Discover MicroPython on the ESP32 with your first script

(Updated at 11/28/2022)

One of the easiest things to do on a new board is to make the built-in LED blink. It's similar to the famous "*Hello World*" we display in the terminal when we discover a new programming language. Here is a script that makes the built-in LED of the ESP32 blink every second. The idea here is to show you an overview of programming in MicroPython.

```
from machine import Pin
import time

pin_led = Pin(2, mode=Pin.OUT)

while True:

    pin_led.on()
    time.sleep(1)
    pin_led.off()
    time.sleep(1)
```

Note

It's best to know the basics of Python syntax before jumping into MicroPython.

Run the MicroPython script on your ESP32 board

First of all, make sure you have configured your IDE software for your board.

On Thonny IDE, it is better to save the code on the computer and not directly on the board. Thonny IDE will send the code directly to the MicroPython interpreter (via the REPL) when executing the script. To save the script on the board afterward, go to *File → Save as…*

You will notice that **the Python script does not start automatically when the ESP32 is turned on** . To solve this problem, you must save the script on the board with the name **main.py** . It's not a concern in general during the development phase, but it is essential to know when using a MicroPython script on a board.

Warning

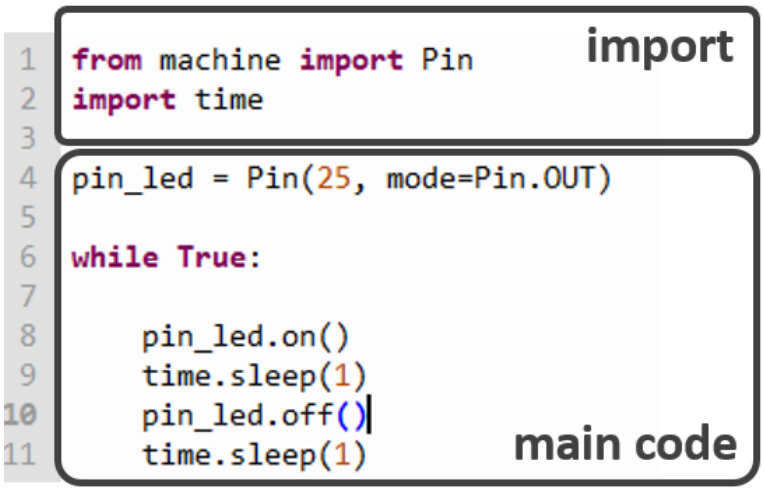
Be careful not to have opened the serial monitor of the Arduino IDE, otherwise you will be confronted with the following error `PermissionError(13, 'Access is denied.', None, 5)` :

```
Unable to connect to COM5: could not open port 'COM5': PermissionError(13, 'Accès refusé.', None, 5)
If you have serial connection to the device from another program, then disconnect it there first.
Backend terminated or disconnected. Use 'Stop/Restart' to restart.
```

Indeed, two software cannot access the same serial COM port simultaneously.

A detailed explanation of a MicroPython script

The layout of a MicroPython script is as follows:




```
1  from machine import Pin
2  import time
3
4  pin_led = Pin(25, mode=Pin.OUT)
5
6  while True:
7
8      pin_led.on()
9      time.sleep(1)
10     pin_led.off()
11     time.sleep(1)
```

Layout of a MicroPython script

Each MicroPython script starts with an import phase of the modules necessary for proper functioning. This is equivalent to the `#include` Arduino code for external libraries `#include <SPI.h>` , `#include <SD.h>` ..

The difference with MicroPython is that even **the basic objects must be imported** . As a reminder, Python is an object-oriented language: all elements in Python are considered objects. For example, the input/output pins of the boards will be seen and manipulated as objects **MicroPython objects are used easily** without being an expert in object-oriented programming.

In the example script, we import from the machine module the sub-module `Pin` . **The module** `machine` **contains most of the objects specific to MicroPython** . The module `Pin` object is, in fact, a class that defines the operation of an input/output pin (GPIO). This is why we create a `pin_led` which corresponds to the pin that controls the built-in LED. We define this pin as an electrical output. The object `Pin` , which we have created, has functions to modify its state `.on()` turns on the LED and `.off()` on turns off the LED.

Note

The integrated LED of the ESP32 is connected to pin number **2 (GPIO2)**

We add a one-second delay with the `sleep()` function of the `time` module to see the blinking visually.

The code that makes the LED blink is **in an infinite loop** to blink… to infinity.

You will notice that if you used to use Arduino code, there are no more `setup()` and `loop()` functions: the script is executed once from bottom to top. So don't forget the infinite loop to have a behavior similar to the Arduino sketch. A code is available to have the same structure as in Arduino code with a `setup()` and a `loop()` .

Different functions to pause a MicroPython script

```
from machine import Pin
import time

pin_led = Pin(2, mode=Pin.OUT)

while True:
    pin_led.on()
    time.sleep_ms(250)
            pin_led.off()
            time.sleep_ms(250)
```

In the code above, you will notice that `time.sleep()` has been replaced by `time.sleep_ms()` for more clearness. In fact, there are 3 functions that allow you to stop the program for a certain period, each with a different precision level:

```
time.sleep(duration_in_seconds)
time.sleep_ms(duration__milliseconds)
time.sleep_us(duration__microseconds)
```

Note

1 second = 103 milliseconds = 106 microseconds

Thus these three lines will block the program for 5 ms:

```
time.sleep(0.005)
time.sleep_ms(5)
time.sleep_us(5000)
```

Note

As for the Arduino code, these functions block the program: it cannot run any other tasks in the background like detecting the pressure of a button and measuring voltages.

You could see an overview of the programming of ESP32 boards with MicroPython. I hope this has made you want to discover more about this new language. I encourage you to read a more detailed tutorial on I/O management with MicroPython with a concrete example at the end.

Using ESP32 I/O with MicroPython

(Updated at 01/23/2023)

The control of the inputs/outputs of the pins (GPIO) is straightforward in MicroPython, thanks to the `Pin` object of the `machine` library. You must associate a `Pin` variable to interact with a "physical" pin on the board. Each physical pin will therefore be represented by a variable (or an object) of type `Pin` . You just have to use `my_pin.a_function()` on the "virtual" pin to change the state of the "physical" pin.

You must therefore remember to import this sub-module at the beginning of your Python script:

```
from machine import Pin
```

Configure pins as input or output with MicroPython

The configuration of a pin in input/output is done during the creation of an object `Pin` .

To configure an output pin (for example, for the integrated LED of the ESP32), we create an object `Pin` specifying the pin number and its output mode *(OUT)* :

```
pin_led = Pin(25, mode=Pin.OUT)
```

If you wish, you can directly set an output voltage when creating the object:

```
pin_led = Pin(25, mode=Pin.OUT, value=1) # 3.3V on output -> the led will be on
pin_led = Pin(25, mode=Pin.OUT, value=0) # 0V on output -> the led will be off
```

When creating a `Pin` object, the output voltage is 0V, and the `value` is 0.

To configure a pin as input, change the `mode` in input *(IN)* when creating the pin.

```
pin_24 = Pin(24, mode=Pin.IN)
```

```

```

It is possible to specify the type of input you want to have. You can choose between input with a pull-up resistor, a pulldown resistor or without a pull resistor. If you don't understand these terms, I encourage you to consult the function and usefulness of pull-up and pulldown resistors. We use the optional argument `pull` , which allows you to choose the type of pull resistor.

```
pin_24 = Pin(24, mode=Pin.IN, pull=Pin.PULL_DOWN) # Input with a pulldown resistor
pin_24 = Pin(24, mode=Pin.IN, pull=Pin.PULL_UP)   # Input with a pullup resistor (the most used)
```

Various ways in MicroPython to set or read a voltage

MicroPython offers several functions to impose an output voltage:

- `Pin.on()` and `Pin.off()`
- `Pin.high()` and `Pin.low()`
- `Pin.value(value)`

The function names are self-explanatory:

```
pin_led = Pin(25, mode=Pin.OUT)

# Set a voltage of 3.3V at the output (high logic state)
pin_led.on()
pin_led.high()
pin_led.value(1)

# Set a voltage of 0V at the output (low logic state)
pin_led.off()
pin_led.low()
pin_led.value(0)
```

Note

Choose functions couple that speaks to you the most. For example, `.on()` | `.off()` function to turn on LEDs and `high()` | `.low()` for a more general case. The function `Pin.value()` is useful when the value is stored in a variable `Pin.value(my_variable)` . For example:

```
pin_led = Pin(2, mode=Pin.OUT)
output_state=1
pin_led.value(output_state) # 3.3V output
output_state=0
pin_led.value(output_state) # 0V output
```

To read the state of a pin, i.e. 3.3V for a high logic level and 0V for a low logic level, we use the same function `Pin.value()` without specifying a value.

```
pin_24 = Pin(24, mode=Pin.IN, pull=Pin.PULL_UP)
pin_24.value() # Returns 1 or 0 depending on the measured voltage
```

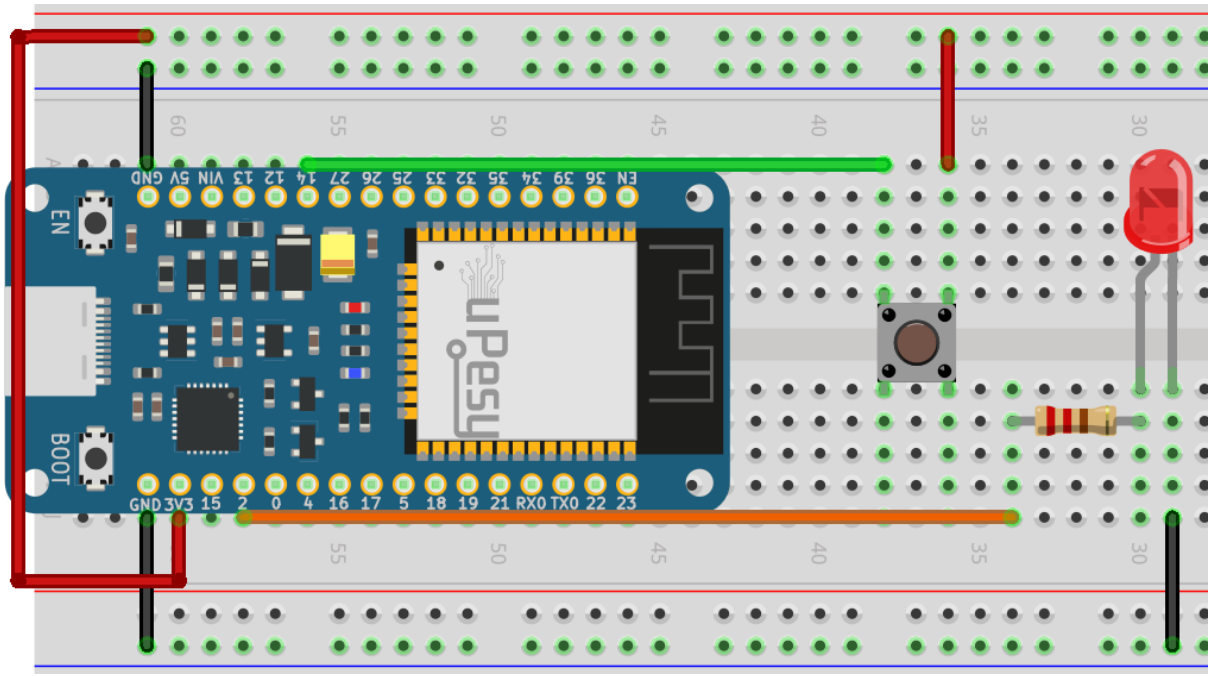
Mini-Project: Turn on the LED when a push button is pressed

To put GPIO pin control into practice, here's a script that turns on the built-in LED when a push button is pressed. There are two variants of the script: one that uses a digital input with a pull-up resistor and another with a pulldown resistor.

Note

Pull-up resistors are generally used rather than pulldown resistors for digital inputs.
With the internal pull-down resistor of the digital input

Here is the electrical circuit to run the MicroPython script on an ESP32 board.



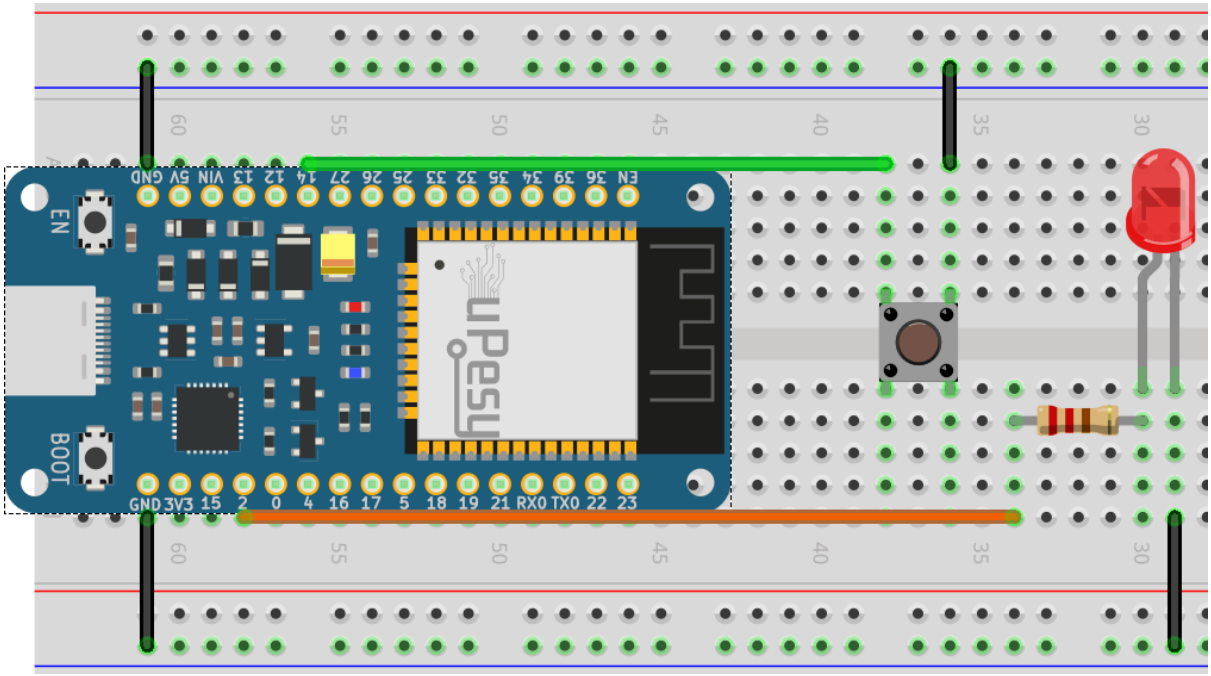
Electrical circuit to be made

```
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_DOWN)
pin_led    = Pin(2, mode=Pin.OUT)

while True:
    if pin_button.value() == 1:
        pin_led.on()
    else:
```

```
        pin_led.off()
```

With the internal pull-up resistor of the digital input

Here is the electrical schematic associated with the MicroPython script on an ESP32 board. You just have to connect the button to the ground instead of 3.3V.



Electrical circuit to be made

```
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(2, mode=Pin.OUT)

while True:
    if not pin_button.value():
        pin_led.on()
    else:
        pin_led.off()
```

Note

You will notice that we can condense `pin_button.value() == 1` by `not pin_button.value()` in the condition of `if` .

We can condense the code because a condition `if` can be written on a single line in Python :

```
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(2, mode=Pin.OUT)
```

```
while True:
    pin_led.on() if not pin_button.value() else pin_led.off()
```

We could simplify the code further by removing the `if` even if we lose clarity.

```
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(2, mode=Pin.OUT)

while True:
    pin_led.value(not pin_button.value())
```

Note

These minor optimizations can be crucial if you want to run the code as quickly as possible because MicroPython is very slow compared to C/C++ language.

Generate variable voltages using PWM in MicroPython

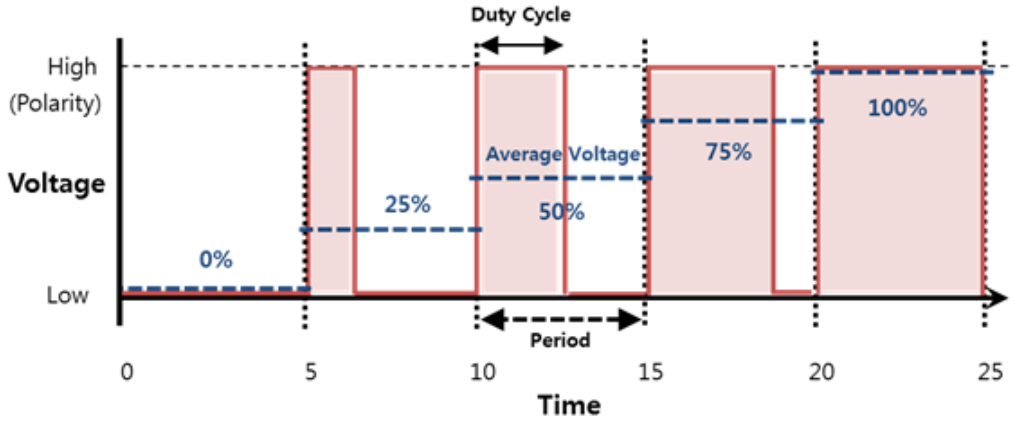
(Updated at 12/22/2022)

PWM: a trick to generate variable voltages on digital pins

The **PWM** is a technique that allows generating a voltage between 0 and 3.3V using **only digital outputs** . PWM is the acronym for *Pulse With Modulation* . Indeed, this trick is based on the temporal proportion of a logic signal at its high state (3.3V) and its low state (0V): the PWM consists in varying the width of an electrical pulse.

Note

The PWM allows for generating constant voltages during a certain period of time. It does not generate alternating voltages as a DAC could.



PWM principle

The succession of pulses with a given width is seen on average as a constant voltage between 0V and 3.3V, whose value is determined by :

$$\text{Voltage} = \text{Voltage} \times \alpha$$

with α , the duty cycle (the pulse width in percent)

A PWM signal is configured via the duty cycle and the pulse frequency. We can modify these two parameters in MicroPython.

In practice, PWM is used to :

- Controlling the speed of a motor
- Controlling the brightness of LEDs
- Generate square signals (with α=0.5)
- Generate music notes (sound similar to retro consoles)

PWM performance on the ESP32

The microprocessor generates PWM signals not continuously but by dedicated hardware blocks. You only have to configure the PWM blocks once in the script so that the signal is constantly generated in the background. We associate an output of a PWM block to a pin of our board. The processor resources will be free to execute other tasks.

Each PWM block can have an independent frequency.

PWM characteristics of the	**ESP32**
Frequency range of the PWM signal	1Hz to 40 Mhz
Independent PWM frequency	8
PWM output	16
Pulse width resolution	10 bits

Note

In most cases, a PWM frequency of around 1000 Hz will be sufficient.

PWM on MicroPython

Using PWM with MicroPython is very simple. MicroPython automatically selects an available PWM block: it is unnecessary to indicate which one you intend to use. The hardware part described above is completely hidden.

The configuration consists in associating a `PWM` object to a `Pin` object and choosing the PWM frequency.

```
from machine import Pin, PWM

pin = Pin(25, mode=Pin.OUT)
pin_with_pwm = PWM(pin) # Attach PWM object on a pin
```

Note

Since the PWM object is also in the machine module, we can combine the two imports on a single line:

```
from machine import Pin
from machine import PWM
```

equals to

```
from machine import Pin, PWM
```

Once we have linked a pin of the board to our `PWM` object, all the functions are done directly on the `PWM` object (unlike the Arduino code where we specify the pin number with the function `analogWrite(pin_number)` .

Note

In python, you can insert underscores _ to make the numbers easier to read (and thus avoid counting zeros on large numbers). For example, to write `1 MHz` instead of having `1000000` , we can put `1_000_000` .

We use the function `.duty_()` to select the duty cycle with a value between 0 and 210 (0-1024). The voltage is set directly on the output of the previously selected pin.

To see the voltage variation, you can use the integrated LED of your board with the following script:

```
from machine import Pin, PWM
```

```
LED_BUITLTIN = 2 # For ESP32

pwm_led = PWM(Pin(LED_BUITLTIN, mode=Pin.OUT)) # Attach PWM object on the
LED pin

# Settings
pwm_led.freq(1_000)

while True:
    for duty in range(0,1024):
        pwm_led.duty(duty) # For ESP32
```

To clarify, you could specify the duty cycle percentage and then apply a formula.

```
from machine import Pin, PWM

LED_BUITLTIN = 2 # For ESP32

pwm_led = PWM(Pin(LED_BUITLTIN, mode=Pin.OUT)) # Attach PWM object on the
LED pin
pwm_led.freq(1_000)

duty_cycle = 50 # Between 0 - 100 %
pwm_led.duty(int((duty_cycle/100)*1024))
```

We can change the brightness of the LED by varying the duty cycle between 0 and 100%.

```
from machine import Pin, PWM
import time

LED_BUITLTIN = 2 # For ESP32

pwm_led = PWM(Pin(LED_BUITLTIN, mode=Pin.OUT)) # Attach PWM object on the
LED pin
pwm_led.freq(1_000)

while True:
    for duty in range(100): # Duty from 0 to 100 %
        pwm_led.duty(int((duty/100)*1024))
        time.sleep_ms(5)
```

Note

We add a 5ms delay with `time.sleep_ms(5)` to slow down the code and see the variation of the brightness with the eye.

Mini-Project: Dimming the ESP32 built-in LED

With all that we have seen, the script is straightforward:

```
from machine import Pin, PWM
import time

LED_BUITLTIN = 2 # For ESP32
pwm_led = PWM(Pin(LED_BUITLTIN, mode=Pin.OUT)) # Attach PWM object on the
LED pin

# Settings
pwm_led.freq(1_000)

while True:
    for duty in range(0,1024, 5):
        pwm_led.duty(duty)
        time.sleep_ms(5)
    for duty in range(1023,-1, -5):
        pwm_led.duty(duty)
        time.sleep_ms(5)
```

Note

To vary the voltage faster, we change the duty from 5 to 5 with `range(0, 1024, 5)` . To decrement a value, we put a step of -5 `range(1023, -1, -5)` .

Measure an analog voltage with the ESP32 ADC in MicroPython

(Updated at 11/28/2022)

On the ESP32, some pins are connected to the integrated ADC of the microcontroller. An **ADC is an analog to digital converter** that returns a digital value proportional to the measured voltage.

Features

On the ESP32, pins 36, 39, 34 and 35 can be analog inputs with MicroPython. The resolution of the measurements is 12 bits, so the measurements are between 0 and 4095.

Warning

Unlike the Arduino, the analog inputs on the ESP32 must not exceed a voltage of 3.3V. This could damage or even destroy the inputs of the ADC. It is, therefore, necessary to check that you do not use a supply voltage of 5V to measure the value of a sensor. There are solutions to limit the input voltage of the ADC.

Read an analog input from the ESP32 with MicroPython

The voltage of analog input is measured using the sub-module `ADC` of the `machine` . As for PWM, a physical pin is associated with the `ADC` object.

```
from machine import Pin, ADC

# Pin definitions
adc_pin = Pin(36, mode=Pin.IN)
adc = ADC(adc_pin)
adc.atten(ADC.ATTN_11DB)
```

Do not forget to import the ADC sub-module. You will note that for the ESP32, you must also specify attenuation of 11 dB (with `ADC.ATTN_11DB`) to read a voltage over the range of 0 to 3.3V.

You can also create an `ADC` by specifying the number of the analog channel you want to use.

```
from machine import ADC
adc = ADC(0) # Select the ADC_0 ()
```

Warning

On the ESP32, the analog channels are assigned to the pins disorderly. It is, therefore, better to use the definition by pin number.

Once you have configured an analog input in MicroPython, you only need to use the function `read()` to read the analog value.

```
from machine import Pin, ADC
adc = ADC(Pin(36, mode=Pin.IN))
print(adc.read())
```

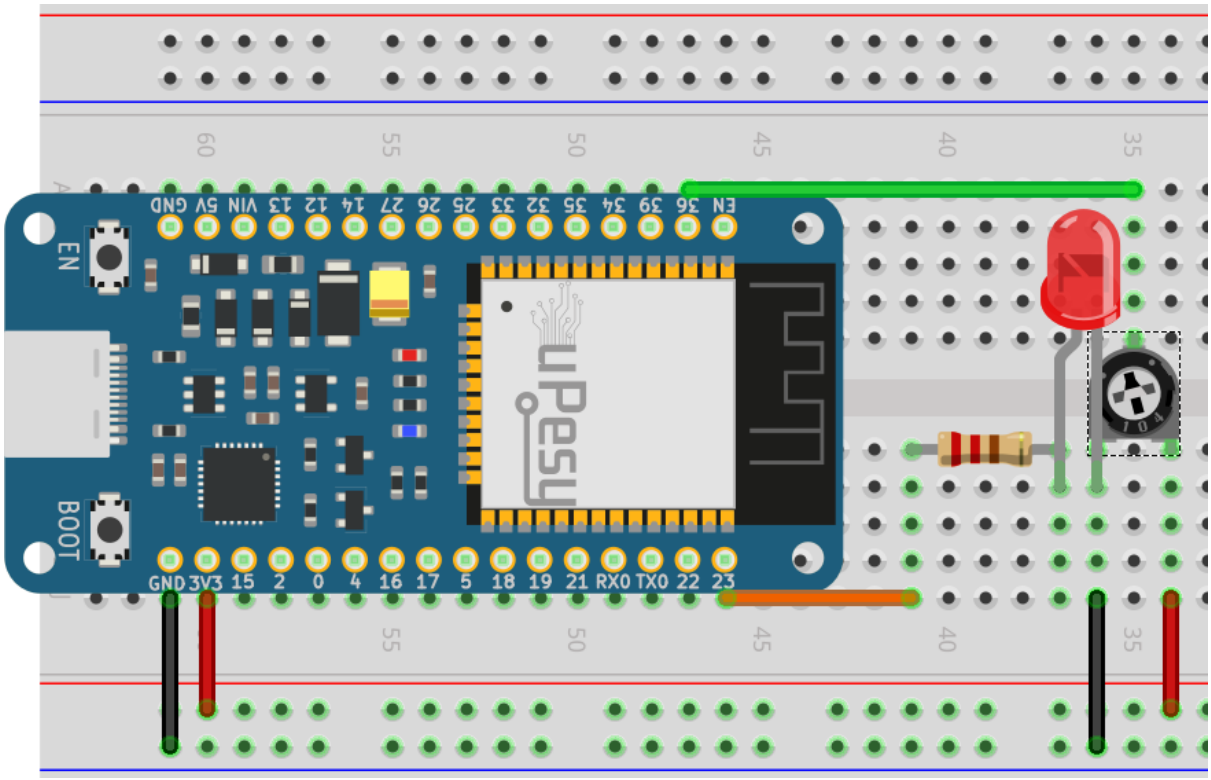
It will be necessary to use a cross product to have the value measured in volt:

$$\text{?????} = 3.3 \text{?} 4095 \times \text{?????}$$

Mini-Project: Control the intensity of an LED via a potentiometer on the ESP32

We will use PWM and ADC with MicroPython to control the brightness of an external LED.

Electrical schematic :



Electrical circuit to be made

Note

Remember putting a resistor in series with the LED to prevent it from burning (330 Ω, for example)

To read the position of the potentiometer alone:

```
from machine import Pin, ADC
import time


# Create an ADC object linked to pin 36
adc = ADC(Pin(36, mode=Pin.IN))

while True:

    # Read ADC and convert to voltage
    val = adc.read()
    val = val * (3.3 / 4095)
    print(round(val, 2), "V") # Keep only 2 digits

    # Wait a bit before taking another reading
    time.sleep_ms(100)
```

For reading the position of the potentiometer and changing the brightness:

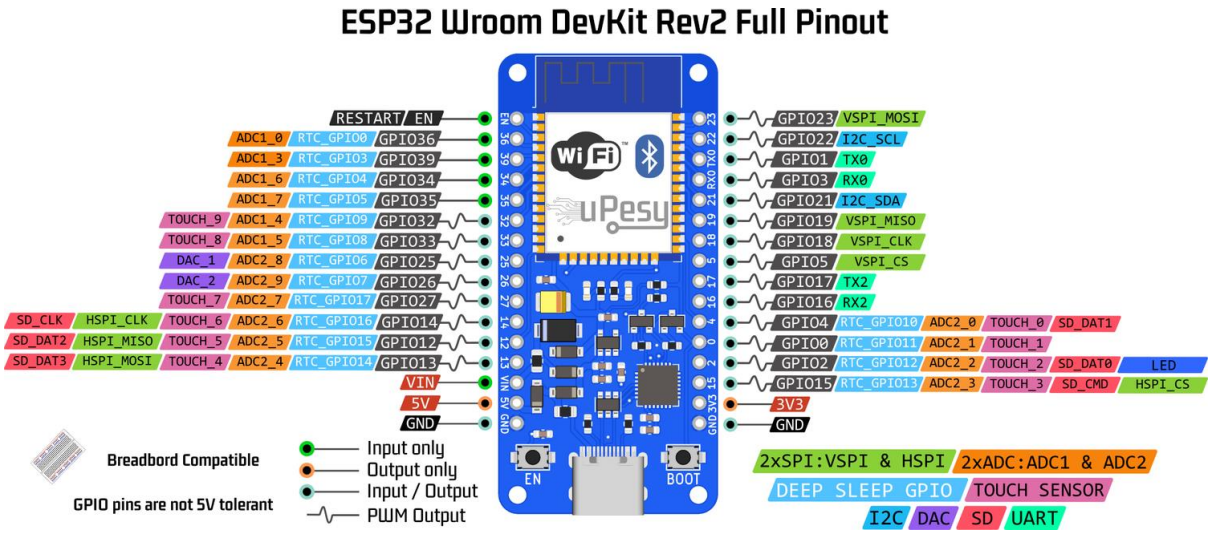
```
from machine import Pin, ADC, PWM
```

```
import time


def map(x, in_min, in_max, out_min, out_max):
    """ return linear interpolation like map() fonction in Arduino"""
    return (x - in_min) * (out_max - out_min) // (in_max - in_min) +
out_min


if __name__ == "__main__":

    # Create a PWM object linked to pin 23
    pwm_led = PWM(Pin(23,mode=Pin.OUT))
    pwm_led.freq(1_000)

    # Create an ADC object linked to pin 36
    adc = ADC(Pin(36, mode=Pin.IN))
    adc.atten(ADC.ATTN_11DB)

    while True:

        val = adc.read()
        pwm_value = map(x=val, in_min=0, in_max=4095,
out_min=0,out_max=1023)
        pwm_led.duty(pwm_value)
        time.sleep_ms(10)
```

Using the ESP32 capacitive sensors in MicroPython

(Updated at 01/20/2023)

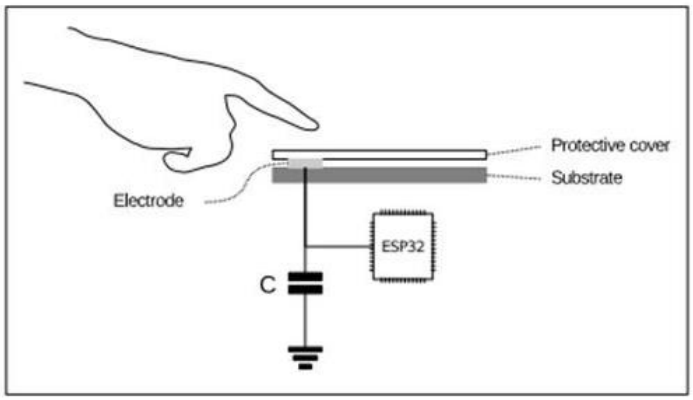
The ESP32 comes with capacitive sensors that can be used instead of conventional push buttons. These are sometimes known as “TOUCH” on ESP32 pinouts found online. There are 10 on the uPesy ESP32 boards that can be utilized with MicroPython.



Pinout of the uPesy ESP32 Wroom Devkit board

How does a capacitive sensor work?

Capacitive sensors are frequently used to detect the pressure of our fingers on everyday devices, such as touch screens. They are famous for this type of application.



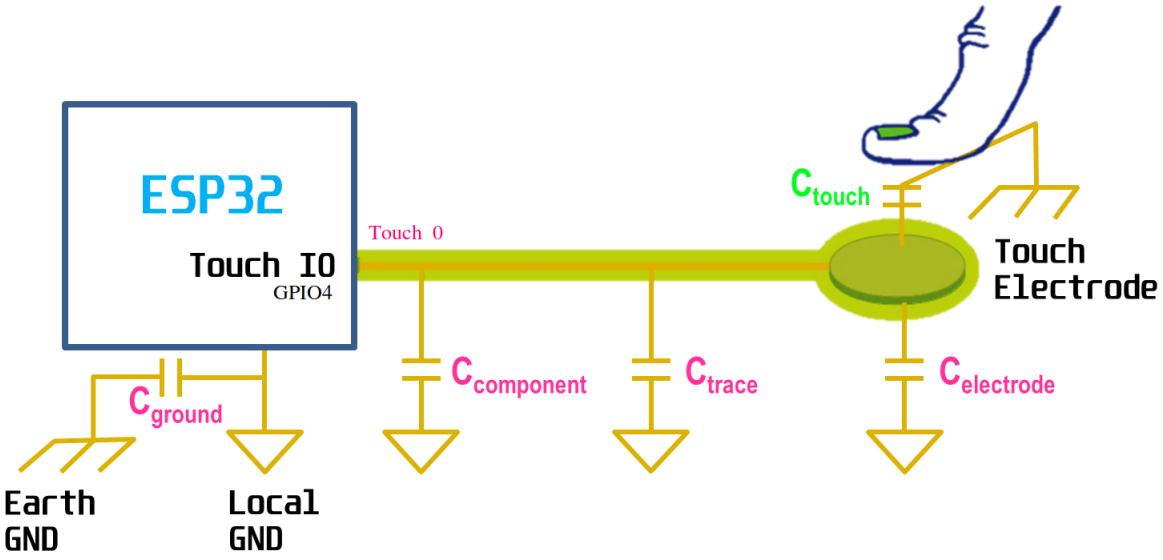
Capacitive touch sensors

Capacitive touch sensors detect changes in capacitance when touched. A capacitor typically has two metal plates facing each other, separated by a non-conductive substrate. On a capacitive sensor, one plate is fixed, while the other is your finger. The ESP32's Analog-to-Digital Converter (ADC) converts this change into an analog value.

Note

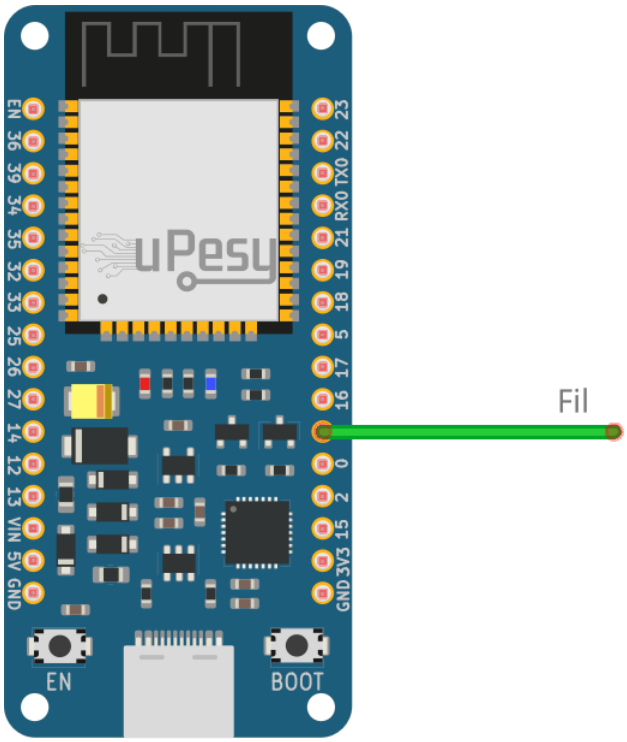
Capacitive sensors are less reliable than mechanical pushbuttons, which are more prone to environmental noise interference. It is essential to consider this when using them in your projects.

Understanding the values obtained can sometimes be challenging due to interference from outside factors. To gain a better comprehension of this, take a look at this diagram which illustrates the different sources of disturbances.



Parasitic capacities

Good results can be achieved by considering all the required parameters and designing an electrical circuit that is more sophisticated than the one provided. A simple copper wire is enough to detect a finger's pressure.



Circuit consisting of a wire (jumer)

Using the ESP32 capacitive sensors with MicroPython

Micro-Python can measure analog values instead of just binary values of `LOW` or `HIGH` . The values measured range from 0 to 4095.

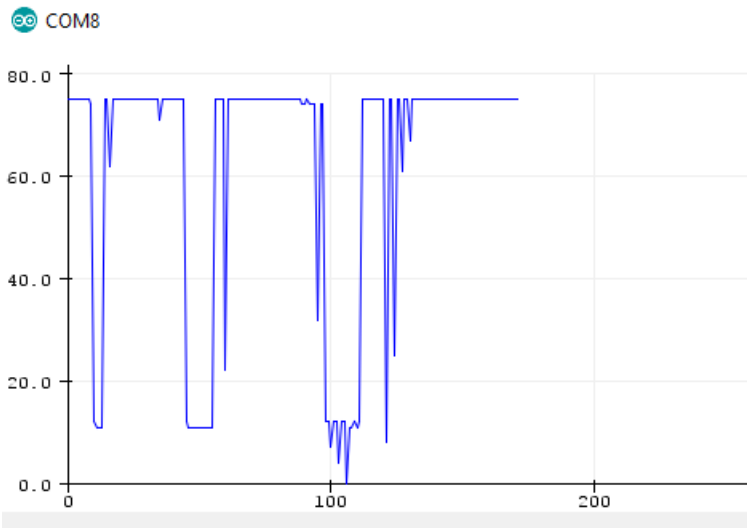
Note

We indicate the PIN with a `TOUCH` functionality in the Python script. For instance, we use `TOUCH0` , connected to the `GPIO4` pin.

```
from machine import TouchPad, Pin
import time

touch_pin = TouchPad(Pin(4, mode=Pin.IN))
while True:
    touch_value = touch_pin.read()
    print(touch_value)
    time.sleep_ms(500)
```

As for the code, it is identical to that of the ADC, except for the change of `ADC(Pin(36, mode=Pin.IN))` to `TouchPad(Pin(4, mode=Pin.IN))` .

Analog values must be given a particular reference point (threshold value) to detect pressure. When nothing is pressing on the wire, the value is usually high. When the wire is touched, however, the value drops significantly.



Curve displayed in the serial plotter

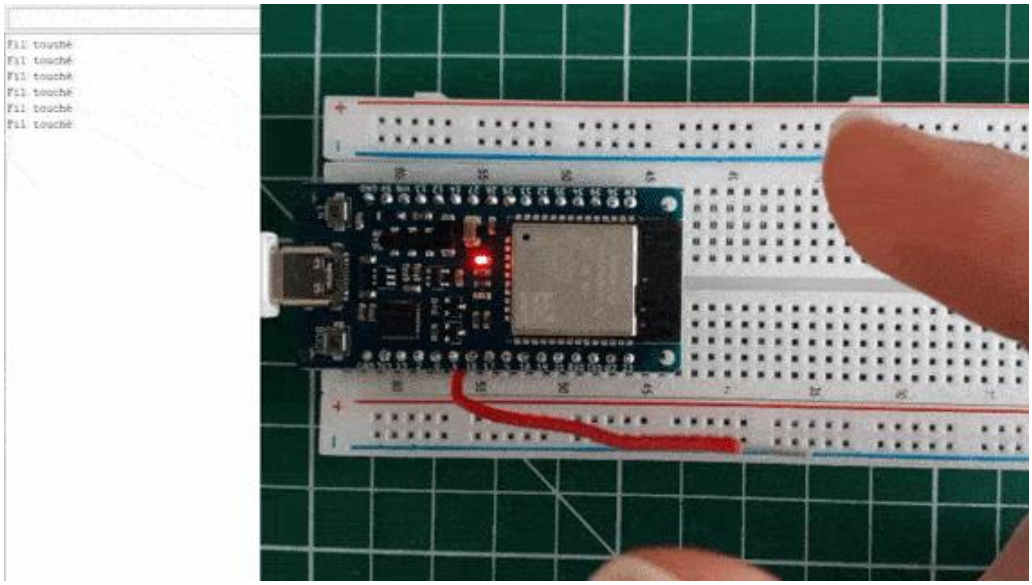
Note

The threshold value depends on the type of material used, such as the wire, length, or breadboard, so you may need to fine-tune the current for your particular application.

Here is an example with a threshold of `150` :

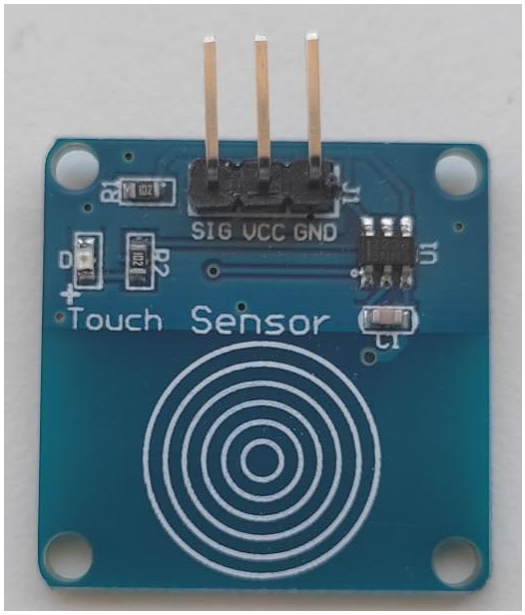
```
from machine import TouchPad, Pin
import time

capacitiveValue = 500
threshold = 150 # Threshold to be adjusted
touch_pin = TouchPad(Pin(4))

print("\nESP32 Touch Demo")
while True: # Infinite loop
  capacitiveValue = touch_pin.read()
  if capacitiveValue < threshold:
    print("Fil touché")
    time.sleep_ms(500)
```

See for yourself the result :

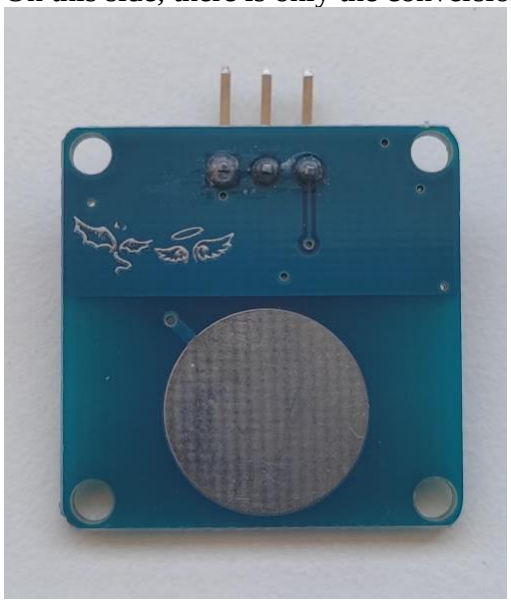


Amazing isn’t it 😎?

That’s all! There are readily available capacitive sensing modules that contain a circuit that can convert the capacitance levels into binary values.



On this side, there is only the conversion circuit, no plate



On the other side, a metal plate

If you want to obtain a signal for your ESP32 to convert, avoid modules that come with a conversion circuit. Instead, opt for modules that provide a raw signal, which the ESP32 can convert using its internal circuit. An example of such a module is a PCB with a designated space for a finger.

How to use interrupts in MicroPython on an ESP32 ?

(Updated at 01/23/2023)

In electronics, an interrupt is a signal sent to a processor to indicate that an important task must be executed immediately, thus interrupting the execution of the program in progress.

Interrupts can be generated in different ways, for example following an external event, a `timer` they allow certain tasks to be executed asynchronously, i.e. independently of the main program.

Interrupts in MicroPython

In practice, interruptions are generally used:

- To execute portions of critical code when an external event occurs. For example, when a button is pressed, a Python function will automatically execute.
- To perform functions periodically. For example, to flash an LED every 5 seconds.

You are going to tell me that we can already do this kind of script, without using interrupts. Yes, it's true, but there are 2 major flaws without using them. Let's take the example of a script that turns on a LED when the button is pressed:

```
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(2, mode=Pin.OUT)

while True:
    if not pin_button.value():
        pin_led.on()
    else:
        pin_led.off()
```

The first flaw is that the script spends its time scanning the value of the `pin_button` pin to know if the button has been pressed. The script can't do other things in addition because otherwise the second flaw will occur: missing events. If the script performs other tasks in the loop, it may not be able to detect the temporary button press.

The advantage of using a hardware interrupt is that the detection is completely detached from the processor (and therefore from the Python script). With an interrupt the `while` loop of the script will be empty. The hardware block in charge of the detection is also much more reactive than the MicroPython script.

Note

With interrupts, there is no need to constantly scan the value of a pin: a function is executed automatically when a change is detected.

Whether with a timer or an external event, the interrupt is triggered following a signal change. Let's discover the different possible variations 😊.

Triggering a hardware interrupt: detection modes

The detection of an event is based on the shape of the signal that arrives at the pin.

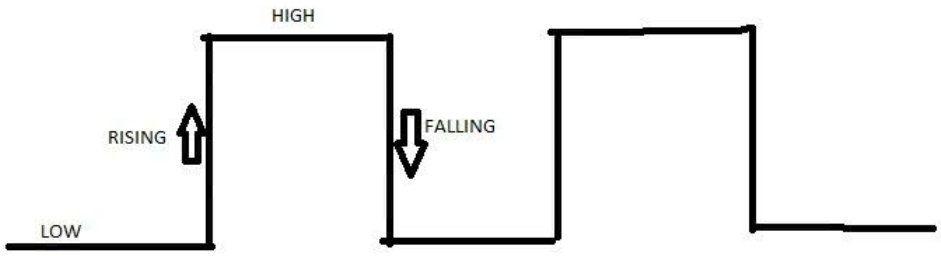


Figure: Digital Signal

Different detection modes

Here are the different types of possible detection of an interruption:

- `Pin.IRQ_LOW_LEVEL` triggers the interrupt as soon as the signal is at 0V
- `Pin.IRQ_HIGH_LEVEL` triggers the interrupt as soon as the signal is at 3.3V
- `Pin.IRQ_RISING` : Triggers the interrupt as soon as the signal goes from `LOW` à `HIGH` (From 0 to 3.3V)
- `Pin.IRQ_FALLING` triggers the interrupt as soon as the signal changes from `HIGH` à `LOW` (From 3.3V to 0)

Note

The modes `RISING` and `FALLING` modes are the most used. Note that if you use the `LOW` and `HIGH` modes, the interrupt will be triggered in a loop as long as the signal does not change state.

Configuring and using interrupts in MicroPython on the ESP32

A basic skeleton script

Here is a skeleton code, to trigger an interrupt via an external signal on your ESP32 board with MicroPython :

```
from machine import Pin
```

```
pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)

def interruption_handler(pin):
    ...

pin_button.irq(trigger=Pin.IRQ_FALLING,handler=interruption_handler)

while True:
    ...
```

The script uses the function `Pin.irq()` function which allows to create an interrupt request on a falling edge of the signal present on the pin `pin_button` .

Note

`irq` stands for *Interrupt Request* to request an interrupt request.

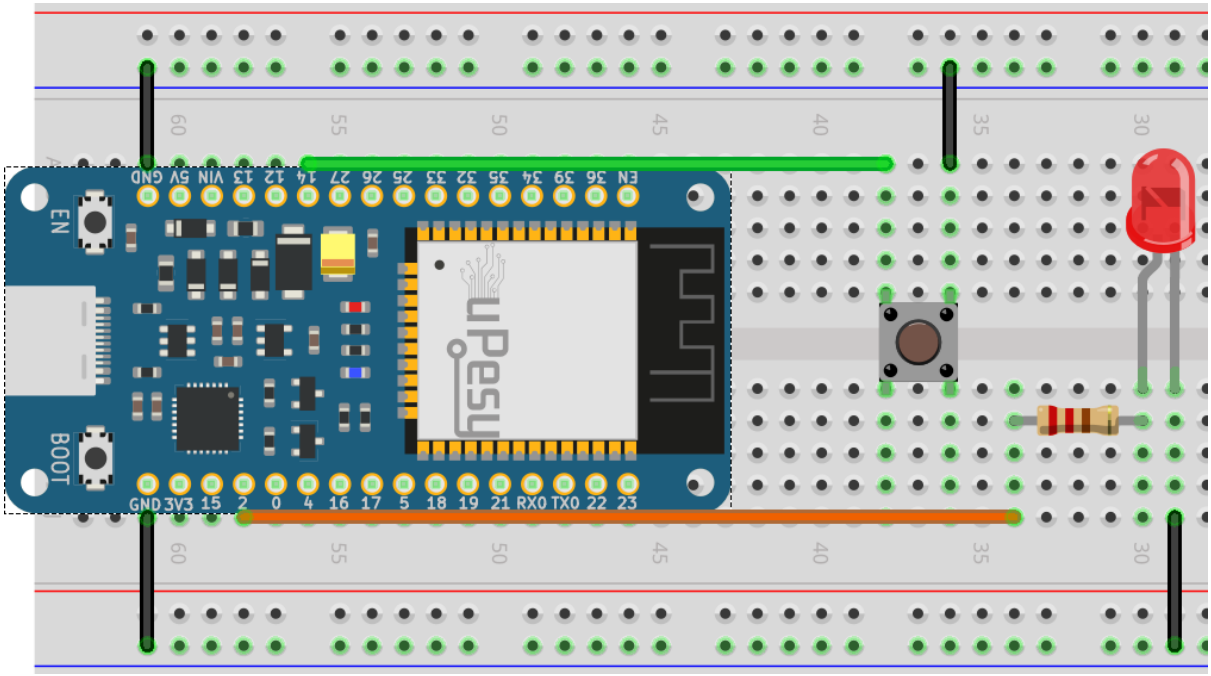
When an interrupt is triggered the `interruption_handler()` function will be executed automatically. The interrupt routine will have as input argument the pin on which the event was detected.

It is a good practice to have an interrupt function (`isr`) as fast as possible to avoid disturbing the main program which has been interrupted. For example, we will avoid sending data via I2C, SPI directly from an interrupt. We can use **flags** in the form of boolean to store the detection of an event and then process it in the main loop.

Note

Interrupt handling in microPython will always be much slower than in Arduino or pure C code! However, it is possible to minimize this latency by using advanced parameters .

Example: Turn on an LED when a push button is pressed



Circuit électrique à réaliser

Here is the complete script that detects when a button is pressed with an interrupt and turns on the LED accordingly:

```
import time
from machine import Pin

pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)
pin_led    = Pin(2, mode=Pin.OUT)


def button_isr(pin):
  pin_led.value(not pin_led.value())

pin_button.irq(trigger=Pin.IRQ_FALLING,handler=button_isr)

while True:
  ...
```

Note

In this example, the interrupt is triggered on a falling edge. It is possible to combine the modes so that the interrupt is triggered on both a rising and a falling edge with the OR operator `|` :

```
pin_button.irq(trigger=Pin.IRQ_FALLING | Pin.IRQ_RISING,handler=button_isr)
```

Here are some functions that may be useful to you:

- `irq.init()` : Re-initialize the interrupt. It will be automatically reactivated.
- `irq.enable()` : Enable the interrupt.
- `irq.disable()` : Disable the interrupt.
- `irq()` : Manually launch the call of the interrupt routine.
- `irq.flags()` : Know the type of event that triggered the interrupt. Can only be used in the``isr`` .

```
def pin_handler(pin):
    print('Interrupt from pin {}'.format(pin.id()))
    flags = pin.irq().flags()
    if flags & Pin.IRQ_RISING:
        # handle rising edge
    else:
        # handle falling edge
    # disable the interrupt
    pin.irq().deinit()
```

To use these functions, you have to use the variable of the pin that is attached to an interrupt, for example `pin_button.irq().enable()` .

You now know the basics of using interrupts in MicroPython. You may want to` consult the official documentation <https://docs.micropython.org/en/latest/library/machine.Pin.html#machine.Pin.irq>`__ to use all their potential, for example to set priorities between several interrupts that are triggered at the same time. Some tips and optimization are presented in the advanced section 😎.

Use global variables to manage events in the main program

We try to limit the number of actions done in an interrupt. It is common to increment a variable in the``isr`` and to do the long tasks in the main code according to the value of this variable. Here is an example that counts the number of times you press a push button.

```
import time
from machine import Pin

button_pressed_count = 0 # global variable
pin_button = Pin(14, mode=Pin.IN, pull=Pin.PULL_UP)

def button_isr(pin):
    global button_pressed_count
    button_pressed_count += 1


if __name__ == "__main__":
    button_pressed_count_old = 0
    pin_button.irq(trigger=Pin.IRQ_FALLING,handler=button_isr)

    while True:
        if button_pressed_count_old != button_pressed_count:
            print('Button value:', button_pressed_count)
            button_pressed_count_old = button_pressed_count
```

```
        if button_pressed_count > 10: # heavy task here
            ...
```

We use a global variable to be able to write to it in the interrupt routine.

Warning

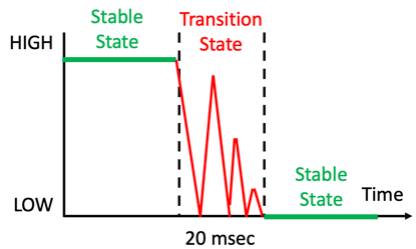
Even if the variable is defined at the very top of the Python script, you must add the keyword `global` when the variable is used in a function. This tells the Python interpreter to use the global variable instead of creating a local variable (with the same name), which would be used only in the execution context of the function.

```
MicroPython v1.19.1 on 2022-06-18; ESP32 module with ESP32
Type "help()" for more information.
>>> %Run -c $EDITOR_CONTENT
 Button value: 1
 Button value: 2
 Button value: 3
 Button value: 4
 Button value: 5
 Button value: 7
 Button value: 8
 Button value: 11
 Button value: 15
 Button value: 16
 Button value: 17
 Button value: 21
 Button value: 24
 Button value: 27
 Button value: 30
 Button value: 33
 Button value: 36
 Button value: 39
 Button value: 42
 Button value: 45
 Button value: 47
```

When running the MicroPython script, the increment value is much larger than the number of presses we did. Weird 🤔? It's because of the transitions of the logic levels at the push button that are not perfect..

Improve reliability of outage detection

You may have noticed that with push buttons, there are false positives: the interrupt routine runs more times than it should. This is because the signal received by the ESP32 is not perfect: it is as if it had received a "double press" of the button:



This is called the bounce effect. We can reduce the **bounce of a push-button** via the Python script directly. This is called **debouncing software.** It consists in not taking into account the transitory period between the 2 logical states by waiting a certain delay.

Using ESP32 timers with MicroPython

(Updated at 01/23/2023)

In this article, we will explore the use of timers on the ESP32 with MicroPython. We'll discover the steps necessary to set up a timer on the ESP32, as well as the important parameters for optimal operation. Let's go ! 😊

The operation of a timer on the ESP32

The theoretical functioning of the timer is not discussed in this article in order to keep it as clear as possible. If you are a beginner and you don't know the inner workings of a timer, I strongly invite you to read the theoretical article on the functioning of a timer . This will allow you to better understand how to select the right parameter values to use it in your Micro Python scripts.

Configuring and using an ESP32 timer in MicroPython

With MicroPython, all the complexity of the timer is hidden: you just have to enter the desired timer period. That is to say, the length of time the counter waits before an interrupt is triggered.

Note

It's even easier than with Arduino code, because you don't have to worry about the values of the `prescaler` and the``autoreload`` values, to have the right period. MicroPython takes care of that.

Script squellete minimal

Here is a minimal skeleton Python script to use an ESP32 timer. It allows to trigger the interrupt `interruption_handler()` at the end of each timer period.

```
from machine import Timer

timer_0 = Timer(0) # Between 0-3 for ESP32

def interruption_handler(timer):
    ...
```

```
if __name__ == "__main__":
    timer_0.init(mode=Timer.PERIODIC, period=1000,
callback=interruption_handler)
```

In our example, the period of the timer is set to 1000ms or 1 second. To use a timer, you must :

- Choose the hardware timer to be used with `Timer(id)` . In general we create the object at the beginning of the script.
- Set up the timer at its initialization with the function `timer_0.init(mode=, period=, callback=)` which contains the following arguments:
    - The mode `Timer.PERIODIC` mode so that the interrupt is triggered each time the timer period is reached or `Timer.ONE_SHOT` to have a single triggering
    - The desired timer period **in milliseconds**
    - The interrupt routine (the function) that will be invoked via the interrupt triggered by the timer, here `interruption_handler`

Warning

The minimum timer period is one millisecond in MicroPython, whereas with Arduino code, it can easily be reduced to one microsecond. Timer performances are limited so that MicroPython can keep up with the pace..

Example: Blink script only with a timer

With this script, the built-in blue LED of the ESP32 starts blinking every second, without using loops: you can write other functions without the blinking being disturbed.

```
from machine import Timer, Pin

pin_led    = Pin(2, mode=Pin.OUT)
timer_0 = Timer(0) # Between 0-3 for ESP32

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

if __name__ == "__main__":
    timer_0.init(mode=Timer.PERIODIC, period=1000,
callback=interruption_handler)
```

Here, I use a little trick to reverse the state of the LED. We get the current state of the LED with `pin_led.value()` which is 0 or 1, and invert the value with the operator `not` . We use the same function `pin_led.value()` to update the new value.

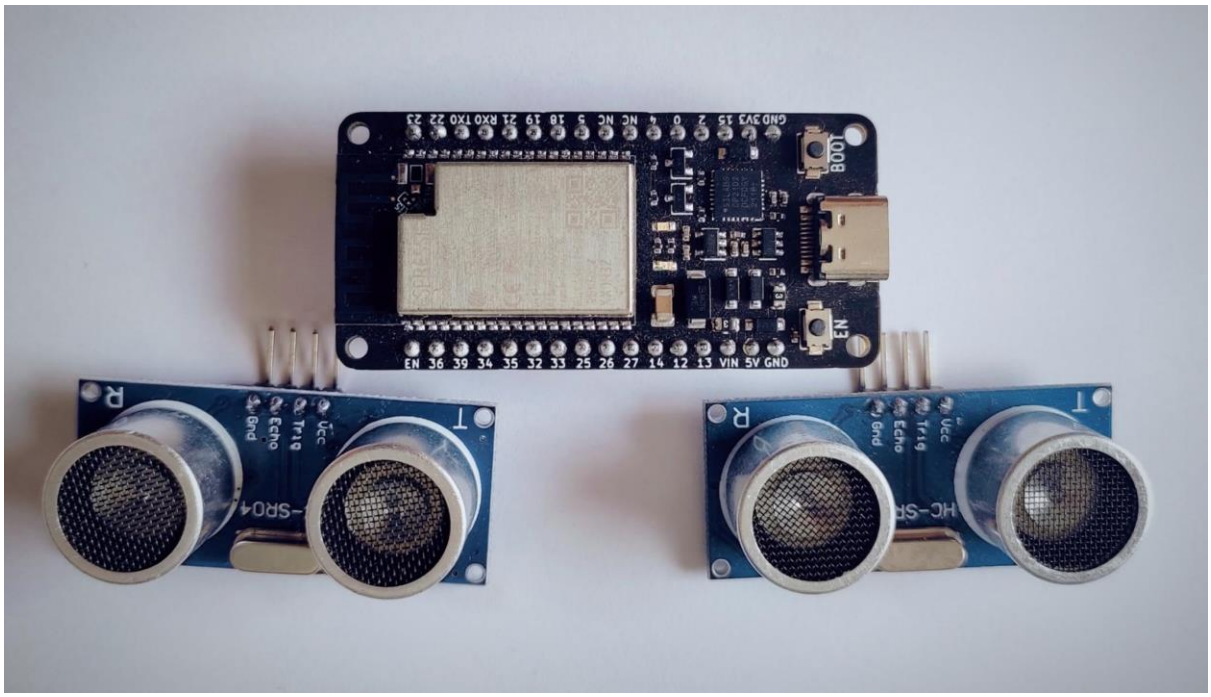
Example : Increment a variable

Here is a Python script that increments a variable each time the timer triggers an interrupt. In the main loop, when the variable is 10, a specific task is performed.

```
from machine import Timer, Pin

timer_0 = Timer(0) # Between 0-3 for ESP32
timer_count = 0 # global variable

def interruption_handler(pin):
    global timer_count
    timer_count += 1


if __name__ == "__main__":
    timer_count_old = 0
    timer_0.init(mode=Timer.PERIODIC, period=100,
callback=interruption_handler)

    while True:
        if timer_count > 10:
            timer_count = 0
            print("10x")
            # heavy task here
```

Measuring distances with the HC-SR04 ultrasonic sensor with an ESP32 in MicroPython

(Updated at 01/04/2023)



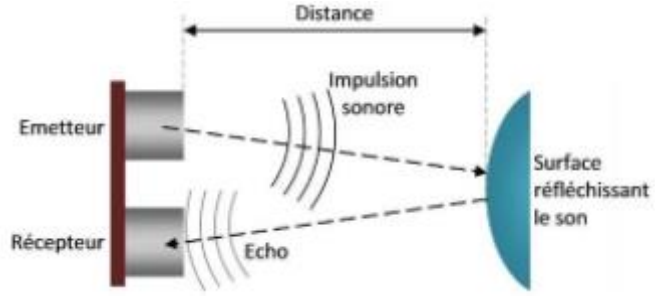
The **HC-SR04** sensor is a device that can measure the distance to an obstacle using ultrasonic waves. It consists of three components: an ultrasonic transmitter, a receiver, and a chip to control them.

Tip

*Please check that your module is an **HC-SR04** . It is usually colored blue and needs 5V to work. Please distinguish it from the **RCWL-1606** , which is generally green and needs 3.3V. Although they work differently, they are both driven in the same way.*

Getting started with the HC-SR04 module

Operation of an ultrasonic sensor



If you'd like to understand how the HC-SR04 module works, you can check out this article which explains the module's physical function in detail.

Technical specifications of the HC-SR04 module

- The module requires a 5V power supply.
- Consumption: The sensor requires 20mA of power to function correctly.
- Range: The module can measure distances between 3cm to 4m. However, I recommend using it for lengths between 10cm and 2.5m for the best results.

  Warning

  For accurate readings, the obstacle's surface must be even.

- Measurement angle: < 15°
- Ultrasonic frequency: 40 kHz (inaudible to the ear)

HC-SR04 Sensor Wiring on ESP32

In addition to the power pins, we can use two pins to control the module. It is more convenient to use both pins, but if you don't have more pins available, you can use just one.

Pin Wiring

HC SR04 Module	ESP32
TRIG	GPIO5
ECHO	GPIO18

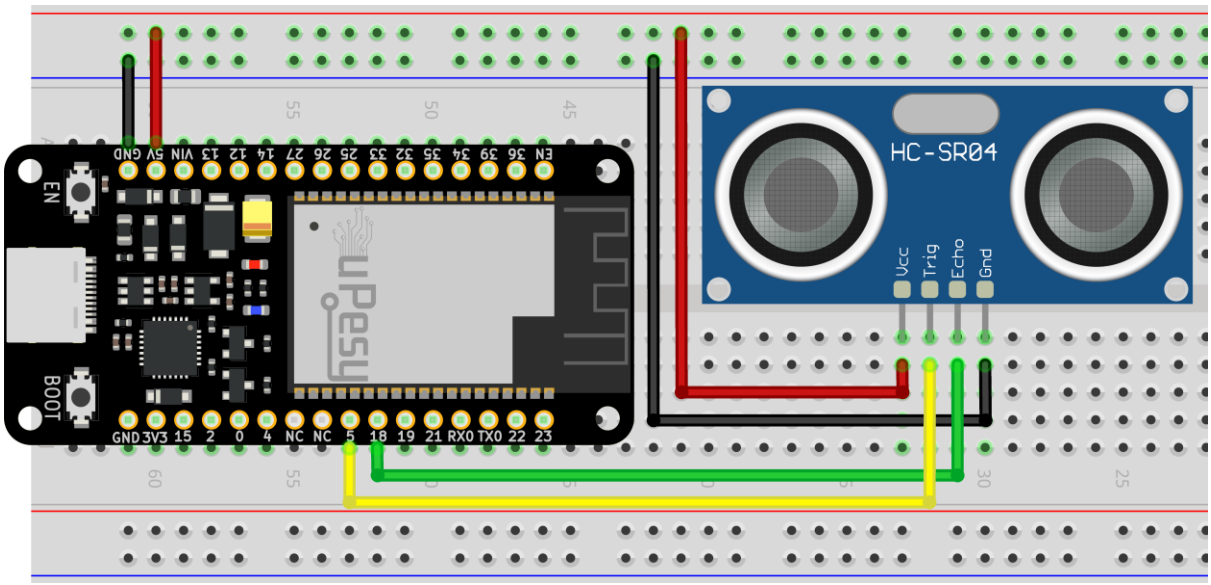
Pin Wiring

HC SR04 Module	ESP32
GND	GND
VCC	5V

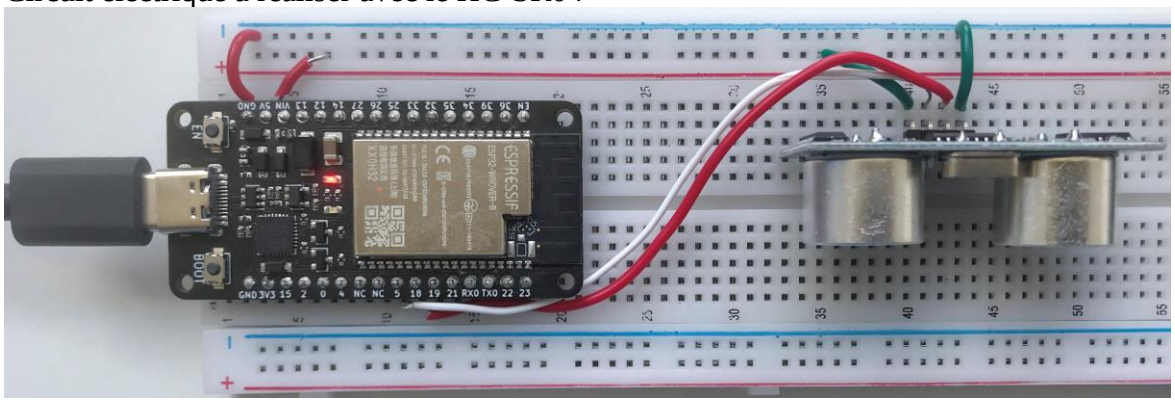
Note

For this example, pins `5` and `18` have been chosen for the output `TRIG` and the input `ECHO` , respectively. However, you can use other compatible pins of the ESP32 .

Circuit with the HC-SR04 module and an ESP32



Circuit électrique à réaliser avec le HC-SR04



Circuit fait sur breadboard

How power the module: 5V or 3.3V?

The HC-SR04 module is not the best choice for ESP32 boards because it operates with 5V (with 5V logic levels), whereas ESP32 works in 3.3V. It must be powered with 5V, and either level shifters or voltage dividers must be added to the `ECHO` pin. This will allow the ESP32 `GPIO18` pin to receive 3.3V instead of the original 5V, making the circuit a little more complex.

Tip

The ESP32 pins can support a voltage of 5V for temporary use. However, using this voltage for long-term use is not recommended as it can damage the pins and cause them to stop working. **For this reason, this tutorial has not included any instructions on how to use 5V for long-term use.**

Note

Many people have reported success in powering the module with 3.3V and obtaining accurate readings by increasing the duration of the pulse sent. However, I could not make the ones I had work with 3.3V.

If you're a beginner and use many boards that operate on 3.3V (e.g., ESP8266, ESP32, Raspberry Pi Pico), I recommend using a model specifically designed for this voltage, such as the **RCWL-1601** .

Easily measure a distance with the HC-SR04 and a MicroPython script

The ESP32 can easily calculate the distance from an obstacle without needing external libraries. When the ESP32 sends an impulse on the `TRIG` , the module sends a pulse to the `ECHO` , which is proportional to the distance from the obstacle.

It is enough to implement the triggering of measurement, retrieve the value and apply the following formula (the obtaining of this formula is detailed in the article on the operation of an ultrasonic sensor)

µ������������(��)=������������(µ�)×����(�/�)×10−42

This is what the code below does:

```
from machine import Pin, time_pulse_us
import time

SOUND_SPEED=340 # Vitesse du son dans l'air
TRIG_PULSE_DURATION_US=10

trig_pin = Pin(5, Pin.OUT)
echo_pin = Pin(18, Pin.IN)

while True:
    # Prepare le signal
    trig_pin.value(0)
    time.sleep_us(5)
    # Créer une impulsion de 10 µs
    trig_pin.value(1)
    time.sleep_us(TRIG_PULSE_DURATION_US)
    trig_pin.value(0)

    ultrason_duration = time_pulse_us(echo_pin, 1, 30000) # Renvoie le
temps de propagation de l'onde (en µs)
```

```
    distance_cm = SOUND_SPEED * ultrason_duration / 20000

    print(f"Distance : {distance_cm} cm")
    time.sleep_ms(500)
```

The program produces a 10µs pulse sent to the `GPIO5` of the ESP32 microcontroller.

The function `time_pulse_us()` pauses the program until it receives a response pulse from the HC-SR04 connected to pin `18` . After that, the code calculates the distance from the duration of the ultrasound wave. The program then shows the distance between the module and an object (like my hand in this example 🙂) in the serial monitor.

```
19      ultrason_duration = time_pulse_us(echo_pin, 1, 30000)
20      distance_cm = SOUND_SPEED * ultrason_duration / 20000
21
22      print(f"Distance (cm) : {distance_cm} cm")
23      time.sleep_ms(500)
24
```

Shell

```
Distance (cm) : 19.618 cm
Distance (cm) : 19.703 cm
Distance (cm) : 19.805 cm
Distance (cm) : 19.873 cm
Distance (cm) : 18.632 cm
Distance (cm) : 18.581 cm
Distance (cm) : -0.017 cm
```

Note

If you obtain negative values close to 0 (`Distance: -0.017 cm`), the ultrasonic module didn't receive the wave it sent out. This could be because the `time_pulse_us()` function timed out. To get accurate measurements, the surface of the obstacle should be flat.

MicroPython libraries for the HC-SR04

When you need to control multiple ultrasonic sensors and keep the calculations hidden, a library can be a great help.

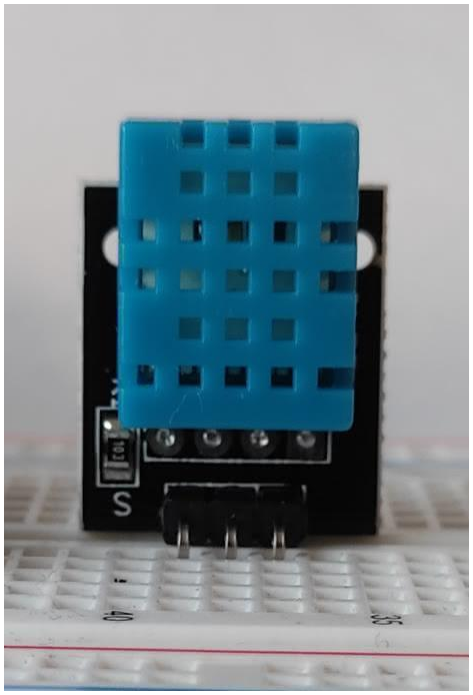
Basic library, with blocking measures

Here is a short library code to drive the ultrasonic sensor. You can either save it in a separate file on the ESP32 (with the name `hcsr04.py`) or include it directly in your main script:

```
import machine, time
from machine import Pin

class HCSR04:
            # © 2021 Roberto Sánchez Apache License 2.0

            # echo_timeout_us is based in chip range limit (400cm)
```

DHT11: Measuring temperature and humidity with ESP32 in MicroPython

(Updated at 01/05/2023)



The DHT11 sensor with its characteristic blue protection

The DHT11 sensor is viral among makers and DIY enthusiasts. It is so popular that it is included in MicroPython by default. Its low price and ease of use are the main reasons for this.

Warning

The main drawback is that you can only take a measurement once a second. However, this will be fine for making a DIY IoT weather station. 🙂

Getting started with the DHT11 sensor

The DHT11 and DHT22 sensors are used similarly but have some differences. The DHT11 has a blue housing, while the DHT22 has a white housing and is slightly larger. If you'd like to learn more about the differences, please look at this comparison. The DHT22 is the more advanced model and is more complete.

Some technical characteristics of the DHT11

The DHT11 can be powered directly by the 3.3V voltage of the ESP32.

DHT11 Specifications

Features	DHT11
Temperature accuracy	± 2°C
Humidity accuracy	± 5%
Temperature range	0-50 °C
Humidity range	20-90%

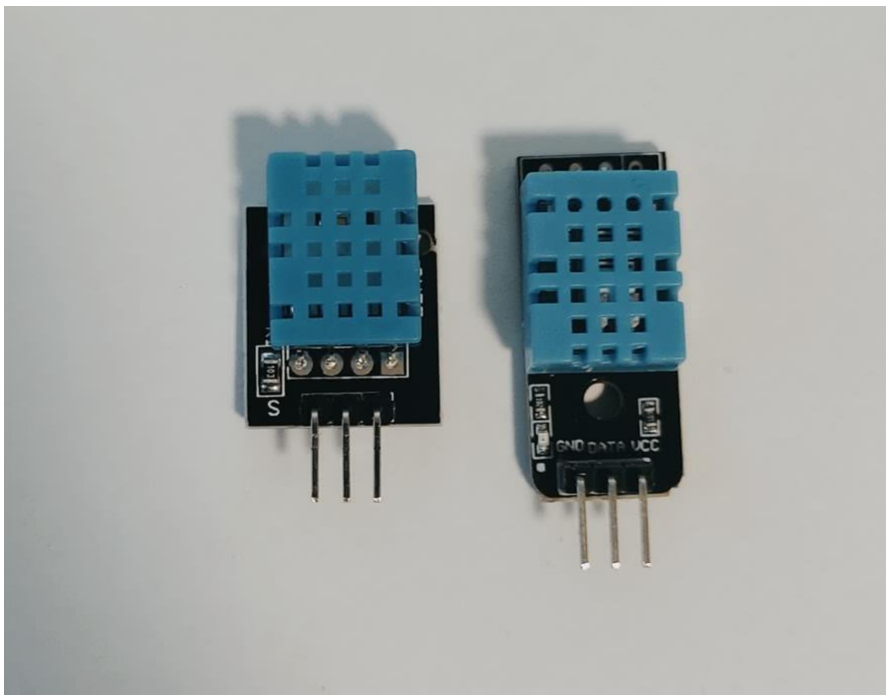
DHT11 Specifications

Features	DHT11
Sampling	1/s
Supply voltage	3-5.5V
Current	~2.5mA

Since the DHT11 measurements are straightforward, I suggest you use a BMExxx series sensor from the manufacturer Bosh for more accurate measurements. Of course, the cost is not the same. 😉

DHT11 Sensor Wiring on ESP32

The DHT11 sensor has between 3 and 4 pins depending on your module. If it has 4, one of them is not needed. Unfortunately, the type of pin-outs used depends on the module manufacturer. The numbering is done from the left when you hold the sensor facing you (the part with the grid in front of you). Here is a summary of the different possibilities:



Different pin-outs are possible (Elegoo's is on the left)

◉ 3 pins (Elegoo)

Pin Wiring

DHT11 Module	ESP32
1 (S)	GPIO17
2	3V3
3 (-)	GND

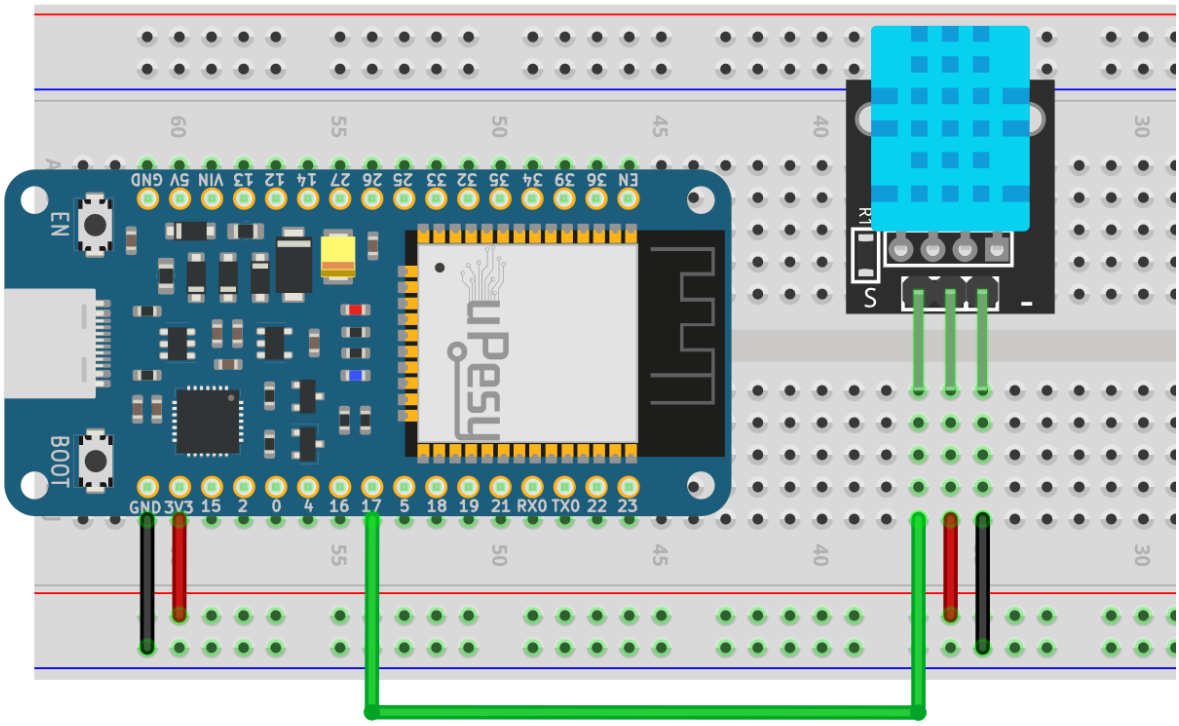
○ 3 pins (bis) ○ 4 pins

In addition to the pins supplying power to the DHT11, a single pin is used to send out the sensor data.

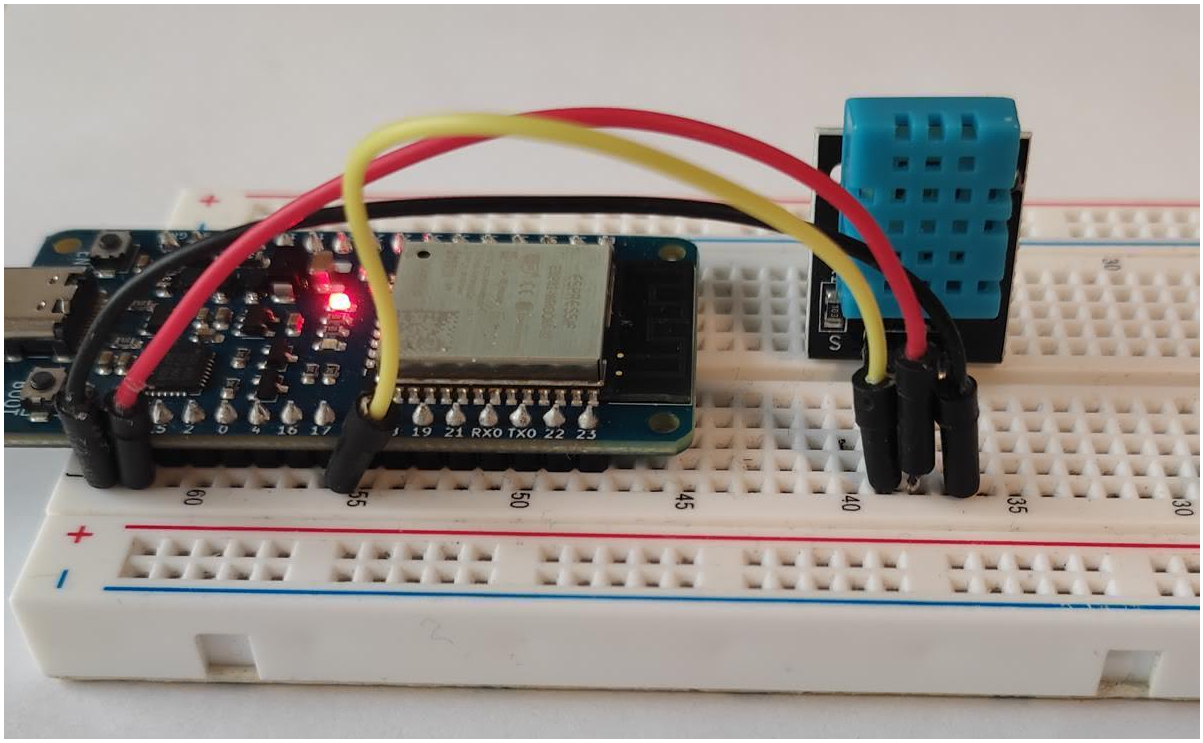
Electrical circuit for using the DHT11 module with an ESP32

If your module does not have a pullup resistor, you may need to add one. Connect a 4.7kΩ to 10kΩ resistor between pins ``3V3`` and the signal ``(GPIO17)`` . It doesn't hurt to add a resistor, even if it's not strictly necessary.

Any output pin can be used for the following circuits, with ``GPIO17`` as an example. The DHT11 sensor is from Elegoo, so modify it according to your model if it differs. 🙂



Wiring the DHT11 with a uPesy ESP32 Wroom DevKit board



Wiring the DHT11 with a uPesy ESP32 Wroom DevKit on a breadboard

Warning

Please ensure the module is provided with 3.3V instead of 5V to get a digital signal at 3.3V.

Measuring the temperature and humidity of the DHT11 with MicroPython

To communicate with a DHT11 sensor, you'll need to use a library as it uses a proprietary protocol. The good news is that MicroPython has this library built in, so you won't have to download anything extra! 😎

The code is simple: instantiating a `dht` with a PIN and using the `sensor.measure()` .

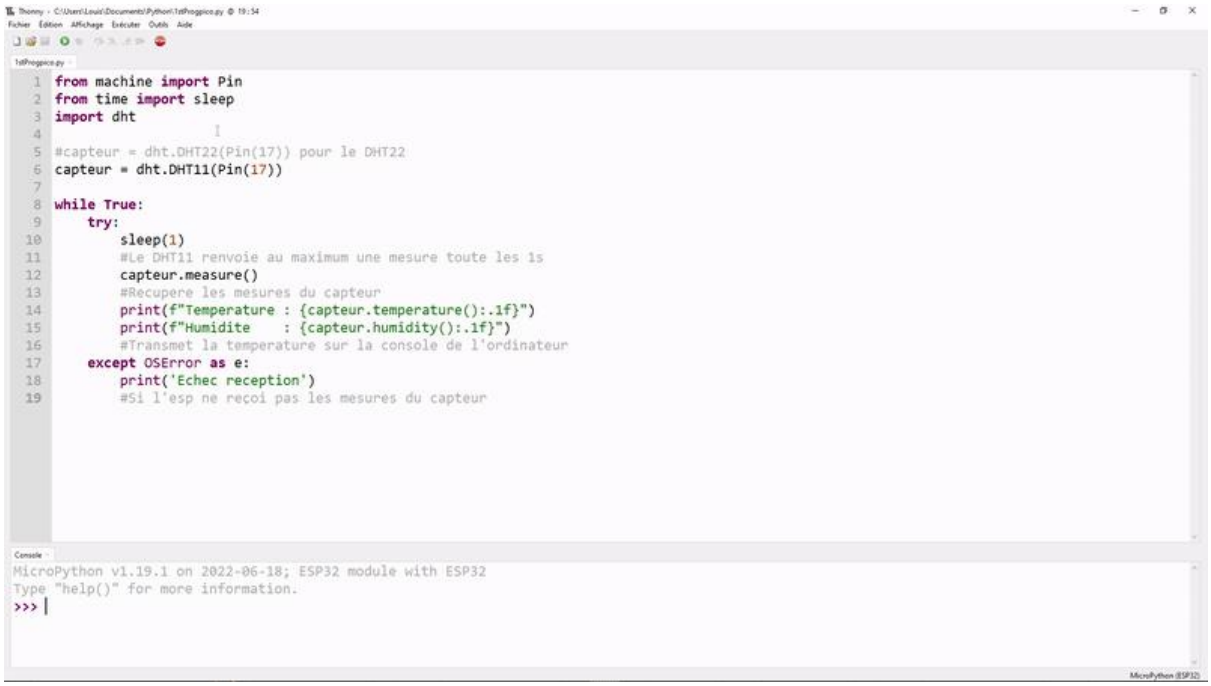
```
from machine import Pin
from time import sleep
import dht

sensor = dht.DHT11(Pin(17))

while True:
    try:
        sleep(1)
        # The DHT11 returns at most one measurement every 1s
        sensor.measure()
        # Retrieves measurements from the sensor
        print(f"Temperature : {capteur.temperature():.1f}")
        print(f"Humidity    : {capteur.humidity():.1f}")
        # Transmits the temperature to the computer console
    except OSError as e:
```

```
        print('Failed reception')
        # If esp does not receive the measurements from the sensor
```

Note

One second is added between each measurement because the DHT11 cannot go faster😮💨. If the DHT11 fails to keep up, an error will be captured thanks to the `try: except:` !

This is what is displayed in the Thonny IDE terminal. The temperature is expressed in degrees Celsius, while the humidity is expressed in percent.



Note

You can blow on the DHT11 sensor like a frosted window to remove fog.

In Python, you can use `f-string` to display variables with text easily `f"Humidity {capteur.humidity():.1f}"` . The string must start with an `f` , and the variable should be enclosed in square brackets. You can also specify the number of digits after the decimal point that should be displayed with `.1f`

Warning

You could even put zero because the sensor is only accurate to one degree!

Making asynchronous measurements with the DHT11 sensor

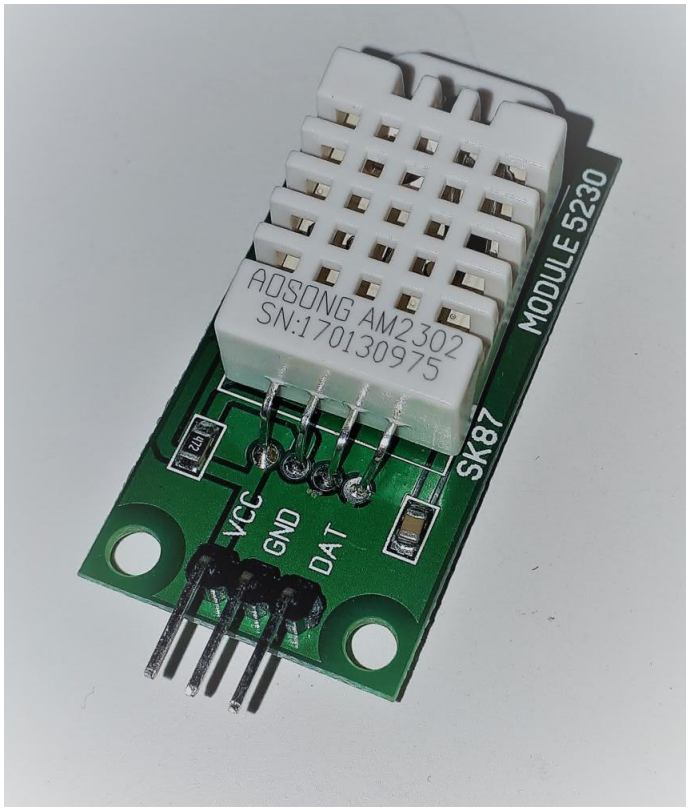
When using an ESP32, you may observe that it cannot perform any other tasks while measurements are being taken. This is due to the measurement process blocking the device from completing other operations.

The DHT22 is a more advanced model of the DHT11 sensor. A simple way to differentiate the two models is to look at the housing color: the DHT11 is blue, while the DHT22 is white. Additionally, the DHT22 is slightly larger. Despite the differences, the two sensors are used in the same way. For more information on the differences between the two, check out this comparison.

How to use the DHT22 Sensor with an ESP32 and a MicroPython script?

(Updated at 01/04/2023)

The **DHT22** is an improved version of the popular DHT11 sensor for DIY projects. It offers superior accuracy but has the identical drawback: it takes few measurements per second: only 1 every 2 seconds. Despite this, it may be suitable for creating a DIY IoT weather station.



The DHT22 sensor with its white housing

Note

The **DHT22** , also known as **AM2302.**

Getting started with the DHT22 sensor

Some characteristics of the DHT22 sensor

Technical characteristics of DHT22

Features	DHT22
Temperature accuracy	± 0.5°C
Humidity accuracy	± 2%
Temperature range	-40-80 °C
Humidity range	0-100%
Sampling	0.5/s
Supply voltage	3-6V
Current	~1.5mA

Note

The DHT22 is a better option than the DHT11 for outdoor temperature measurements since it can handle temperatures as low as -40°C. While it provides basic performance, it is not as dependable as Bosh's BMExxx sensors. Ultimately, it depends on your particular needs.

Connections of the DHT22 sensor

The DHT22 is a module with between 3 and 4 pins. It usually contains 4 pins, though one may not be utilized. Depending on the module, only 3 pins may be visible.

When connecting a sensor module, it is crucial to know that the pins' order may differ based on the manufacturer. To properly determine the order of the pins, hold the module, so the grid is facing you, and the numbering will start from the left. It should be noted that the power pin is always the first one, and the other pins may vary.

3 pins

Pin Wiring

DHT22 Module	ESP32
1 (VCC)	3V3
2 (GND)	GND
3 (OUT)	GPIO23

3 pins (bis) 4 pins (Raw DHT22)

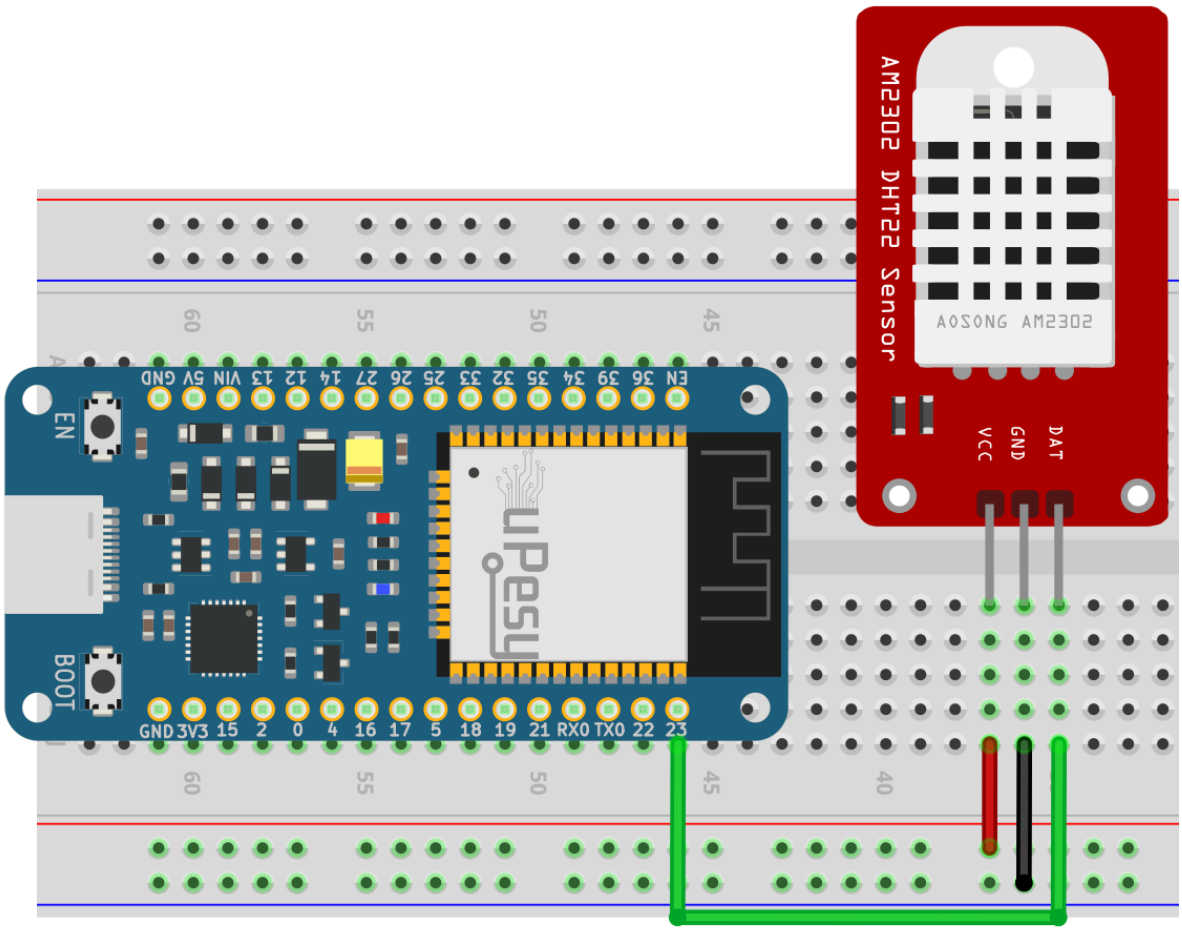
The DHT22 has only one data pin outside its power supply (One Wire Protocol).

Wiring diagram for using the DHT22 module with an ESP32

If your module does not have a pull-up resistor, you must add one between 4.7kΩ and 10kΩ between the 3V3 pin and the signal (GPIO23). Any output pin can be used on the ESP32, and

here we will use the `GPIO23` . Don’t forget to modify the circuit according to your model if it is slightly different.

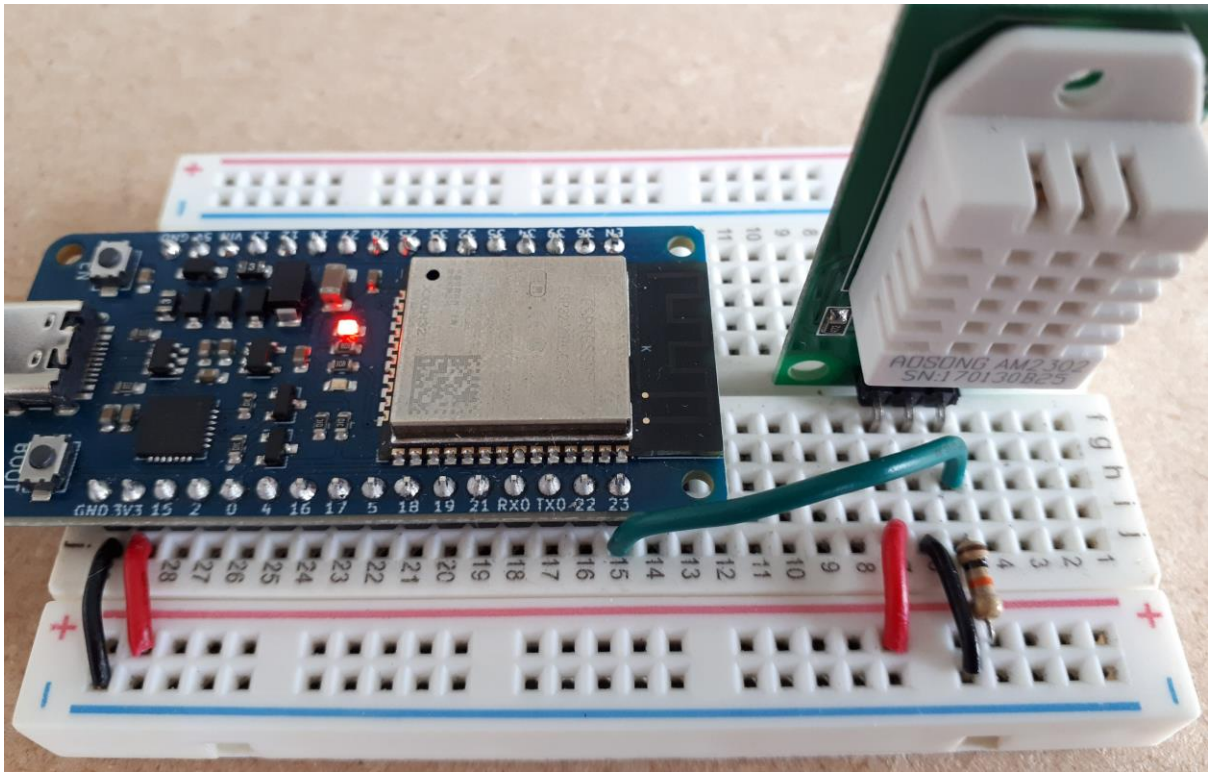
3 pins



Pin Wiring

3 pins (bis) 4 pins (Raw DHT22)

Here is an example of a circuit for the first type of DHT22 module:



Wiring the DHT22 with a uPesy ESP32 DevKit board

Measuring the temperature and humidity on the DHT22 with MicroPython

Utilizing the DHT22 requires a specific protocol, so it is essential to use a library to communicate with the sensor. Fortunately, no additional downloads are necessary as the library is included in the most recent versions of MicroPython. You can quickly import the module using the `import dht` command.

```
from machine import Pin
from time import sleep
import dht

sensor = dht.DHT22(Pin(23))

while True:
  try:
    sleep(1)      # the DHT22 returns at most 1 measurement every 2s
    sensor.measure()      # Recovers measurements from the sensor
    print(f"Temperature : {sensor.temperature():.1f}°C")
    print(f"Humidite    : {sensor.humidity():.1f}%")
  except OSError as e:
    print("Failed reception")
```

In the Thonny IDE terminal, you will see the temperature in °C and the humidity in percentage:

In Python, you can use an `f-string` to display variables with text. This starts with an `f` , and the variable must be enclosed in square brackets. You can also specify the number of decimal places to be displayed by adding `:.1f` . For example, `f"Humidite: {sensor.humidity():.1f}"` will display a number after the decimal point because the sensor is accurate within 0.5°C.

Calculate the heat index | Felt air temperature

The heat index is a way to measure the temperature. This value, expressed in °C, considers the temperature and the amount of moisture in the air. When the air is more humid, it feels hotter than the temperature indicates.

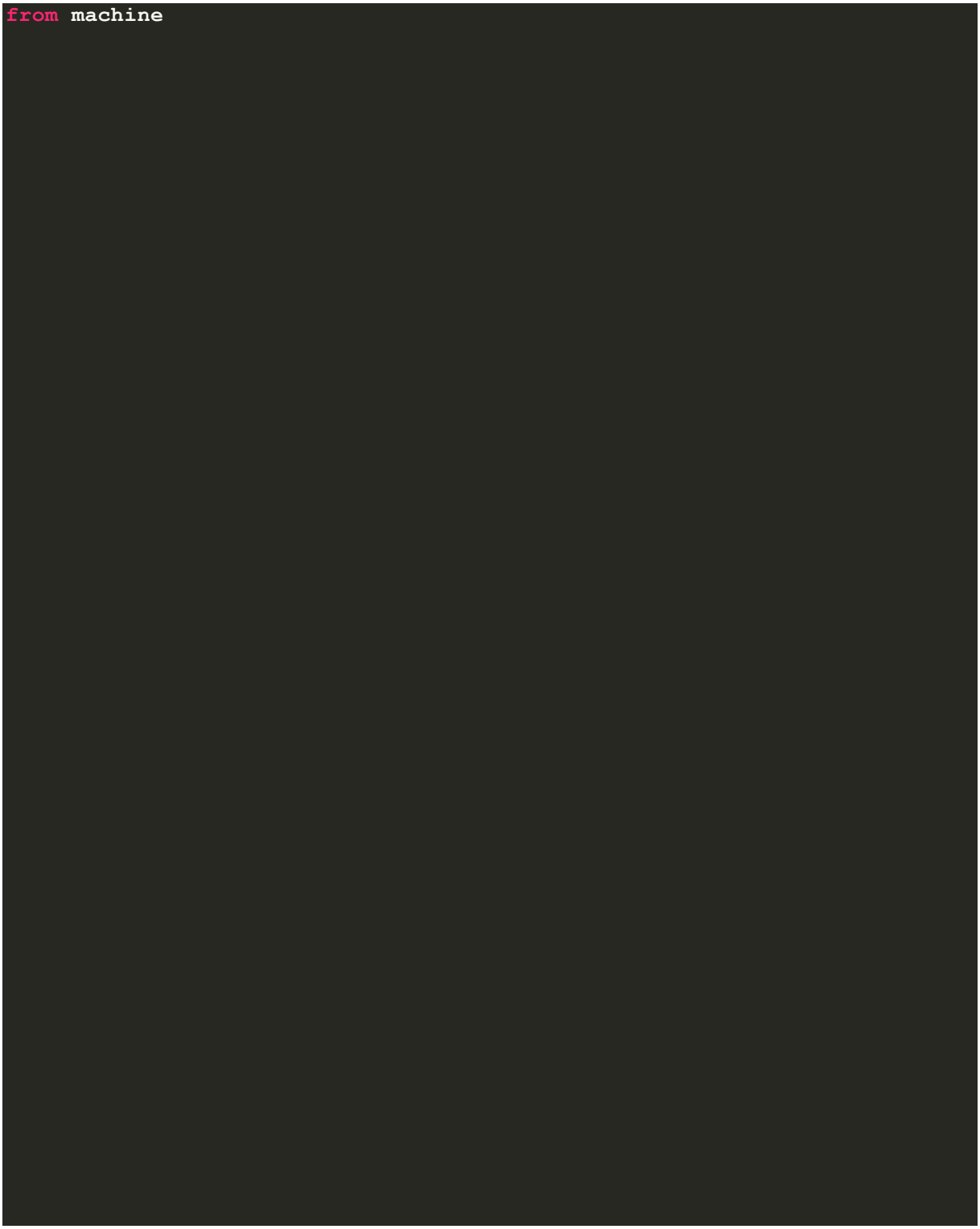
$$\mathrm{HI} = c_1 + c_2T + c_3R + c_4TR + c_5T^2 + c_6R^2 + c_7T^2R + c_8TR^2 + c_9T^2R^2$$

- HI = heat index (in degrees Celsius)
- *T* = ambient dry-bulb temperature (in degrees Celsius)
- *R* = relative humidity (percentage value between 0 and 100)

$$\begin{aligned} c_1 &= -8.784\,694\,755\,56, & c_2 &= 1.611\,394\,11, & c_3 &= 2.338\,548\,838\,89, \\ c_4 &= -0.146\,116\,05, & c_5 &= -0.012\,308\,094, & c_6 &= -0.016\,424\,827\,7778, \\ c_7 &= 2.211\,732 \times 10^{-3}, & c_8 &= 7.2546 \times 10^{-4}, & c_9 &= -3.582 \times 10^{-6}. \end{aligned}$$

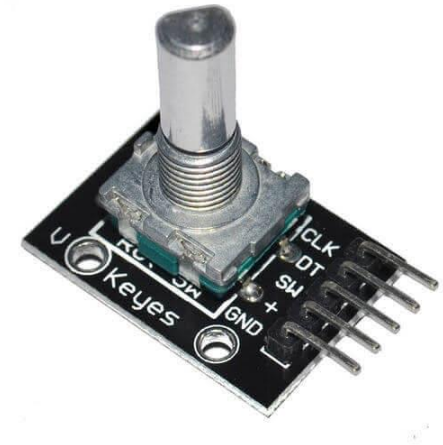
Formula from Wikipedia

Although the calculation formula is complex, I have provided a ready-to-use function that displays the result in the console. 😉



Using a rotary encoder in MicroPython with an ESP32

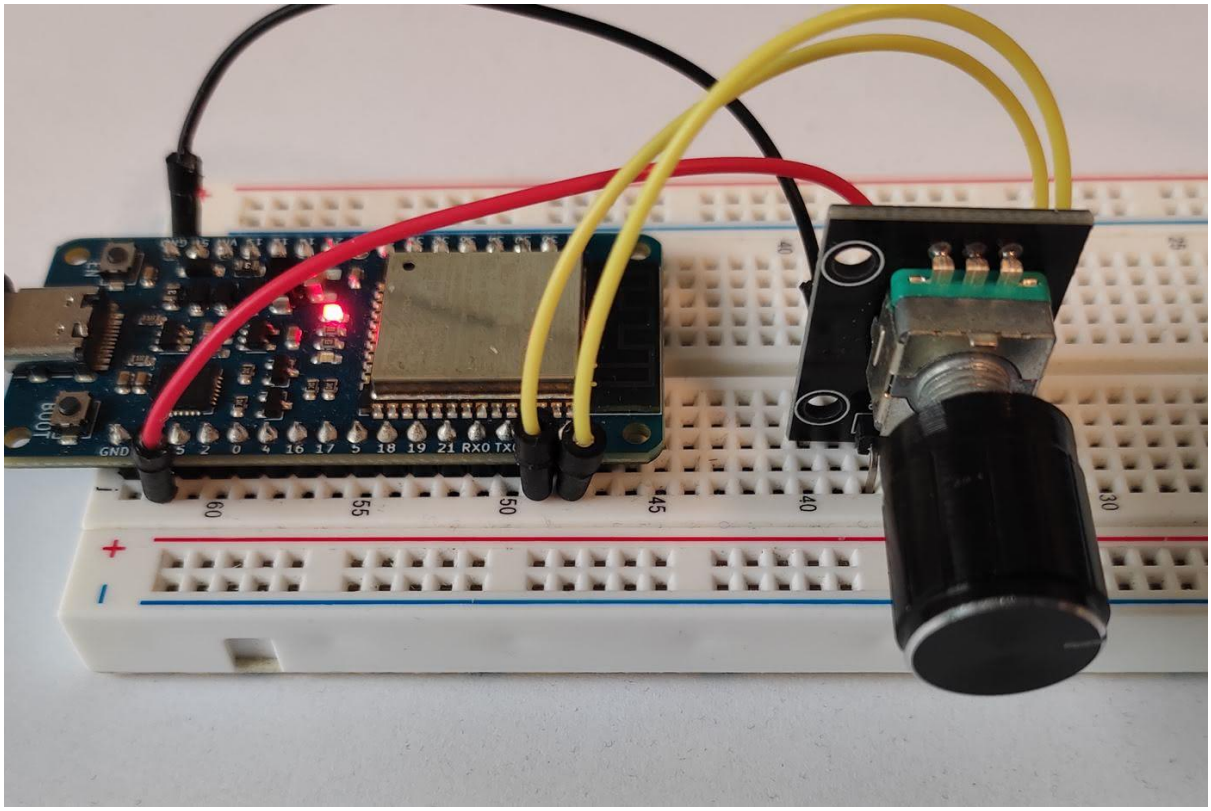
(Updated at 01/31/2023)



A rotary encoder is a kind of sensor that translates rotational motion into an output signal used to determine the direction and position of a spinning axis. It is mainly used for controlling servo motors but can also replace potentiometers in user interfaces. Additionally, many DIY kits come with a built-in push-button!

The control button on 3D printers is usually a rotary encoder

Handling a rotary encoder: the KY-040



This tutorial will explore using a `KY-040` rotary encoder module in a DIY project. A rotary encoder is an excellent device for creating a user interface that allows you to choose a menu, increase or decrease a variable, etc. This feature is commonly found in many DIY kits.

Difference between rotary encoder and potentiometer

The rotary encoder `KY-040` and the potentiometer may look similar, but they are different. The potentiometer is an analog sensor whose value is determined by its number of turns. On the other hand, the rotary encoder `KY-040` is a digital sensor and does not have a limited number of turns like the potentiometer.

The rotary encoder can detect a certain amount of “steps” for each revolution it goes through and sends a signal at each step. It can also turn infinitely, but the difference between it and a potentiometer can be felt through touch. A potentiometer turns smoothly while the rotary encoder turns jerkily.

A rotary encoder can be connected directly to a microcontroller and sends logic levels. Its resolution is based on the number of steps per revolution. In contrast, the resolution of a potentiometer is determined by the resolution of the analog-to-digital converter (ADC) needed to measure its position.

Note

A potentiometer is a device used to adjust values in an analog system, like the volume on an amplifier. A rotary encoder accurately measures the angle and direction of rotation.

How works a rotary encoder work?

This guide does not dive into technical aspects; however, it covers a lot of ground. If you would like to learn more about the physical operation, the various technologies, and the conventional topologies (such as understanding how an optical quadrature phase rotary encoder works 😨), please refer to the theoretical presentation of a rotary encoder.

Wiring the KY-040 rotary encoder to the ESP32

This rotary encoder has 2 signals - `CLK` and `DT` - to determine position. Additionally, the `SW` pin is connected to the integrated push-button switch.

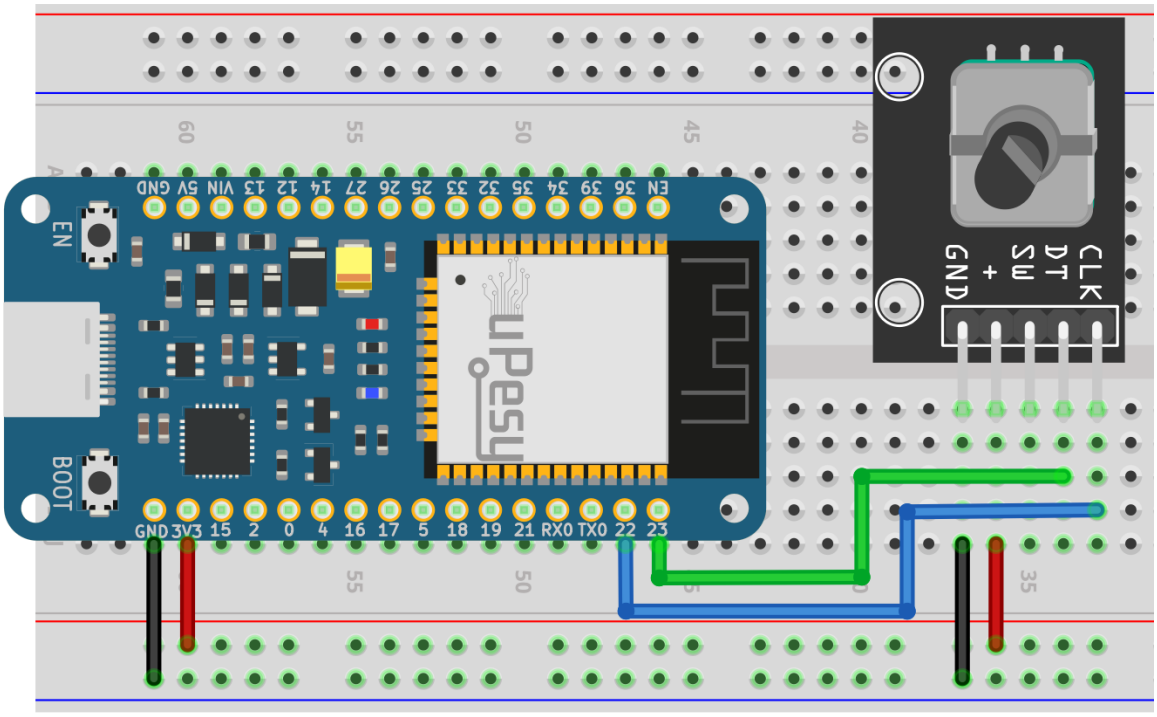
Warning

If you're new to using rotary encoders, you'll need to add pull-up resistors to the 3 logic pins. This is the same as when using a primary push-button circuit and is necessary for distinguishing between logic levels.

Here is a suggestion to connect to the uPesy ESP32 Wroom DevKit board:

Rotary Encoder	ESP32
`CLK`	`GPIO22`
`DT`	`GPIO23`
`SW`	Not used here
`+`	`3V3`
`GND`	`GND`

The scheme to be done is as follows:

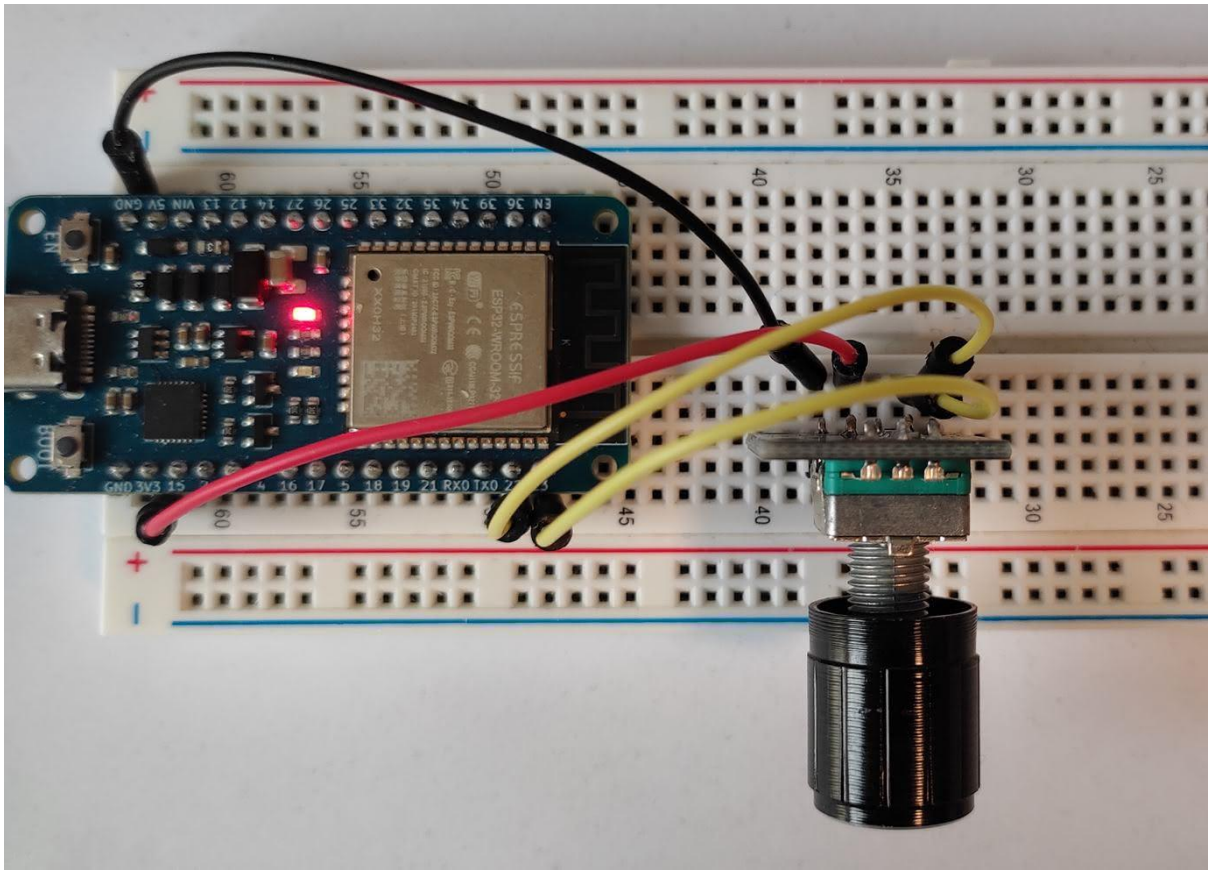


Electronic circuit to be replicated

Note

It would help if you supplied the encoder with 3.3V to have 3.3V logic levels (ESP32 friendly).

And here is an example of breadboard mounting:



Get the angular position of the KY-040 rotary encoder

Incremental encoders are helpful because they provide information on how many increments are made rather than the angle in degrees. To determine the angle, this information needs to be interpreted by a program. This allows us to determine the increment's value without knowing the exact angle.

Finding an efficient way to count the steps of an encoder is a critical task. A straightforward approach would be to continuously check if logical signals are being received from the encoder, but this method has its limitations; if the encoder is rotated rapidly, there is a risk that some steps may be missed. Thus, it is ideal to have a reliable method that accurately counts all the steps without blocking the program.

Note

At times, a program may end up counting in the wrong direction. (A classic bug)

An efficient solution is to incorporate hardware interrupts that respond to logic-level changes. This method allows the CPU to remain uninvolved, meaning the code is not blocked.

Note

The ESP32 can attach an interrupt to any output pin.

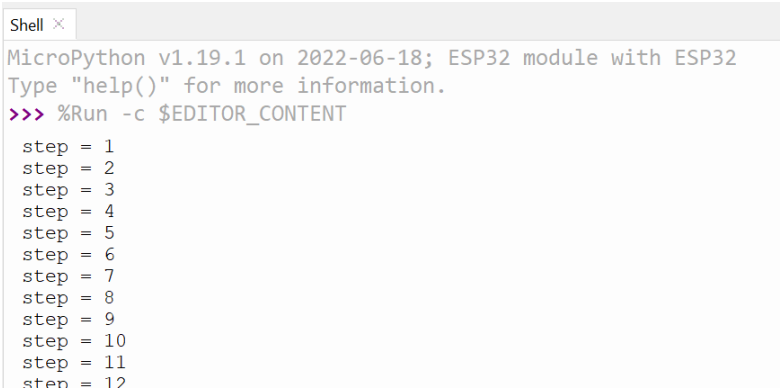
I recommend utilizing the [microPython library micropython-rotary] for asynchronously reading the increments of a rotary encoder. This library consists of two files that must be moved to your board, like any other microPython library. This process can be quickly completed within the Thonny IDE.

- `rotary.py <https://github.com/miketeachman/micropython-rotary/blob/master/rotary.py>`__
- `rotary_irq_esp.py <https://github.com/miketeachman/micropython-rotary/blob/master/rotary_irq_esp.py>`__

Here is a basic script that allows you to test the library and validate the electrical setup:

```
import time
from rotary_irq_esp import RotaryIRQ

r = RotaryIRQ(
    pin_num_clk=22,
    pin_num_dt=23,
    reverse=True,
    incr=1,
    range_mode=RotaryIRQ.RANGE_UNBOUNDED,
    pull_up=True,
    half_step=False,
)

val_old = r.value()
while True:
    val_new = r.value()

    if val_old != val_new:
        val_old = val_new
        print("step =", val_new)

    time.sleep_ms(50)
```

Creating an object of type `RotaryIRQ` lets you set a variety of options. Please visit the [library presentation page] for complete details. `RotaryIRQ` will take care of the work in the background, and the value can be accessed using the `r.value()` function.



Note

To make it turn the other way, you have two options: reverse the wires `CLK` and `DT` of the object, or change the value of `reverse` when creating it in the script.

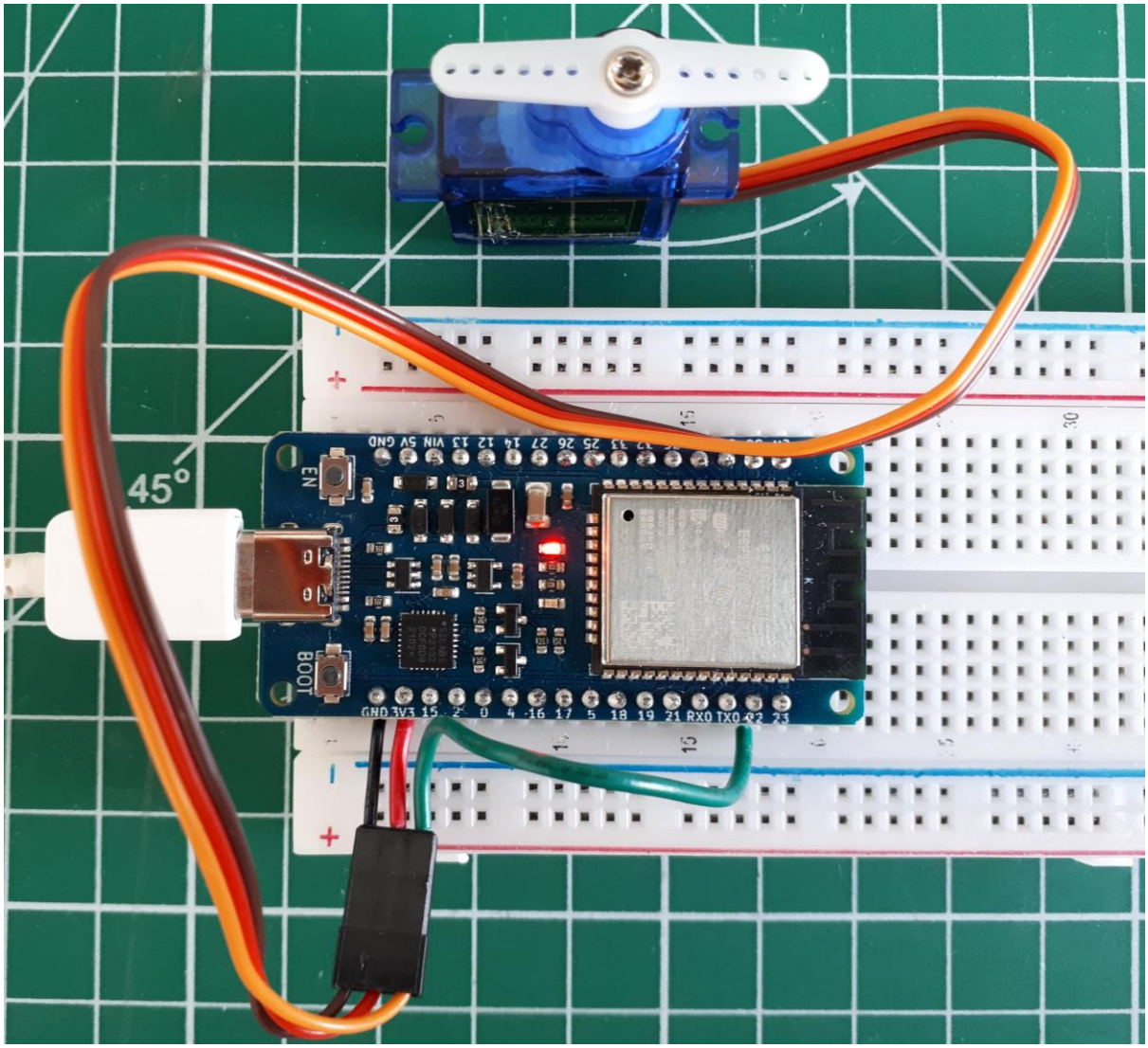
In this example, the encoder has no upper limit on the amount it can increment. The variable will increment without bounds until it is full. However, you can set a limit for the increment, for example, `-20` to `+20` , by altering the `range_mode` parameter.

Detecting the pressure of the push button

The library does not include a feature to detect a push button. Fortunately, it can be done with MicroPython code similar to how a classic button is detected. An interrupt can also be used to detect the pressing of a button asynchronously. For instance, it can be set up so that pressing the button will reset the count of the current counter.

Using a servo motor with an ESP32 board in MicroPython

(Updated at 01/06/2023)



Servo motors, frequently shortened to “servo”, are a special form of motor that can be fixed to a specific position with great precision. This position is maintained until a new instruction is given.

They have an excellent power-to-weight ratio. TowerPro’s popular SG90 blue servo has a torque of 1.5 kg/cm for only 9g. Its low price and ease of control from an ESP32 make it a popular choice for makers!

Note

Servos are widely used in model making (wheel steering in remote controlled cars, control of rudder and elevator on airplanes, etc.), but also in robotics and industry, for example to regulate liquid flows in valves.

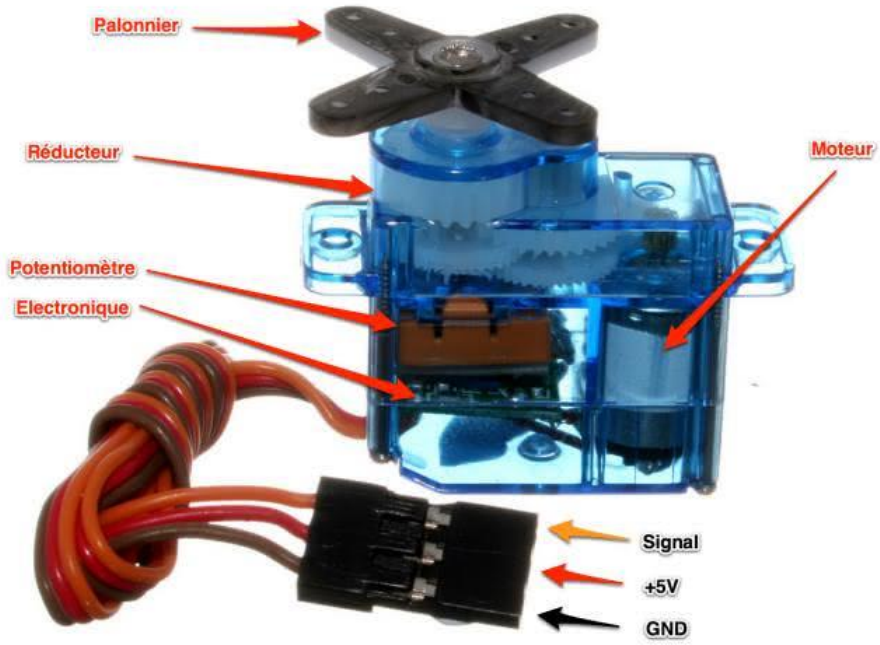
Getting to grips with the SG90 actuator

The main difference between a servo motor and a conventional motor is that most servo motors can only rotate between 0 and + 180 degrees. A servomotor consists of a conventional DC motor, various gears to increase torque (and reduce speed) and the servo system. The feat is to have managed to pack all this mechanism in such a small box!

Warning

The plastic stop on the actuator that limits rotation is relatively fragile. It is important to avoid turning the actuator shaft by hand and forcing it to the stop.

Operation of a servomotor



A small DC motor is connected to a potentiometer via an electronic circuit, which allows the speed of the motor to be finely regulated according to the position of the potentiometer. A series of gears is attached to the motor's output shaft to multiply the torque while reducing its speed. When the motor turns, the gears drive the movement of the arm which in turn drives the potentiometer. If the movement stops, the electronic circuit continuously adjusts the motor speed to keep the potentiometer and therefore the arm at the same position. This feature is particularly useful for robot arms that do not fall back under their own weight when the movement stops.

Note

It's kind of like an intelligent engine 🙂

This small servomotor is controlled using a pulse width modulated (PWM) signal with a frequency of 50 Hz, i.e. one pulse every 20ms. The position of the actuator is determined by the duration of the pulses, usually varying between 1ms and 2ms.

How to power the actuator with an ESP32

In the SG90 servo data sheet the optimal power supply is 5V. However, it seems to work also with 3.3V.

Note

With a voltage of 5V, the rotation will be slightly faster!

A servo motor consumes a lot of current, especially when it exerts a lot of torque. Since the 5V pin on most ESP32 boards comes directly from the USB bus, you will be limited to a maximum of 500mA. With 1 or 2 servo motors connected the ESP32 should hold the load.

Beyond 2, use a separate power supply instead. In this case, be sure to connect a pin `GND` from the board to the negative terminal of the actuator power supply 😉.

Warning

On uPesy ESP32 boards, the self-resetting fuse may trip if the current is too high.

SG90 actuator connections to the uPesy ESP32

An SG90 servo motor contains 3 wires: 2 for the power supply and 1 for the PWM control signal. The colors of the wires allow to differentiate them:

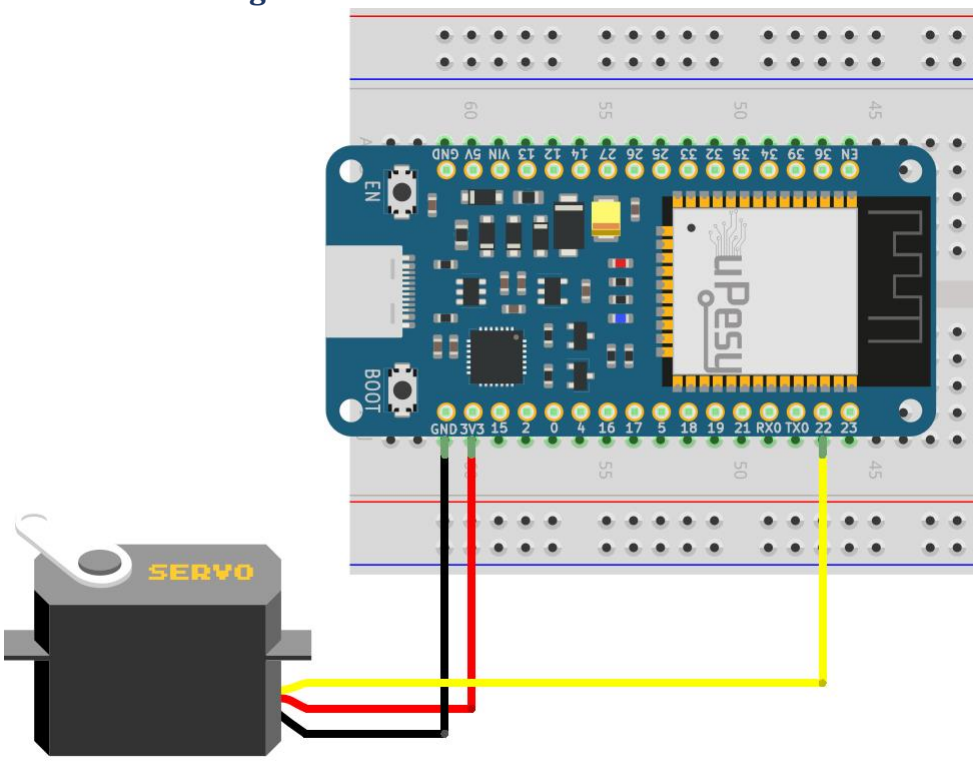
Servomoteur SG90	Couleur du fil	ESP32
`GND`	Marron	`GND`
`5V`	Rouge	`5V` ou `3V3`
Signal `PWM`	Orange	`GPIO22`

Note

On some servo motor models, the signal wire color is yellow or white instead of orange.

Any output pin of the ESP32 can be used to control the servo motor because the ESP32 pins are all capable of producing a PWM output.

Circuit for driving a servomotor with an ESP32



Control a servo motor from the ESP32 with a Python script

In fact, since it is the pulse width of the PWM signal that tells the servo motor the desired angular position, there is no real need for a library to master the beast 😎. But it can quickly become tedious to calculate the right values, especially when you want to drive several at the same time. That's why I'll recommend using a ready-made library instead to simplify your life in our future DIY projects.

Basic Python script to drive the servo with its own calculations

With the SG90 servo from TowerPro, the minimum position corresponds to a pulse width of 0.5ms and the maximum position to one of 2.4ms. By doing some calculations, we can deduce the duty-cycle of the PWM, then the value in bits of the pulse width. I will not detail the calculations, because we will rather use a ready-made library for the continuation.

Note

The difficulty is to find the right PWM pulse width to obtain a given angular position.

Here is a basic script:

```
from machine import Pin,PWM
```

```
import time

sg90 = PWM(Pin(22, mode=Pin.OUT))
sg90.freq(50)

# 0.5ms/20ms = 0.025 = 2.5% duty cycle
# 2.4ms/20ms = 0.12 = 12% duty cycle

# 0.025*1024=25.6
# 0.12*1024=122.88

while True:
    sg90.duty(26)
    time.sleep(1)
    sg90.duty(123)
    time.sleep(1)
```

I grant you that it is not very easy to understand. That's why we are going to use a library!

Control a servo via a Python script with the library `servo.py`

I suggest you to use this following MicroPython library to easily control the servo motor. It is installed like other MicroPython libraries: copy and paste the file on your ESP32 board in the MicroPython file manager.

Warning

In the current version of MicroPython for the ESP32 (v1.19), this library is not present as standard. Only the Pyboard board has a library `servo` included as standard in MicroPython.

```
from machine import Pin, PWM

class Servo:
    # these defaults work for the standard TowerPro SG90
    __servo_pwm_freq = 50
    __min_u10_duty = 26 - 0 # offset for correction
    __max_u10_duty = 123- 0  # offset for correction
    min_angle = 0
    max_angle = 180
    current_angle = 0.001


    def __init__(self, pin):
        self.__initialise(pin)


    def update_settings(self, servo_pwm_freq, min_u10_duty, max_u10_duty,
min_angle, max_angle, pin):
        self.__servo_pwm_freq = servo_pwm_freq
        self.__min_u10_duty = min_u10_duty
        self.__max_u10_duty = max_u10_duty
        self.min_angle = min_angle
        self.max_angle = max_angle
        self.__initialise(pin)


    def move(self, angle):
```

```
        # round to 2 decimal places, so we have a chance of reducing
unwanted servo adjustments
        angle = round(angle, 2)
        # do we need to move?
        if angle == self.current_angle:
            return
        self.current_angle = angle
        # calculate the new duty cycle and move the motor
        duty_u10 = self.__angle_to_u10_duty(angle)
        self.__motor.duty(duty_u10)

    def __angle_to_u10_duty(self, angle):
        return int((angle - self.min_angle) *
self.__angle_conversion_factor) + self.__min_u10_duty


    def __initialise(self, pin):
        self.current_angle = -0.001
        self.__angle_conversion_factor = (self.__max_u10_duty -
self.__min_u10_duty) / (self.max_angle - self.min_angle)
        self.__motor = PWM(Pin(pin))
        self.__motor.freq(self.__servo_pwm_freq)
```

Warning

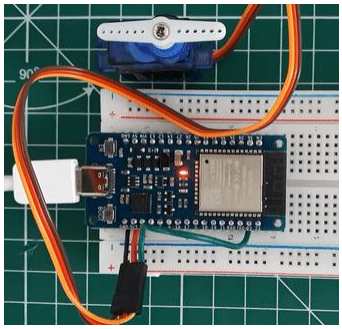
The code of this library works only for ESP32 boards (the code is slightly different for the Raspberry Pi Pico for example)

To use the library, it is very simple. After importing the class `Servo` object, we define a `Servo` which represents our blue servomotor. We specify the pin used to drive it in the manufacturer's parameters.

Then we indicate the desired angular position with the function `.move(angle)` .

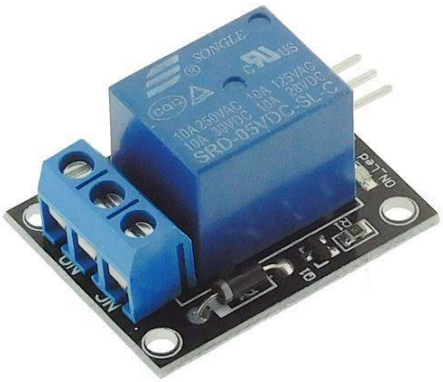
```
from servo import Servo
import time

motor=Servo(pin=22) # A changer selon la broche utilisée
motor.move(0) # tourne le servo à 0°
time.sleep(0.3)
motor.move(90) # tourne le servo à 90°
time.sleep(0.3)
motor.move(180) # tourne le servo à 180°
time.sleep(0.3)
motor.move(90) # tourne le servo à 90°
time.sleep(0.3)
motor.move(0) # tourne le servo à 0°
time.sleep(0.3)
```

Here is a demonstration video with the above code:



Control electrical devices with a Relay and an ESP32 in MicroPython

(Updated at 01/09/2023)



Relay is essential for creating DIY home automation projects. Connecting it to a WiFi-enabled ESP32 allows you to control your home appliances from your mobile device, laptop, or remote location.

Warning

However, handling mains voltages can be hazardous. If you must use it, do so with extreme caution and for continuous use, it is best to opt for quality relay modules and to confirm its installation by a professional before use.

Getting started with a Relay

Using a relay is very simple, mainly when it is sold as a ready-to-use module.

Note

I strongly recommend using a module that incorporates a relay with its minimal circuitry rather than utilizing the relay alone and making the circuit for the logic part yourself (the relay itself does not fit on a breadboard and should not be placed on it anyway).

Presentation and functioning of a Relay

If you have never used a relay before or if you are curious to understand how it works, its uses and its limits, I recommend you to read the theoretical article on how a relay works .

Relay is a device that can be used with ESP32 to control the power of a DC electrical circuit, such as RGB LED strips or pumps. It can also control appliances connected to the 220V mains, such as fans, lights, and heaters.

Note

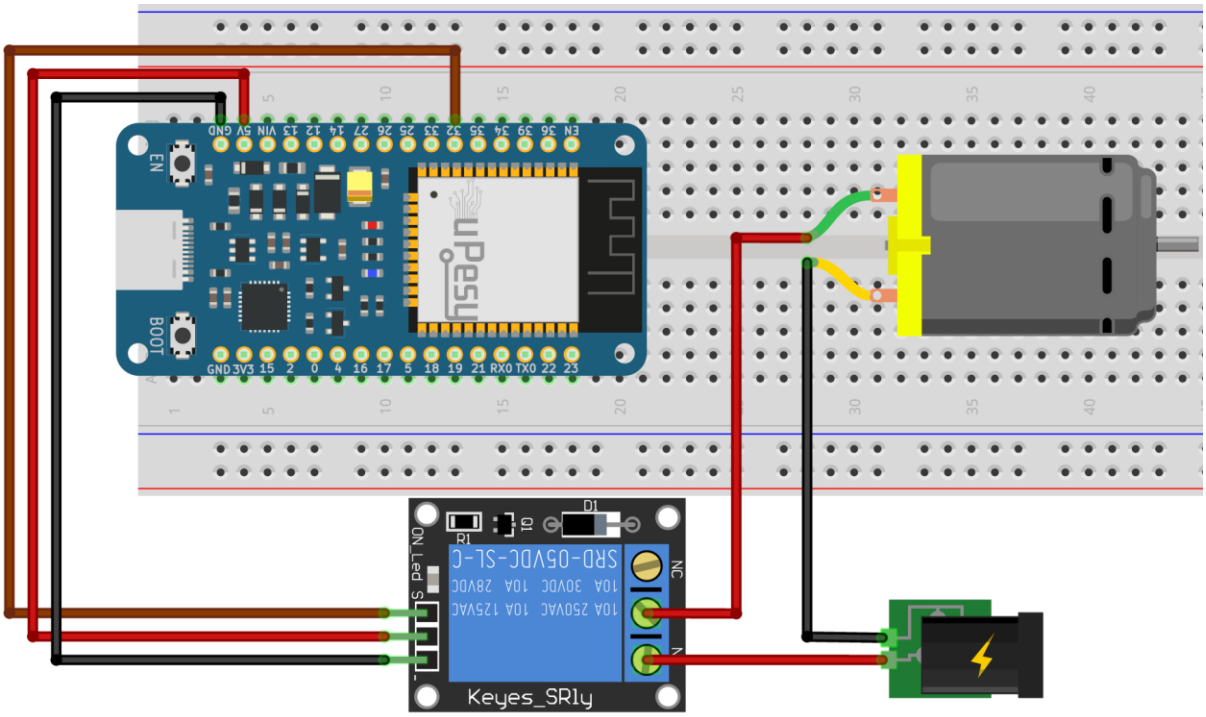
If the purpose is to switch a motor on and off frequently, the transistor circuit would be more appropriate (for direct current, of course).

Connection of the SRD-05VDC-SL-C Relay

In most Arduino kits, the modules use the relay **SRD-05VDC-SL-C** from the manufacturer **Songle** . There are generally two varieties: a module with a single relay and one with several relays simultaneously. If you plan to drive several power circuits separately in a project, choose the multi-relay module instead.

Note

If you’re using an ESP32 powered by a LiPo battery, the available voltage will usually fall in the range of 3.3V to 4.2V. As such, the **SRD-03VDC-SL-C** model, which runs on 3.3V, is more suitable than the **SRD-05VDC-SL-C** model, which requires 5V.



Electrical circuit to be made

Warning

When a relay is used, the grounds of the logic part and the power part are entirely separate and must never be connected, primarily if the circuit is supplied with 220V AC.

Pin Wiring

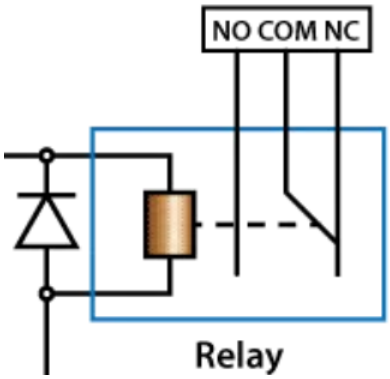
Relay Module	ESP32
S	32
VCC	5V
GND	GND

The model **SRD-05VDC-SL-C** has three pins to drive the relay:

- The signal pin `S` drives the relay, which is connected to a pin of the ESP32.
- The power supply pin is connected to the `5V` module of the ESP32 (or the `3V3` if it is the SRD- module **03VDC** -SL-C).
- The final pin, indicated by – is the ground, is linked to pin `GND` .

There is also a 3-pin terminal block at the power side:

- `NC` → Normally Closed contact.
- `COM` → the middle pin is called common (COM).
- `NO` → Normally Open: Normally Open contact.



Output terminals of the relay

Depending on the application, the power circuit device must be connected to the terminal COM and either NO or NC . If you select NO , the relay will be in an open state by default (the electrical circuit will not be complete). The relay will only close the circuit when a signal is received.

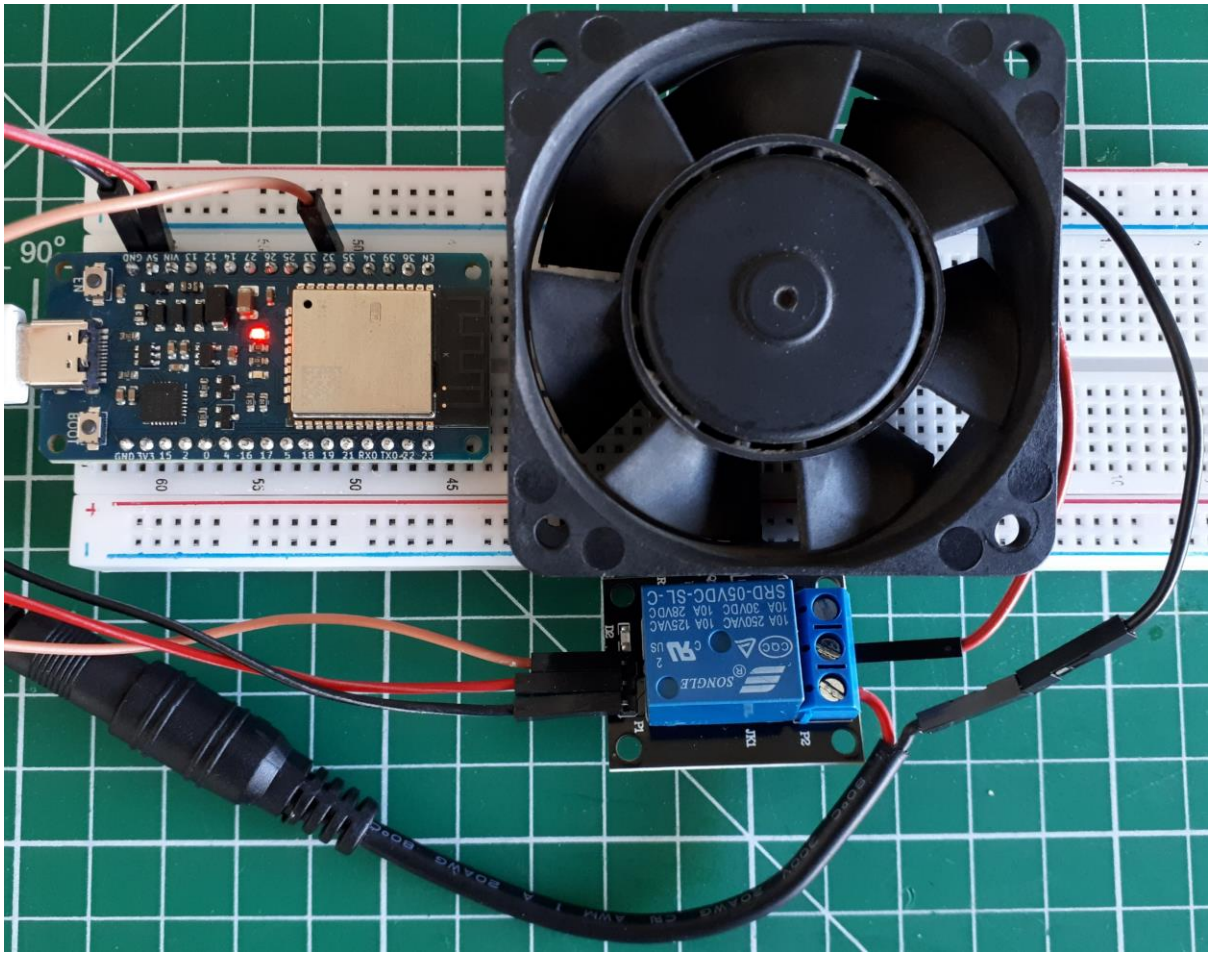
The circuit is closed by default with the COM and NC terminals selected, meaning the connected device is on. When the relay activates, the circuit opens, and the device no longer receives power.

In the event of a control fault (when the relay is out of service), selecting the safest possible mode is essential. For instance, if a heating resistor needs to be powered, it is best to ensure the circuit remains open if the relay stops working.

Note

I suggest using the combination COM and NO to operate any device, such as a lamp, motor, or pump.

Here is an example of using a relay to turn on a large computer fan powered by 12V.



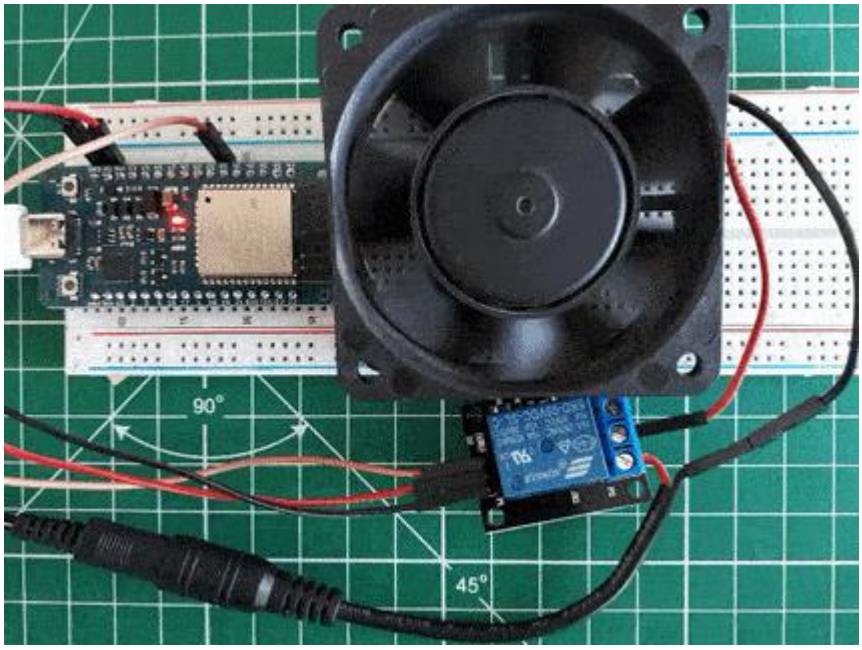
The NO and COM terminals are used for this configuration

Controlling the Relay with Python on the ESP32

The MicroPython script is straightforward, like turning on or off an LED. The code is identical to the script `blink` .

```
from machine import Pin
import time

pin_relay = Pin(32, mode=Pin.OUT)

while True:
    pin_relay.on()
    time.sleep_ms(200)
    pin_relay.off()
    time.sleep_ms(5000)
```

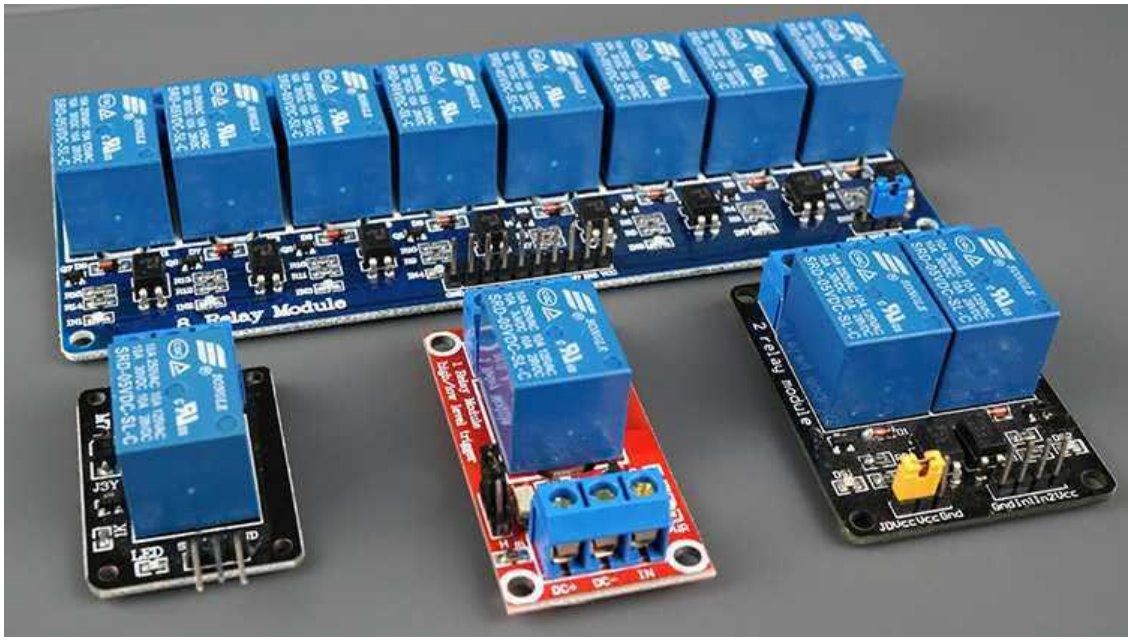
And here is what it gives with the proposed circuit:



Note

The module has an LED to indicate the status of the relay. It lights up when the relay is activated.

Des modules relais avancés multicanaux



There are a variety of relay modules available, ranging from one to eight channels. Each module can control a corresponding number of outputs and typically contains a blue SRD-05VDC-SL-C relay.

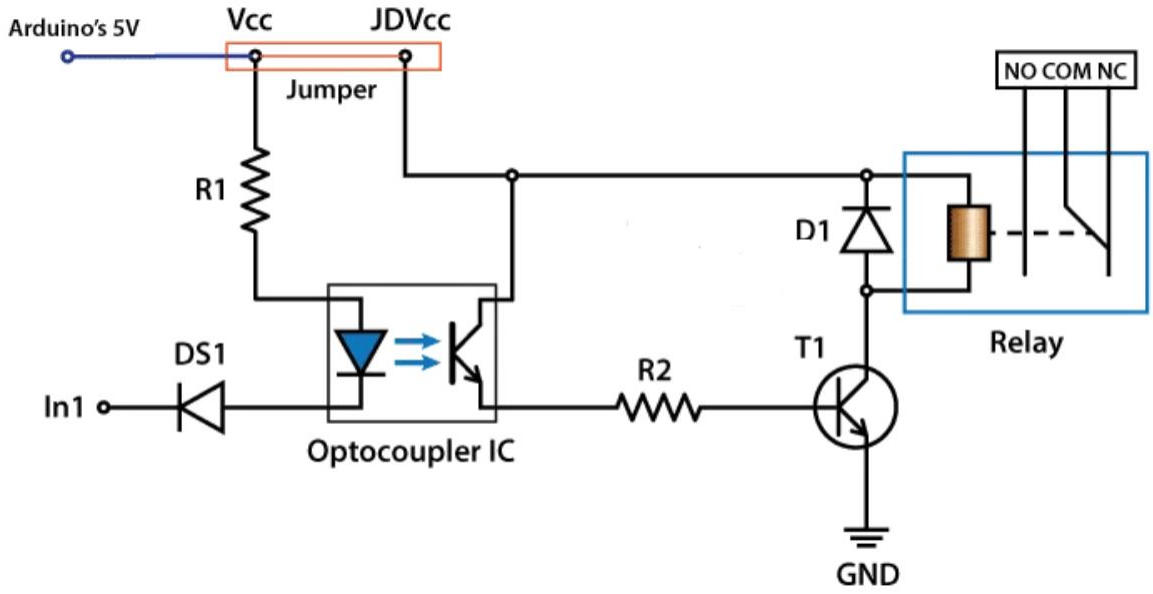
Separate the ESP32 from the Relay with the Optocouplers

Models with two or more channels usually incorporate an optocoupler to isolate the relay from the ESP32 completely.

Note

An optocoupler is a device that consists of a light-sensitive transmitter and receiver, enabling complete isolation of the logic signal.

For this layer of protection to be effective, the relay circuit must be**supplied with an external voltage source of 5V** on pin `JD-VCC` . These relay modules contain additional pins with a jumper `JD-VCC` , `VCC` and `GND` to enable or disable this option.



The optocoupler is useless if the jumper is positioned between `JD-VCC` and `VCC` . It may be necessary to use an external power supply if the relays draw too much current for the electromagnet.

Use multi-channel Relay modules

The operation of a multi-channel relay is similar to that of a single relay. Multiple pins `VIN` `VIN1` , `VIN2` can be used to control the relays independently:

```
from machine import Pin
import time

pin_relay_0 = Pin(32, mode=Pin.OUT)
pin_relay_1 = Pin(33, mode=Pin.OUT)

while True:
    pin_relay_0.on()
            pin_relay_1.on()
    time.sleep_ms(200)
```

```
    pin_relay_0.off()
            pin_relay_1.off()
    time.sleep_ms(5000)
```