

# **Micropython on STM32F401RE**

## INDEX:

<b>1. Local filesystem and SD card</b>	<b>3</b>
2. Using the REPL prompt	4
2.1 Resetting the Board	5
<b>3. Turning on LEDs</b>	<b>5</b>
3.1 Led.py	5
<b>4. Switch()</b>	<b>6</b>
4.1 Switch callback	7
4.2 Technical details of interrupts	8
<b>5. Timers</b>	<b>9</b>
5.1 Timer counter	10
5.2 Timer callbacks	11
5.3 Microsecond counter	11
<b>6. Power control</b>	<b>13</b>
<b>7. UART</b>	<b>13</b>
<b>8. GPIO</b>	<b>14</b>
8.1 GPIO OUPUT(led blinking)	14
8.2 GPIO INPUT(push button)	15
<b>9. RTC(Real time clocks)</b>	<b>16</b>
<b>10. PWM</b>	<b>16</b>
10.1 What is Pulse Width Modulation (PWM) and when is it used?	16
10.2 What is duty cycle? Hertz?	17
10.3 Fading LEDs	18
<b>11. INTERRUPTS</b>	<b>19</b>
11.1 Multiple Interrupts	21
11.1.1 Interrupt related functions	21

# 1. Local filesystem and SD card

There is a small internal filesystem(drive) on the pyboard ,called /flash,which is stored within the microcontroller's flash memory.

```
(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> ampy --port COM7 --baud 115200  
ls  
  
/flash
```

- If a micro SD card is inserted into the slot , it is available as /sd
- When the board boots up , it need to choose the filesystem to boot from.if there is no SD card,them it uses the internal filesystem /flash as the boot filesystem.Otherwise it uses the SD card /sd. After boot the current directory is set to one of the directories above
- If needed , you can prevent the use of the SD card by creating an empty file called /flash/SKIPSD. If this file exists when the pyboard boots up then the SD card will be skipped and the pyboard will always boot from the internal filesystem
- The boot filesystem is used for 2 things: it is the filesystem from which the boot.py and main.py files are searched for, and it is the filesystem which is made available on your PC over the USB cable.

```
(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> ampy --port COM7 --baud 115200  
ls /flash  
  
/flash/boot.py  
  
/flash/main.py
```


If you power up normally, or press the reset button, the pyboard will boot into standard mode: the boot.py file will be executed first, then the USB will be configured, then main.py will run.

- boot.py – the various configuration options for the pyboard. It is executed when the pyboard boots up.
- main.py – the Python program to be run. It is executed after boot.py.

## 2. Using the REPL prompt

With your serial program open (puTTY) you may see a blank screen with a flashing cursor

Press the reset button on the board you will find the REPL of micropython, you should be able to find the Micropython prompt i.e, >>>



```
COM7 - PuTTY
MicroPython v1.23.0-preview.379.gcf5a8ea3 on 2024-05-23; NUCLEO-F401RE with STM32F401xE
Type "help()" for more information.
>>> print("hello")
hello
```

In the above, you should not type in the `>>>` characters. They are there to indicate that you should type the text after it at the prompt. In the end, once you have entered the text `print("hello pyboard!")` and pressed Enter, the output on your screen should look like it does above.

If you already know some python you can now try some basic commands here. If any of this is not working you can try either a hard reset or a soft reset; see below.

Go ahead and try typing in some other commands. For example:

[illegible]

## 2.1 Resetting the Board

If something goes wrong, you can reset the board in two ways. The first is to press CTRL-D at the MicroPython prompt, which performs a soft reset. You will see a message something like

```
>>>
MPY: sync filesystems
MPY: soft reboot
MicroPython v1.23.0-preview.379.gcfd5a8ea3 on 2024-05-23; NUCLEO-F401RE with STM32F401xE
Type "help()" for more information.
>>> █
```

If that isn't working you can perform a hard reset (turn-it-off-and-on-again) by pressing the RST switch (the small black button closest to the micro-USB socket on the board). This will end your session, disconnecting whatever program (PuTTY, screen, etc) that you used to connect to the board

### 3. Turning on LEDs

1. The easiest thing to do on the pyboard is to turn on the LEDs attached to the board. Connect the board, We will start by turning and LED on in the interpreter, type the following

```
>>> myled = pyb.LED(1)
>>> myled.on()
>>> myled.off()
>>> █
```

2. To run the code by using any “.py” .
  - Open the pycharm and write the required code to toggle the LED
  - Install the ampy using “pip install Adafruit-ampy” in the terminal of the venv of pycharm
  - Using the command “ampy –port COM7 run led.py”

#### 3.1 Led.py

```
led = pyb.LED(1)
while True:
    led.toggle()
    pyb.delay(1000)
```

## 4. Switch()

- The STM32 board has 2 small switches , called USR(blue button) and RST(black button).
- The RST switch is a hard-reset switch, and if you press it restarts the board from scratch, equivalent to turning the power off then back on.
- The USR switch is for general use, and is controlled via a switch object. To make switch object do:  
`>>> sw = pyb.Switch()`
- With the switch object you can get its status:  
`>>> sw.value()`
- There is also a shorthand notation to get the switch status, by “callig” the switch object:  
`>>> sw()`

```
import pyb
sw = pyb.Switch()
while True:
    print(sw.value())
    pyb.delay(1000)
```

### OUTPUT:

(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> ampy --port COM7 ru

n push\_button.py

False

False

False

True

True

True

False

False

False

True

False

True

False

False

- So, when you press the push button it will print “True” otherwise it will print “False”

## 4.1 Switch callback

The switch is a very simple object, but it does have one advanced feature: the `sw.callback()` function. The callback function sets up something to run when the switch is pressed, and uses an interrupt.

### Example 1:

```
>>> sw.callback(lambda:print('press!'))
```

```
sw = pyb.Switch()
while True:
    sw.callback(lambda:print('pressed!!'))
    pyb.delay(1000)
```

### OUTPUT:

```
(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> ampy --port COM7 ru
```

```
n switch_callback.py
```

```
pressed!!
```

```
pressed!!
```

```
pressed!!
```

```
pressed!!
```

```
pressed!!
```

```
pressed!!
```

This tells the switch to print `press!` each time the switch is pressed down. Go ahead and try it: press the USB switch and watch the output on your PC. Note that this print will interrupt anything you are typing, and is an example of an interrupt routine running asynchronously.

### Example 2:

```
>>> sw.callback(lambda:pyb.LED(1).toggle())
```

```
sw = pyb.Switch()
#method1
```

```
while True:
    sw.callback(lambda:pyb.LED(1).toggle())
```

This will toggle the red LED each time the switch is pressed. And it will even work while other code is running.

To disable the switch callback, pass None to the callback function:

```
>>> sw.callback(None)
```

You can pass any function (that takes zero arguments) to the switch callback. Above we used the lambda feature of Python to create an anonymous function on the fly. But we could equally do:

```
sw = pyb.Switch()
def led():
    pyb.LED(1).toggle()

while True:
    sw.callback(led)
```

This creates a function called `f` and assigns it to the switch callback. You can do things this way when your function is more complicated than a lambda will allow.

Note that your callback functions must not allocate any memory (for example they cannot create a tuple or list). Callback functions should be relatively simple. If you need to make a list, make it beforehand and store it in a global variable (or make it local and close over it). If you need to do a long, complicated calculation, then use the callback to set a flag which some other code then responds to.

## 4.2 Technical details of interrupts

Let's step through the details of what is happening with the switch callback. When you register a function with **sw.callback()**, the switch sets up an external interrupt trigger (falling edge) on the pin that the switch is connected to. This means that the microcontroller will listen on the pin for any changes, and the following will occur:

1. When the switch is pressed a change occurs on the pin (the pin goes from low to high), and the microcontroller registers this change.
2. The microcontroller finishes executing the current machine instruction, stops execution, and saves its current state (pushes the registers on the stack). This has the effect of pausing any code, for example your running Python script.
3. The microcontroller starts executing the special interrupt handler associated with the switch's external trigger. This interrupt handler gets the function that you registered with `sw.callback()` and executes it.
4. Your callback function is executed until it finishes, returning control to the switch interrupt handler.



5. The switch interrupt handler returns, and the microcontroller is notified that the interrupt has been dealt with.
6. The microcontroller restores the state that it saved in step 2.
7. Execution continues of the code that was running at the beginning. Apart from the pause, this code does not notice that it was interrupted.

The above sequence of events gets a bit more complicated when multiple interrupts occur at the same time. In that case, the interrupt with the highest priority goes first, then the others in order of their priority. The switch interrupt is set at the lowest priority.

## 5. Timers

The STM32F401RE microcontroller, which is the main component of the Nucleo-F401RE board, comes with several timers with different functionalities. Here's a general overview of the timers available on the STM32F401RE:

1. **General-Purpose Timers (TIM1, TIM2, TIM3, TIM4):** These timers offer versatile timing and control capabilities suitable for a wide range of applications.
2. **Basic Timers (TIM6, TIM7):** Basic timers are simpler in functionality compared to general-purpose timers but can still be useful for basic timing tasks.
3. **Advanced-Control Timers (TIM1, TIM8):** These timers offer additional features suitable for advanced control applications, such as motor control or power conversion.
4. **Low-Power Timers (LPTIM1, LPTIM2):** Low-power timers are designed to operate in low-power modes and are suitable for applications requiring precise timing with low energy consumption.
5. **Pulse-Width Modulation (PWM) Timers (TIM1, TIM2, TIM3, TIM4, TIM5):** These timers can generate PWM signals, which are commonly used for controlling motor speed, LED brightness, and other analog-like functions.

- Let's create a timer object:

```
>>> tim = pyb.Timer(1)
```

- Now let's see what we just created:

```
>>> tim.Timer(1)
```

- The pyboard is telling us that tim is attached to timer number 4, but it's not yet initialised. So let's initialise it to trigger at 10 Hz (that's 10 times per second):

```
>>> tim.init(freq=10)
```

- Now that it's initialised, we can see some information about the timer:

```
>>> tim
```

```
Timer(4, prescaler=624, period=13439, mode=UP, div=1)
```

## 5.1 Timer counter

So what can we do with our timer? The most basic thing is to get the current value of its counter:

```
>>> tim.counter() 21504
```

This counter will continuously change, and counts up

```
tim = pyb.Timer(1)
print(tim)
tim.init(freq=10)
print(tim)
print(tim.source_freq())

#timer counter
print(tim.counter())
```

### OUTPUT:

(.venv) PS C:\Users\vlav\PycharmProjects\MicroPython>ampy --port COM7 run timer.py

```
Timer(1)
```

```
Timer(1, freq=10, prescaler=624, period=13439, mode=UP, div=1, deadtime=0,
brk=BRK_OFF)
```

```
84000000
```

```
1201
```

- The information means that this timer is set to run at the peripheral clock speed divided by 624+1, and it will count from 0 up to 13439, at which point it triggers an interrupt, and then starts counting again from 0. These numbers are set to make the timer trigger at 10 Hz: the source frequency of the timer is 84MHz (found by running `tim.source_freq()`) so we get  $84\text{MHz} / 625 / 13440 = 10\text{Hz}$ .

## 5.2 Timer callbacks

The next thing we can do is register a callback function for the timer to execute when it triggers (see the switch tutorial for an introduction to callback functions):

```
>>> tim.callback(lambda t:pyb.LED(1).toggle())
```

This should start the red LED flashing right away. It will be flashing at 5 Hz (2 toggle's are needed for 1 flash, so toggling at 10 Hz makes it flash at 5 Hz). You can change the frequency by re-initialising the timer:

```
>>> tim.init(freq=20)
```

You can disable the callback by passing it the value None:

```
>>> tim.callback(None)
```

Because the callbacks are proper hardware interrupts, we can continue to use the board for other things while these timers are running

```
import pyb

def f():
    pyb.LED(1).toggle()

# Initialize Timer 1 with a frequency of 20 Hz
tim1 = pyb.Timer(1, freq = 20)

# Set the callback function for Timer 1
tim1.callback(f)
```

## 5.3 Microsecond counter

You can use a timer to create a microsecond counter, which might be useful when you are doing something which requires accurate timing. We will use timer 2 for this, since timer 2 has a 32-bit counter (so does timer 5, but if you use timer 5 then you can't use the Servo driver at the same time).

We set up timer 2 as follows:

```
>>> micros = pyb.Timer(2, prescaler=83, period=0x3fffffff)
```

The prescaler is set at 83, which makes this timer count at 1 MHz. This is because the CPU clock, running at 168 MHz, is divided by 2 and then by prescaler+1, giving a frequency of  $168 \text{ MHz} / 2 / (83 + 1) = 1 \text{ MHz}$  for timer 2. The period is set to a large number so that the timer can count up to a large number before wrapping back around to zero. In this case it will take about 17 minutes before it cycles back to zero.

**Period = (CPU frequency / (Frequency × (Prescaler + 1))) - 1**

To use this timer, it's best to first reset it to 0:

**>>> `micros.counter(0)` and then perform your timing:**

**>>> `start_micros = micros.counter()`**

**... do some stuff ...**

**>>> `end_micros = micros.counter()`**

**Example:**

```
micros = pyb.Timer(2, prescaler=0, period=0xffff)
micros.counter(0)

start_micros = micros.counter()
for i in range(5):
    print(i)
end_micros = micros.counter()

elapsed_micros = end_micros - start_micros
print("Elapsed time:", elapsed_micros, "microseconds")
```

**OUTPUT:**

**(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> `ampy --port COM7 run microsecond_counter.py`**

0

1

2

3

4

Elapsed time: 2416 microseconds

This code snippet is setting up Timer 2 on a board (assuming you're using MicroPython on a board) with a prescaler of 0 and a period of 65535 (0xFFFF). Then it's using the timer to measure the elapsed time in microseconds for a loop to execute.

Here's a breakdown of what each part does:

1. **`micros = pyb.Timer(2, prescaler=0, period=0xffff)`:** This line initializes Timer 2 with a prescaler of 0 and a period of 65535 (0xFFFF). This configuration will make Timer 2 count up to its maximum value before overflowing and resetting.
2. **`start_micros = micros.counter()`:** This line reads the current value of Timer 2, storing it in **`start_micros`**.

3. **for i in range(5): print(i):** This loop iterates five times, printing the value of **i** in each iteration. This loop is simply meant to simulate some code execution.
4. **end\_micros = micros.counter():** This line reads the current value of Timer 2 after the loop execution, storing it in **end\_micros**.
5. **elapsed\_micros = end\_micros - start\_micros:** This line calculates the difference between the **end\_micros** and **start\_micros**, giving the elapsed time in microseconds.

## 6. Power control

- **pyb.wfi()** is used to reduce power consumption while waiting for an event such as an interrupt. You would use it in the following situation:

**while True:**

**do\_some\_processing()**

**pyb.wfi()**

- Control the frequency using **pyb.freq()**:

**pyb.freq(30000000)** # set CPU frequency to 30MHz

## 7. UART

```
import machine
import time

# Initialize UART (use UART2 which is available on the Nucleo-F401RE)
uart = machine.UART(2, baudrate=115200)

# Main loop to send "hello" over UART
while True:
    uart.write('hello\n') # Send the message "hello"
    time.sleep(1) # Wait for 1 second
    if uart.any(): # Check if there is any incoming data
        msg = uart.read() # Read the received data
        print(msg)
```

**OUTPUT:**

(.venv) PS C:\Users\vlav\PycharmProjects\MicroPython> **ampy --port COM7 run uart.py**

hello

hello

hello

hello

hello

hello

hello

hello

hello

Aborted!

## 8. GPIO

### 8.1 GPIO OUTPUT(led blinking)

```
import machine
import time

# Configure PA5 as an output
led = machine.Pin('PA5', machine.Pin.OUT)

while True:
    led.value(1)  # Turn the LED on
    print("led turned on: ", led.value())
    time.sleep(1) # Wait for 1 second
    led.value(0)  # Turn the LED off
    print("led turned off: ", led.value())
    time.sleep(1) # Wait for 1 second
```

This script creates a simple blinking LED effect, where the LED turns on for one second and then turns off for one second, repeating indefinitely. Adjust the timings in the **time.sleep** calls as needed to change the blink rate.

#### OUTPUT:

(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> [ampy --port COM7 run GPIO\\_output.py](#)

led turned on: 1

led turned off: 0

led turned on: 1

led turned off: 0  
led turned on: 1  
led turned off: 0  
led turned on: 1  
led turned off: 0

## 8.2 GPIO INPUT(push button)

```
import machine
import time

# Configure PA1 as an input
button = machine.Pin('PC13', machine.Pin.IN, machine.Pin.PULL_DOWN) #
PULL_UP or PULL_DOWN based on your circuit

while True:
    if button.value() == 0: # Button pressed (assuming active-low)
        print("Button pressed!")
        time.sleep(0.1) # Debounce delay
```

This script continuously checks the state of pin **PC13**. If the button connected to this pin is pressed (assuming it's wired so that pressing the button connects the pin to ground, making it read **0**), it prints "Button pressed!" to the console. The **time.sleep(0.1)** call introduces a small delay to debounce the button, preventing multiple presses from being detected due to mechanical bouncing. Adjust the delay as needed based on the characteristics of your button.

### OUTPUT:

(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> [ampy --port COM7 run GPIO\\_input.py](#)

Button pressed!

Button pressed!

Button pressed!

Button pressed!

## 9. RTC(Real time clocks)

The code you provided sets a specific date and time using the **RTC** (Real-Time Clock) module in MicroPython and then retrieves the date and time

In MicroPython, you would typically set the date and time using the **datetime** attribute of the **RTC** object and retrieve it using the same attribute

```
from pyb import RTC
rtc = RTC()
rtc.datetime((2017, 8, 23, 1, 12, 48, 0, 0)) # set a specific date and time
print(rtc.datetime()) # get date and time
```

**OUTPUT:**

```
(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> ampy --port COM7 run
RTC_setDate.py
```

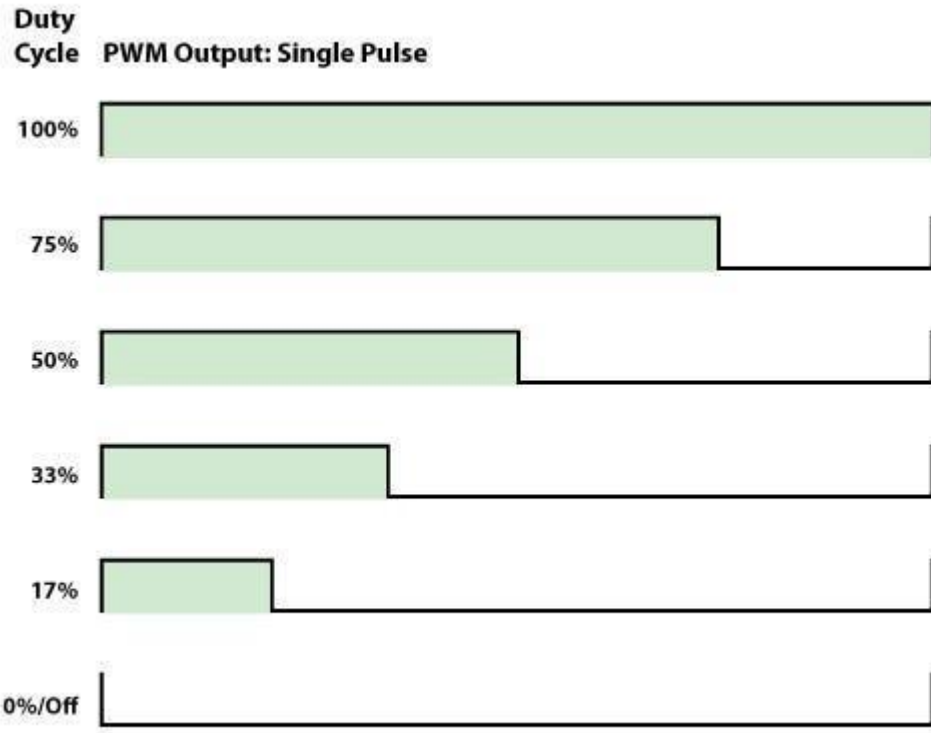
```
(2017, 8, 23, 1, 12, 48, 0, 255)
```

## 10. PWM

### 10.1 What is Pulse Width Modulation (PWM) and when is it used?

Our LED circuits at Environmental Lights are designed to operate at a constant voltage; Typically 5, 12 or 24 volts DC. For this reason, to dim our lights, we cannot simply reduce the voltage or the current flowing through to them. We use a Pulse Width Modulation (PWM) to dim our products. PWM is a very common method of dimming LED lights that works by very rapidly turning them on and off (pulsing) for periods that visually appear as a steady dimmed light. We adjust the brightness level by adjusting the percentage of the time the lights are on (100%) to the time they are off (0%).





## 10.2 What is duty cycle? Hertz?

Duty cycle refers to the percentage of the period that the light is on. As you dim the light the duty cycle (and power consumption) will decrease. At full brightness the duty cycle will be 100%. Hertz is a unit here equivalent to periods per second.

For the STM32F401RE microcontroller on the Nucleo board, the available timers and their corresponding pins with alternate functions are as follows:

- **Timer 1 (TIM1):** Pins A8, A9, A10, A11
- **Timer 2 (TIM2):** Pins A0, A1, A2, A3, A5
- **Timer 3 (TIM3):** Pins B4, B5, B0, B1, A6, A7
- **Timer 4 (TIM4):** Pins B6, B7, D12, D13, D14, D15

```
from pyb import Pin, Timer

# Define the pin connected to the LED
led_pin = Pin('PA5') # Change this to the appropriate pin for your setup

# Create a Timer object
tim = Timer(2, freq=1000) # Timer 2, with a frequency of 1000 Hz

# Configure the Timer channel for PWM
ch = tim.channel(1, Timer.PWM, pin=led_pin)
```

```
# Set the duty cycle to achieve 10% brightness (50% duty cycle)
ch.pulse_width_percent(10)
```

## 10.3 Fading LEDs

- In addition to turning LEDs on and off, it is also possible to control the brightness of an LED using Pulse-Width Modulation (PWM), a common technique for obtaining variable output from a digital pin. This allows us to fade an LED
- Brightness of the LED in PWM is controlled by controlling the pulse-width, that is the amount of time the LED is on every cycle. With a timer frequency of 100 Hz, each cycle takes 0.01 second, or 10 ms.
- If we want to have a breathing effect, where the LED fades from dim to bright then bright to dim, then we simply need to reverse the sign of wstep when we reach maximum brightness, and reverse it again at minimum brightness.

```
import pyb
from time import sleep

# Configure the timer for PWM
tim = pyb.Timer(2, freq=100) # Set frequency to 1kHz
tchannel = tim.channel(1, pyb.Timer.PWM, pin=pyb.Pin.board.PA5,
pulse_width=0) # PA5 corresponds to the first LED

# Initialize variables for the breathing effect
cur_width = 0
wstep = 500 # Adjust the step size for speed of breathing effect
max_width = 65535 # Maximum value for 16-bit timer
min_width = 0 # Minimum pulse width

while True:
    tchannel.pulse_width(cur_width)
    sleep(0.01) # Small delay for smooth transition
    cur_width += wstep

    if cur_width > max_width:
        cur_width = max_width
        wstep *= -1 # Reverse direction

    elif cur_width < min_width:
        cur_width = min_width
        wstep *= -1 # Reverse direction
```

# 11. INTERRUPTS

□ **Pin.irq(handler=None, trigger=Pin.IRQ\_FALLING | Pin.IRQ\_RISING, \*, priority=1, wake=None, hard=False)**

Configure an interrupt handler to be called when the trigger source of the pin is active. If the pin mode is `Pin.IN` then the trigger source is the external value on the pin. If the pin mode is `Pin.OUT` then the trigger source is the output buffer of the pin. Otherwise, if the pin mode is `Pin.OPEN_DRAIN` then the trigger source is the output buffer for state '0' and the external pin value for state '1'.

The arguments are:

- **handler** is an optional function to be called when the interrupt triggers. The handler must take exactly one argument which is the Pin instance.

- **trigger** configures the event which can generate an interrupt. Possible values are:

- **Pin.IRQ\_FALLING** interrupt on falling edge.
- **Pin.IRQ\_RISING** interrupt on rising edge.
- **Pin.IRQ\_LOW\_LEVEL** interrupt on low level.
- **Pin.IRQ\_HIGH\_LEVEL** interrupt on high level.

These values can be OR'ed together to trigger on multiple events.

- **priority** sets the priority level of the interrupt. The values it can take are port-specific, but higher values always represent higher priorities.

- **wake** selects the power mode in which this interrupt can wake up the system. It can be `machine.IDLE`, `machine.SLEEP` or `machine.DEEPSLEEP`. These values can also be OR'ed together to make a pin generate interrupts in more than one power mode.

- **hard** if true a hardware interrupt is used. This reduces the delay between the pin change and the handler being called. Hard interrupt handlers may not allocate memory; see Writing interrupt handlers. Not all ports support this argument.

This method returns a callback object

```
import machine
import time
# Define a function to be called when the interrupt occurs
def button_pressed(b):
    print("Button ",b," pressed!")

# Initialize the button pin as an input with pull-down resistor
button = machine.Pin('PC13', machine.Pin.IN, machine.Pin.PULL_DOWN)
uart = machine.UART(2, baudrate=115200)

# Attach an interrupt to the button pin
button.irq(trigger=machine.Pin.IRQ_RISING, handler=button_pressed)
```

```
# Main loop
while True:
    uart.write('hello\n') # Send the message "hello"
    time.sleep(1) # Wait for 1 second
    if uart.any(): # Check if there is any incoming data
        msg = uart.read() # Read the received data
        print(msg)
```

## OUTPUT:

- Whenever we press the button it trigger the interrupt , and runs it by blocking the present running task
- After the execution of interrupt the blocked tasks resume its function

(.venv) PS C:\Users\vlab\PycharmProjects\MicroPython> [ampy --port COM7 run interrupts.py](#)

hello

Button Pin(Pin.cpu.C13, mode=Pin.IN, pull=Pin.PULL\_DOWN) pressed!

hello

hello

Button Pin(Pin.cpu.C13, mode=Pin.IN, pull=Pin.PULL\_DOWN) pressed!

hello

Button Pin(Pin.cpu.C13, mode=Pin.IN, pull=Pin.PULL\_DOWN) pressed!

hello

Button Pin(Pin.cpu.C13, mode=Pin.IN, pull=Pin.PULL\_DOWN) pressed!

hello

hello

## 11.1 Multiple Interrupts

### 11.1.1 Interrupt related functions

The following functions allow control over interrupts. Some systems require interrupts to operate correctly so disabling them for long periods may compromise core functionality, for example watchdog timers may trigger unexpectedly. Interrupts should only be disabled for a minimum amount of time and then re-enabled to their previous state.

For example:

```
import machine

# Disable interrupts

state = machine.disable_irq()

# Do a small amount of time-critical work here

# Enable interrupts

machine.enable_irq(state)
```

#### **machine.disable\_irq() :**

Disable interrupt requests. Returns the previous IRQ state which should be considered an opaque value. This return value should be passed to the `enable_irq()` function to restore interrupts to their original state, before `disable_irq()` was called.

#### **machine.enable\_irq() :**

Re-enable interrupt requests. The state parameter should be the value that was returned from the most recent call to the `disable_irq()` function.

```
from pyb import Pin, Timer, UART
import time
import machine

# Initialize UART for debugging
uart = UART(2, baudrate=115200)

# Define interrupt flags
button_pressed_flag = False
timer_interrupt_flag = False

# Button press interrupt handler
def button_pressed(pin):
    global button_pressed_flag
    button_pressed_flag = True

# Timer interrupt handler
def timer_interrupt(timer):
    global timer_interrupt_flag
    timer_interrupt_flag = True
```

```

# Initialize the button pin as an input with pull-down resistor
button = Pin('PC13', Pin.IN, Pin.PULL_DOWN)

# Attach an interrupt to the button pin
button.irq(trigger=Pin.IRQ_RISING, handler=button_pressed)

# Initialize the timer
try:
    timer = Timer(1, freq=1) # Set timer to trigger every second
    timer.callback(timer_interrupt)
except ValueError as e:
    uart.write("Timer error: {}\n".format(e))
    raise e

# Main loop
while True:
    # Handle button press first as it has higher priority
    if button_pressed_flag:
        machine.disable_irq()
        button_pressed_flag = False
        uart.write('Button pressed!\n')
        if uart.any():
            print(uart.read())
        machine.enable_irq()

    # Handle timer interrupt next
    if timer_interrupt_flag:
        machine.disable_irq()
        timer_interrupt_flag = False
        uart.write('Timer interrupt!\n')
        if uart.any():
            print(uart.read())
        machine.enable_irq()

    time.sleep(0.1) # Small delay to debounce and yield processor

```

## OUTPUT:

(.venv) PS C:\Users\vlav\PycharmProjects\MicroPython> `ampy --port COM7 run multiple_interrupts.py`

Timer interrupt!

Button pressed!

Timer interrupt!

Button pressed!

Timer interrupt!

Timer interrupt!

Timer interrupt!

Button pressed!

Timer interrupt!

Button pressed!

Timer interrupt!