
>>> %Run -c $EDITOR_CONTENT
395552
>>>
```

Interrupts

WDT

Interrupts

WatchDogTimer

Der Wachhundtimer überwacht den Programmlauf. Dazu wird dieser Timer in regelmäßigem Abstand zurückgesetzt. Stürzt das Programm ab, wird der WDT nicht zurückgesetzt und löst einen Neustart aus. So fällt das Programm wieder auf die Beine. Einmal gestartet kann WDT nicht verändert oder gestoppt werden.

```
from machine import WDT

# Es wird eine WDT Instanz
# mit 2s Wartezeit erzeugt

wdt = WDT(timeout=2000)

# Im Hauptprogramm muss der WDT nach
# < 2s zurückgesetzt werden.

wdt.feed()
```

M5Stack Micropython

Key Interrupt

Interrupts

Um einen **Keyinterrupt** zu ermöglichen braucht es 2 Dinge:

1. Eine Funktion die tut was zu tun ist wenn der Taster gedrückt wird und
2. Eine spezielle Methode für den Taster (A/B) der diese Funktion übergeben wird.

```
from m5import import *

# Funktion für den Taster schreiben:

def buttonA_wasPressed():
  pass


# Funktion an die TasterA Methode
# übergeben

btnA.wasPressed(buttonA_wasPressed)
```

Interrupts

Der **Keyinterrupt** im M5Micropython ist also sehr einfach zu benutzen.

Der Funktionsname ist frei wählbar.

Für **TasterB** ist dann die Methode btnB.xxx zuständig.

```
>>> from m5import import *

    def mickey_maus():
        lcd.clear()
        Lcd.print('Micky Maus')

    btnA.wasPressed(mickey_maus)

>>> Micky Maus
Micky Maus
Micky Maus
Micky Maus
Micky Maus
Micky Maus
Micky Maus
Micky Maus
Micky Maus
```

Interrupts

Rechts sind alle Methoden aufgeführt die M5Stack Micropython für den M5Stick C/Plus anbietet.

Interessant ist die Methode **pressFor()** bei der die Mindestdauer des Tastendrucks angegeben werden kann.

```
# Methoden für eine Taste
btnA.wasPressed(<funktions_name>)
btnA.wasReleased(<funktions_name>)
btnA.pressFor(0.8, <funktions_name>)
btnA.wasDoublePress(<funktions_name>)

btnB.wasPressed(<funktions_name>)
btnB.wasReleased(<funktions_name>)
btnB.pressFor(0.8, <funktions_name>)
btnB.wasDoublePress(<funktions_name>)

# Methoden für 2 Tasten
btn.multiBtnCb(btnA,btnB,<funktions_name>)
```

M5Stack Micropython

Timer Interrupt

Interrupts

Der Timerinterrupt ist bei M5Stack auf den ersten Blick etwas seltsam implementiert:

```
@timerSch.event('timer1')
def ttimer1():
```

Die erste Zeile stellt einen sogenannten **Decorator** dar, der die anschließend definierte Funktion dekoriert.
Was das bedeutet werden wir uns jetzt gleich ansehen:

```
import time

@timerSch.event('timer1')
def ttimer1():
    pass

timerSch.setTimer('timer1', 100, 0x01)
timerSch.run('timer1', 100, 0x00)
timerSch.stop('timer1')
```

Interrupts

Der Einsatz von Decorators beim M5Stack Micropython.

Decorators sind ein Thema für einen fortgeschrittenen Kurs, deshalb werden wir dieses Konzept hier auch nur oberflächlich betrachten,
In Micropython gibt besonders für die OOP noch eine Reihe weiterer Decorators.
Uns interessiert hier nur der Decorator

@timerSch.event('timerx')

. Was dieser Docorator konkret tut ist nicht bekannt.

Interrupts

Rechts ein kleines Demo für die Anwendung von Dekoratoren.

```
def ein_dekorateur(func):
    def hilfsfunktion(x):
        print("Vor dem Aufruf " + func.__name__)
        func(x)
        print("Nach dem Aufruf " + func.__name__)
    return hilfsfunktion

@ein_dekorateur
def foo(x):
    print("Hallo, foo wurde mit aufgerufen " + str(x))


>>> foo('Hallo')
Vor dem Aufruf foo
Hallo, foo wurde mit aufgerufen Hallo
Nach dem Aufruf foo
```

Interrupts

Um einen **Timerinterrupt** im M5Stack Stil zu erstellen, muss **zuerst ein Timer eingerichtet** werden.

Anschließend kann die Funktion geschrieben werden die dann ausgeführt werden soll.
Dabei muss dem Funktionsnamen ein **t** voran gestellt werden.

```
# Timer einrichten

timerSch.setTimer('timer1',
                  <Zeit in ms>,
                  <PERIOGIG = 0x00 |
                   ONESHOT = 0x01>)

timerSch.run('timer1', 100, 0x00)


@timerSch.event('timer1')
def ttimer1():
  # global params
  pass
```

Interrupts

Die UIFlow-IDE verlangt, dass zuerst der Timer eingerichtet wird, bevor die Funktion dazu geschrieben werden kann. In Micropython ist das nötig.

```
>>> @timerSch.event('timer1')
    def ttimer1():
        lcd.clear()
        print('timer1')

>>> timerSch.run('timer1', 1000, 0x00)

>>> timer1
    timer1
    timer1
    timer1
    timer1
    timer1
    timer1
    timer1
    timer1
```

Interrupts

Hier habe ich die Namensgebung getestet:

```
>>> @timerSch.event('micky_mouse')
    def tmicky_mouse():
        lcd.clear()
        print('micky_mouse')

>>> timerSch.run('micky_mouse', 1000, 0x00)

>>> micky_mouse
micky_mouse
micky_mouse
micky_mouse
micky_mouse
micky_mouse
micky_mouse
micky_mouse
micky_mouse
```

Interrupts

Im M5Stack Micropython gibt es 3 Methoden zur **Steuerung des Timers.**

`timerSch.setTimer('timer1', 100, 0x01)` setzt die Einstellungen für den Timer, startet diesen aber noch nicht.

`timerSch.run('timer1', 100, 0x00)` startet den Timer. Allerdings müssen auch hier die Einstellungen übergeben werden.

`timerSch.stop('timer1')` hält den Timer an.

Interrupts

Es gibt 3 Parameter für die Einstellung des Timers:

Name
Der Name des Timers ist frei wählbar. In der Funktionsdefinition muss ihm aber ein ‚**t**‘ voran gestellt werden.

Dauer
Die Timerzeit wird im **ms** angegeben werden.

Mode
Als Mode lässt sich periodisch **0x00** oder einmalig **0x01** einstellen.

Interrupts

```
>>> @timerSch.event('micky_mouse')
    def tmicky_mouse():
        x = 5 / 2
        lcd.clear()
        print('micky_mouse = '+ x)
>>> timerSch.run('micky_mouse', 1000,
0x00)

# es passiert nichts

>>> timerSch.stop('micky_mouse')
```

```
>>> @timerSch.event('micky_mouse')
    def tmicky_mouse():
        lcd.clear()
        print('micky_mouse = '+ 5/2)

>>> timerSch.run('micky_mouse', 1000,
0x00)

# es passiert nichts

>>> timerSch.stop('micky_mouse')
```

Interrupts

ENDE