8 Timers en Interrupts

8.1 Timers in python

We kunnen de knop-acties zoals we die in hoofdstuk 7 hebben geleerd iets anders maken. Deze methode hoeft niet altijd beter te zijn, maar het is een andere manier om ermee om te gaan.

We beginnen met het uitleggen van deze methode met acties die iedere tijd moet worden uitgevoerd.

Als we iedere 10 seconden willen dat er iets op het scherm moet verschijnen kunnen we onderstaande code uitvoeren in python.

```
1 from time import sleep
2
3 while True:
4     sleep(10)
5     print("10 seconden later")
```

Als we nu iedere seconden een punt op het scherm willen, en iedere 10 seconden bovenstaande tekst moeten we van alles gaan verbouwen aan onze code.

```
1 from time import sleep
2
3 teller = 0
4 while True:
5     sleep(1)
6     print(".", end="")
7     teller += 1
8     if teller % 10 == 0:
9         print("10 seconden later")
```

Dit is allemaal mogelijk, maar als we nu nog iets willen toevoegen moet de gehele code omgegooid worden als je iedere halve seconden een ander teken op het scherm wilt hebben.

We gaan hiervoor een timer gebruiken. Deze timer zit in python NIET in time zoals je zou verwachten, maar in threading. Threading is een term dat acties tegelijk uitgevoerd kunnen worden.

Op regel 16 gaan we de functie `16 periodic_action() # Start the periodic action` uitvoeren. In die functie wordt de counter geprint op regel 12. Op regel 13 en 14 gebeurt alleen de magie van de timer. Er wordt een timer gestart die los staat van de rest van je programma. Als de timer afgaat na TIME_INVERVAL (in seconden) dan wordt de functie periodic_action() weer uitgevoerd.

Dit wordt op deze manier maar 1x uitgevoerd vanaf timer.start(). Om te zorgen dat dit iedere seconden gebeurt wordt deze timer iedere keer opnieuw aangemaakt en gestart.

Op regel 19, 20 en 21 zie je de lus die je vaker ziet. In deze lus hoeft dus niets te gebeuren. Tijdens de sleep van 20 seconden wordt er via de Timer steeds een getal op het scherm getoond.

De sleep op regel 20 hebben we nu op 20 seconden gezet. Regel 21 is erbij gezet zodat je kan zien dat iedere 20 seconden dat even wordt uitgevoerd, maar normaal laat je die regel weg.

```python
import threading
import time
TIME_INTERVAL = 1.0

_counter = 0
def print_counter():
    global _counter
    print(_counter%10, end="")
    _counter += 1

def periodic_action():
    print_counter()
    timer = threading.Timer(TIME_INTERVAL, periodic_action)  # Schedule next execution
    timer.start()

periodic_action() # Start the periodic action

try: # Keep the main thread alive so the periodic action can run
    while True:
        time.sleep(20)  # Keep the main thread alive
        print("\n----")
except KeyboardInterrupt: #CTRL-C
    print("Program interrupted by user.")

```

Uit onderstaande regel kan je zien wat het effect is op de CPU als je de timer te kort gaat zetten.

```python
import time
while True: time.sleep(0.2)      # 0% CPU
while True: time.sleep(0.02)     # 0% CPU
while True: time.sleep(0.002)    # 0.5% CPU
while True: time.sleep(0.0002)   # 6% CPU
while True: time.sleep(0.00002)  # 18% CPU
```

We kunnen deze kennis nu omzetten naar het programma dat we eerder al hadden gemaakt. Willen we en nog iedere halve seconden iets bijzetten, hoeven we alleen een nieuwe functie aan te maken.

```python
import threading
import time
TIME_INTERVAL_1 = 1.0
TIME_INTERVAL_2 = 10

def periodic_action_1():
    print(".", end="")
    timer = threading.Timer(TIME_INTERVAL_1, periodic_action_1)
    timer.start()

def periodic_action_2():
    print("10 seconden later")
    timer = threading.Timer(TIME_INTERVAL_2, periodic_action_2)
    timer.start()

periodic_action_1() # Start the periodic action
periodic_action_2() # Start the periodic action

try: # Keep the main thread alive so the periodic action can run
    while True:
        time.sleep(20)  # Keep the main thread alive
except KeyboardInterrupt: #CTRL-C
    print("Program interrupted by user.")

```

8.2 Timers in microPython

Via microPython kunnen we dit nog mooier doen.

Op regel 9 wordt een timer gemaakt met de index 0. Op regel 11 wordt die timer geïnitieerd. Er staat bij mode dat dit periodiek uitgevoerd moet worden. Dat betekent dat dit standaard steeds wordt hervat. Je had hier ook ONE_SHOT kunnen kiezen.

De callback-functie is de functie die wordt uitgevoerd als de timer afgaat. Ten slotte period is de tijd in milliseconden dat de timer afgaat.

```
1  import machine
2  import utime
3
4  # Function to be executed by the timer
5  def periodic_action(timer):
6      print("Performing periodic action...") # Add the action you want to perform here
7
8  # Create a timer object (Timer 0)
9  timer = machine.Timer(0)
10 # Configure the timer to call the periodic_action function every 5 seconds
11 timer.init(period=5000, mode=machine.Timer.PERIODIC, callback=periodic_action)
12
13 # Keep the main program running indefinitely
14 try:
15     while True:
16         utime.sleep(10)  # Sleep to reduce CPU usage
17 except KeyboardInterrupt:
18     print("Program interrupted by user.")
```

Ook hier kunnen we zien dat we op regel 16 de microcomputer laten slapen, en er niets op de gewone lus gedaan moet worden. Er is geen beperking op de hoeveelheid timers. De beperking zit op het geheugen en de rekenkracht van je microcomputer.

Let op dat voor iedere timer die je nu maak, je ook een nieuwe id moet meegeven.

In de documentatie van microPython staat dat er alternatieven zijn voor regel 16, de sleep van je microcomputer.

machine.idle()

Gates the clock to the CPU, useful to reduce power consumption at any time during short or long periods. Peripherals continue working and execution resumes as soon as any interrupt is triggered (on many ports this includes system timer interrupt occurring at regular intervals on the order of millisecond).

machine.sleep()

> **Note**
>
> This function is deprecated, use `lightsleep()` instead with no arguments.

machine.lightsleep([*time_ms*])

machine.deepsleep([*time_ms*])

Stops execution in an attempt to enter a low power state.

If *time_ms* is specified then this will be the maximum time in milliseconds that the sleep will last for. Otherwise the sleep can last indefinitely.

With or without a timeout, execution may resume at any time if there are events that require processing. Such events, or wake sources, should be configured before sleeping, like `Pin` change or `RTC` timeout.

The precise behaviour and power-saving capabilities of lightsleep and deepsleep is highly dependent on the underlying hardware, but the general properties are:

- A lightsleep has full RAM and state retention. Upon wake execution is resumed from the point where the sleep was requested, with all subsystems operational.
- A deepsleep may not retain RAM or any other state of the system (for example peripherals or network interfaces). Upon wake execution is resumed from the main script, similar to a hard or power-on reset. The `reset_cause()` function will return `machine.DEEPSLEEP` and this can be used to distinguish a deepsleep wake from other resets.

8.3 Interrupts in Python

Interrupts werken gelijk aan de timers die je hiervoor hebt gezien. Als er een toets ingedrukt wordt, wordt er een actie uitgevoerd. Zolang de toets niet ingedrukt wordt zal de code niets doen, en in slaapmodes zitten.

We kunnen dit tegenwoordig in python alleen nog op de onderstaande methode gebruiken. In verband met veiligheid mag je niet zomaar de toetsaanslagen afvangen. Je zou op deze manier namelijk een programma kunnen maken die “meeluistert” wat een gebruiker aan het intypen is. In microPython kunnen we hier gelukkig wel volop gebruik van maken.

```
1  import signal
2  from time import sleep
3
4  def signal_handler(sig, frame):
5      print('@', end="")
6
7  # Interrupt from keyboard SIGINT => (CTRL + C).
8  signal.signal(signal.SIGINT, signal_handler)
9
10 # druk Ctrl+C voor melding
11 while True:
12     print(".", end="")
13     sleep(60)
14
```

8.4 Interrupts in microPython

MicroPython heeft geen beperkingen op de interrupts. Dit komt ook grotendeels dat er standaard geen toetsenbord is aangesloten op een microComputer.

Op de microcomputer zal je de interrups voornamelijk gebruiken om veranderingen op pinnen te detecteren. Als een drukknop wordt ingedrukt, en een pin wordt van LAAG naar HOOG wil je dat er direct een actie ondernomen wordt. Als je dit is een lus zou opnemen moet je wachten dat de code op het punt is beland dat de knop wordt uitgelezen.

De callback functie op regel 5 moet altijd de parameter pin hebben. Pin wordt bij het aanroepen automatisch meegegeven, en op deze manier krijg je als argument mee welke pin de actie heeft getriggerd.

Op regel 9 wordt op de bekende manier het gebruik van een pin geïnitieerd.

Op regel 10 wordt de irq gezet (interrupt request pin). *trigger=machine.Pin.IRQ_FALLING | machine.Pin.IRQ_RISING* zorgt ervoor dat de trigger afgaat als de knop wordt ingedrukt of de knop wordt losgelaten. Het streepje omhoog is een logische OR, en zegt dus dat of het ene of het andere moet gebeuren.

Let op dat de aanroep van *irq* per microcomputer kan verschillen. Raadpleeg hiervoor altijd de documentatie van de microcomputer en de versie van microPython.

```python
import machine
import time

# Define the ISR (Interrupt Service Routine)
def pin_interrupt_handler(pin):
    print("Interrupt triggered by pin", pin)

# Configure an interrupt on GPIO pin 5, triggered by pin change
interrupt_pin = machine.Pin(5, machine.Pin.IN, machine.Pin.PULL_UP)
interrupt_pin.irq(trigger=machine.Pin.IRQ_FALLING | machine.Pin.IRQ_RISING,
                  handler=pin_interrupt_handler)

try:
    while True:
        time.sleep_ms(100) # Your main program logic here
except KeyboardInterrupt:
    print("Program interrupted by user.")

```

8.5 Opdrachten

Onderstaande opdrachten heb je al gemaakt in hoofdstuk 7. Deze keer moet je de opdracht maken met gebruik te maken van timers en interrupts.

8.5.1 Indrukken tellen

Maak een schakeling die telt hoeveel keer je de knop hebt ingedrukt. Let dus op dat als je de knop ingedrukt houdt, dit maar 1x geteld moet worden. Gebruik timers en interrupts.

8.5.2 Links of rechts

Sluit twee drukknoppen aan. Led 8 staat in het begin aan. Als je op de ene drukknop druk gaat de Led die aanstaat 1 plaats naar links, Als je op de andere drukknop druk gaat de led die aanstaat 1 naar rechts. Als Led 0 aan staat, kan zal deze blijven branden als de knop naar links ingedrukt wordt. Als Led 15 aanstaan zal deze blijven branden als de knop naar rechts wordt ingedrukt. Gebruik timers en interrupts.

8.5.3 Spel met Knoppen en LEDs

Sluit twee drukknoppen aan op de pico. Alle LEDs op de break-out board knipperen om de seconden. 1 knop is om het spel opnieuw op te starten (begint langzaam). 6 Leds Gaan willekeurig aan en uit. De leds blijven in eerste instantie 3 seconden aan, en dan gaan ze uit en gaat de volgende aan. Als LED 2 of LED 4 aan gaat moet op de andere knop gedrukt worden. Als je op tijd ben dan gaat de volgende .1 seconde sneller uit (2.9 seconden aan). Dit gaat net zolang door dat de snelheid 0.5 seconden is.

Als je te laat ben met indrukken… Dan gaan alle LEDs tegelijk knipperen om de seconden, totdat de startknop weer is ingedrukt. Maak eerst een flowchart van het spel, en daarna de code van het spel.

Gebruik timers en interrupts.