Kapitel 8 • Interrupts und Timer

Im Mikrocontroller-Umfeld versteht man unter Interrupts (englisch to interrupt, "unterbrechen") temporäre Unterbrechungen eines ablaufenden Programms. Sie dienen dazu, einen meist zeitlich kritischen Vorgang abzuarbeiten. Typische Anwendungen sind Alarmmeldungen oder die Sicherheitsabschaltung von Komponenten und Maschinenteilen. Wird beispielsweise ein Generator elektronisch überwacht, so muss bei einer Übertemperaturmeldung eines Sensors unverzüglich reagiert werden. Zeitverzögerungen aufgrund einer aktuell hohen Prozessorlast können in diesem Fall katastrophale Folgen haben und sind daher nicht tolerierbar.

Auch für den Einsatz von Timern werden Interrupts benötigt. Hier geben eigne Hardwareeinheiten im Controller bestimmte Zeitintervalle vor. Sind diese abgelaufen, wird ein Interrupt ausgelöst. Damit kann unabhängig von der Auslastung des Hauptprozessorkerns ein sehr genaues Timing erreicht werden. Typische Anwendungen sind Uhren und Zeitsteuerungen aller Art. Die Präzision der Timer ist letztendlich nur von der Ganggenauigkeit des Prozessorquarzes abhängig. Da sich damit problemlos eine Präzision im Promillebereich erreichen lässt, kommen Timer in vielen praktischen Anwendungen zum Einsatz.

8.1 Unterbrechung erwünscht: Interrupts

Interrupts werden eingesetzt, wenn auf nicht vorhersehbare oder zeitlich unregelmäßige Ereignisse rasch reagiert werden muss. Wenn die Status-Änderung an einem interruptfähigen Pin erkannt wird, löst eine eigene Hardware-Einheit im Controller ein Interrupt-Ereignis aus. Damit ist es nicht notwendig, den entsprechenden Pin über eine Abfrage-Routine ständig zu überwachen. Nach der Auslösung des Interrupts kann dann eine vordefinierte Funktion aufgerufen werden, welche auf das Ereignis angemessen reagiert.

Nach dem Auftreten eines Interrupts stoppt der Prozessor die Ausführung des Hauptprogramms, um die Interrupt-Routine auszuführen. Erst wenn diese abgearbeitet sie, kehrt er zum Hauptprogramm zurück. Die folgende Abbildung verdeutlicht diesen Vorgang:

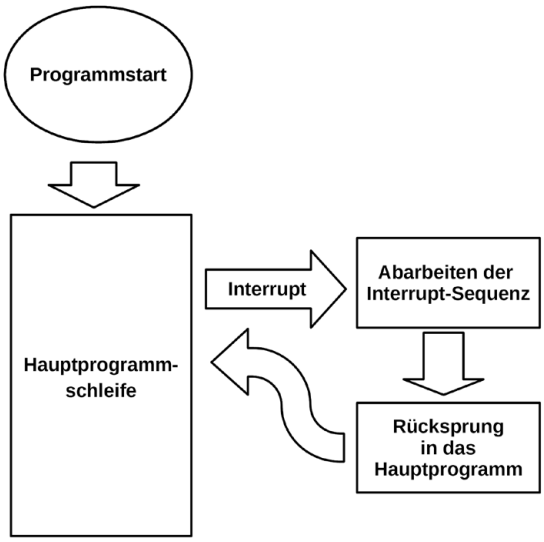


Abbildung 8.1: Interrupts

Interrupt-Routinen laufen also quasi-parallel zum Hauptprogramm ab. Allerdings handelt es sich hierbei nicht um eine echte Parallelverarbeitung. Der Prozessor arbeitet vielmehr das Hauptprogramm und die Interrupt-Routine nacheinander ab.

Interrupt-Routinen sollten daher prinzipiell möglichst kurz sein, d. h. ihre Abarbeitung sollte möglichst wenig Zeit in Anspruch nehmen. Innerhalb einer Interrupt-Routine dürfen also keine zeitaufwändigen Berechnungen erfolgen oder umfangreiche Programmschleifen etc. abgearbeitet werden.

8.2 Automatisches Nachtlicht

In diesem Abschnitt wird erläutert, wie Interrupts in MicroPython konfiguriert werden. Als Beispiel dient das Auslesen eines Tasters. Dieser soll eine LED für eine vorgegebene Zeitdauer einschalten. Anschließend soll die LED selbständig verlöschen. Anwendungen für diese Schaltung wären z. B. automatische Badezimmer- oder Treppenhausbeleuchtungen. Aber auch in bei Innenbeleuchtungen von Fahrzeugen finden solche Schaltungen Anwendung. Mit einer hellen weißen LED lässt sich das ESP-Board aber auch als automatische Nachttischlampe verwenden. Die folgende Abbildung zeigt das Schaltbild dazu.

Weitere Informationen zu den Hardware-Timern des ESP32 finden sich im nächsten Kapitel.

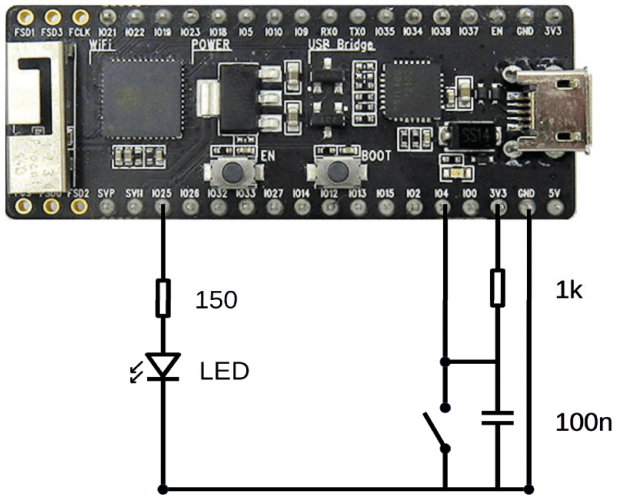


Abbildung 8.2: Aufbaubild zum automatisch Nachtlicht

Auch für den Einsatz von Interrupts wird zunächst wieder das Maschinenmodul importiert, mit dem auf die hardwarenahen Funktionen zugegriffen werden kann:

```
from machine import Pin
```

Zudem werden die Sleep-Methoden benötigt, da später im Programm eine Wartezeit erforderlich ist:

```
from time import sleep
```

Über eine Variable wird abgefragt, ob ein Tastendruck erfolgt ist. Diese Variable wird global definiert, da sie in der Interrupt-Handling-Funktion geändert wird. Es genügt, wenn diese die booleschen Werte True oder False annehmen kann. Die Initialisierung erfolgt mit False:

```
key_pressed = False
```

Im Falle eines Interrupts soll die folgende Routine abgearbeitet werden:

```
def interrupt_handler(pin):
  global key_pressed
  key_pressed = True
```

Diese Funktion wird bei jedem Tastendruck aufgerufen. Die interrupt_handle-Funktion hat einen Eingabeparameter (pin), in dem ein Objekt der Klasse Pin übergeben wird, wenn der Interrupt auftritt. Der Parameter gibt an, an welchem Pin der Interrupt ausgelöst wurde. Falls nur ein Interrupt-Pin verwendet wird, wird diese Information nicht benötigt. Wenn jedoch mehrere Interrupts dieselbe Interrupt-Behandlungsfunktion auslösen können, kann es durchaus von Interesse sein, welcher GPIO den Interrupt ausgelöst hat.

Im Beispiel ändert der interrupt_handler die Variabel key_pressed in True. Da in einer Interrupt-Funktionen die Bearbeitungszeit so kurz wie möglich gehalten werden soll, sind dort die Funktion wie sleep() oder print() möglichst zu vermeiden. Die Anweisungen, welche beim Auftreten des Interrupts auszuführen sind, sollten daher im Hauptprogramm stehen.

Damit eine Variable sowohl innerhalb der Funktion als auch im gesamten Code verwendet werden kann, muss sie als "global" deklariert werden. Andernfalls würde das Schalten der LED nicht funktionieren, da sich die Variablenänderung nur innerhalb der Funktion auswirken würde, nicht jedoch im Hauptprogramm.
Über

```
led = Pin(25, Pin.OUT)
button = Pin(4, Pin.IN)
```

werden die Pins für die LED und den Taster festgelegt. Die Anweisung

```
button.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)
```

bewirkt, dass immer beim Drücken des Taste ein Interrupt ausgelöst wird. Der Parameter "trigger" kann dabei die folgenden Werte annehmen:

- IRQ_RISNING: Auslösung bei steigender Flanke
- IRQ_FALLING: Auslösung bei fallender Flanke

Wird der Interrupt ausgelöst, springt der Controller in die Interrupt-Handler-Routine und die Variable key_pressed wird auf True gesetzt.

Nach dem Rücksprung in das Hauptprogramm wird dort dann die Meldung "Interrupt detected!" ausgegeben. Zusätzlich wird gemeldet, an welchem Pin der Interrupt ausgelöst wurde. Anschließend wird die LED für den vorgegebenen Zeitraum ein- und anschließend wieder ausgeschaltet. Abschließend wird key_pressed wieder auf False gesetzt und der Controller steht für den nächsten Interrupt zu Verfügung. Das vollständige Programm sieht damit so aus:

```
# LED_interrupt.py

from machine import Pin
from time import sleep

timeOn=3
key_pressed=False

def handle_interrupt(pin):
  global key_pressed
  key_pressed=True

led = Pin(25, Pin.OUT)
button1 = Pin(4, Pin.IN)
button1.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)

while True:
  if key_pressed:
    print(,Interrupt detected')
    led.value(1)
    sleep(timeOn)
    led.value(0)
    key_pressed = False
```

In diesem Beispiel wird der bereits bekannte sleep()-Befehl verwendet, um die LED drei Sekunden lang eingeschaltet zu lassen. An dieser Stelle wäre jedoch eine Timer-Anwendung die bessere Lösung. Wie und warum Timer eingesetzt werden, zeigt der nächste Abschnitt.

8.3 Herren der Zeit: Timer

Timer können für eine Vielzahl von Aufgaben verwendet werden. Zu den häufigsten Anwendungsfällen gehören:

- das periodische Aufrufen einer Funktion
- das Zählen von Ereignissen und
- das Erzeugen von PWM-Signalen

Jeder Timer besteht aus zwei 16-Bit-Kanälen, die zu einem 32-Bit-Timer verknüpft werden können. Die Betriebsart muss pro Timer konfiguriert werden, die Periode oder die Frequenz kann dagegen für jeden Kanal unabhängig gewählt werden. Mit der Callback-Methode kann

das Timer-Ereignis eine Python-Funktion aufrufen. Das folgende Programm lässt auf diese Weise eine LED blinken:

```
# LED_timer.py

import machine

led = machine.Pin(25, machine.Pin.OUT)
timer = machine.Timer(0)

def handleInterrupt(timer):
  led.value(not led.value())

timer.init(period=1000, mode=machine.Timer.PERIODIC,
callback=handleInterrupt)
```

Der Unterschied zum klassischen Blink-Programm mit sleep()-Funktion liegt darin, dass der Controller durch die Verwendung einer Timer-Funktion nicht blockiert wird. Während der Ausführung der sleep()-Funktion ist der Prozessorkern ausschließlich damit beschäftigt, Taktzyklen zu zählen. Der Kern kann darüber hinaus keine weiteren Aufgaben ausführen. Damit ist die sleep()-Funktion also "blockierend". Für die Timer stehen dagegen eigene Hardwareeinheiten zur Verfügung, die den Hauptprozessorkern nicht belasten.

Dass bei Verwendung des Interrupts der Prozessorkern für andere Aufgaben frei bleibt, zeigt sich auch daran, dass die Eingabeaufforderung >>> in der Shell erscheint. Man kann hier also simultan weiter arbeiten:

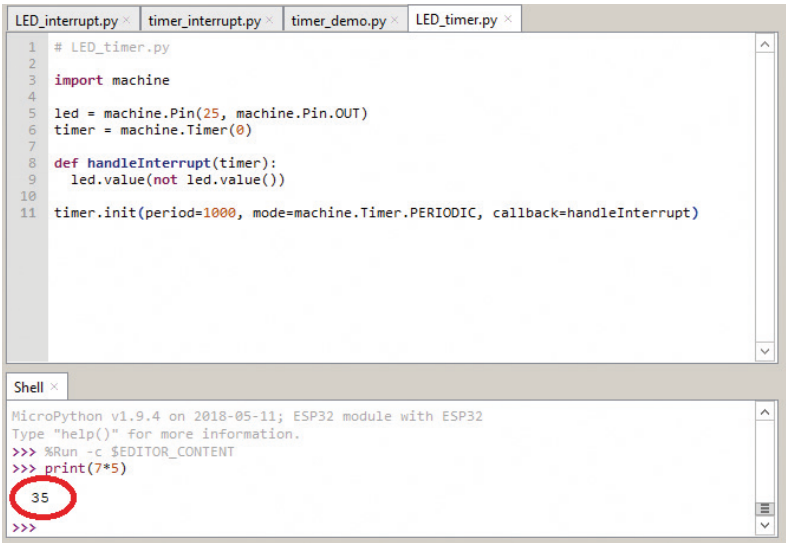


Abbildung 8.3: Laufender Timer und aktive Shell

Auch im Programm können weitere Funktionen laufen, währen der Timer aktiv ist. Der folgende Code bewirkt, dass led1 an Pin 25 mit einer Periode von 125 ms blinkt, während gleichzeitig led2 an Port 26 im Sekundentakt aufblitzt:

```
# LED_timer_and_mainprogram.py

from machine import Pin, Timer
from time import sleep

led1 = Pin(25, Pin.OUT)
led2 = Pin(26,Pin.OUT)
timer = Timer(0)

def handleInterrupt(timer):
  led1.value(not led1.value())

timer.init(period=125, mode=Timer.PERIODIC, callback=handleInterrupt)

while True:
  led2.value(1)
  sleep(.1)
  led2.value(0)
  sleep(1))
```

Die zweite LED muss hierfür mit einem Vorwiderstand an Port 26 angeschlossen werden.

Zum Abschluss zeigt das folgende Programm, dass unter Verwendung von zwei Timern zwei LEDs mit völlig unterschiedlichen Frequenzen geschaltet werden können:

```
# LED_double_timer.py

import machine

led1 = machine.Pin(25, machine.Pin.OUT)
led2 = machine.Pin(26, machine.Pin.OUT)

timer1 = machine.Timer(0)
timer2 = machine.Timer(1)

def handleInterrupt1(timer1):
  led1.value(not led1.value())
def handleInterrupt2(timer2):
  led2.value(not led2.value())

timer1.init(period=125, mode=machine.Timer.PERIODIC,
callback=handleInterrupt1)
timer2.init(period=517, mode=machine.Timer.PERIODIC,
callback=handleInterrupt2)
```

Diese relativ einfache Aufgabe wäre ohne Interrupts und Timer bereits nicht mehr praktikabel lösbar.

8.4 Ein multifunktionales Blinklicht

Rote Leuchtstrahler für Fahrräder sind heutzutage mit mehreren leistungsfähigen LEDs bestückt. Darüber hinaus bieten sie häufig die Möglichkeit, neben dem Dauerlichtbetrieb auch verschiedene Signalmuster einzustellen. Damit lässt sich eine erhöhte Aufmerksamkeit erzeugen. Fest am Fahrrad montierte blinkende Rücklichter sind zwar nicht legal, das Mitführen von Zweitlichtern am Helm oder am Rucksack ist dagegen unter Umständen gestattet. Man sollte sich aber in jedem Fall vor der Verwendung eines "Blinkies" im Straßenverkehr über die aktuelle Rechtslage informieren.

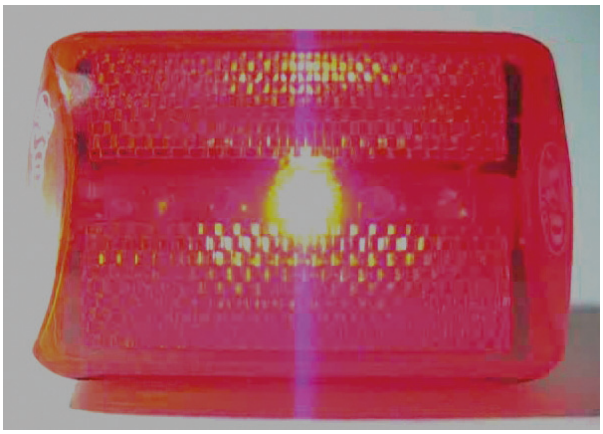


Abbildung 8.4: LED-Fahrradrücklicht

Mit dem ESP Controller kann leicht ein multifunktionales Blinklicht nachgebaut werden. Das projektierte "Rücklicht" soll die folgenden Leistungsmerkmale aufweisen:

- 5 LEDs
- Dauerlicht
- Blinken
- Blitzen
- Lauflichteffekt

Das Umschalten zwischen den einzelnen Betriebszuständen ist eine klassische Interrupt-Anwendung. Ohne den Einsatz der Interrupt-Technik wäre eine effiziente Umschaltsteuerung nicht möglich.

Das folgende Programm erfüllt die Aufgabe:

```
# bike light.py

from machine import Pin
from time import sleep

LED = [25,26,32,27,14]
```

```
del_time = 0.1
mode=1

def handle_interrupt(button1):
  global mode
  mode+=1
  if(mode>4):
    mode=1
  print(mode)

button1 = Pin(4, Pin.IN)
button1.irq(trigger=Pin.IRQ_RISING, handler=handle_interrupt)

for n in range(len(LED)):
  LED[n] = Pin(LED[n],Pin.OUT)

while True:
  if (mode==1):     # LEDs all on
    for n in range(len(LED)):
      LED[n].value(1)
  if (mode==2):    # LEDs blink
    for n in range(len(LED)):
      LED[n].value(1)
    sleep(del_time)
    for n in range(len(LED)):
      LED[n].value(0)
    sleep(del_time)
  if (mode==3):    # LEDs flash
    for n in range(len(LED)):
      LED[n].value(1)
    sleep(del_time)
    for n in range(len(LED)):
      LED[n].value(0)
    sleep(del_time*5)
  if (mode==4):    # LEDs chaser
    for n in range(len(LED)):
      LED[n].value(1)
      sleep(del_time)
      LED[n].value(0)
      sleep(del_time)
```

Da nun im Gegensatz zum einfachen Lauflicht aus Abschnitt 6.2 mehrere LEDs gleichzeitig aktiv sind, muss der Hardwareaufbau etwas modifiziert werden:

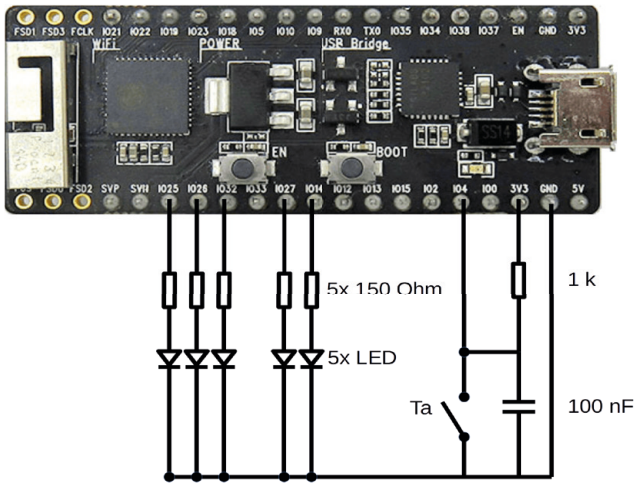


Abbildung 8.5: Schaltbild zum LED-Rücklicht

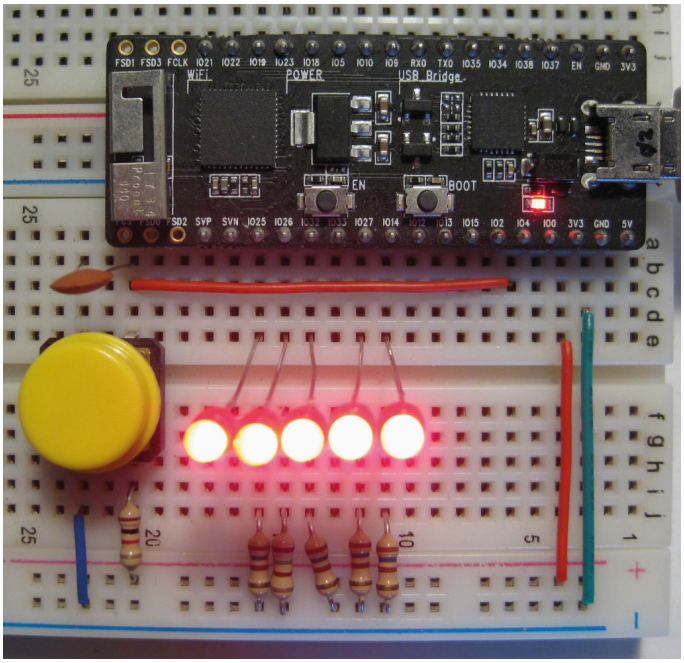


Abbildung 8.6:Aufbau zum LED-Blinkie

Zum Umschalten des Betriebsmodus wurde ein Taster an Pin 4 inklusive Pull-Up-Widerstand und Entprellungskondensator hinzugefügt. Zudem erhält nun jede LED ihren eigenen Vorwiderstand. Mit einem gemeinsamen Widerstand würden die einzelnen Leuchtdioden mit der Anzahl der eingeschalteten LEDs zunehmend weniger hell leuchten.