

# Lab 3: Timers

---

## Learning objectives

After completing this lab you will be able to:

- Use internal microcontroller timers
- Understand overflow and PWM
- Combine different interrupts in MicroPython

The purpose of this laboratory exercise is to acquire the skills to interact with timers and interrupts of the ESP32 microcontroller.

## Table of contents

- [Pre-Lab preparation](#)
- [Part 1: Interrupts](#)
- [Part 2: ESP32 timer overflows](#)
- [Part 3: PWM and LED dimming](#)
- [\(Optional\) Experiments on your own](#)
- [References](#)

## Components list

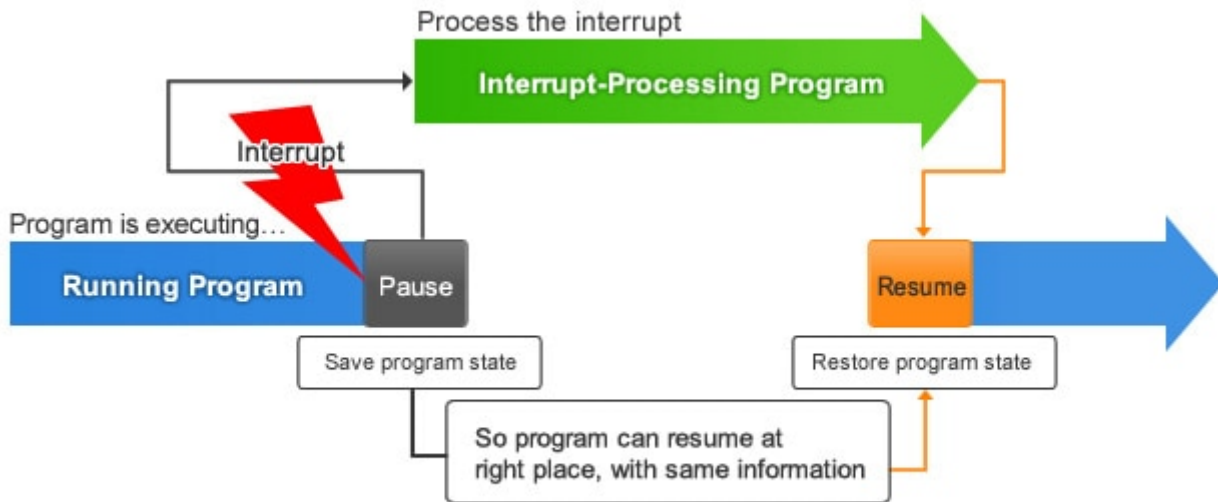
- ESP32 board, USB cable
- Breadboard
- 3 LEDs, 3 resistors
- Jumper wires

## Pre-Lab preparation

1. Remind yourself how to write and run functions in MicroPython.

## Part 1: Interrupts

Interrupts can be triggered by both internal and external devices within the microcontroller unit (MCU). It represents a signal **Interrupt request** sent to the processor by hardware or software, signifying an event that requires immediate attention. When an interrupt is triggered, the controller finishes executing the current instruction and proceeds to execute an **Interrupt Service Routine (ISR)** or Interrupt Handler. ISR tells the processor or controller what to do when the [interrupt occurs](#). After the interrupt code is executed, the program continues exactly where it left off.



Interrupts can be set up for events such as a counter's value, a pin changing state, receiving data through serial communication, or when the Analog-to-Digital Converter has completed the conversion process.

## Part 2: ESP32 timer overflows

A timer (or counter) is an integral hardware component in a microcontroller unit designed for measuring time-based events. Timers count from 0 to  $(2^n - 1)$ , where  $n$  being the number of bits of the counter. Thus, an 8-bit counter will count from 0 to 255, a 16-bit counter will count from 0 to 65535, and so on.

ESP32 has two timer groups, each group containing two 64-bit timers. Thus, there are four 64-bit timers in total, designated as Timer0, Timer1, and Timer2. They are all 64-bit generic timers based on 16-bit prescalers and 64-bit up/down counters which are capable of being auto-reloaded.



Prescalers help divide the base clock frequency. ESP32 generally has a base clock frequency of 80 MHz and it can be a bit too high. Having a 16-bit pre-scaler means that you can divide the base clock frequency by at least 2 and by as much as 65536.  $80 \text{ MHz} / 65535 = 1.22 \text{ KHz}$ . Now, this means that the frequency of a timer can be adjusted from 1.22 KHz to 80 MHz (of course in discrete steps). This wide range of frequency, along with the fact that these are 64-bit timers ensures that almost any interval is possible with ESP32 timers.

The timer interrupts are the best way to run non-blocking functions at a certain interval. For that, we can configure and attach a particular timer interrupt to a specific Interrupt Service Routine or ISR.

The MicroPython's `machine.Timer` class defines a baseline operation of initializing and executing a callback with a given period or once after some delay.

```
from machine import Timer
tim = Timer(0) # Timer0

def mycallback(t):
    pass

# Periodic at 1kHz
tim.init(mode=Timer.PERIODIC, freq=1000, callback=mycallback)

# Periodic with 100ms period
tim.init(period=100, callback=mycallback)

# One shot firing after 1000ms
tim.init(mode=Timer.ONE_SHOT, period=1000, callback=mycallback)
```

The init keyword arguments are:

- `mode` can be `Timer.ONE_SHOT` or `Timer.PERIODIC`
- `freq` is the timer frequency, in units of Hz
- `period` is the timer period in milliseconds
- `callback` is executed whenever a timer is triggered. The callback must take one argument, which is passed the Timer object.

To blink the on-board LED with a period of 1 sec, use the following code:

```
from machine import Pin, Timer

pin_led = Pin(2, mode=Pin.OUT)
timer0 = Timer(0) # Between 0-3 for ESP32

def interruption_handler(timer):
    pin_led.value(not pin_led.value())

timer0.init(mode=Timer.PERIODIC, period=1000,
callback=interruption_handler)
```

1. Use breadboard, jumper wires and connect three LEDs and resistors to ESP32 GPIO pins in active-high way. Use GPIO pin numbers 3, 1, 25.
2. Create a new source file, write the code for continuous blinking of all LEDs, save the file as `01-blink_leds.py` to your local folder, and run the application.

```
# Load `Pin` class from `machine` module to access hardware
from machine import Pin, Timer
from time import sleep_ms

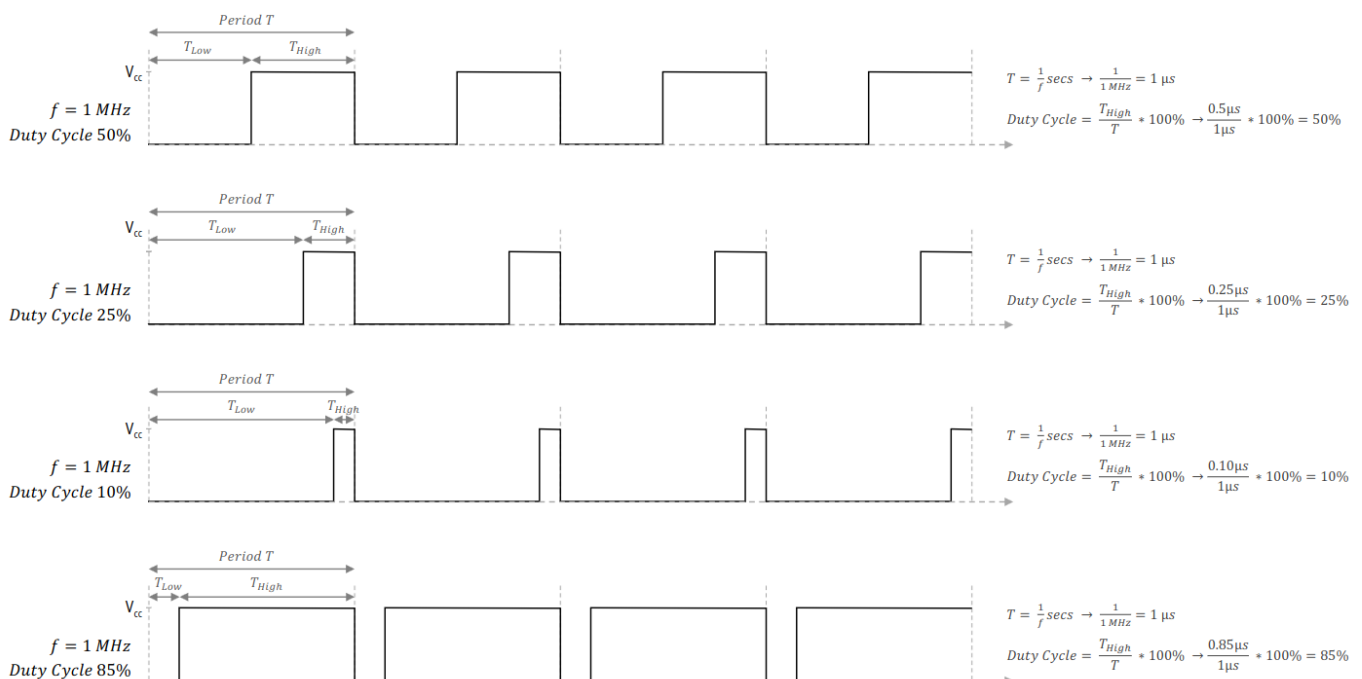
# Define three LED pins
led0 = Pin(3, Pin.OUT)
timer0 = Timer(0)
timer0.init(mode=Timer.PERIODIC, period=1000,
callback=tim0_interrupt_handler)

# COMPLETE THE CODE
```

## Part 3: PWM and LED dimming

**Pulse Width Modulation (PWM)** changes the average power delivered from a signal by chopping the square wave signal into discrete parts. It is not an actual continuous signal. Instead, a periodic digital signal's ON, and OFF duration is modulated to reduce the effective voltage/power output. For example, if a microcontroller's GPIO outputs 3.3V in digital output while generating a PWM signal of **50% duty cycle** from the pin, it will output an effective voltage of approximately 1.65V, i.e.,  $3.3/2$ .

### DIFFERENT DUTY CYCLE, SAME FREQUENCY (1MHZ)



The PWM class is written to provide pulse width modulation in MicroPython supported boards. This class can be imported into a MicroPython script using the following statements.

```
from machine import PWM
```

After importing, an object has to be instantiated from the PWM class. For this, `machine.PWM()` method is provided. This method has the following syntax.

```
class machine.PWM(dest, *, freq, duty_u16, duty_ns)
```

1. Create a new source file, write the code for change a duty cycles of one LED, save the file as `03-dimmer.py` to your local folder, and run the application.
2. Add another two PWM channels and control dimming of all three LEDs.

## (Optional) Experiments on your own

1. To combine timers and PWM in MicroPython on an ESP32, create a smooth fading effect using an RGB LED or two-colour LED. By controlling the intensity of the red, green, and blue components, you can achieve various colors and dynamic color transitions. Timers will be used to control the PWM signals for each color component.

## References

1. MicroPython documentation. [class Timer -- control hardware timers](#)
2. Physical Computing. [Lesson 3: Fading an LED with PWM](#)
3. Nikhil Agnihotri. [MicroPython -- Generating PWM on ESP8266 and ESP32](#)
4. MicroPython documentation. [class PWM -- pulse width modulation](#)