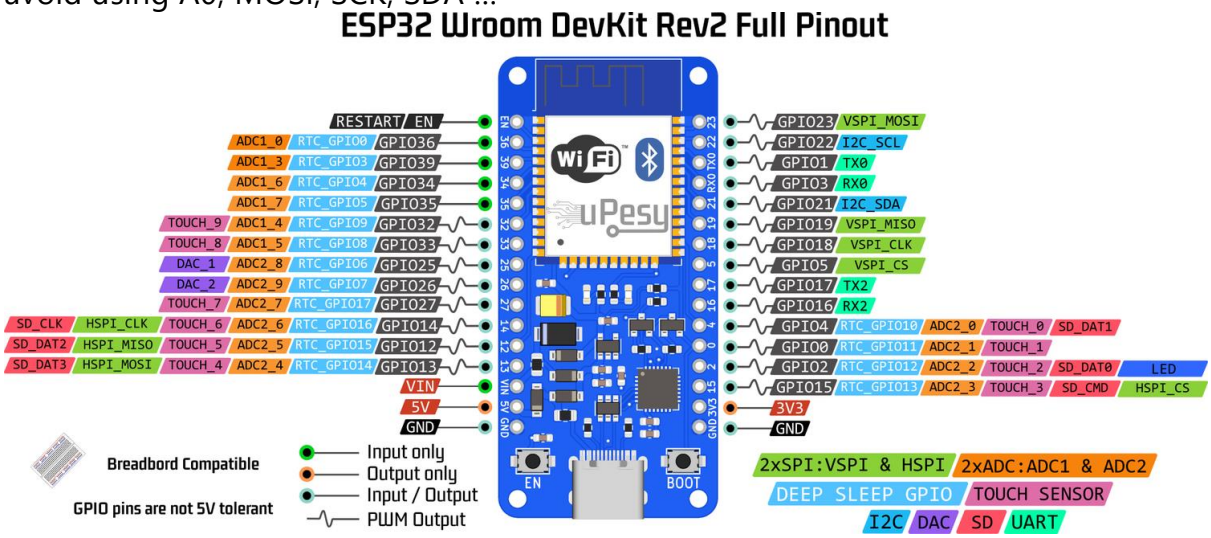
Control ESP32 GPIO Pin

(Updated at 11/28/2022)

The functions to be used are the same as with an Arduino. We use the `pinMode()` method to configure a pin as a digital input or output, the `digitalWrite()` function to impose a voltage **of 0V or 3.3V** on the output and the `digitalRead()` function to read a logic level (either 0V or 3.3V) on the input.

Note

The pin numbers are those indicated in the grey “GPIO” insert on the board’s pinout. Even if there is a link between the number of GPIO pins of the ESP32 and those usually used on the Arduino, to avoid confusion, it is better not to **use them and to use the native pins number of ESP32 exclusively.** You should therefore avoid using A0, MOSI, SCK, SDA ...



Pinout of the uPesy ESP32 Wroom Devkit board

Configuration of ESP32 input or output pins

- Configuration of the GPIO2 pin as output:

```
pinMode(2, OUTPUT);
```

- Configuration of the GPIO15 pin as input:

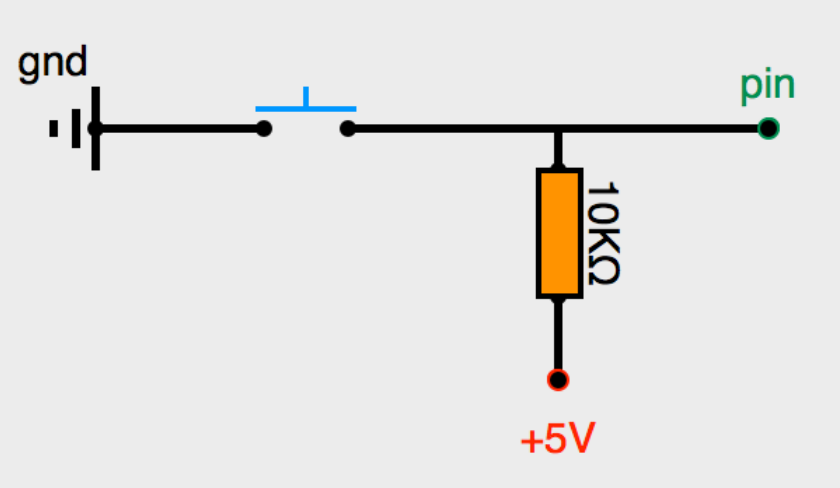
```
pinMode(15, INPUT);
```

A quick reminder on pull-up and pull-down resistors

The measured value is random if we measure a voltage from a pin that is not connected to the ground or a non-zero voltage. This is due to the electrical noise generated by several factors (the wire behaves in particular like an antenna). The most concrete example is when you use a push button.

Two ways to eliminate noise and get reliable measurements :

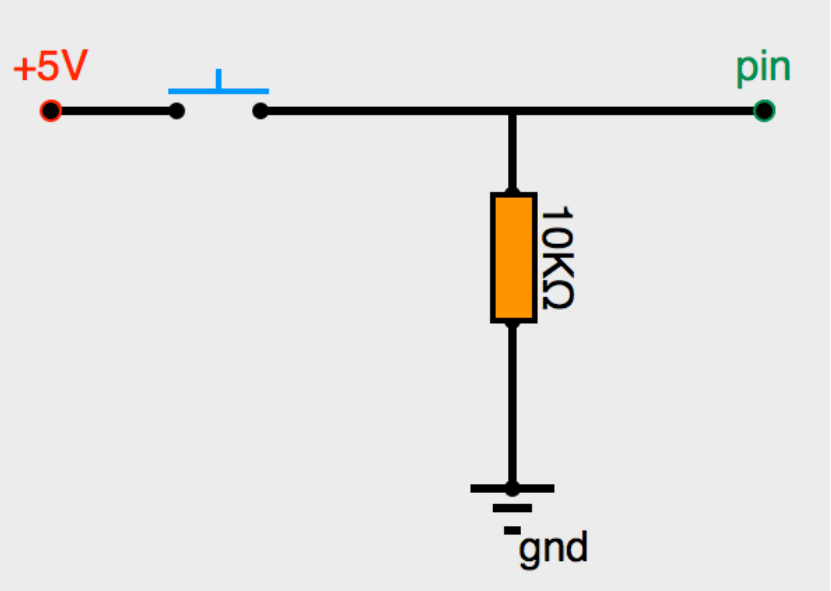
- With a pull-up resistor



Using a pullup resistor with a push button

This circuit allows only two possible voltages: **0V when the button is pressed; otherwise +3.3V**

  - When the switch is opened (button released), the + 3.3V feeds the pin of the ESP32, and the measured value of the pin will give HIGH (or 1).
  - When the switch is closed (button pressed), the +3.3V and the pin are absorbed by the mass. The pin measurement will give LOW (or 0).
- With a pull-down resistor



Using a pulldown resistor with a push-button

This circuit allows you to have only two possible voltages: **+3.3V when you push the button; otherwise, 0V.**

- If the push button is pressed, the current goes from +3.3V to the pin of the ESP32. It will not take the path of the mass because the current always flows by the least resistive path. The ESP32 pin will receive +3.3V and indicate HIGH (or 1).
- If the push button is released, the circuit is opened. The mass will absorb the very low residual current. The pin will therefore be in LOW (or 0).

All GPIO pins (except GPIO36, GPIO39, GPIO34, and GPIO35 pins) have these two circuits internally in the ESP32.

Use ESP32 internal pull-up and pull-down resistors

- Configuration of the GPIO15 pin as input with the internal pull-up resistor:

```
pinMode(15, INPUT_PULLUP);
```

- Configuration of the GPIO15 pin as input with the internal pull-down resistor :

```
pinMode(15, INPUT_PULLDOWN);
```

Note

Unlike the Arduino, on the ESP32, there are internal pull-down resistors.

Set or read a voltage

- To set a voltage **of 3.3V** at the output of pin 2 of the ESP32:

```
pinMode(2, OUTPUT); // set the pin as output
digitalWrite(2, HIGH)
```

Note

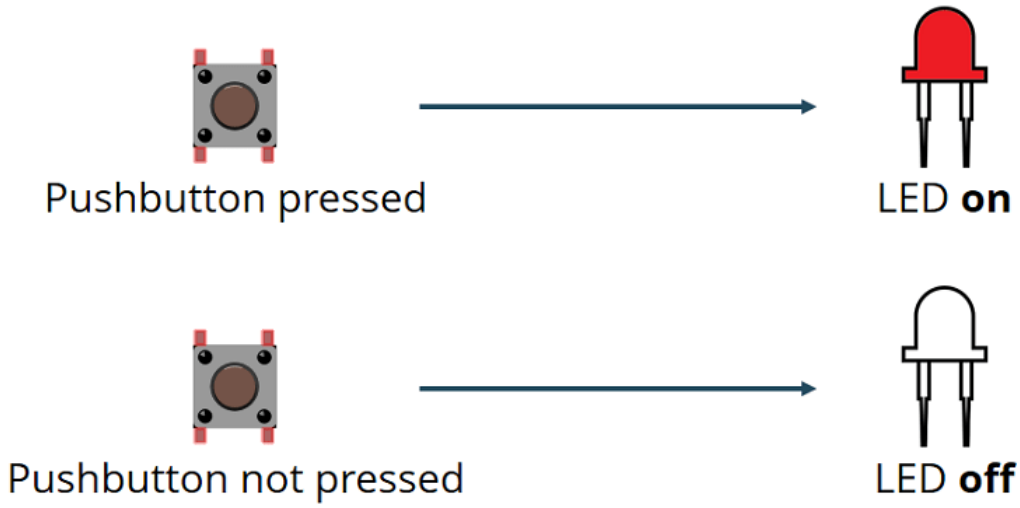
The output voltage of pins is 3.3V (and not 5V like on Arduino)

- To read a voltage **of a logic level of 0V or 3.3V** on pin 15 of the ESP32:

```
pinMode(15, INPUT); // set the pin as input
digitalRead(15);
```

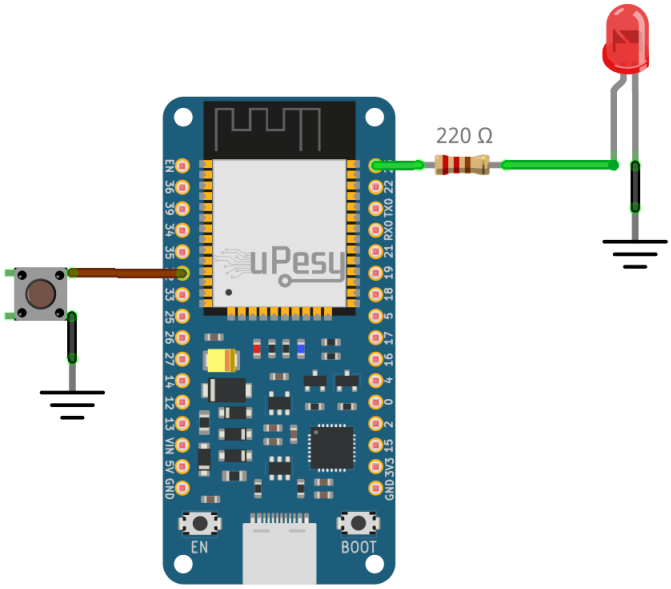
Mini-Project: Controlling an LED with a button

To show how to use the digital inputs/outputs of the ESP32, we will accomplish a short project that turns on the LED when a button is pressed.



Electrical schematic

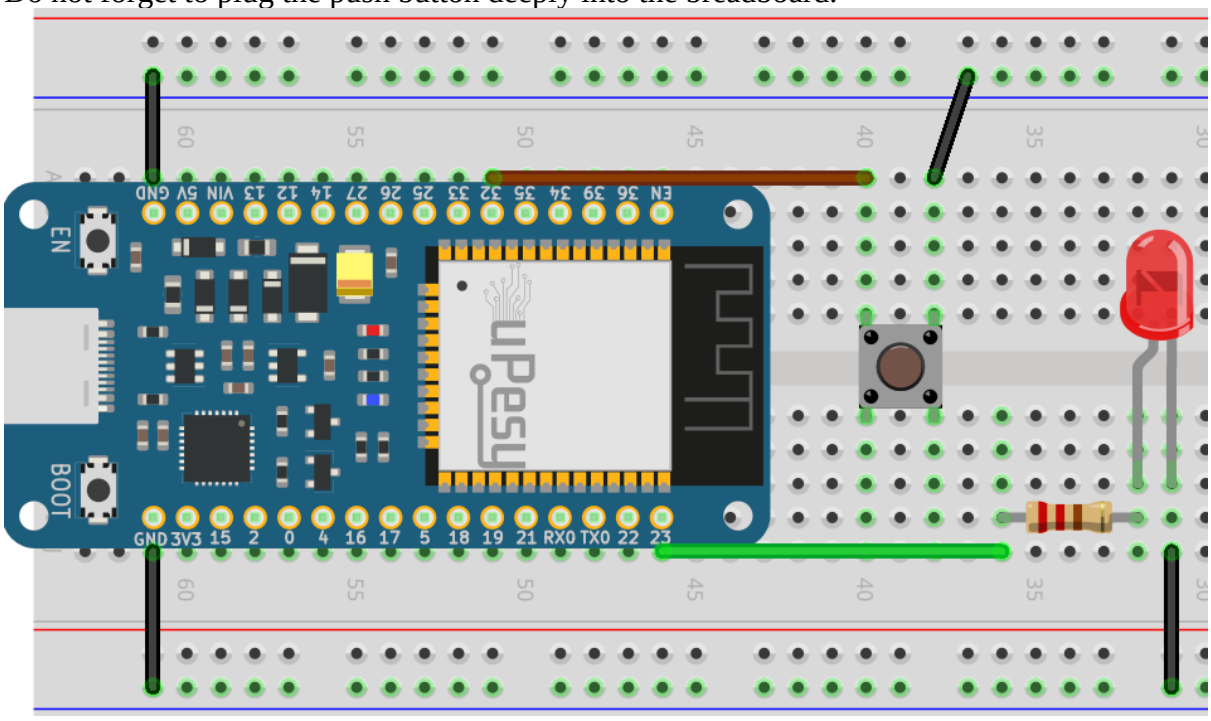
We will use the internal pull-up resistors to simplify the electronic schematic.



Electrical circuit

Warning

Do not forget to plug the push button deeply into the breadboard.



Electrical circuit

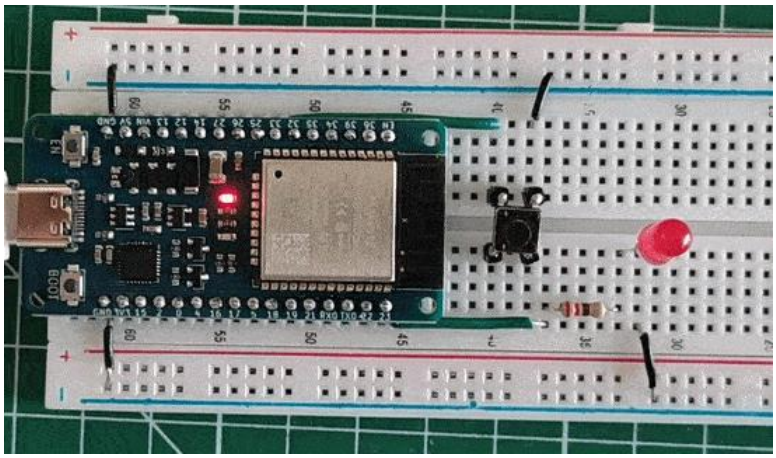
Code to turn on the LED when the button is pushed

Solution

```
const int buttonPin = 32;
const int ledPin =  23;
```

```
// State of the push button
int buttonState = 0;

void setup() {
    Serial.begin(115200);

    //Set the pin as an input pullup
    pinMode(buttonPin, INPUT_PULLUP);
    pinMode(ledPin, OUTPUT);
}

void loop() {

    buttonState = digitalRead(buttonPin);
    Serial.println(buttonState);

    if (buttonState == LOW) {
        // Switch on the led
        digitalWrite(ledPin, HIGH);
    } else {
        // Switch off the led
        digitalWrite(ledPin, LOW);
    }
}
```

Final result

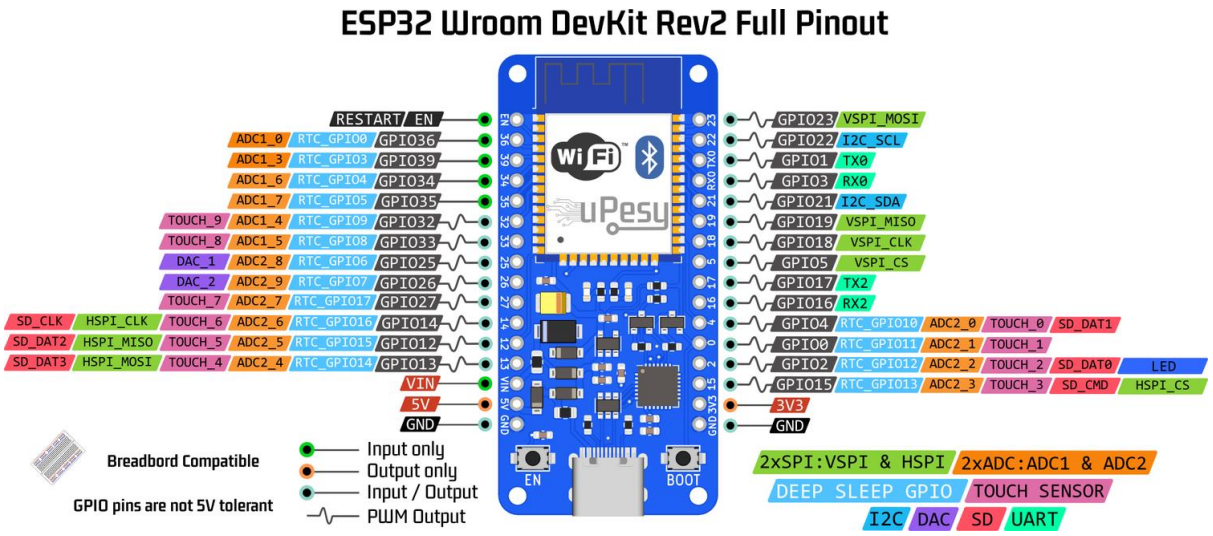


Measure analog voltage on ESP32 with ADC

(Updated at 11/28/2022)

As its name suggests, the ADC (Analog to Digital Converter) allows it to convert an analog voltage into a binary value.

There are 2x12 bits ADCs on the ESP32, the ADC1 with 8 channels and the ADC2 with 10 channels. Each channel of the ADC allows measuring a pin.



Pinout of the uPesy ESP32 Wroom Devkit board

ADC limitation on the ESP32

The ADC is not a strong point of the ESP32 because it has many flaws. Use the Arduino one or an external ADC if you want to make accurate measurements.

Warning

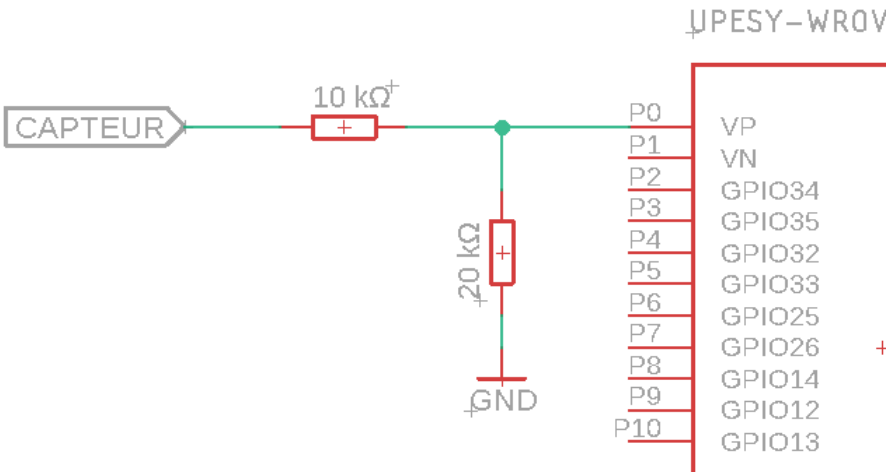
Although it sounds strange, the Arduino's 10-bit ADC (1024 values) is more accurate and reliable than the ESP32's 12-bit (4096 values).

The ADC of the ESP32 has several flaws:

- **ADC2 cannot be used with enabled WiFi** since it is used internally by the WiFi driver. Since there is a good chance of using WiFi on a microcontroller designed to use it, only the ADC1 and its 8 channels can be used.
- **The ADC can only measure a voltage between 0 and 3.3V.** You cannot directly measure analog voltages between 0 and 5V.

  Note

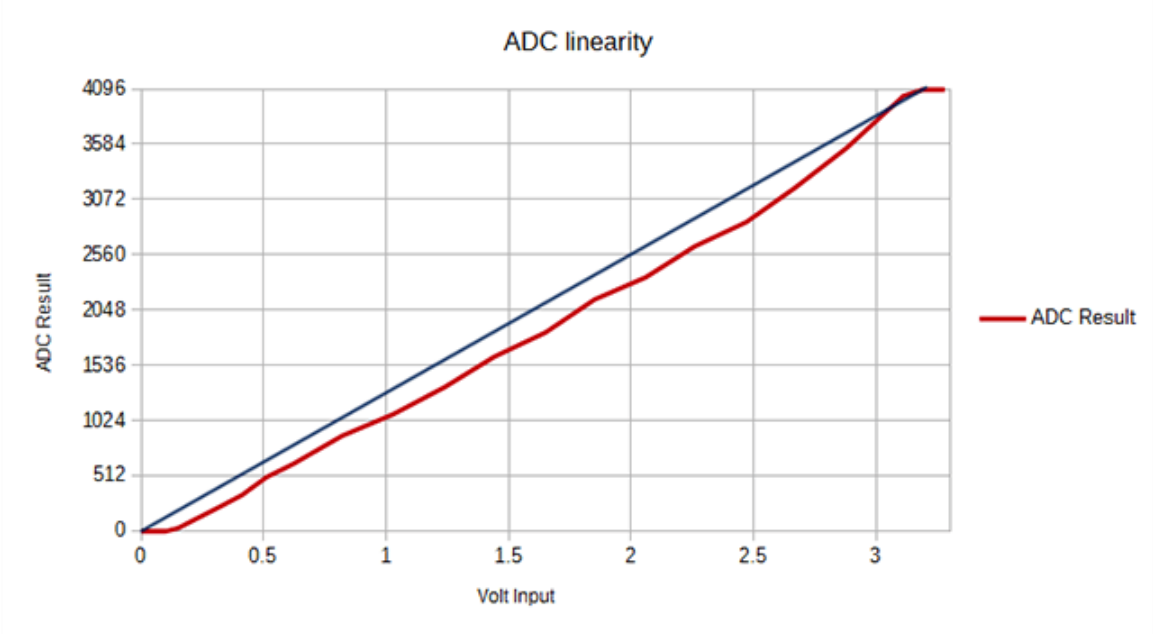
  A voltage divider bridge can reduce a voltage between 0 and 5V to a voltage between 0 and 3.3V.



Voltage divider bridge to switch from a voltage between 0-5V to 0-3.3V

- **Non-linearity**

  The ADC of the ESP32 is not very linear (the ADC response curve is not a linear line), especially at the ends of its operating range (around 0V and 3.3V)



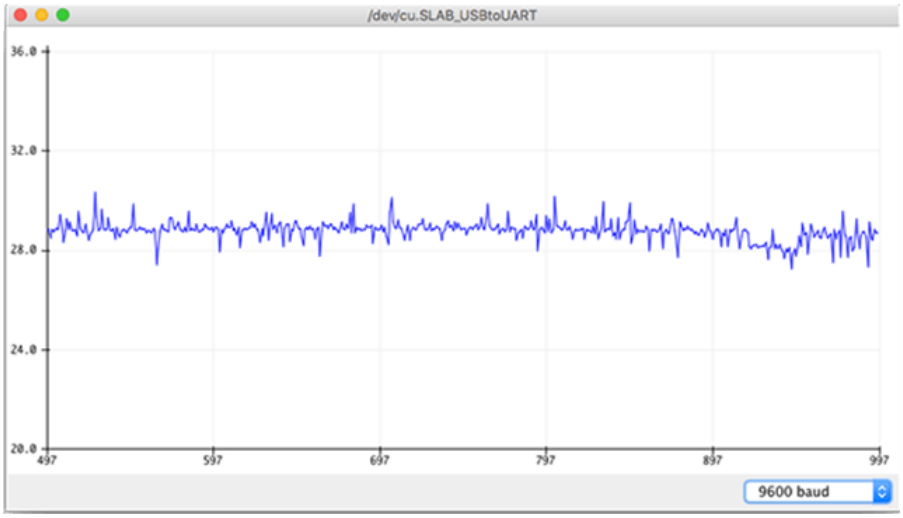
Non-linearity of the ESP32 ADC

Basically, this means that the ESP32 cannot distinguish a signal of 3.2V and 3.3V: the measured value will be the same (4095). Likewise, the ESP32 will not differentiate between a 0V and 0.2V signal for small voltages.

Note

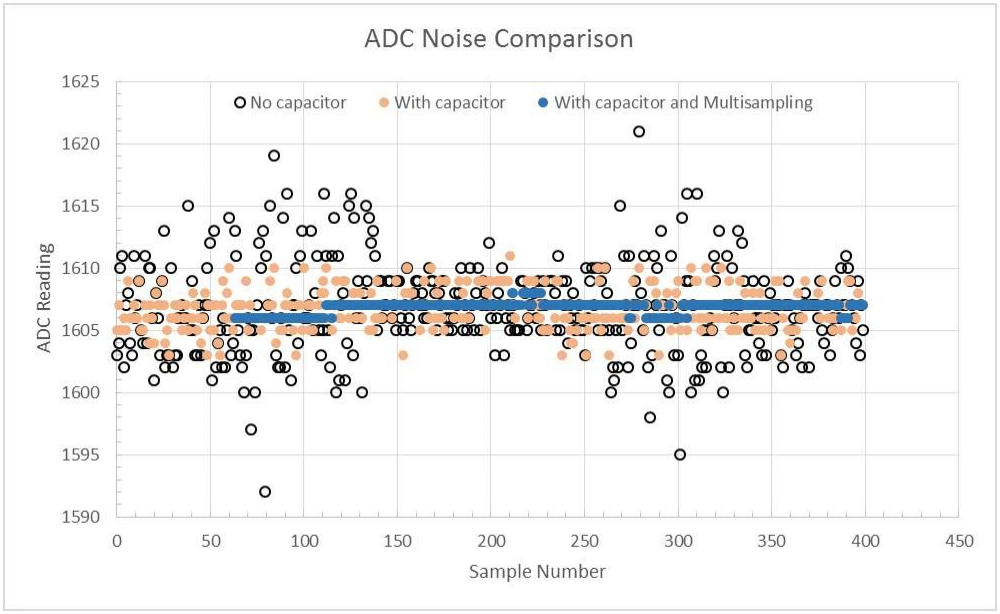
It is possible to calibrate the ADC to reduce this linearity flaw. An example is available here .

- **The electrical noise** of the ADC implies a slight fluctuation of the measurements:



Electrical noise of the ESP32 ADC

Here too, it is possible to try to “correct” this defect by adding a capacitor at the output and with oversampling :



Correction of the electrical noise of the ADC of the ESP32

Usage

The primary use of the ESP32 ADC is the same as on the Arduino with the function `analogRead()` .

- To read the voltage of the VP pin (GPIO36) of the ESP32:

```
pinMode(36, INPUT); //It is necessary to declare the input pin
analogRead(36);
```

Note

There are also more advanced functions.

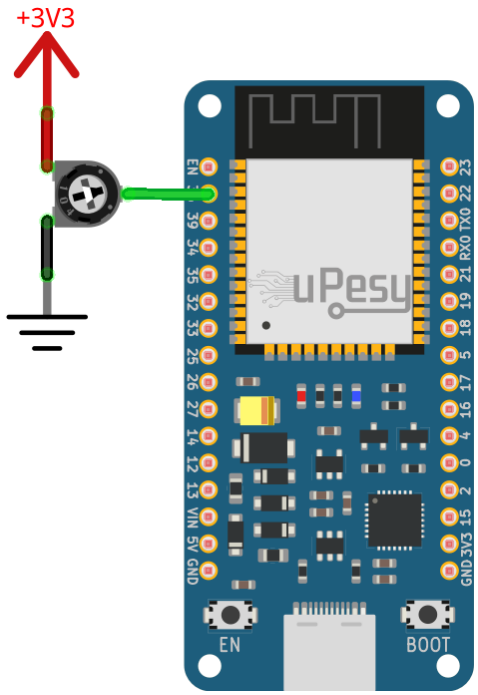
- To change the ADC resolution:

```
analogReadResolution(resolution) // Resolution between 9-12 bits
```

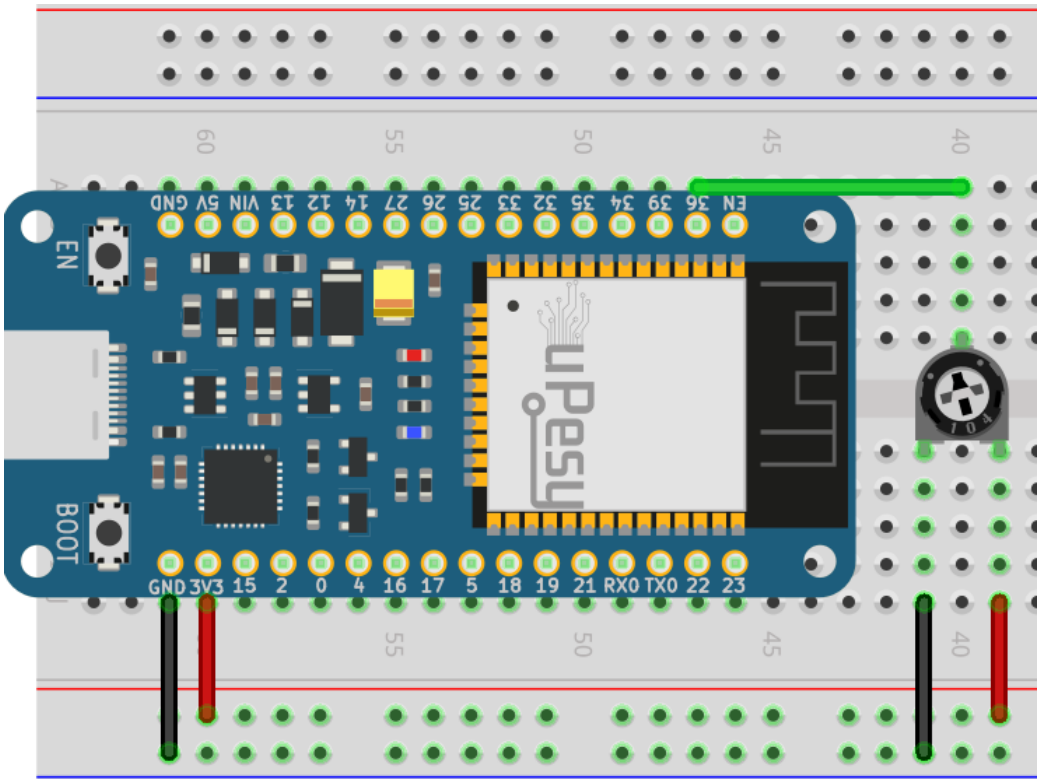
Mini-Project

We will test the ADC using a potentiometer (variable resistor).

Electrical schematic for the potentiometer



Electrical Circuit



Electrical circuit of the potentiometer on a breadboard

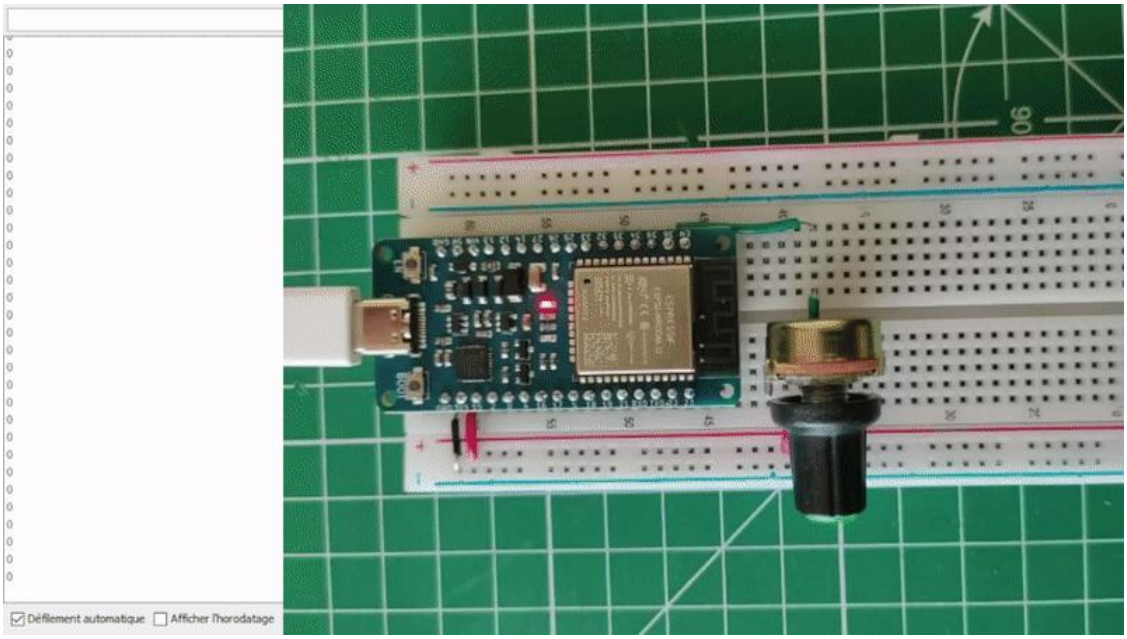
Code for reading potentiometer values

Solution

```
// The potentiometer is connected to GPIO36 (Pin VP)
const int potPin = 36;

// Potentiometer value
int potValue = 0;

void setup() {
Serial.begin(115200);
delay(1000);
pinMode(potPin,INPUT_PULLUP);
}

void loop() {
// Measures the value of the potentiometer
potValue = analogRead(potPin);
Serial.println(potValue);
delay(250);
}
```

Final results

When you turn the potentiometer, you get :



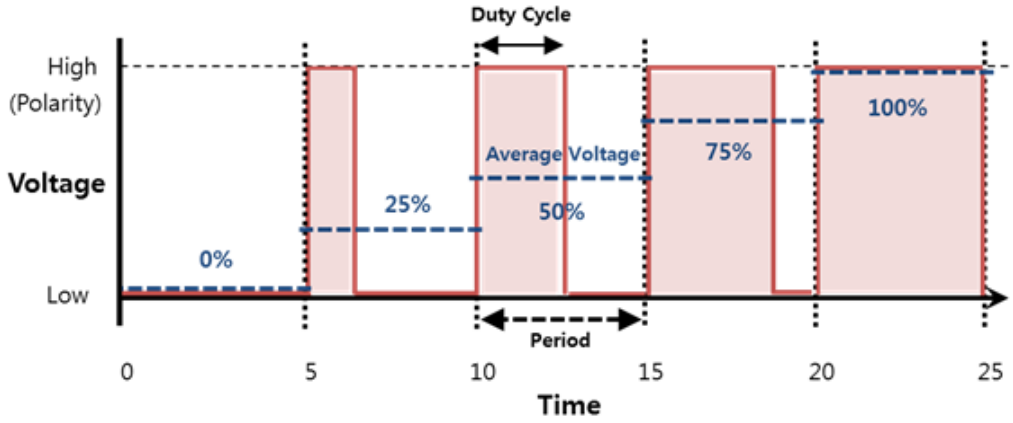
Create analog voltage on ESP32 with PWM

(Updated at 12/23/2022)

The PWM (Pulse With Modulation) artificially creates a variable voltage **between 0 and 3.3V.** The PWM on the ESP32 is much more complete than on the Arduino.

A reminder of the PWM working principle

PWM is a method for obtaining analog-looking signals on digital pins. We create a square wave, a signal switching between a HIGH and LOW level, between 0V and 3.3V. This succession of HIGH / LOW levels can simulate an intermediate voltage between the two levels by playing on the balance when the signal is HIGH and LOW. The duration of the HIGH level is called the “duty cycle.” You must change or modify this pulse width to obtain an analog variation.



Principle of PWM

Note

The signal frequency is fixed, and we can change only the duty cycle.

PWM usage on ESP32 with Arduino code

Using PWM on the ESP32 is different from using the Arduino. Sixteen independent PWM channels can be assigned to GPIO pins (except GPIO36, GPIO39, GPIO34, and GPIO35 pins). The PWM configuration is a bit more complex on the ESP32 but is more powerful. The `ledc` module takes care of the PWM, and three primary functions will allow it to be used: `ledcSetup()`, `ledcAttachPin()`, `ledcWrite()`

To generate a PWM signal, for example, on pin GPIO23, you must:

- Choose a PWM channel (0 - 15)
- Choose the PWM frequency
- Choose the resolution of the pulse width between 1 and 16 bits
- Choose the GPIO pin which will generate the PWM signal
- Assign the value of the voltage you want at the output

```
int pwmChannel = 0; // Selects channel 0
int frequence = 1000; // PWM frequency of 1 KHz
int resolution = 8; // 8-bit resolution, 256 possible values
int pwmPin = 23;

void setup(){
    // Configuration of channel 0 with the chosen frequency and resolution
    ledcSetup(pwmChannel, frequence, resolution);

    // Assigns the PWM channel to pin 23
    ledcAttachPin(pwmPin, pwmChannel);

    // Create the selected output voltage
    ledcWrite(pwmChannel, 127); // 1.65 V
}

void loop(){
}
```

Once the configuration with the `ledcSetup()` and `ledcAttachPin()` functions is done, we only use the `ledcWrite()` function to set the voltage.

Warning

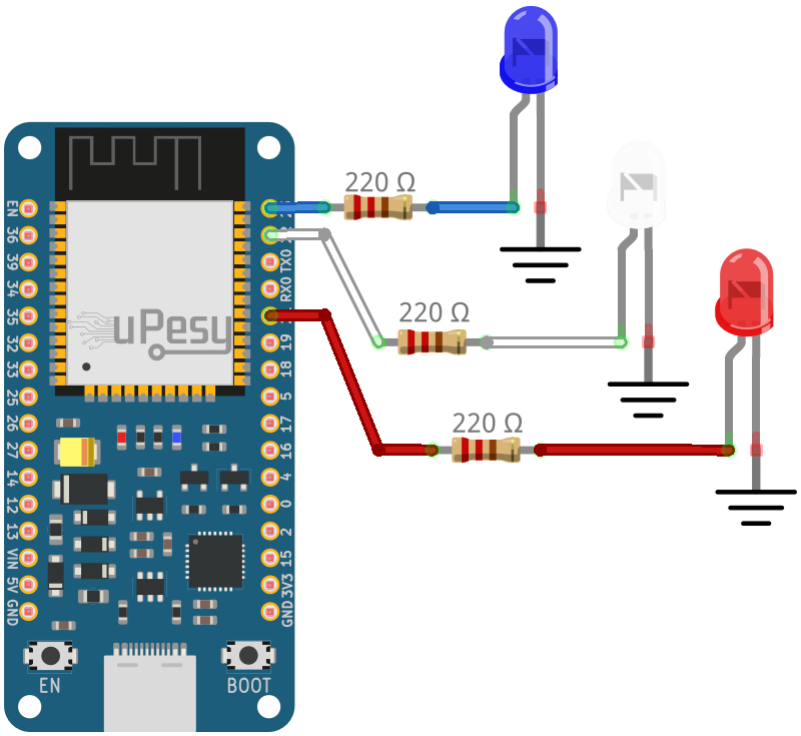
The `analogWrite()` function will not work on the ESP32. You should use `ledcWrite()` instead.

Mini-Project: Dimming LEDs via PWM

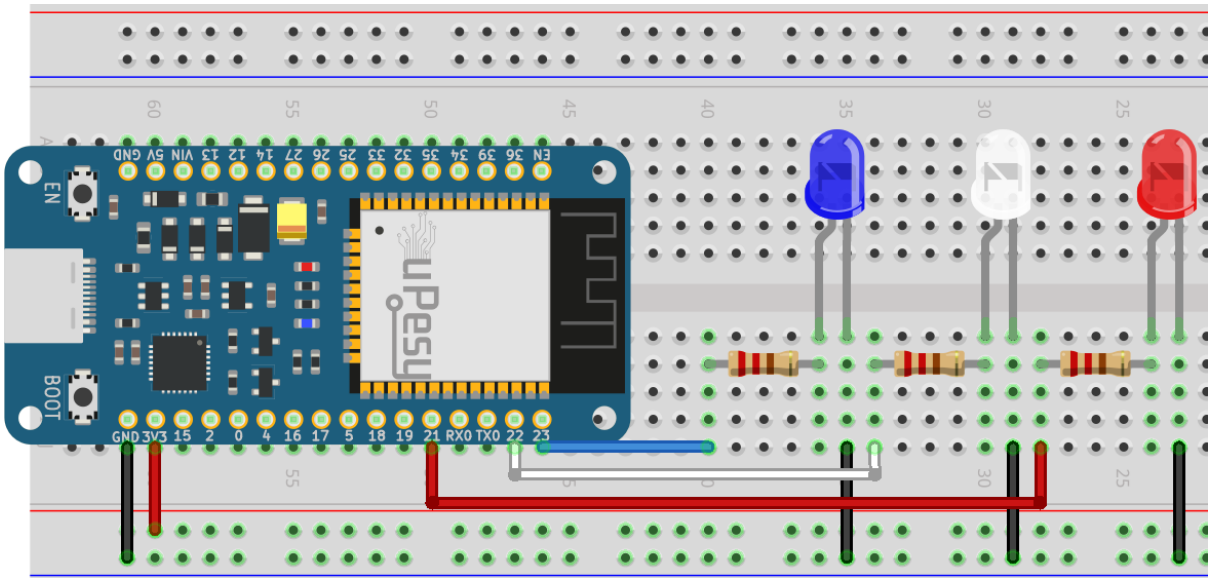
We will dim three LEDs with a PWM signal.

Electrical schematic

Choose the colors of LEDs that you want.



Electrical circuit



Electrical circuit (breadboard version)

Warning

Don’t forget to put resistors in series with the LEDs to avoid burning them out. A value between 220Ω and 330Ω will do the job nicely.

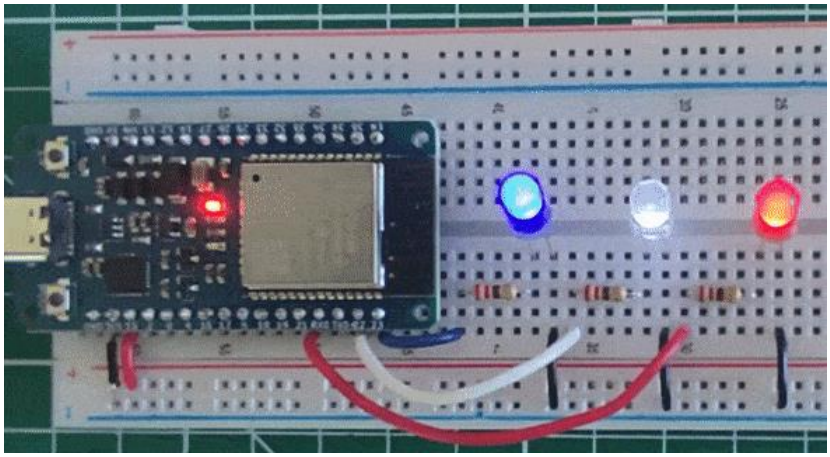
Code to control the brightness of the LEDs

Solution

```
const int ledPin = 23;
const int ledPin2 = 22;
const int ledPin3 = 21;

// PWM channel 0 parameter
const int freq = 5000; // 5000 Hz
const int ledChannel = 0;
const int resolution = 8; // 8-bit resolution

void setup(){
    // Configure the channel 0
    ledcSetup(ledChannel, freq, resolution);

    // Attach the channel 0 on the 3 pins
    ledcAttachPin(ledPin, ledChannel);
    ledcAttachPin(ledPin2, ledChannel);
    ledcAttachPin(ledPin3, ledChannel);
}

void loop(){
    // Increase the brightness of the led in the loop
    for(int dutyCycle = 0; dutyCycle <= 255; dutyCycle++){
        ledcWrite(ledChannel, dutyCycle);
        delay(15);
    }
}
```

Instead of using three different PWM channels for each pin, only one is used and assigned to the three pins.

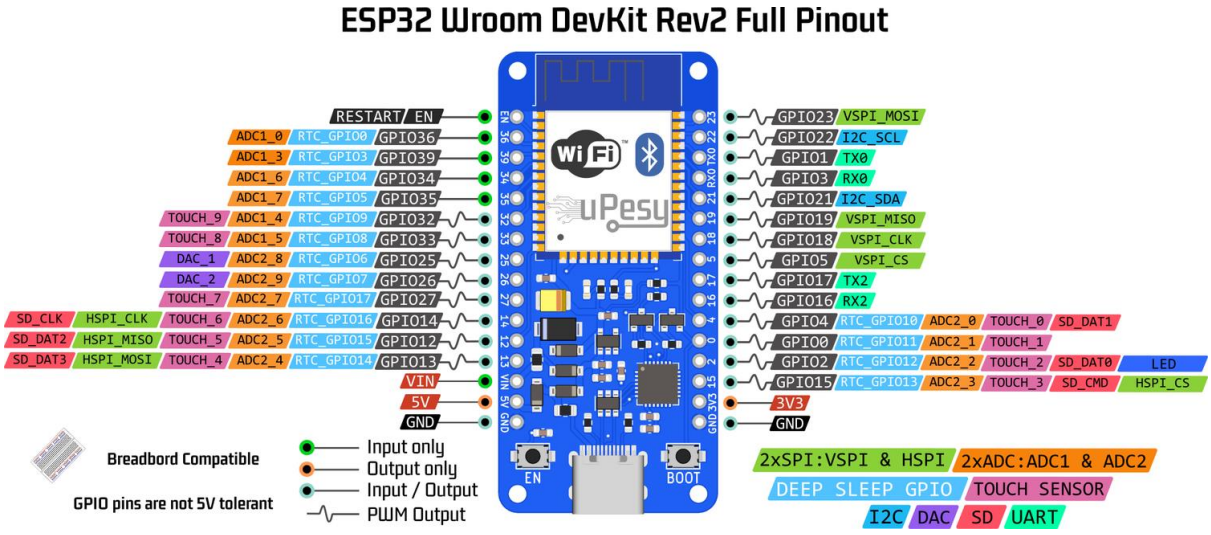
Results



The capacitive sensors of the ESP32

(Updated at 11/28/2022)

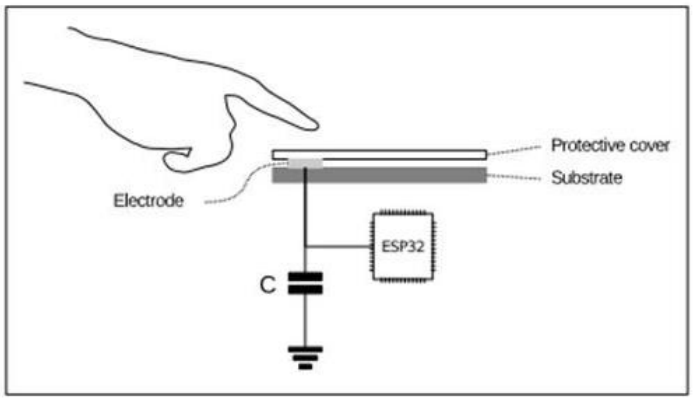
The ESP32 has capacitive sensors that can be used as a touch button. These are the famous **“TOUCH”** pins found on the pinouts. Ten are available on the uPesy ESP32 boards.



Pinout of the uPesy ESP32 Wroom Devkit board

What is it?

Capacitive sensors are widely used to detect the pressure of our fingers, especially on touch screens. We can use them on the ESP32 to replace push buttons.



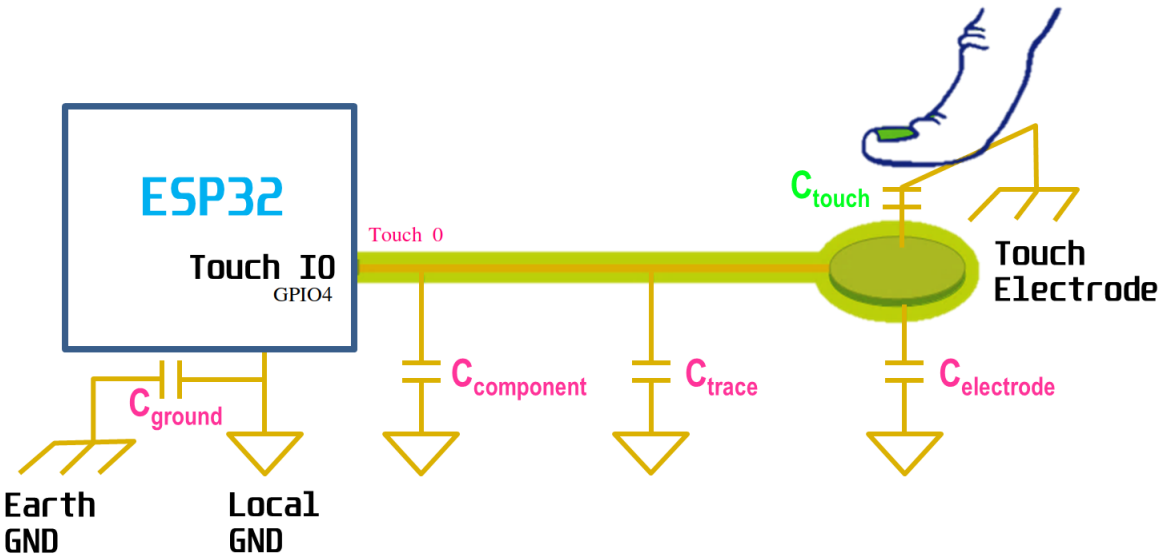
Capacitive touch sensors

Capacitive sensors are based on the variation of the capacitance (of a capacitor) when the sensor is touched. The ADC (Analog to Digital Converter) reads and converts this variation.

Note

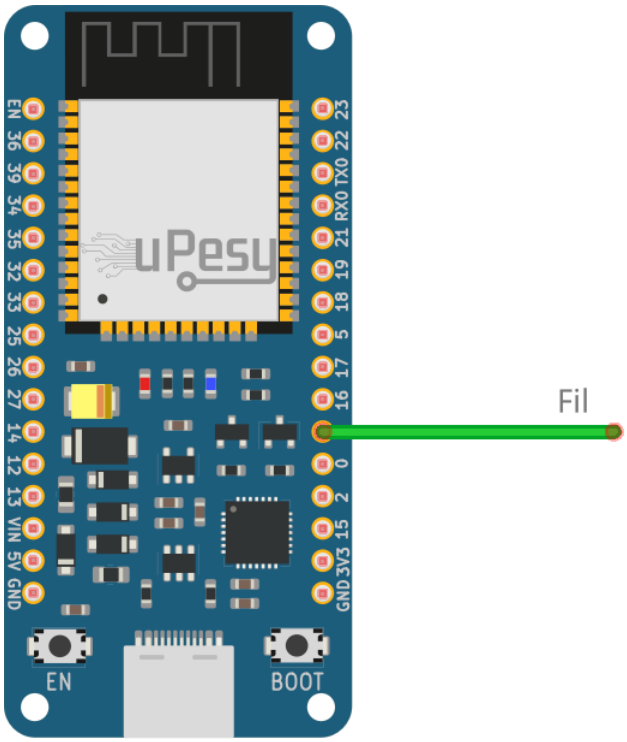
It is essential to remember that these capacitive sensors will not be as reliable as push buttons, especially for the use we will make of them.

Indeed, many parasitic capacitances come into play and more or less disturb the measured values.



Parasitic capacitors

It is necessary to consider all these parameters and make a much more rigorous and complex electrical circuit to have good measures. We can resume the circuit **to a copper wire to show how it works!**



It’s hard to make a simpler circuit 🙂

Use on the ESP32

The code to use the capacitive sensors is straightforward. Only one function is needed `touchRead()` . The code to read the capacitive measurement of pin 4 is :

```
touchRead(4);
//or
touchRead(T0);
```

Note

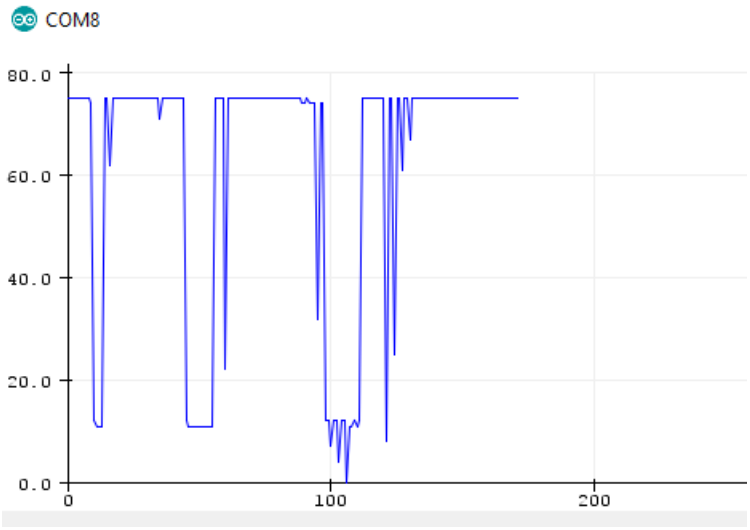
The function parameter is either the pin number (here GPIO4) or the number of the capacitive sensor associated with the pin (here T0).

The usage is very similar to the `analogRead()` .

With a short code, the measured values are displayed in the serial monitor:

```
void setup() {
    Serial.begin(115200);
    delay(1000); // Delay to launch the serial monitor
    Serial.println("ESP32 Touch Demo");
```

```
}

void loop() {
    Serial.println(touchRead(4));
    delay(500);
}
```



Waveform displayed in the serial plotter

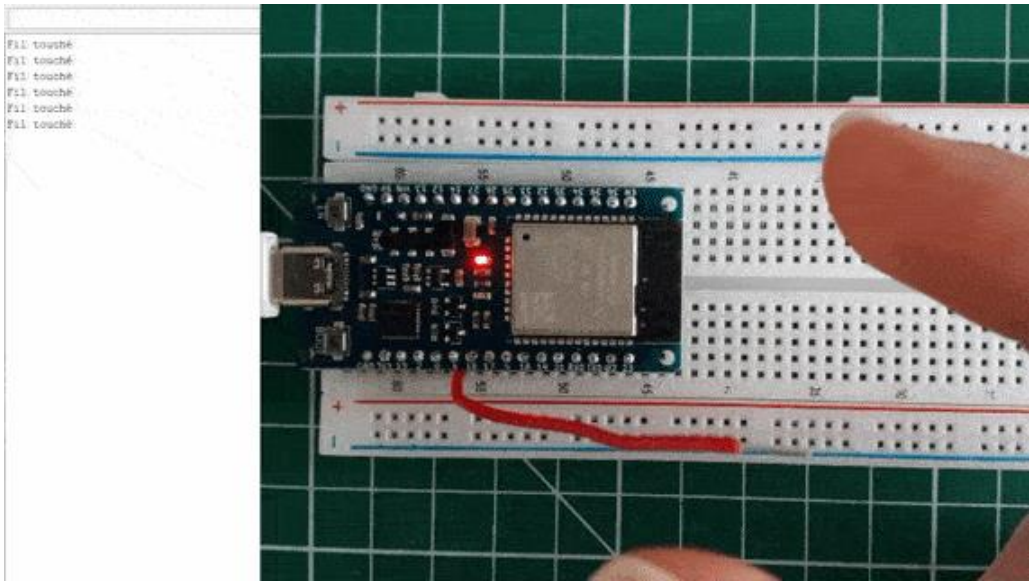
The hollows correspond to the moment when the wire is touched. A threshold value must be defined to determine if the cable has been touched. If we are below this threshold value, we have touched the button.

Note

The threshold value depends on the material used (wire, length, breadboard) and should be adjusted.

The code **with the threshold** is :

```
int capacitiveValue = 100;
int threshold = 20; //Threshold to adjust

void setup() {
    Serial.begin(115200);
    delay(1000); // Delay to launch the serial monitor
    Serial.println("ESP32 Touch Demo");
}

void loop() {
    capacitiveValue = touchRead(4);
    if(capacitiveValue < threshold ){
        Serial.println("Wire touched");
        }
    delay(500);
}
```



Create ESP32 GPIO Interrupts to reduce CPU usage

(Updated at 11/28/2022)

An interrupt is a function triggered asynchronously by an external event that momentarily interrupts the current code's execution to execute more critical code.

What's the point?

Imagine that you want to turn on an LED when you press a button connected to a GPIO pin of the ESP32. The simplest way is to look permanently in the function `loop()` if you have pressed the :

```
const int buttonPin = 33;
const int ledPin =  2;

// Push button status
int buttonState = 0;

void setup() {
    Serial.begin(115200);

    //Configuration of the input pin in pullup
    pinMode(buttonPin, INPUT_PULLUP);
    pinMode(ledPin, OUTPUT);
}
```

```
void loop() {
    buttonState = digitalRead(buttonPin);

    if (buttonState == LOW) {
        digitalWrite(ledPin, HIGH);
    } else if(buttonState == HIGH){
        digitalWrite(ledPin, LOW);
    }
}
```

The problem is that this task keeps the microcontroller's processor busy. So we can tell the microcontroller to do other tasks in the `loop()` , but in this case, the microcontroller will only look at the state of the button once at each iteration of the `loop()` . We may miss an event. We can't process real-time external events. Interrupts allow detecting an event in real-time while letting the microcontroller processor do other tasks. Thus the operation of an interrupt is as follows :

Detection of an event → Interruption of the main program → Execution of the interrupt code → The processor picks up where it left off.

Note

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

Detection modes

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

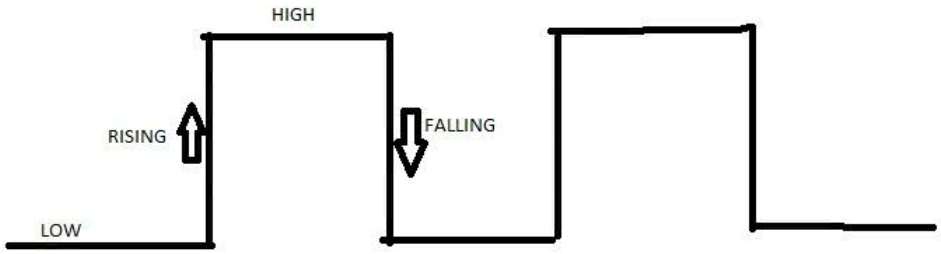


Figure: Digital Signal

Different modes of detection

You can choose the interruption detection mode:

- `LOW` triggers the interrupt as soon as the signal is at 0V
- `HIGH` triggers the interrupt as soon as the signal is 3.3V
- `RISING` triggers the interrupt as soon as the signal changes from `LOW` to `HIGH` (0 to 3.3V)
- `FALLING` triggers the interrupt as soon as the signal changes from `HIGH` to `LOW` (3.3V to 0)
- `CHANGE` triggers the interrupt as soon as the signal changes from `LOW` to `HIGH` or from `HIGH` to `LOW` .

Note

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

Use on the ESP32

The use of interrupts on the ESP32 is similar to that on the Arduino with the `attachInterrupt()` . We can use **any GPIO pin for interrupts.**

Thus to **create an interrupt on a pin,** you must :

- Assign a pin to detect the interrupt `attachInterrupt()`

```
attachInterrupt(GPIOPin, function_ISR, Mode);
```

With `Mode` , the detection mode can be `LOW` , `HIGH` , `RISING` , `FALLING` or `CHANGE`

- Create the function that will be executed when the interrupt is triggered

```
void IRAM_ATTR function_ISR() {
  // Content of the function
}
```

Note

It is recommended to add the flag `IRAM_ATTR` so that the function code is stored in RAM (and not in Flash) so that the function runs faster.

The entire code will be of the form :

```
void IRAM_ATTR function_ISR() {
  // Function code
}
```

```
void setup() {
  Serial.begin(115200);
  pinMode(23, INPUT_PULLUP);
  attachInterrupt(23, function_ISR, FALLING);
}

void loop() {
}
```

As soon as the voltage goes from 3.3V to 0V, the function `fonction_ISR()` will be executed. Then we can do other tasks in the `loop()` .

You must remember that an interrupt's function must be executed as quickly as possible so as not to disturb the main program. The code must be as concise as possible, and it is not recommended to exchange SPI, I2C, or UART data from an interrupt.

Warning

You can't use the `delay()` nor `Serial.println()` with an interrupt. However, you can display messages in the serial monitor by replacing `Serial.println()` with `ets_printf()` , which is compatible with interrupts.

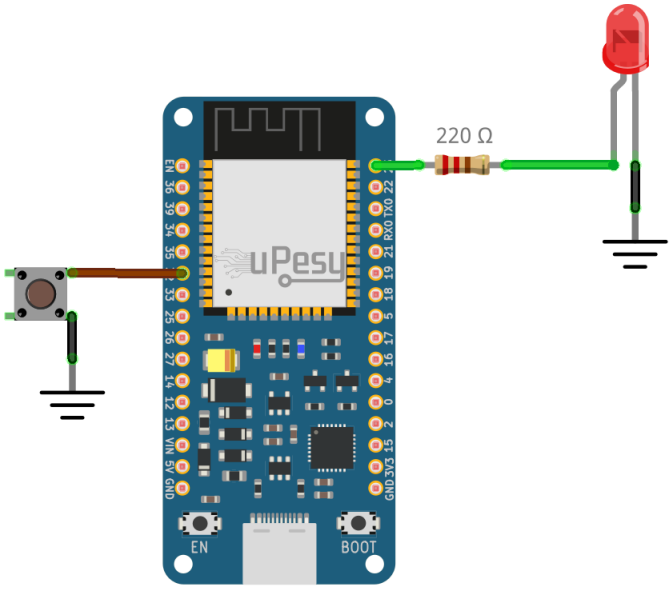
The code below displays *"Button pressed"* when a button connected to pin 33 is pressed.

```
void IRAM_ATTR fonction_ISR() {
    ets_printf("Boutton pressé\n");
    // Code de la fonction
}

void setup() {
    Serial.begin(115200);
    pinMode(33, INPUT_PULLUP);
    attachInterrupt(33, fonction_ISR, FALLING);
}

void loop() {
}
```

Mini-Project

We will redo the first mini-project, which consisted in making a LED blink when a button is pressed . We will use interrupts to manage the event and free the processor to do other tasks.

Electrical schematic



Schematic

Code

Solution

```
const int buttonPin = 32;
const int ledPin = 23;
int buttonState = 0;
int lastMillis = 0;

void IRAM_ATTR function_ISR() {
 if(millis() - lastMillis &gt; 10){ // Software debouncing buton
         ets_printf("ISR triggered\n");
         buttonState = !buttonState;
         digitalWrite(ledPin,buttonState);
 }
 lastMillis = millis();
}

void setup() {
 Serial.begin(115200);
 pinMode(buttonPin, INPUT_PULLUP);
 pinMode(ledPin,OUTPUT);
 attachInterrupt(buttonPin, function_ISR, CHANGE);
 digitalWrite(ledPin, buttonState);
}

void loop() {
 // Code ...
}
```

Using ESP32 timers in Arduino code

(Updated at 02/02/2023)

In this article, we will explore the operation of a timer on the ESP32 using Arduino code. We'll look at how to set up and use a timer effectively, using practical examples to help you understand the basic concepts. We will discover the steps to configure a timer on the ESP32 with the critical parameters for optimal operation. Here we go. 😊

How do ESP32 timers work?

This article will not cover the timer's theoretical workings to keep it manageable. If you are a beginner and need to learn how a timer works, read the theoretical article on how it works . It will allow you to understand better how to choose the parameters' values to use in your Arduino code.

Configuring and using an ESP32 timer with Arduino code

Here is the minimal skeleton code to use a timer on the ESP32 with Arduino code. It allows you to trigger an interrupt when the timer reaches a `threshold` value, commonly called `autoreload` .

```
hw_timer_t * timer = NULL;

void IRAM_ATTR timer_isr() {
    // This code will be executed every 1000 ticks, 1ms
}

void setup() {
  Serial.begin(115200);
  uint8_t timer_id = 0;
  uint16_t prescaler = 80; // Between 0 and 65 535
  int threshold = 1000000; // 64 bits value (limited to int size of 32bits)

  timer = timerBegin(timer_id, prescaler, true);
  timerAttachInterrupt(timer, &timer_isr, true);
  timerAlarmWrite(timer, threshold, true);
  timerAlarmEnable(timer);
}

void loop() {
}
```

Timer selection and basic configuration

We define an object `hw_timer_t` object outside the `setup()` function to be able to access it from different functions. The function `timerBegin(uint8_t id, uint16_t prescaler, bool countUp)` allows to configure the timer :

The ESP32 has 4 independent timers, selected by an id between 0 and 3. Then we select the `prescaler` to apply to the timer clock signal. On the ESP32, this is the `APB_CLK` clock, clocked at 80 MHz.

Warning

If you use external libraries in your code, they may use timers. In this case, you must be careful not to choose one they already use; otherwise, your program will undoubtedly have bugs!

In most examples available, a `prescaler` of `80` is used to obtain a counting period of 1 µs. From a period of the timer in microseconds, we can directly know the value of the corresponding autoreload without any complicated calculations:

$$\text{autoreload} = \text{period} \times 106 \quad \text{period} = \text{autoreload} \times 10^{-6}$$

The argument `countUp` argument specifies the direction we want to count: `true` in ascending order, `false` in descending order.

Configuring the alarm and triggering an interrupt routine

We attach an interrupt to the timer, which is triggered every time the threshold value is exceeded with `timerAttachInterrupt(hw_timer_t *timer, void (*fn)(void), bool edge)`.

The central argument must be the name of the interrupt routine to be executed (here, `timer_isr()`). You can also choose whether the interrupt is to be triggered on the rising or fall in select: we generally prefer the increasing edge for timers.

The threshold value of the counter is defined by `timerAlarmWrite(hw_timer_t *timer, uint64_t alarm_value, bool autoreload)`. We activate the mode `autoreload` mode by setting `autoreload` to `true`. As soon as the timer has exceeded the value `alarm_value` value, it starts counting again from zero, and the interrupt function is triggered.

Note

With a `prescaler` of 80, the value of `alarm_value` corresponds directly to the global period of the timer in microseconds.

Once the timer is fully configured, you can enable it with its alarm using `timerAlarmEnable(timer)`.

If we run this code, the function `timer_isr()` will be executed every second. Nothing will happen because the function is empty in this “skeleton” code. I suggest a version that flashes the blue LED of the ESP32 on pin `GPIO2`.

Blink code with only one timer

Here is the famous “blink” rewritten using only a hardware timer for flashing the blue LED every second:

```
#define PIN_LED 2

hw_timer_t * timer = NULL;
volatile uint8_t led_state = 0;

void IRAM_ATTR timer_isr(){
  led_state = ! led_state;
  digitalWrite(PIN_LED, led_state);
}

void setup() {
  Serial.begin(115200);
  pinMode(PIN_LED, OUTPUT);
  uint8_t timer_id = 0;
  uint16_t prescaler = 80;
  int threashold = 1000000;
  timer = timerBegin(timer_id, prescaler, true);
  timerAttachInterrupt(timer, &timer_isr, true);
  timerAlarmWrite(timer, threashold, true);
  timerAlarmEnable(timer);
}

void loop() {
  // put your main code here to run repeatedly
}
```

We use the keyword `volatile` for the variable `led_state` to force the compiler not to optimize it. In fact, for the compiler, the function `timer_isr()` is never called in the code because it is called via an external interrupt (which the compiler does not know about).

Without the `volatile` attribute, the compiler could remove the unused variable to free up memory. Using the `volatile` attribute keyword, we force the compiler to ignore this variable for optimization.

To make the LED blink, we use a little trick here to invert the state of the LED with the operator``!`` . When used on a variable; it replaces all zeros in its binary representation with 1s (and vice versa).

So the value of led_state will be inverted at each entry in the interrupt routine: `0b0000000` → `0b11111111` → `0b0000000` … As the `digitalWrite()` function only looks at the value of the first bit, we will have an alternation between 0 and 1.

Note

We add the `IRAM_ATTR` attribute to the function declaration to tell the compiler to load all the interrupt routine code into the ESP32’s internal RAM, making it run faster.

With this program, the blu LED starts blinking without the function, which can be used to do other tasks. With this program, the blue LED starts blinking without the `loop()` function; different can be used for other tasks.

Increment a variable and generate flags

The interrupt routine must be executed as quickly as possible to avoid disturbing the main program. So we often use `flags` that change state (usually a boolean: `true` or `false`) in the `isr` . These changes are then processed in the main loop:

```
hw_timer_t * timer = NULL;
volatile boolean tick_flags = false;

void IRAM_ATTR timer_isr() {
  tick_flags = true;
}

void setup() {
  Serial.begin(115200);

  uint8_t timer_id = 0;
  uint16_t prescaler = 80;
  int
```


Advanced use of ESP32 timers in Arduino code

Manage read/write access to a shared variable using semaphores

It is good practice to frame the interrupt code of a timer in a semaphore to manage access to global variables. This helps predict how the program will behave if the interrupt routine and the main program want to simultaneously write to the same variable. Since the Arduino code for ESP32 uses freeRTOS as its real-time OS, the **semaphores** are already integrated: we need to use them directly in our program:

Save energy on the ESP32 with Deep Sleep

(Updated at 11/28/2022)

The ESP32 has different power-saving modes, which allow you to reduce power consumption by disabling some features. This is similar to putting a computer on standby to save power. Among these modes, the best known is Deep Sleep which put the ESP32 in a deep sleep state.

Power mode	Description			Power consumption
Modem-sleep	The CPU is powered on.	240 MHz *	Dual-core chip(s)	30 mA ~ 68 mA
			Single-core chip(s)	N/A
		160 MHz *	Dual-core chip(s)	27 mA ~ 44 mA
			Single-core chip(s)	27 mA ~ 34 mA
		Normal speed: 80 MHz	Dual-core chip(s)	20 mA ~ 31 mA
			Single-core chip(s)	20 mA ~ 25 mA
Light-sleep	-			0.8 mA
Deep-sleep	The ULP coprocessor is powered on.			150 μA
	ULP sensor-monitored pattern			100 μA @1% duty
	RTC timer + RTC memory			10 μA
Hibernation	RTC timer only			5 μA
Power off	CHIP_PU is set to low level, the chip is powered off.			1 μA

Consumption of the different modes of the ESP32, according to the datasheet

During this mode, the ESP32 can perform simple tasks and be woken up to operate normally.

Note

The keyword “Deep Sleep” is used a bit excessively to represent the most energy-saving mode of the ESP32, whereas it’s the “Hibernation” mode that consumes the least energy.

What is the Deep Sleep mode for?

This power-saving mode is useful when the ESP32 is battery-powered and the ESP32 “works” periodically. (For example, reading a value from a sensor and sending it through WiFi every 10 minutes). The battery will be discharged quickly if the ESP32 is 100% on 24 hours a day. With the Deep Sleep mode, the batteries will last much longer.

Deep Sleep mode puts the ESP32 in a rudimentary state. Indeed, in Deep Sleep mode, the 2 CPUs of the ESP32 do not work anymore, and it is the ULP (Ultra Low Processor) processor which takes over. This processor consumes very little power and can perform basic actions. The Flash and RAM are not powered anymore; only the RTC memory is still powered and can be used. WiFi and Bluetooth are, of course, also disabled.

Warning

The consumption of the ESP32 in this mode, in the µA range, will be slightly different depending on the tasks performed in this mode and the chosen wake-up source.

The wake-up sources of the ESP32 Deep Sleep mode

After putting the ESP32 in Deep Sleep mode, there are several ways to wake it up:

- Use an internal timer to wake up the ESP32 at a selected time (internal wake-up)
- Using capacitive touch sensors
- Use the RTC pins

Note

We can combine different wake-up sources.

It is also possible to activate the Deep Sleep mode without configuring any wake-up sources. In this case, the ESP32 will be indefinitely in Deep Sleep mode until a manual reset is done by pressing the `EN/RST` button (or by reflashing the board). So you can't block the ESP32 with the Deep Sleep mode.

For more information about the ESP32 Deep Sleep, I encourage you to consult the official documentation on this subject .

Put the ESP32 in Deep Sleep mode with Arduino code

When you want to use the Deep Sleep mode, you have to think about :

- Configure the type of wake-up source of the ESP32: The consumption of the ESP32 in the µA range will be slightly different from the chosen wake-up source.
- Optionally choose which peripherals you want to turn off or on during Deep Sleep mode, for example, which pins should remain on. By default, the ESP32 turns off all peripherals that are not needed to detect the wake-up trigger.
- Use the function `esp_deep_sleep_start()` to enter the Deep Sleep mode.

Warning

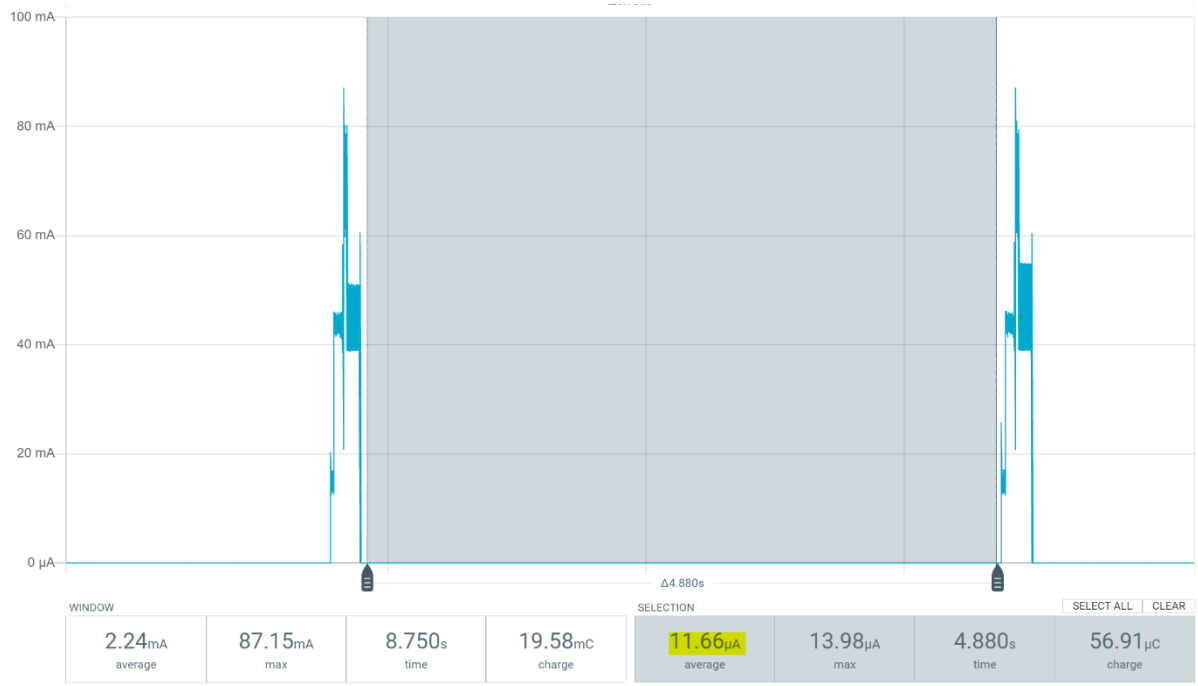
The measurements were created with the board uPesy ESP32 Low Power Devkit , a board optimized for very low power consumption when the ESP32 is in Deep Sleep. If you have a "generic" ESP32 board the power consumption will probably be much higher.

Note

The same uPesy ESP32 Low Power Devkit unit has been used to compare the consumption according to the wake-up sources. Indeed, the values vary from a few µA depending on the unit chosen.

Use a Timer as a wake-up source

This is how to wake up the ESP32 that consumes the least power. Only the `RTC Timer` is on. It is a counter that triggers an alarm (like an alarm clock) whose period is configurable.



Consumption of 11.5µA in timer mode powered by 4.2V

Here is an example code that wakes up the ESP32 every 5 seconds with a timer as the wake up source:

```
#define uS_TO_S_FACTOR 1000000
#define TIME_TO_SLEEP  5

RTC_DATA_ATTR int bootCount = 0;

void print_wakeup_reason(){
   esp_sleep_wakeup_cause_t wake_up_source;

   wake_up_source = esp_sleep_get_wakeup_cause();

   switch(wake_up_source){
      case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wake-up from external
signal with RTC_IO"); break;
      case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wake-up from external
signal with RTC_CNTL"); break;
      case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wake up caused by a
timer"); break;
      case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wake up caused by a
touchpad"); break;
      default : Serial.printf("Wake up not caused by Deep Sleep:
%d\n",wake_up_source); break;
   }
}

void setup(){
   Serial.begin(115200);

   ++bootCount;
   Serial.println("----------------------");
   Serial.println(String(bootCount)+ "th Boot ");
```

```
   // Displays the reason for the wake up
   print_wakeup_reason();

   //Timer Configuration
   esp_sleep_enable_timer_wakeup(TIME_TO_SLEEP * uS_TO_S_FACTOR);
   Serial.println("ESP32 wake-up in " + String(TIME_TO_SLEEP) + "
seconds");

   // Go in Deep Sleep mode
   Serial.println("Goes into Deep Sleep mode");
   Serial.println("----------------------");
   delay(100);
   esp_deep_sleep_start();
   Serial.println("This will never be displayed");
}

void loop(){
}
```

The function `print_wakeup_reason()` displays the ESP32 wake-up source. During the first ESP32 boot, the wake-up was not caused by the Deep Sleep but by the *"Hard resetting via RTS pin…"* on the Arduino IDE. Then during the next boots, the ESP32 is woken up from deep sleep thanks to a timer every 5 seconds.

You can change the Deep Sleep duration by modifying the timer with the function `esp_sleep_enable_timer_wakeup()` .

```
ets Jun 8 2016 00:22:57

rst: 0x10 (RTCWDT_RTC_RESET), boot: 0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP: 0xee
clk_drv: 0x00, q_drv: 0x00, d_drv: 0x00, cs0_drv: 0x00, hd_drv: 0x00,
wp_drv: 0x00
mode: DIO, clock div: 1
load: 0x3fff0018, len: 4
load: 0x3fff001c, len: 1216
ho 0 tail 12 room 4
load: 0x40078000, len: 9720
ho 0 tail 12 room 4
load: 0x40080400, len: 6352
entry 0x400806b8
----------------------
1th Boot
Wake up not caused by Deep Sleep: 0
ESP32 wake-up in in 5 seconds
Goes into Deep Sleep mode
----------------------
ets Jun 8 2016 00:22:57

rst: 0x5 (DEEPSLEEP_RESET), boot: 0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP: 0xee
clk_drv: 0x00, q_drv: 0x00, d_drv: 0x00, cs0_drv: 0x00, hd_drv: 0x00,
wp_drv: 0x00
mode: DIO, clock div: 1
load: 0x3fff0018, len: 4
load: 0x3fff001c, len: 1216
ho 0 tail 12 room 4
```

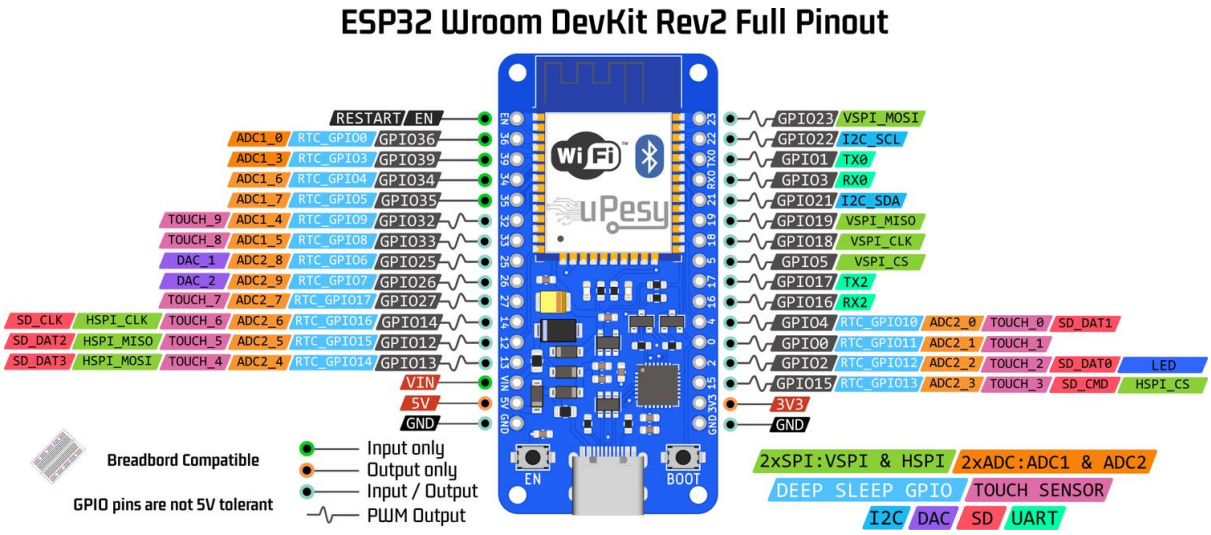
```
load: 0x40078000, len: 9720
ho 0 tail 12 room 4
load: 0x40080400, len: 6352
entry 0x400806b8
----------------------
2th Boot
Wake up caused by a timer
ESP32 wake-up in in 5 seconds
Goes into Deep Sleep mode
----------------------
```

Use a single GPIO pin as a wake-up source

We can also use a GPIO pin with a push button to wake up the Deep Sleep ESP32. This is typically what devices that wake up instantly when a button is pressed (phones, for example) do.

Power consumption of 13.5µA in deep-sleep mode powered by 4.2V

The peak corresponds to the consumption of the ESP32 when it wakes up. The consumption is slightly higher than with a timer.



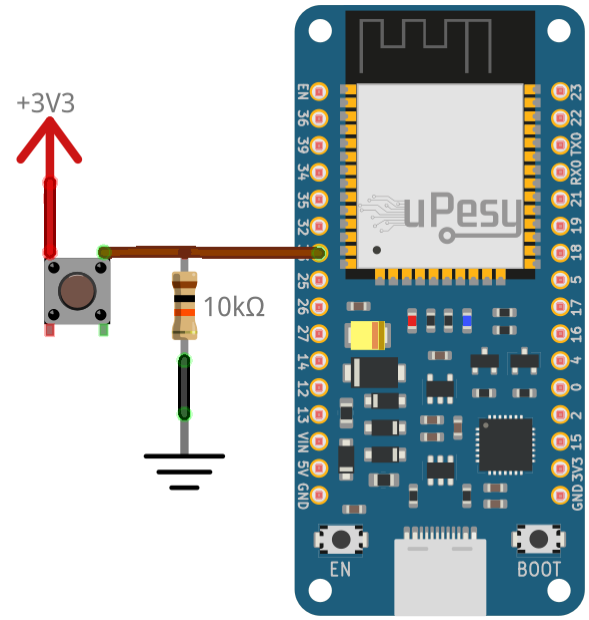
Pinout of the uPesy ESP32 Wroom Devkit board

Warning

Only the RTC_GPIO pins can be used (light blue labels on the pinout below)

In this example, a push button with an external pulldown resistor connected to pin 33 is used:

- Schematic



Wake up ESP32 with a button

- Breadboard

```
RTC_DATA_ATTR int bootCount = 0;

void setup(){
   Serial.begin(115200);

    ++bootCount;
```

```
    Serial.println("----------------------");
    Serial.println(String(bootCount)+ "th Boot ");

    //Displays the wake-up source
    print_wakeup_reason();

    //Configure GPIO33 as a wake-up source when the voltage is 3.3V
    esp_sleep_enable_ext0_wakeup(GPIO_NUM_33,HIGH);

    //Rentre en mode Deep Sleep
    Serial.println("Goes into Deep Sleep mode");
    Serial.println("----------------------");
    esp_deep_sleep_start();
}

void loop(){}

void print_wakeup_reason(){
   esp_sleep_wakeup_cause_t wake_up_source;

   wake_up_source = esp_sleep_get_wakeup_cause();

   switch(wake_up_source){
      case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wake-up from external
signal with RTC_IO"); break;
      case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wake-up from external
signal with RTC_CNTL"); break;
      case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wake up caused by a
timer"); break;
      case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wake up caused by a
touchpad"); break;
      default : Serial.printf("Wake up not caused by Deep Sleep:
%d\n",wake_up_source); break;
   }
}
```

An `EXT0` interrupt is generated when the button is pressed and wakes up the ESP32

In the serial monitor, we obtain :

```
ets Jun 8 2016 00:22:57

rst: 0x10 (RTCWDT_RTC_RESET), boot: 0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP: 0xee
clk_drv: 0x00, q_drv: 0x00, d_drv: 0x00, cs0_drv: 0x00, hd_drv: 0x00,
wp_drv: 0x00
mode: DIO, clock div: 1
load: 0x3fff0018, len: 4
load: 0x3fff001c, len: 1216
ho 0 tail 12 room 4
load: 0x40078000, len: 9720
ho 0 tail 12 room 4
load: 0x40080400, len: 6352
entry 0x400806b8
----------------------
1th Boot
Wake up not caused by Deep Sleep: 0
Goes into Deep Sleep mode
```

```
----------------------
ets Jun 8 2016 00:22:57

rst: 0x5 (DEEPSLEEP_RESET), boot: 0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP: 0xee
clk_drv: 0x00, q_drv: 0x00, d_drv: 0x00, cs0_drv: 0x00, hd_drv: 0x00,
wp_drv: 0x00
mode: DIO, clock div: 1
load: 0x3fff0018, len: 4
load: 0x3fff001c, len: 1216
ho 0 tail 12 room 4
load: 0x40078000, len: 9720
ho 0 tail 12 room 4
load: 0x40080400, len: 6352
entry 0x400806b8
----------------------
2th Boot
Wake-up from external signal with RTC_IO
Goes into Deep Sleep mode
-----------------
```

Use several GPIO pins to wake up the ESP32

It is also possible to use several GPIO pins to trigger the wake-up of the ESP32 when certain conditions are met:

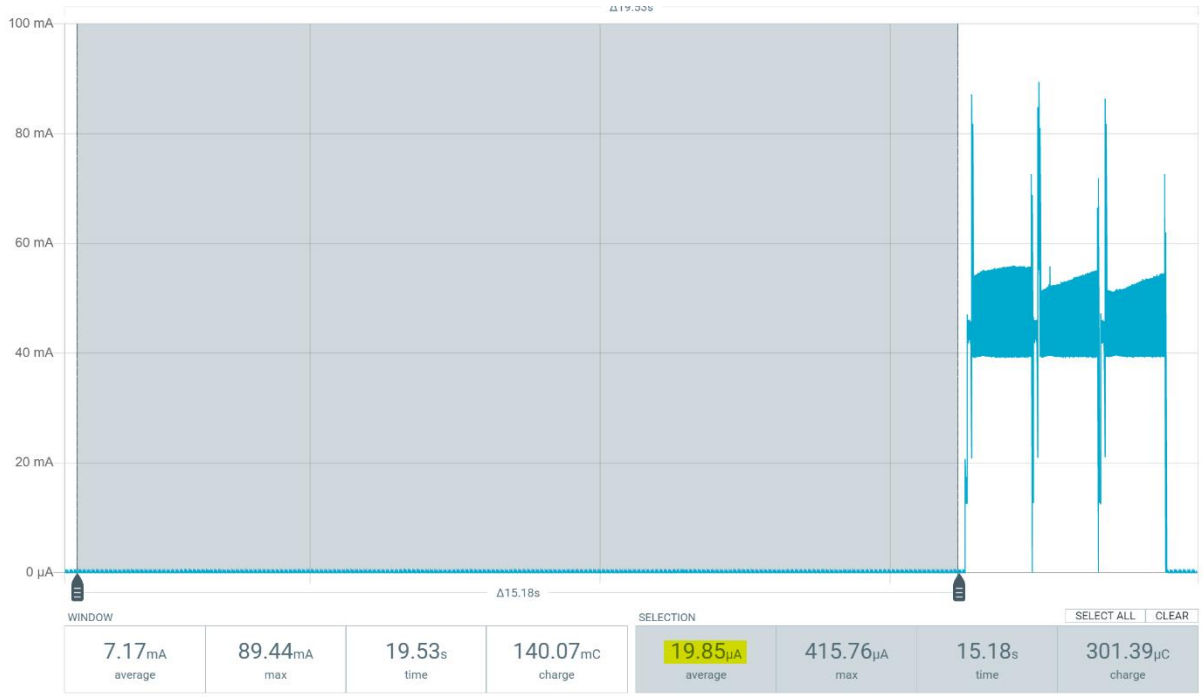
- `ESP_EXT1_WAKEUP_ANY_HIGH` : ESP32 wakes up when all selected GPIO pins are low
- `ESP_EXT1_WAKEUP_ALL_LOW` : ESP32 wakes up when one of the selected GPIO pins is high

The power consumption is the same as with a single GPIO pin.

The configuration of this mode is a little more complicated because it is necessary to create a bitmask. We create a mask where the position of the bits of the mask corresponds to the numbers of the pins. Let's suppose that we choose pins 27, 33 and 36, then we create a binary mask where the positions `27` , `33` and `36` will be `1` and the others `0` :

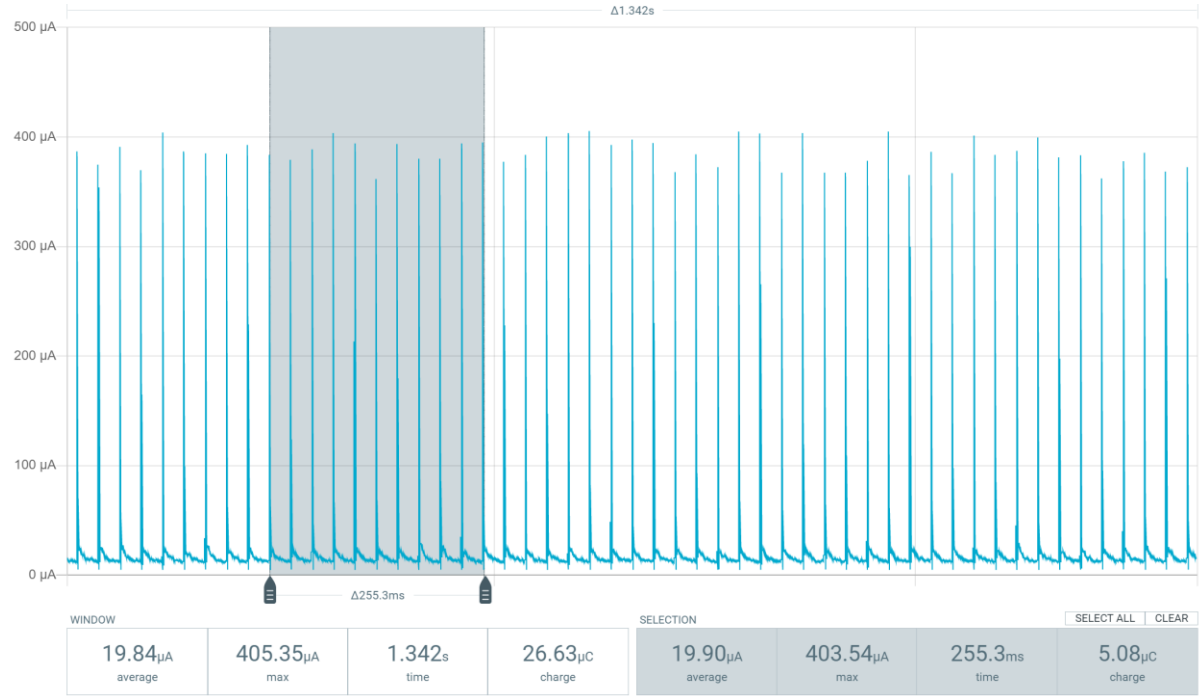
Use a touchpad as a wake-up source

It is the most consuming wake-up source



Consumption of 20µA in deep-sleep mode powered by 4.2V

The ULP processor turns on 40 times per second to measure the capacitance of the capacitive sensor: this generates current peaks of about 400µA.



Zoom on the current peaks when measuring the touchpad with the ESP32 in Deep Sleep

The schematic consists of a single wire connected to the pin `GPIO4` capacitive sensors, identical to the one proposed in [the article that explains the operation of the capacitive sensors of the ESP32] . Here is the code used to wake up the ESP32 with the touchpads :

```
#define seuil 30 //Detection threshold for the capacitive sensor

RTC_DATA_ATTR int bootCount = 0;
touch_pad_t touchPin;

void fonction_isr(){
}

void setup(){
   Serial.begin(115200);

   ++bootCount;
   Serial.println("----------------------");
   Serial.println(String(bootCount)+ "th Boot ");

   //Displays the source of the wake-up call and the touchpad number
   print_wakeup_reason();
   print_wakeup_touchpad();

   //Configuration of an interrupt for the T0 touchpad (GPIO4)
   touchAttachInterrupt(T0, fonction_isr, seuil);

   //Activate wake up by touchpads
   esp_sleep_enable_touchpad_wakeup();

   //Goes into Deep Sleep mode
   Serial.println("Goes into Deep Sleep mode");
   Serial.println("----------------------");
   esp_deep_sleep_start();
}

void loop(){
}

void print_wakeup_reason(){
   esp_sleep_wakeup_cause_t wake_up_source;

   wake_up_source = esp_sleep_get_wakeup_cause();

   switch(wake_up_source){
      case ESP_SLEEP_WAKEUP_EXT0 : Serial.println("Wake-up from external
signal with RTC_IO"); break;
      case ESP_SLEEP_WAKEUP_EXT1 : Serial.println("Wake-up from external
signal with RTC_CNTL"); break;
      case ESP_SLEEP_WAKEUP_TIMER : Serial.println("Wake up caused by a
timer"); break;
      case ESP_SLEEP_WAKEUP_TOUCHPAD : Serial.println("Wake up caused by a
touchpad"); break;
      default : Serial.printf("Wake up not caused by Deep Sleep:
%d\n",wake_up_source); break;
   }
}

void print_wakeup_touchpad(){
   touch_pad_t pin;
   touchPin = esp_sleep_get_touchpad_wakeup_status();
   switch(touchPin){
      case 0  : Serial.println("Wire touched at GPIO 4"); break;
      case 1  : Serial.println("Wire touched at GPIO 0"); break;
      case 2  : Serial.println("Wire touched at GPIO 2"); break;
      case 3  : Serial.println("Wire touched at GPIO 15"); break;
```

```
      case 4  : Serial.println("Wire touched at GPIO 13"); break;
      case 5  : Serial.println("Wire touched at GPIO 12"); break;
      case 6  : Serial.println("Wire touched at GPIO 14"); break;
      case 7  : Serial.println("Wire touched at GPIO 27"); break;
      case 8  : Serial.println("Wire touched at GPIO 33"); break;
      case 9  : Serial.println("Wire touched at GPIO 32"); break;
      default : Serial.println("Wake up not caused the touchpads"); break;
   }
}
```

Erasing RAM: how to keep data in Deep Sleep

When the Deep Sleep mode is activated, the CPU and the RAM are not powered anymore. This means that when the ESP32 wakes up from Deep Sleep, all the variables contained in the RAM are erased. Fortunately, an 8 KB RTC RAM remains on during Deep Sleep. To store variables in this memory, you must add the attribute `RTC_DATA_ATTR` .

For example, the variable that stores the number of ESP32 restarts in the example is stored in this memory.

```
RTC_DATA_ATTR int bootCount = 0;
```

Optimize consumption in Deep Sleep as much as possible

Since we are talking about saving current to the nearest µA, every detail counts. Even if it is impossible in practice to have the consumption indicated in the datasheet, because the measurements indicated are ideal and especially because the ESP32 is not the only component on the board that consumes energy, you can still save a few µA depending on your project.

Moreover, at this stage, it is more relevant to use the ESP-IDF framework directly in C (no Arduino or MicroPython code) to be able to modify the configuration of the ESP32 via the menuconfig file, which generates the `sdkconfig.h`

How to connect to a WiFi network with the ESP32

(Updated at 01/20/2023)

The built-in `WiFi.h` library will allow us to use the WiFi features of the ESP32 board easily.

The ESP32 has 2 WiFi modes:

- **STATION (** `WIFI_STA` **)** : The Station mode (STA) is used to connect the ESP32 module to a WiFi access point. The ESP32 behaves like a computer that is connected to our router. If the router is connected to the Internet, then the ESP32 can access the Internet. The ESP32 can behave as a client: make requests to other devices connected to the network, or as a server: other devices connected to the network will send requests to the ESP32. In both cases, the ESP32 can access the Internet.
- **AP (Access Point)** (`WIFI_AP`): In Access Point mode, the ESP32 behaves like a WiFi network (a bit like a router): other devices can connect to it. In this mode, the ESP32 is not connected to any other network and is therefore not connected to the Internet. This mode is more computationally and energy-intensive (the ESP32 board will heat up) since the ESP32 has to simulate a full WiFi router (Soft AP). The latency and the bandwidth will be less efficient than in a classic router.

Note

The ESP32 is, by default in STATION mode.

For mode selection :

- In general, we use the STATION mode. We can access the Internet to retrieve information from an API, have a home automation server with sensors ...
- The AP mode is usually used temporarily to enter the credentials of the WiFi network (SSID and password). It can also be used to have a separate network from your home network and not be connected to the Internet.

Connecting to a Wi-Fi Access Point

The code to connect to his AP is as follows:

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void setup(){
    Serial.begin(115200);
    delay(1000);
```

```
    WiFi.mode(WIFI_STA); //Optional
    WiFi.begin(ssid, password);
    Serial.println("\nConnecting");

    while(WiFi.status() != WL_CONNECTED){
        Serial.print(".");
        delay(100);
    }

    Serial.println("\nConnected to the WiFi network");
    Serial.print("Local ESP32 IP: ");
    Serial.println(WiFi.localIP());
}

void loop(){}
```

Important

You need to change `"yourNetworkName"` to the name of your WiFi network and `"yourNetworkPassword"` to your network password.

Terminal output

```
Connecting
.............
Connected to the WiFi network
Local ESP32 IP: 192.168.43.129
```

Tip

An easy way to have a WiFi access point to test the code is by sharing a WiFi connection from your smartphone.

The code functions as follows:

- We must include the `WiFi.h` library.
- Then we enter the name of the network and its password.
- We put the ESP32 in STATION mode with the function `WiFi.mode(WIFI_STA)`
- The ESP32 tries to connect to the Wi-Fi network using the function `WiFi.begin(ssid, password)`
- The connection is not instantaneous! It is therefore necessary to regularly check the connection status: as long as the ESP32 is not connected to the network, we will remain blocked inside the `while` loop. We add a slight delay to avoid constantly checking the status.
- Once the connection has been established, the local IP address of the ESP32 on this network will be displayed.

If it is an open network (without a password), then the code becomes simpler:

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";

void setup(){
Serial.begin(115200);
delay(1000);
WiFi.begin(ssid);
Serial.println("\nConnecting");

while(WiFi.status() != WL_CONNECTED){
    Serial.print(".");
    delay(100);
}

Serial.println("\nConnected to the WiFi network");
Serial.print("Local ESP32 IP: ");
Serial.println(WiFi.localIP());
}

void loop(){}
```

Note

Storing logins and passwords in clear text in the code is a bad practice. However, doing so simplify tutorials.

Get WiFi network information

You can get information about the network once you are connected to it:

- WiFi signal strength (RSSI) with the `WiFi.RSSI()` function
- The MAC address of the WiFi network with `WiFi.BSSIDstr()` or `WiFi.macAddress()`
- The local IP address of the ESP32 assigned by the DHCP server of the WiFi network with `WiFi.localIP()`
- The local IP address of the WiFi network (gateway) with `WiFi.gatewayIP()` (usually 192.168.0.1)
- The subnet mask with `WiFi.subnetMask()` (usually 255.255.255.0)

The code below displays all this information:

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void get_network_info(){
    if(WiFi.status() == WL_CONNECTED) {
        Serial.print("[*] Network information for ");
        Serial.println(ssid);

        Serial.println("[+] BSSID : " + WiFi.BSSIDstr());
```

```
        Serial.print("[+] Gateway IP : ");
        Serial.println(WiFi.gatewayIP());
        Serial.print("[+] Subnet Mask : ");
        Serial.println(WiFi.subnetMask());
        Serial.println((String)"[+] RSSI : " + WiFi.RSSI() + " dB");
        Serial.print("[+] ESP32 IP : ");
        Serial.println(WiFi.localIP());
    }
}

void setup(){
    Serial.begin(115200);
    delay(1000);

    WiFi.begin(ssid, password);
    Serial.println("\nConnecting");

    while(WiFi.status() != WL_CONNECTED){
        Serial.print(".");
        delay(100);
    }

    Serial.println("\nConnected to the WiFi network");
    get_network_info();
}

void loop(){}
```

Terminal output

```
Connecting
..............
Connected to the WiFi network
[*] Network information for HUAWEI_**
[+] BSSID : F0:43:47:32:1F:4D
[+] Gateway IP : 192.168.43.1
[+] Subnet Mask : 255.255.255.0
[+] RSSI : -25 dB
[+] ESP32 IP : 192.168.43.129
```

Debug Wi-Fi connection issues

By monitoring the connection status

We can know the status of the WiFi connection with the function `WiFi.status()` . This function returns an integer according to the current status of the connection.

The possible statuses are :

- `WL_IDLE_STATUS` : This is the default status before trying to connect to a network.
- `WL_SCAN_COMPLETED` : The WiFi network scan is completed.

- `WL_NO_SSID_AVAIL` : The ESP32 cannot find the name of the WiFi network. Either the network is too far from the ESP32, or the name (SSID) of the network is incorrect.
- `WL_CONNECT_FAILED` : The ESP32 cannot connect to the designated WiFi network.
- `WL_CONNECTION_LOST` : The WiFi connection to the network is lost. If this error occurs repeatedly, it may be a power problem with the ESP32.
- `WL_CONNECTED` : The ESP32 is connected to the WiFi network.
- `WL_DISCONNECTED` : The ESP32 is disconnected from the WiFi network.

Code that displays the status of the WiFi connection

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

String get_wifi_status(int status){
    switch(status){
        case WL_IDLE_STATUS:
        return "WL_IDLE_STATUS";
        case WL_SCAN_COMPLETED:
        return "WL_SCAN_COMPLETED";
        case WL_NO_SSID_AVAIL:
        return "WL_NO_SSID_AVAIL";
        case WL_CONNECT_FAILED:
        return "WL_CONNECT_FAILED";
        case WL_CONNECTION_LOST:
        return "WL_CONNECTION_LOST";
        case WL_CONNECTED:
        return "WL_CONNECTED";
        case WL_DISCONNECTED:
        return "WL_DISCONNECTED";
    }
}

void setup(){
    Serial.begin(115200);
    delay(1000);
    int status = WL_IDLE_STATUS;
    Serial.println("\nConnecting");
    Serial.println(get_wifi_status(status));
    WiFi.begin(ssid, password);
    while(status != WL_CONNECTED){
        delay(500);
        status = WiFi.status();
        Serial.println(get_wifi_status(status));
    }

    Serial.println("\nConnected to the WiFi network");
    Serial.print("Local ESP32 IP: ");
    Serial.println(WiFi.localIP());
}

void loop(){}
```

Examples of possible scenarios

Successful connection

```
Connecting
WL_IDLE_STATUS
WL_DISCONNECTED
WL_DISCONNECTED
WL_CONNECTED

Connected to the WiFi network
Local ESP32 IP: 192.168.43.129
```

SSID not found Wrong password

By restarting the ESP32

Occasionally, the ESP32 may temporarily fail to connect to the WiFi for unknown or strange reasons. The best solution is to say that after `n` seconds, if the ESP32 still hasn't connected to WiFi, we restart the ESP32. Just add a timeout and use the `ESP.restart()` function to restart the ESP32 from the code.

Here is an example that will restart the ESP32 after 10 seconds if it is still not connected to WiFi.

```
#include <WiFi.h>

#define CONNECTION_TIMEOUT 10

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void setup(){
    Serial.begin(115200);
    delay(1000);

    WiFi.mode(WIFI_STA); //Optional
    WiFi.begin(ssid, password);
    Serial.println("\nConnecting");
    int timeout_counter = 0;

    while(WiFi.status() != WL_CONNECTED){
        Serial.print(".");
        delay(200);
        timeout_counter++;
        if(timeout_counter >= CONNECTION_TIMEOUT*5){
        ESP.restart();
        }
    }

    Serial.println("\nConnected to the WiFi network");
    Serial.print("Local ESP32 IP: ");
    Serial.println(WiFi.localIP());
}

void loop(){}
```

The ESP32 is reset after 10 seconds in case of connection failure from the code with the the``SW_CPU_RESET`` flag during the boot.

```
Connecting
..................................................

ets Jun  8 2016 00:22:57

rst:0xc (SW_CPU_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5816
entry 0x400806ac

Connecting
.......
Connected to the WiFi network
Local ESP32 IP: 192.168.43.129
```

Change Wi-Fi channel

If the ESP32 has difficulty connecting to the WiFi depending on the period (or time slot), it may be related to the Wi-Fi channel chosen by your Acces Point. In fact, even if the Wi-Fi frequency is at 2.4GHz, the AP uses many subbands around 2.4GHz, called channels, to reduce the traffic. This makes it possible to have many different Wi-Fi routers in a very close space in buildings without them interfering with each other.

For France, there are 14 channels available. The available channels vary according to the country. In general, the channel is chosen automatically by the router (in `auto`), depending on the other channels used by the AP around. Except that, in practice, the routers are always stuck on the last channels, 12, 13 and 14

One solution is to change the channel number used by the router to number 1, 6 or 11, for example. This change is done via the router's administrator interface, usually on IP `192.168.0.1`.

If you use a lot of Wi-Fi-connected IoT objects, I recommend having a second Wi-Fi router to separate home use from IoT use. You shouldn't have any more Wi-Fi connection issues by choosing channel 1 or 6 for the IoT sensor router and a channel between 11 and 14 for the other router.

Advance WiFi network stuffs for the ESP32

Assign a static IP address

The local IP address of the ESP32 has been assigned automatically by the router's DHCP server. It is convenient to have an IP address automatically on the computer because we do

not have to enter it manually. The disadvantage (or advantage, depending on the case) is that the IP address is dynamic: it can change. This can become annoying if, for example, as soon as the ESP32 is restarted (or the DHCP lease is expired), the IP address changes while a web server is running on the ESP32. We would have to find the IP of the ESP32 every time. We can overcome this problem by setting the IP address of the ESP32 on the network. To do this, use the `WiFi.config(ip, dns, gateway, subnet)` function.

The parameters to be filled in are :

- `IP` : The IP address you wish to assign.
- `DNS` : Service that links a URL to an IP. By default, we use the DNS server of the router: so we indicate the same address as the router (in general, 192.168.0.1).
- `GATEWAY` : This is the IP address of the router (usually 192.168.0.1)
- `SUBNET` : Subnet mask (usually 255.255.255.0)

Note

You need to know the router's IP address: the easiest way is to use the code that displays the WiFi network information in the serial monitor.

In this example, using the WiFi connection sharing of a phone, the router IP address is 192.168.43.1. I choose to have the following static IP 192.168.43.42 for the ESP32.

Code to set the IP address of the ESP32

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

IPAddress ip(192, 168, 43, 42);
IPAddress dns(192, 168, 43, 1);
IPAddress gateway(192, 168, 43, 1);
IPAddress subnet(255, 255, 255, 0);

void setup(){
Serial.begin(115200);
delay(1000);

WiFi.config(ip, gateway, subnet, dns);
WiFi.begin(ssid, password);
Serial.println("\nConnecting");

while(WiFi.status() != WL_CONNECTED){
    Serial.print(".");
    delay(100);
}

Serial.println("\nConnected to the WiFi network");
Serial.print("[+] ESP32 IP : ");
Serial.println(WiFi.localIP());
}

void loop(){}
```

Warning

It is important to remember not to use an IP address already taken by another device on the network. You can also assign a fixed IP address linked to the MAC address directly from the router settings.

For information purposes, we can then ping the computer to the IP 192.168.43.42 to see if the new address has been taken into account:

```
Pinging 192.168.43.42 with 32 bytes of data:
Reply from 192.168.43.42: bytes=32 time=1ms TTL=255
Reply from 192.168.43.42: bytes=32 time=5ms TTL=255
Reply from 192.168.43.42: bytes=32 time=4ms TTL=255
Reply from 192.168.43.42: bytes=32 time=1ms TTL=255

Ping statistics for 192.168.43.42:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 5ms, Average = 2ms
```

Ping the ESP32 from the Windows terminal (cmd.exe)

Note

Suppose you have the error, `ping: transmission failure. General failure` , there is probably a software or firewall that disables the use of pings. This can be the case with VPN clients.

Change the MAC address

In some applications, it can be interesting to change the MAC address of the ESP32. We can change the MAC address with a few lines of code using the `esp_wifi_set_mac()` function.

Note

The MAC address change is temporary and does not replace the original one. You have to upload a new program to find the original MAC address.

```
#include <WiFi.h>
#include <esp_wifi.h>

uint8_t new_mac[] = {0x60, 0x8B, 0x0E, 0x01, 0x5A, 0x32};

void setup(){
    Serial.begin(115200);

    WiFi.mode(WIFI_STA);//Needed to change MAC adress
    Serial.print("[+] Current MAC Address:  ");
```

```
    Serial.println(WiFi.macAddress());

    esp_wifi_set_mac(ESP_IF_WIFI_STA, new_mac);

    Serial.print("[+] New MAC Address:  ");
    Serial.println(WiFi.macAddress());
}

void loop(){}
```

Serial terminal :

```
[+] Current MAC Address:  24:6F:28:BB:2E:E8
[+] New MAC Address:  60:8B:0E:01:5A:32
```

Code that connects to a Wi-Fi network with a modified MAC address

```
#include <WiFi.h>
#include <esp_wifi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

uint8_t new_mac[] = {0x6C, 0x8D, 0xC1, 0x01, 0x5A, 0x32};

void setup(){
    Serial.begin(115200);

    WiFi.mode(WIFI_STA);//Needed to change MAC adress
    esp_wifi_set_mac(ESP_IF_WIFI_STA, new_mac);
    delay(1000);
    WiFi.begin(ssid, password);
    Serial.println("\nConnecting");

    while(WiFi.status() != WL_CONNECTED){
        Serial.print(".");
        delay(100);
    }

    Serial.println("\nConnected to the WiFi network");
    Serial.print("Local ESP32 IP: ");
    Serial.println(WiFi.localIP());
}

void loop(){}
```

A Wireshark capture shows that the address has been changed (here by an Apple MAC address):

```
> Frame 409: 74 bytes on wire (592 bits), 74 bytes capt
> Ethernet II, Src: Apple_01:5a:32 (6c:8d:c1:01:5a:32),
> Internet Protocol Version 4, Src: 192.168.43.244, Dst
> Internet Control Message Protocol
```

Capture a network frame with Wireshark

Save energy

If you are using an ESP32 in a project that must necessarily use WiFi to work, it is a good idea to set the ESP32 to Deep Sleep mode in case of connection failure to minimize power consumption. This is similar to the ESP32 code that sleeps for 10 seconds between each attempt.

Code that allows putting the ESP32 in Deep Sleep between 2 attempts

```
#include <WiFi.h>
#include <esp_wifi.h>

//Time in seconds
#define CONNECTION_TIMEOUT 5
#define DEEP_SLEEP_DURATION 10

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void setup(){
    Serial.begin(115200);
    WiFi.begin(ssid, password);
    Serial.println("\nConnecting");
    int timeout_counter = 0;

    while(WiFi.status() != WL_CONNECTED){
        Serial.print(".");
        delay(100);
        timeout_counter++;
            if(timeout_counter >= CONNECTION_TIMEOUT*10){
                Serial.println("\nCan't establish WiFi connexion");
                //Setup timer
                esp_sleep_enable_timer_wakeup(DEEP_SLEEP_DURATION *
1000000);
                //Start deep sleep
                esp_deep_sleep_start();
            }
    }
    Serial.println("\nConnected to the WiFi network");
    Serial.print("Local ESP32 IP: ");
    Serial.println(WiFi.localIP());
}

void loop(){}
```

Use WiFi events to have an optimized code

Until now, we used to poll the WiFi functions: the ESP32 remains blocked until it receives an event from the ESP32 WiFi driver. We were doing sequential programming. Here's an example:

```
while(WiFi.status() != WL_CONNECTED){
    Serial.print(".");
    delay(100);
}
```

We regularly check if the ESP32 has successfully connected to the WiFi network. While waiting, we're not doing anything (we could make calculations between 2 polls) and are stuck in the loop.

A more efficient way of doing this is to use event-driven programming. Indeed, events are generated when the WiFi changes state. The advantage is that we can execute code automatically depending on the event received. This is very similar to the interrupts that we use on the GPIO pins. A change of state of the pin generates an interrupt, which will execute a portion of high-priority code. Here a change of state of the WiFi generates an event that will also execute a portion of code.

The basic code for managing events is as follows:

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void my_function(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info){
    //Code
}

void setup(){
    Serial.begin(115200);
    delay(1000);

    WiFi.mode(WIFI_STA); //Optional
    WiFi.onEvent(my_function, WIFI_EVENT_ID);

    WiFi.begin(ssid, password);
}

void loop(){}
```

We use the function `WiFi.onEvent(my_function, WIFI_EVENT_ID)` to specify which function will be executed when the event `WIFI_EVENT_ID` is detected. `WIFI_EVENT_ID` must be replaced by the name or number of the event (see table below). The function `my_function()` must have the parameters `WiFiEvent_t wifi_event`, `WiFiEventInfo_t wifi_info` even if they are not used.

Warning

Depending on the version of the ESP32 port in Arduino code you have installed, the name of the events is different.

- Version 2.0x

Liste des évènements Wi-Fi

Number	Event Name	Description
0	ARDUINO_EVENT_WIFI_READY	ESP32 WiFi ready
1	ARDUINO_EVENT_WIFI_SCAN_DONE	ESP32 finish scanning AP
2	ARDUINO_EVENT_WIFI_STA_START	ESP32 station start
3	ARDUINO_EVENT_WIFI_STA_STOP	ESP32 station stop
4	ARDUINO_EVENT_WIFI_STA_CONNECTED	ESP32 station connected to AP
5	ARDUINO_EVENT_WIFI_STA_DISCONNECTED	ESP32 station disconnected from AP
6	ARDUINO_EVENT_WIFI_STA_AUTHMODE_CHANGE	the auth mode of AP connected by ESP32 station changed
7	ARDUINO_EVENT_WIFI_STA_GOT_IP	ESP32 station got IP from connected AP
8	ARDUINO_EVENT_WIFI_STA_LOST_IP	ESP32 station lost IP and the IP is reset to 0
9	ARDUINO_EVENT_WPS_ER_SUCCESS	ESP32 station wps succeeds in enrollee mode
10	ARDUINO_EVENT_WPS_ER_FAILED	ESP32 station wps fails in enrollee mode
11	ARDUINO_EVENT_WPS_ER_TIMEOUT	ESP32 station wps timeout in enrollee mode
12	ARDUINO_EVENT_WPS_ER_PIN	ESP32 station wps pin code in enrollee mode
13	ARDUINO_EVENT_WIFI_AP_START	ESP32 soft-AP start
14	ARDUINO_EVENT_WIFI_AP_STOP	ESP32 soft-AP stop
15	ARDUINO_EVENT_WIFI_AP_STACONNECTED	a station connected to ESP32 soft-AP
16	ARDUINO_EVENT_WIFI_AP_STADISCONNECTED	a station disconnected from ESP32 soft-AP
17	ARDUINO_EVENT_WIFI_AP_STAIPASSIGNED	ESP32 soft-AP assign an IP to a connected station
18	ARDUINO_EVENT_WIFI_AP_PROBEREQRECVED	Receive probe request packet in soft-AP interface
19	ARDUINO_EVENT_WIFI_AP_GOT_IP6	ESP32 ap interface v6IP addr is preferred

19	ARDUINO_EVENT_WIFI_STA_GOT_IP6	ESP32 station interface v6IP addr is preferred
20	ARDUINO_EVENT_ETH_START	ESP32 ethernet start
21	ARDUINO_EVENT_ETH_STOP	ESP32 ethernet stop
22	ARDUINO_EVENT_ETH_CONNECTED	ESP32 ethernet phy link up
23	ARDUINO_EVENT_ETH_DISCONNECTED	ESP32 ethernet phy link down
24	ARDUINO_EVENT_ETH_GOT_IP	ESP32 ethernet got IP from connected AP
19	ARDUINO_EVENT_ETH_GOT_IP6	ESP32 ethernet interface v6IP addr is preferred
25	ARDUINO_EVENT_MAX	

Old version 1.x

The following code does the same thing as the code at the very beginning to connect to a router, but this time with the use of events when connected to a network.

Code that allows connecting to a router with WiFi events

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void connected_to_ap(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info) {
  Serial.println("\nConnected to the WiFi network");
}

void got_ip_from_ap(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info) {
  Serial.print("Local ESP32 IP: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  delay(1000);

  WiFi.mode(WIFI_STA); //Optional
  WiFi.onEvent(connected_to_ap, ARDUINO_EVENT_WIFI_STA_CONNECTED);
  WiFi.onEvent(got_ip_from_ap, ARDUINO_EVENT_WIFI_STA_GOT_IP);

  WiFi.begin(ssid, password);
  Serial.println("\nConnecting");
}

void loop() {}
```

Console output:

```
Connecting
.............
Connected to the WiFi network
Local ESP32 IP: 192.168.43.167
```

You can remove the execution of the function linked to an event during the execution of the program with `wifi.removeEvent(WIFI_EVENT_ID)`

A practical use case of events is the possibility of automatically reconnecting the ESP32 in case of disconnection.

```
#include <WiFi.h>

const char* ssid = "yourNetworkName";
const char* password = "yourNetworkPassword";

void connected_to_ap(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info) {
  Serial.println("[+] Connected to the WiFi network");
}

void disconnected_from_ap(WiFiEvent_t wifi_event, WiFiEventInfo_t
wifi_info) {
  Serial.println("[-] Disconnected from the WiFi AP");
  WiFi.begin(ssid, password);
}

void got_ip_from_ap(WiFiEvent_t wifi_event, WiFiEventInfo_t wifi_info) {
  Serial.print("[+] Local ESP32 IP: ");
  Serial.println(WiFi.localIP());
}

void setup() {
  Serial.begin(115200);
  delay(1000);

  WiFi.mode(WIFI_STA); //Optional
  WiFi.onEvent(connected_to_ap, ARDUINO_EVENT_WIFI_STA_CONNECTED);
  WiFi.onEvent(got_ip_from_ap, ARDUINO_EVENT_WIFI_STA_GOT_IP);
  WiFi.onEvent(disconnected_from_ap, ARDUINO_EVENT_WIFI_STA_DISCONNECTED);

  WiFi.begin(ssid, password);
  Serial.println("\nConnecting");
}

void loop() {}
```

Console output:

```
Connecting
[-] Disconnected from the WiFi AP
[+] Connected to the WiFi network
[+] Local ESP32 IP: 192.168.43.167
[-] Disconnected from the WiFi AP
[-] Disconnected from the WiFi AP
[-] Disconnected from the WiFi AP
```

```
[-] Disconnected from the WiFi AP
[+] Connected to the WiFi network
Local ESP32 IP: 192.168.43.167
```

Application example: Wi-Fi Scan

Here is a concrete application that allows you to scan the WiFi networks around:

```
#include "WiFi.h"

String get_encryption_type(wifi_auth_mode_t encryptionType) {
    switch (encryptionType) {
        case (WIFI_AUTH_OPEN):
            return "Open";
        case (WIFI_AUTH_WEP):
            return "WEP";
        case (WIFI_AUTH_WPA_PSK):
            return "WPA_PSK";
        case (WIFI_AUTH_WPA2_PSK):
            return "WPA2_PSK";
        case (WIFI_AUTH_WPA_WPA2_PSK):
            return "WPA_WPA2_PSK";
        case (WIFI_AUTH_WPA2_ENTERPRISE):
            return "WPA2_ENTERPRISE";
    }
}

void setup(){
    Serial.begin(115200);
    WiFi.mode(WIFI_STA);
}

void loop() {
    Serial.println("uPesy WiFi Scan Demo");
    Serial.println("[*] Scanning WiFi network");

        // WiFi.scanNetworks will return the number of networks found
        int n = WiFi.scanNetworks();
        Serial.println("[*] Scan done");
        if (n == 0) {
            Serial.println("[-] No WiFi networks found");
        } else {
            Serial.println((String)"[+] " + n + " WiFi networks found\n");
            for (int i = 0; i < n; ++i) {
                // Print SSID, RSSI and WiFi Encryption for each network
found
                Serial.print(i + 1);
                Serial.print(": ");
                Serial.print(WiFi.SSID(i));
                Serial.print(" (");
                Serial.print(WiFi.RSSI(i));
                Serial.print(" dB) [");
                Serial.print(get_encryption_type(WiFi.encryptionType(i)));
                Serial.println("]");
                delay(10);
            }
        }
```

```
        Serial.println("");

        // Wait a bit before scanning again
        delay(5000);
}
```

```
uPesy WiFi Scan Demo
[*] Scanning WiFi network
[*] Scan done
[+] 20 WiFi networks found

1: [redacted] (-37 dB) [WPA2_PSK]
2: [redacted] (-65 dB) [WPA2_PSK]
3: [redacted] 2 dB) [WPA2_PSK]
4: [redacted] 8 dB) [WPA2_PSK]
5: [redacted] -89 dB) [WPA2_PSK]
6: [redacted] [Open]
7: [redacted] ) [WPA_WPA2_PSK]
8: [redacted] (-91 dB) [WPA2_ENTERPRISE]
9: [redacted] (-92 dB) [WPA2_ENTERPRISE]
10: [redacted] 92 dB) [WPA_WPA2_PSK]
11: [redacted] VY 5540 series (-93 dB) [WPA2_PSK]
12: [redacted] [WPA_PSK]
13: [redacted] (-93 dB) [WPA2_PSK]
14: [redacted] (-94 dB) [WPA2_ENTERPRISE]
15: [redacted] B) [WPA_WPA2_PSK]
16: [redacted] [Open]
17: [redacted] 94 dB) [WPA2_PSK]
18: [redacted] (-95 dB) [WPA2_PSK]
```

ESP32 program that scans Wi-Fi surroundings router

Create a WiFi access point with an ESP32

(Updated at 01/05/2023)

The Access Point mode allows you to use the ESP32 to create a WiFi network to connect. This is similar to WiFi connection sharing available on phones. As with phones, the operation of a WiFi router is simulated: this is known as a Soft AP (for “software” WiFi access point). You should, therefore, not expect the same performance as with a conventional WiFi router, especially on a microcontroller!

Also, note that, unlike the phone’s connection sharing, the ESP32 is not connected to the Internet. So you can use the Access Point mode to create a private WiFi local area network wholly isolated from the Internet.

Here are some uses of the Access Point mode:

- Connecting to the ESP32 in Access Point mode temporarily to enter the credentials of its WiFi router and allow the ESP32 to connect to our classic WiFi network. Most connected objects use this principle to connect to the home WiFi.
- Have a separate network from your home network and not connected to the Internet.
- To easily communicate between several ESP32 by WiFi.
- To have a local server on an isolated WiFi network.

Set the ESP32 to Access Point mode (AP)

Simply use `WiFi.mode(WIFI_AP)` and `WiFi.softAP()` to enable Access Point mode. Here is a straightforward example of how to create a WiFi network with an ESP32 :

```
#include <WiFi.h>

const char* ssid     = "uPesy_AP";
const char* password = "super_strong_password";

void setup()
{
    Serial.begin(115200);
    Serial.println("\n[*] Creating AP");
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password);
    Serial.print("[+] AP Created with IP Gateway ");
    Serial.println(WiFi.softAPIP());
}

void loop(){}
```

Tip

Don’t forget to import the `WiFi.h` module with `#include <WiFi.h>` .

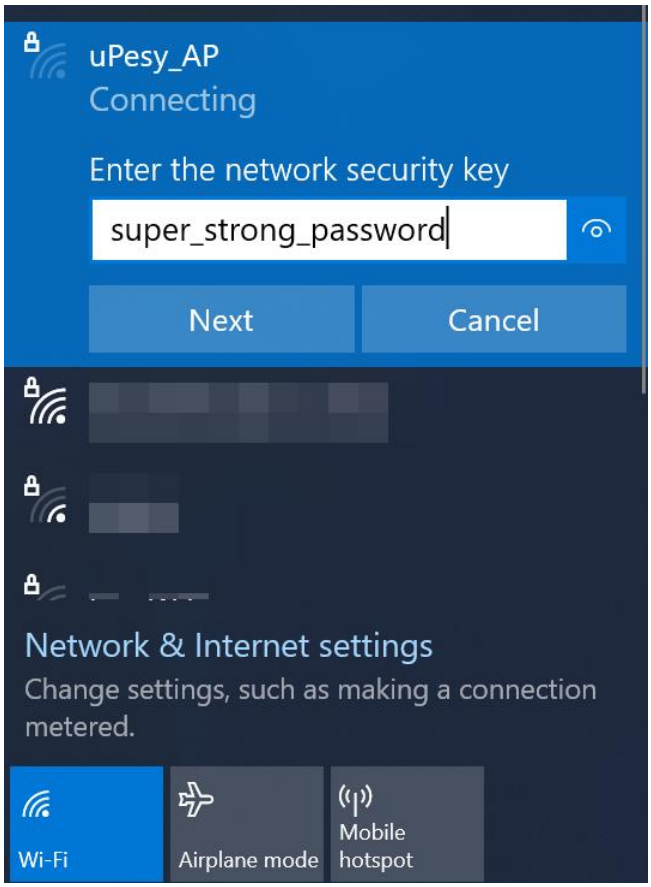
Important

If the chosen password is less than 8 characters, the compiler will return an error. The password must be at least 8 to 63 characters long.

Terminal output

```
[*] Creating AP
[+] AP Created with IP Gateway 192.168.4.1
```

You can then connect to the Wi-Fi network by entering the password. In this example, it's a computer on Windows 10 that will connect to ESP32.

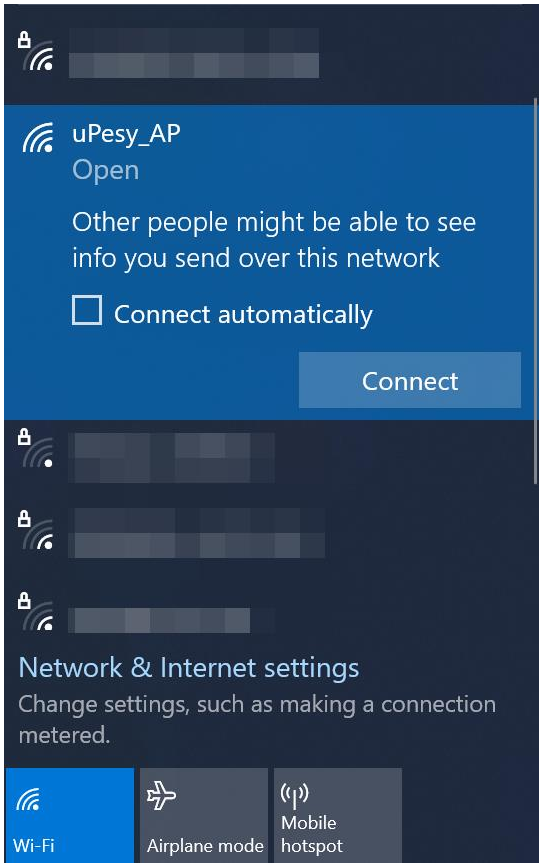


Connection to the ESP32 Wi-Fi network

You can also create an open Wi-Fi network by setting `NULL` for the password :

```
#include <WiFi.h>

const char* ssid     = "uPesy_AP";
const char* password = NULL;

void setup()
{
    Serial.begin(115200);
    Serial.println("\n[*] Creating AP");
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password);
    Serial.print("[+] AP Created with IP Gateway ");
    Serial.println(WiFi.softAPIP());
}

void loop(){}
```

The access point appears as an open network :



Connect to the ESP32 Wi-Fi network without a password

Tip

The IP displayed in the terminal corresponds to the router's local IP on the new WiFi network.

The access point can also be further customized by specifying the WiFi channel used (more information on the WiFi channels is available on Wikipedia), the visibility or not of the network, as well as the maximum number of devices that can be connected at the same time.

```
#include <WiFi.h>

const char* ssid           = "uPesy_AP";               // SSID Name
const char* password       = "super_strong_password";  // SSID Password - Set to NULL to have an open AP
const int   channel        = 10;                       // WiFi Channel number between 1 and 13
const bool  hide_SSID      = false;                    // To disable SSID broadcast -> SSID will not appear in a basic WiFi scan
const int   max_connection = 2;                        // Maximum simultaneous connected clients on the AP

void setup()
{
    Serial.begin(115200);
    Serial.println("\n[*] Creating AP");
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password, channel, hide_SSID, max_connection);
```

```
    Serial.print("[+] AP Created with IP Gateway ");
    Serial.println(WiFi.softAPIP());
}

void loop(){}
```

Once connected to the WiFi network from a device, one can then ping the router, in this case, the ESP32 in the Windows command prompt with:

```
ping 192.168.4.1
```

```
Pinging 192.168.43.42 with 32 bytes of data:
Reply from 192.168.43.42: bytes=32 time=1ms TTL=255
Reply from 192.168.43.42: bytes=32 time=5ms TTL=255
Reply from 192.168.43.42: bytes=32 time=4ms TTL=255
Reply from 192.168.43.42: bytes=32 time=1ms TTL=255

Ping statistics for 192.168.43.42:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 1ms, Maximum = 5ms, Average = 2ms
```

Ping the ESP32 from the Windows terminal

We have our local WiFi router on an ESP32 on which we can add several devices (computer, phone, ESP32, Raspberry Pi …)

Advanced configuration

The IP address `192.168.4.1` of the access point can be modified, as well as the gateway's IP address and the subnet mask. You must use the `WiFi.softAPConfig()` function before creating the access point via `WiFi.softAP()` . The function `WiFi.softAPConfig(local_ip, gateway, subnet)` accepts the following parameters

- `IPAddress local_ip` is the local IP address of our access point in the newly created network
- `IPAddress gateway` is the IP address of the gateway. In our case, it will be the same as the local IP.
- `subnet` the subnet mask of the network created by the ESP32, usually `255.255.255.0`

For example, to have a Wi-Fi access point with a local IP `192.168.0.1` with a subnet mask of `255.255.255.0` , the code would be :

```
#include <WiFi.h>
```

```
const char* ssid     = "uPesy_AP";
const char* password = "super_strong_password";

IPAddress local_ip(192,168,0,1);
IPAddress gateway(192,168,0,1);
IPAddress subnet(255,255,255,0);

void setup()
{
  Serial.begin(115200);
  Serial.println("\n[*] Creating AP");

  WiFi.mode(WIFI_AP);
  WiFi.softAPConfig(local_ip, gateway, subnet);
  WiFi.softAP(ssid, password);

  Serial.print("[+] AP Created with IP Gateway ");
  Serial.println(WiFi.softAPIP());
}

void loop(){}
```

Terminal output

```
[*] Creating AP
[+] AP Created with IP Gateway 192.168.0.1
```

Application examples

An example displays a list of devices connected to the ESP32 with their MAC and local IP addresses. It is a simplified version of the one you can have on a typical WiFi router.

```
#include <WiFi.h>
#include "esp_wifi.h"

const char* ssid           = "uPesy_AP";                 // SSID Name
const char* password       = "super_strong_password";    // SSID Password -
Set to NULL to have an open AP
const int   channel        = 10;                         // WiFi Channel
number between 1 and 13
const bool  hide_SSID      = false;                      // To disable SSID
broadcast -> SSID will not appear in a basic WiFi scan
const int   max_connection = 2;                          // Maximum
simultaneous connected clients on the AP

void display_connected_devices()
{
    wifi_sta_list_t wifi_sta_list;
    tcpip_adapter_sta_list_t adapter_sta_list;
    esp_wifi_ap_get_sta_list(&wifi_sta_list);
    tcpip_adapter_get_sta_list(&wifi_sta_list, &adapter_sta_list);

    if (adapter_sta_list.num > 0)
```

```
        Serial.println("-----------");
    for (uint8_t i = 0; i < adapter_sta_list.num; i++)
    {
        tcpip_adapter_sta_info_t station = adapter_sta_list.sta[i];
        Serial.print((String)"[+] Device " + i + " | MAC : ");
        Serial.printf("%02X:%02X:%02X:%02X:%02X:%02X", station.mac[0],
station.mac[1], station.mac[2], station.mac[3], station.mac[4],
station.mac[5]);
        Serial.println((String) " | IP " + ip4addr_ntoa(&(station.ip)));
    }
}

void setup()
{
    Serial.begin(115200);
    Serial.println("\n[*] Creating AP");
    WiFi.mode(WIFI_AP);
    WiFi.softAP(ssid, password, channel, hide_SSID, max_connection);
    Serial.print("[+] AP Created with IP Gateway ");
    Serial.println(WiFi.softAPIP());
}

void loop()
{
    display_connected_devices();
    delay(5000);
}
```

Terminal output

```
[+] Device 0 | MAC : 44:E2:42:08:D7:11 | IP 192.168.4.2
[+] Device 1 | MAC : 00:0A:F7:42:42:42 | IP 192.168.4.3
-----------
[+] Device 0 | MAC : 44:E2:42:08:D7:11 | IP 192.168.4.2
[+] Device 1 | MAC : 00:0A:F7:42:42:42 | IP 192.168.4.3
```

Alternatively, some devices could be prevented from connecting by filtering MAC addresses. For example, only allow the ESP32 to connect to it. Indeed,
the `esp_wifi_deauth_sta(int device_id)` function can be used by specifying the device number you want to remove.

Tip

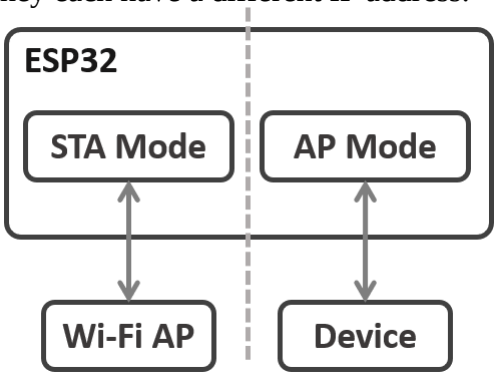
The solution of the first person who will send me an example of a code that allows filtering the MAC addresses on the WIFI router created by the ESP32 will be published here!

Use the STATION and AP mode at the same time

It is possible to have the ESP32 working with the STATION and AP modes at the same time. That is to say that the ESP32 is connected to a classic WiFi router (STATION mode) with also a WiFi access point activated (AP mode). This mode is called `WIFI_AP_STA` .

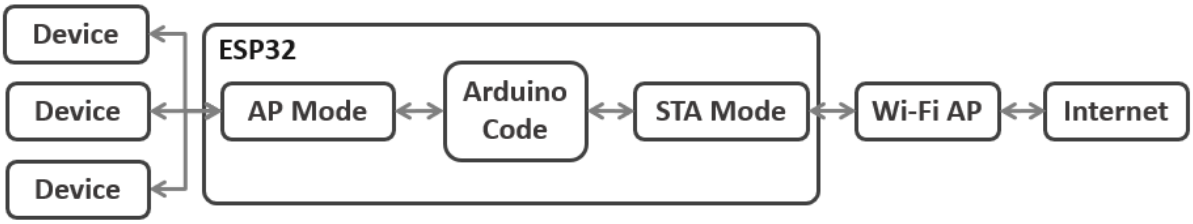
Note

There is a separate network interface for each mode. Since the interfaces are independent, they each have a different IP address.



Use the STATION and AP mode at the same time

One of the possible applications is to connect to the Internet the local WiFi network created by the ESP32 in Soft AP mode.



Possible example of using the 2 modes simultaneously

The Arduino code mixes both the management of the STATION mode and the AP mode. The code is relatively simple when you already use the two modes separately:

- We use the WiFi mode `WIFI_AP_STA`
- We create our local WiFi access point (Soft AP)
- Then we connect to the classic WiFi router

Here is an example that allows you to have the two modes at the same time:

```
#include <WiFi.h>

const char* wifi_network_ssid     = "Lounge";
const char* wifi_network_password =  "cupcakes";

const char *soft_ap_ssid          = "uPesy_AP";
const char *soft_ap_password      = NULL;

void setup()
{
    Serial.begin(115200);
    WiFi.mode(WIFI_AP_STA);
```

```
    Serial.println("\n[*] Creating ESP32 AP");
    WiFi.softAP(soft_ap_ssid, soft_ap_password);
    Serial.print("[+] AP Created with IP Gateway ");
    Serial.println(WiFi.softAPIP());

    WiFi.begin(wifi_network_ssid, wifi_network_password);
    Serial.println("\n[*] Connecting to WiFi Network");

    while(WiFi.status() != WL_CONNECTED)
    {
        Serial.print(".");
        delay(100);
    }

    Serial.print("\n[+] Connected to the WiFi network with local IP : ");
    Serial.println(WiFi.localIP());
}

void loop() {}
```

Terminal output

```
[*] Creating ESP32 AP
[+] AP Created with IP Gateway 192.168.4.1

[*] Connecting to WiFi Network
............................................................
[+] Connected to the WiFi network with local IP: 192.168.85.37
```

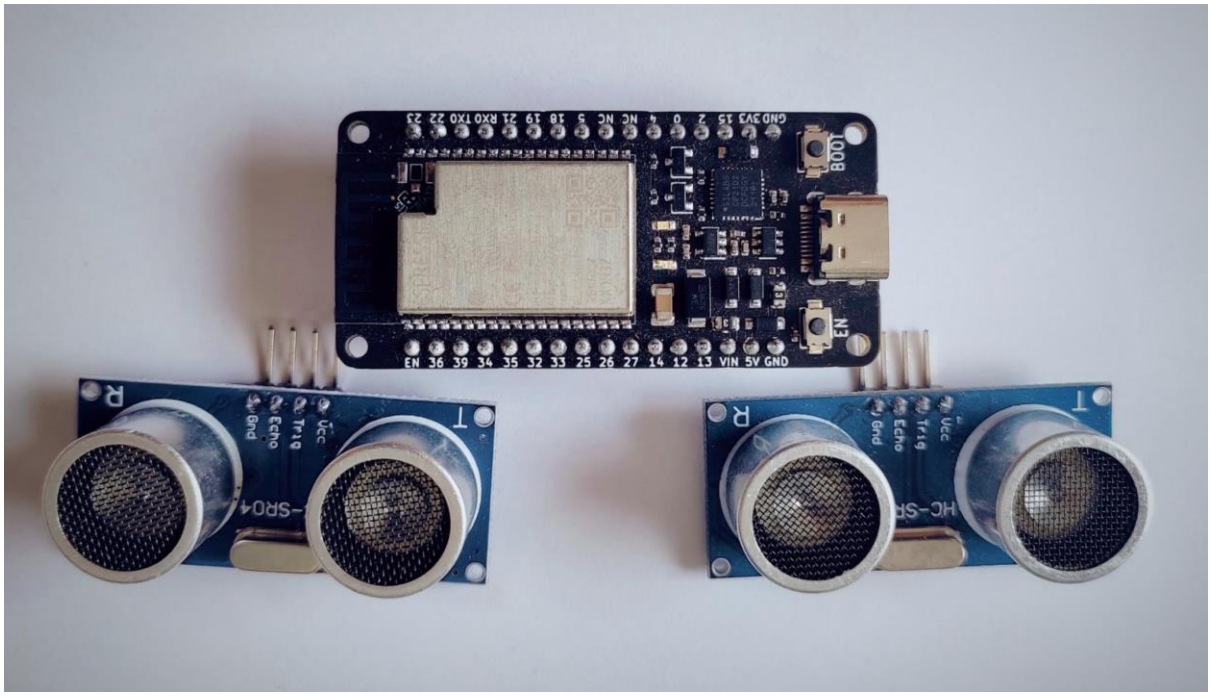
Important

Don’t mix up the credentials:

- `wifi_network` refers to the credentials of the classic WiFi router (AT&T, Verizon, Xfinity)
- `soft_ap` is the credential of the access point that we will create with the ESP32. You will have to enter this password on the devices that want to connect to the ESP32.

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

(Updated at 01/04/2023)



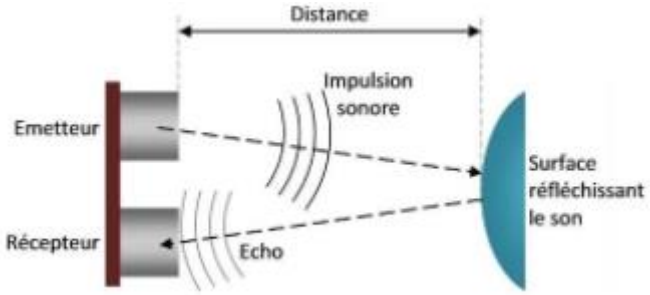
The sensor **HC-SR04** allows measurement distances to an obstacle based on ultrasonic waves. The available modules are generally composed of an ultrasonic transmitter, a receiver, and a chip that controls them.

Tip

Please check that the module you have is an HC-SR04. Several ultrasonic modules look similar but do not use the same chip. The **HC-SR04** (usually in blue) works only with 5V, while the **RCWL-1601** (usually in green) also works with 3.3V. Fortunately, they are generally driven in the same way.

Getting started with the HC-SR04 module

The physical operation of the sensor



If you'd like to understand how the module works, you can consult the article that explains the detailed physical function of the HC-SR04 .

Technical specifications of the HC-SR04

- Supply voltage: 5V
- Consumption: The sensor consumes around 20mA in operation
- Range: The module can measure a distance between 3cm to 4m in theory. In practice, a range between 10cm to 2.5m will give you the best results.
- Measurement angle: < 15°

  Warning

  Obstacles slightly offset from the module can be measured, but the obstacle's surface must be flat to get reliable measurements.

- Ultrasonic frequency: 40 kHz

HC-SR04 sensor pinout

Apart from the power pins, only two pins are required to control the module. You can even use only one! In practice, two pins are used unless not enough pins are available on the ESP32.

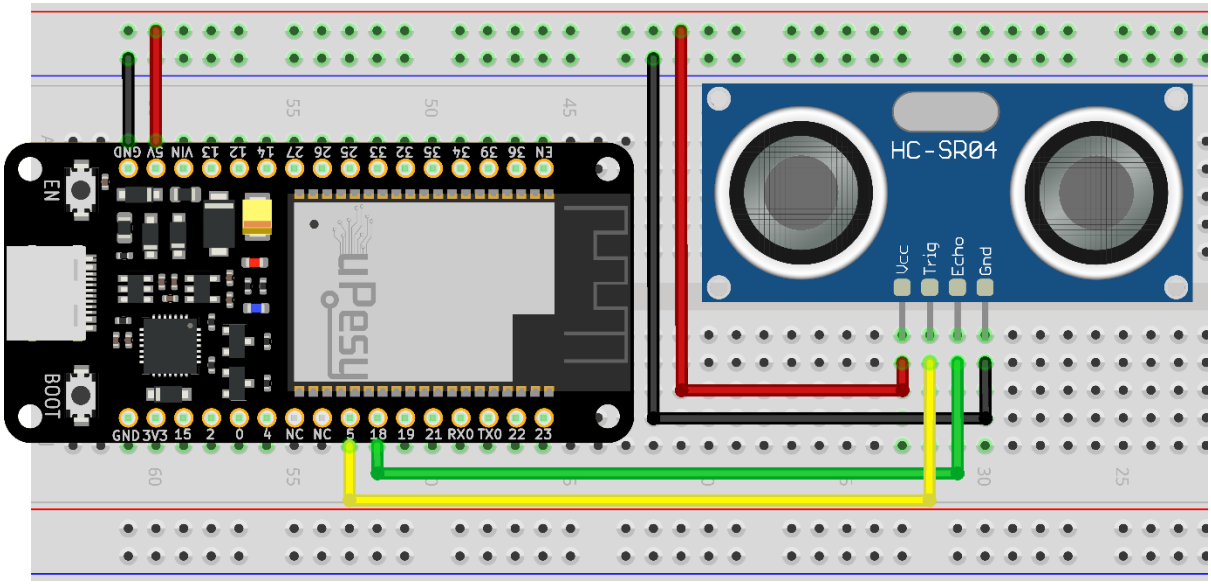
Pin Wiring

HC SR04 Module	ESP32
TRIG	GPIO5
ECHO	GPIO18
GND	GND
VCC	5V

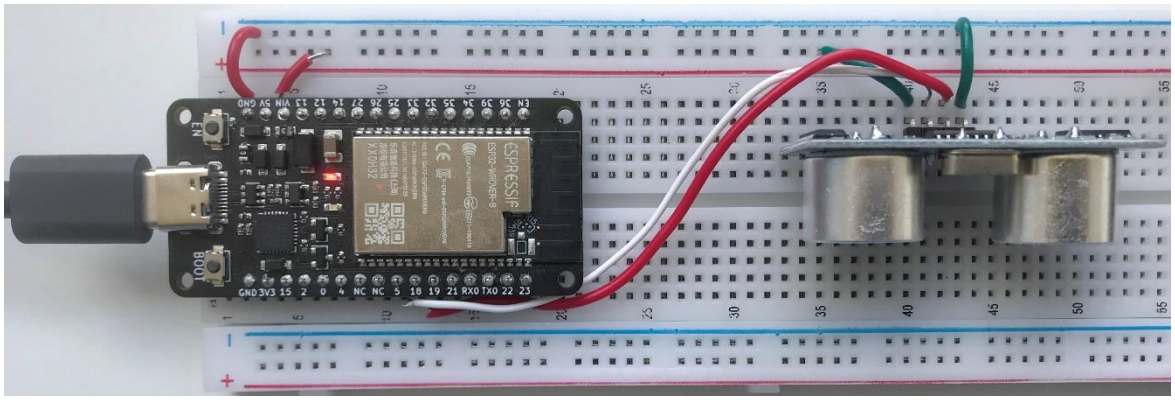
Note

The pins `GPIO5` and `GPIO18` have been chosen arbitrarily: you can take any of the output ESP32 pins for `TRIG` and input for `ECHO` .

HC-SR04 module with an ESP32 Schematic



Schematic to be made



Schematic to be made

What is the power supply of the module: 5V or 3.3V?

The HC-SR04 module is not the most relevant sensor for ESP32 boards because it works in 5V (5V logic levels), whereas ESP32 works in 3.3V. It should be supplied with 5V, and either levels shifters or voltage divider bridges should be added on the pin `ECHO` . In this way, the pin of the ESP32 `GPIO18` will receive 3.3V instead of the original 5V. But this makes the circuit a bit more complex.

Tip

In practice, if it is for temporary use, the pins of the ESP32 can support a voltage of 5V. In the long term, they risk being damaged and not working anymore. **That's why these precautions have been omitted in this tutorial for simplicity.**

Note

Some internet users say that they manage to power the HCSR04 module with 3.3V and get reliable measurements by increasing the duration of the pulse sent. However, I did not manage to make work the ones I had in my possession with a voltage of 3.3V.

If you use many boards with 3.3V (ESP8266, ESP32, Raspberry Pi Pico), I advise you to use an equivalent model designed to work with this voltage: for example, the **RCWL-1601.**

Easily measure a distance with basic Arduino functions

It is not necessary to use external libraries because the process is straightforward. When the ESP32 sends a pulse on the `TRIG` pin, the module sends back a pulse to the `ECHO` pin, whose duration is proportional to the distance from the obstacle.

It corresponds to the time taken by the ultrasonic wave between its emission and its reception by the module. By applying basic physical formulas, we can easily find the measured distance.

µ�������(��)=������(µ�)×������(�/�)×10−4
2

Note

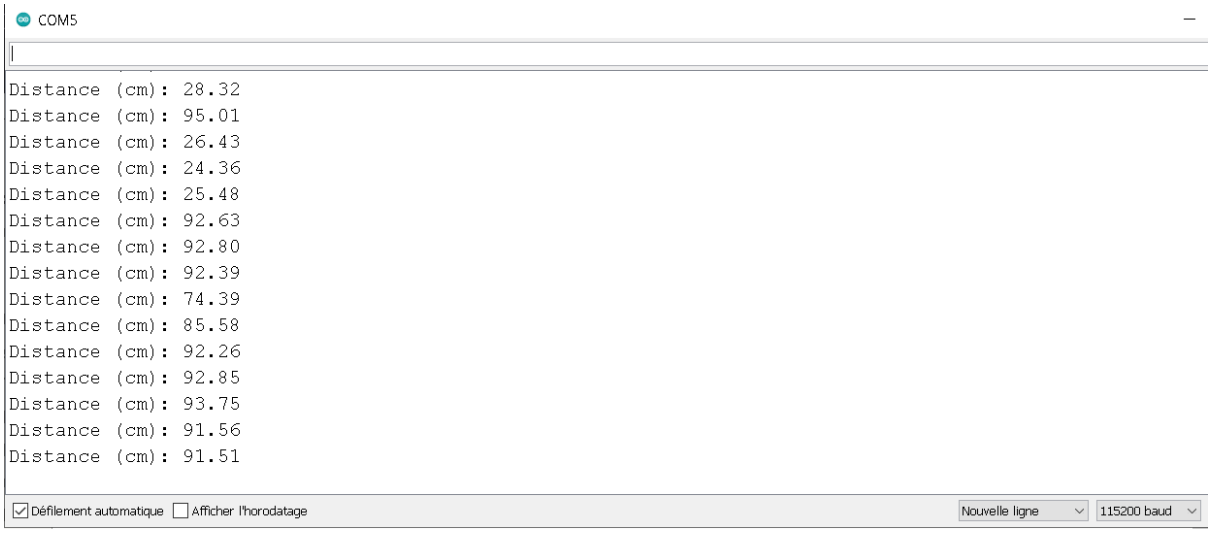
Obtaining this formula is detailed in the article on the operation of an ultrasonic sensor .

You must implement a measurement’s triggering, get the value and apply the formula. This is what the code below does:

```
const int trig_pin = 5;
const int echo_pin = 18;

// Sound speed in air
#define SOUND_SPEED 340
#define TRIG_PULSE_DURATION_US 10


long ultrason_duration;
float distance_cm;

void setup() {
  Serial.begin(115200);
  pinMode(trig_pin, OUTPUT); // We configure the trig as output
  pinMode(echo_pin, INPUT); // We configure the echo as input
}

void loop() {
  // Set up the signal
  digitalWrite(trig_pin, LOW);
  delayMicroseconds(2);
 // Create a 10 µs impulse
  digitalWrite(trig_pin, HIGH);
  delayMicroseconds(TRIG_PULSE_DURATION_US);
  digitalWrite(trig_pin, LOW);

  // Return the wave propagation time (in µs)
  ultrason_duration = pulseIn(echo_pin, HIGH);

//distance calculation
  distance_cm = ultrason_duration * SOUND_SPEED/2 * 0.0001;

  // We print the distance on the serial port
  Serial.print("Distance (cm): ");
  Serial.println(distance_cm);

  delay(1000);
}
```

The program generates a 10µs pulse sent on the `GPIO5` of the ESP32.

The function `pulseIn()` blocks the program until it receives the response pulse from the HC-SR04 from the `18` . Then the code calculates the distance from the duration of the ultrasound wave. The program displays the distance between the module and an obstacle (my hand in this case 🙂) in the serial monitor.



You can decrease the delay `delay(1000)` to have more regular measurements. (Until the ultrasonic module has time to make the measurement)

This approach is excellent for understanding how the HC-SR04 module works. However, as it is, it will not be efficient in more complex programs because it blocks the execution of other tasks via the `pulseIn()` function. The solution is to use interrupts to have asynchronous measurements.

Libraries for the HC-SR04 in ESP32-compatible Arduino code

The use of external libraries dedicated to the HC-SR04 is relevant in a larger project when you want to have more features:

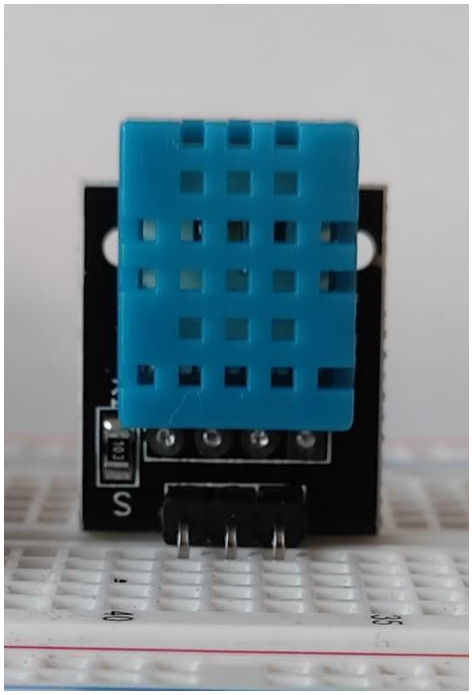
- Efficiently manage multiple ultrasound modules
- Have asynchronous distance readings (with interrupts and timers)
- Managing measurement errors

Warning

If you already use libraries for the HC-SR04 with an Arduino board, the library needs to work correctly with an ESP32. For example, the library NewPing focuses on ultrasonic sensors but still needs to integrate the management of asynchronous measurements for the ESP32.

DHT11: Measuring temperature and humidity with the ESP32 in Arduino code

(Updated at 01/04/2023)



The DHT11 sensor with its typical blue protection

The DHT11 sensor is a trendy temperature and humidity sensor in the Makers and DIY world. Its low price and ease of use have given it this success.

Warning

The fact that you can only get one measurement per second is its main drawback. It will be perfectly suitable for making a DIY IoT weather station. 🙂

Getting started with the DHT11 sensor

A higher-end model, the DHT 22, is more complete. To differentiate them, we can look at the colors of their housing: the DHT11 has blue housing while the DHT22 is white. The DHT22 is also slightly bigger. Apart from their differences, they are used in the same way.
Please compare DHT11 and DHT22 sensors to learn more about the differences.

Some technical characteristics of the DHT11

The DHT11 can be directly powered by 3.3V.

DHT11 Specifications

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

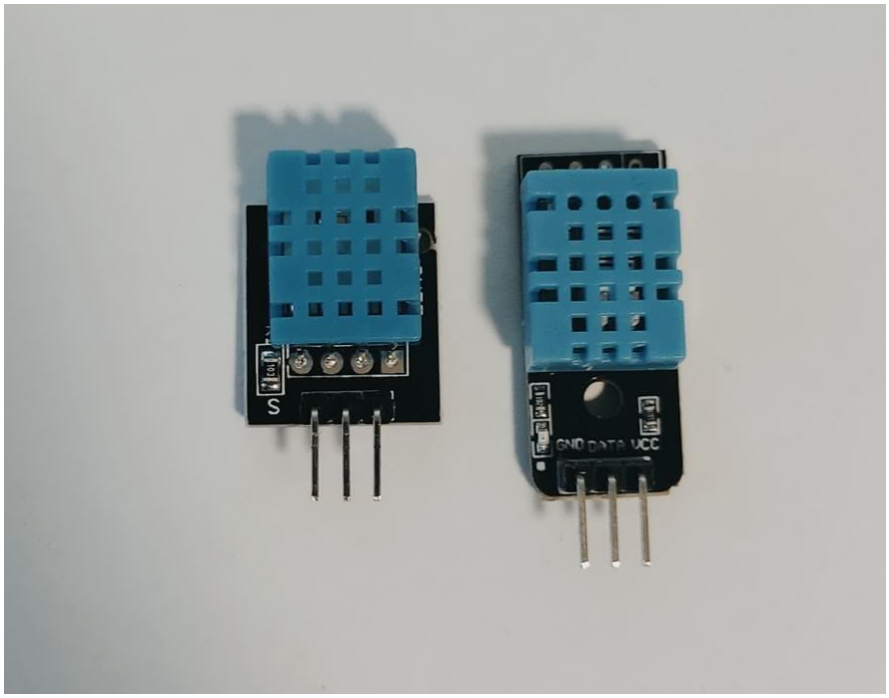
DHT11 Specifications

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

The performances are simple, which limits the use of the sensor. I advise you to use a sensor of the BMExxx series of the manufacturer Bosh for better measurements.

DHT11 Sensor Wiring on ESP32

The DHT 11 sensor has between 3 and 4 pins. Depending on your module on the 4-pin model, one pin is useless. Unfortunately, different pin-outs rely on the module manufacturer. Here is a summary of the other possibilities:



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

The numbering is done from the left when you hold the sensor facing you (the part with the grid in front of you).

- (•) 3 pins (Elegoo)

Pin Wiring

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

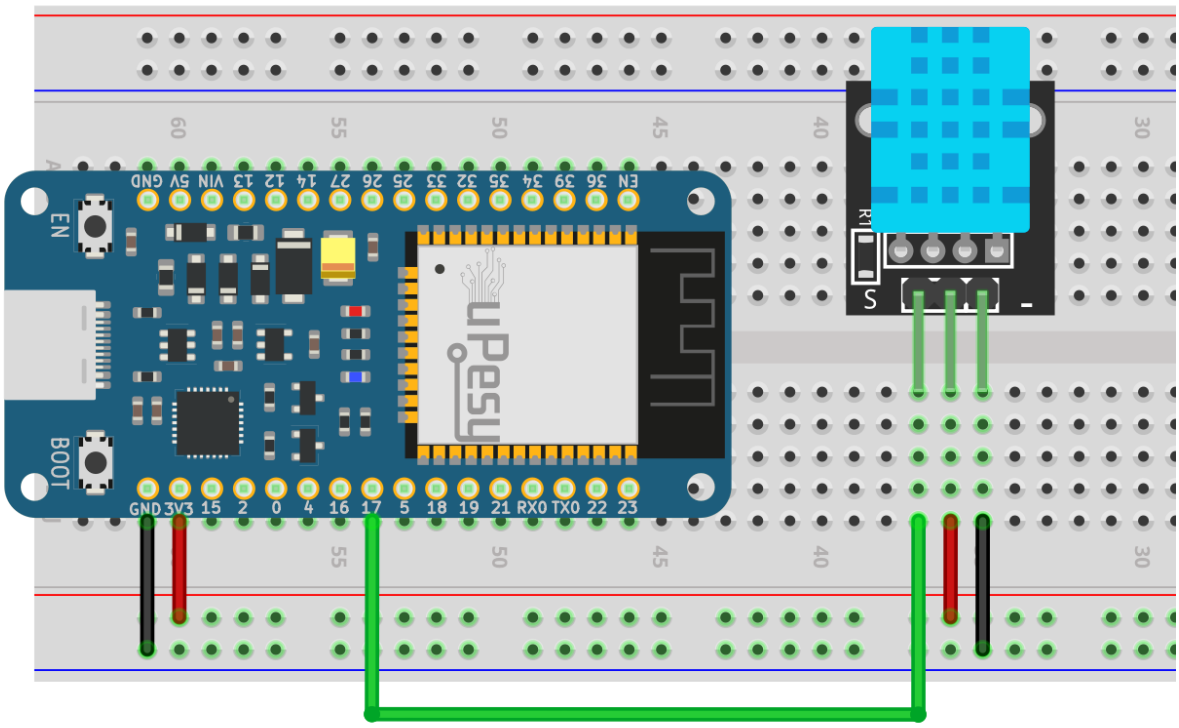
- ( ) 3 pins (bis)
- ( ) 4 pins

Apart from the power supply pins, the DHT11 has a single pin to transmit sensor data.

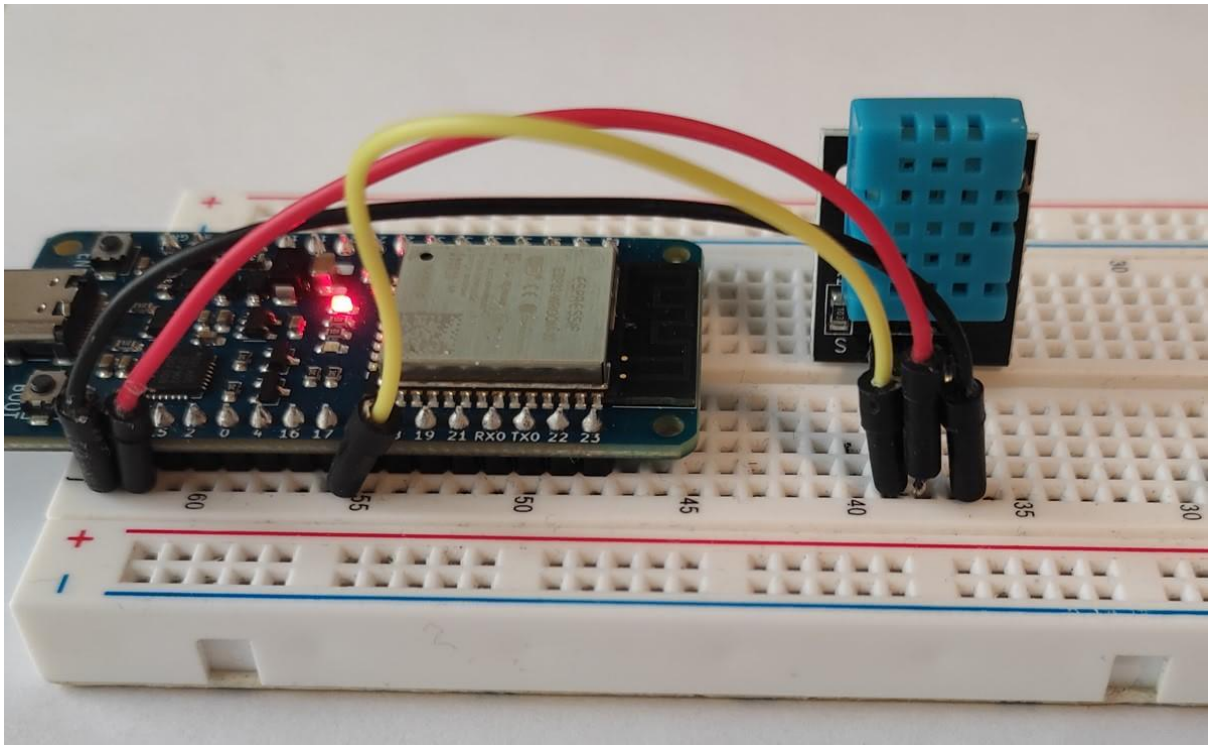
Wiring diagram for using the DHT11 module with an ESP32

If the module does not contain a pullup resistor, you must add one between 4.7kΩ (or 10kΩ) between the pins `3V3` and the signal (`GPIO17`). You can always add one more; it won't hurt.

Any output pin can be used. Here we use the pin `GPIO17` . In the following diagrams, I use the DHT11 sensor from Elegoo; please make sure to adapt it to your model if it is slightly different. 🙂



Wiring the DHT11 with a uPesy ESP32 Wroom DevKit board



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

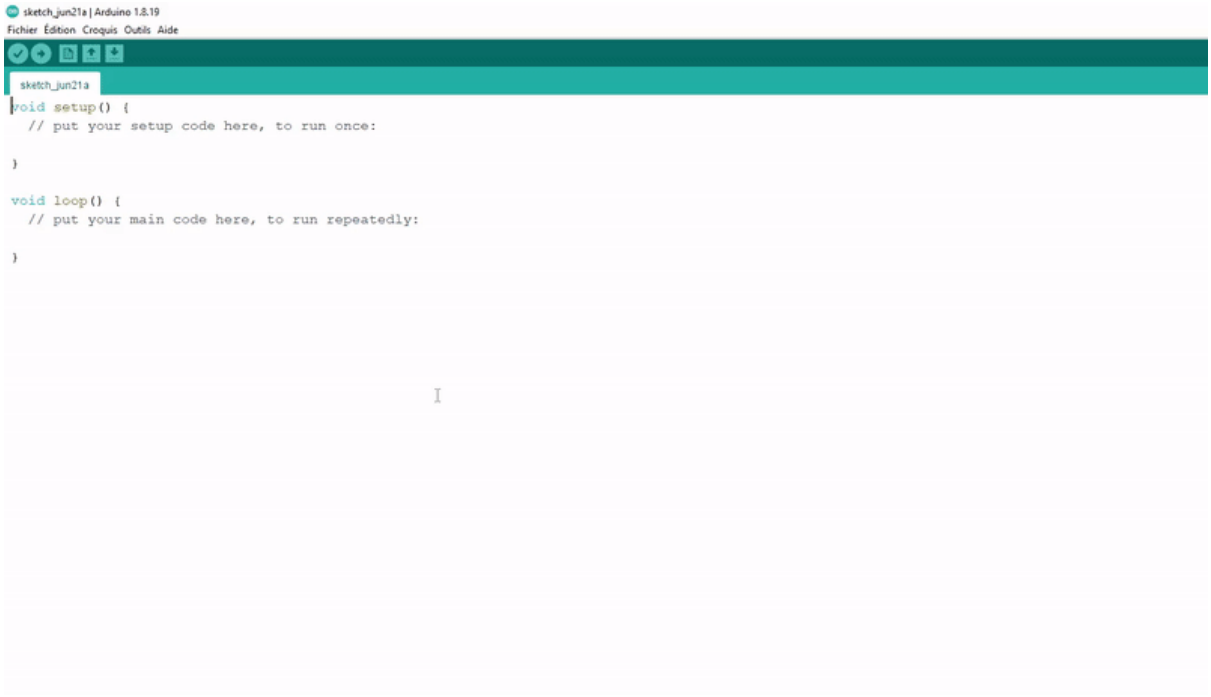
Warning

Make sure you supply the module with 3.3V and not 5V to have a 3.3V data signal.

Measuring the temperature and humidity of the DHT11 with Arduino code

Since the DHT11 uses a proprietary protocol, it is essential to use a library to communicate easily with the sensor. It would be best if you used adafruit's DHT Sensor library, which works properly on the ESP32 .

Here is how to install it from the Arduino IDE :



Installation of the DHT11 library on Arduino IDE

Once the library is installed, we use the following code that retrieves the temperature and humidity of the DHT11.

```
#include "DHT.h"

#define DHTPIN 17
#define DHTTYPE DHT11
//DHTTYPE = DHT11, but there are also DHT22 and 21

DHT dht(DHTPIN, DHTTYPE); // constructor to declare our sensor

void setup() {
  Serial.begin(115200);
  dht.begin();
}

void loop() {
  delay(1000);
  // The DHT11 returns at most one measurement every 1s
  float h = dht.readHumidity();
  //Read the moisture content in %.
  float t = dht.readTemperature();
  //Read the temperature in degrees Celsius
  float f = dht.readTemperature(true);
  // true returns the temperature in Fahrenheit

  if (isnan(h) || isnan(t) || isnan(f)) {
    Serial.println("Failed reception");
    return;
    //Returns an error if the ESP32 does not receive any measurements
  }

  Serial.print("Humidite: ");
  Serial.print(h);
```

```
  Serial.print("%  Temperature: ");
  Serial.print(t);
  Serial.print("°C, ");
  Serial.print(f);
  Serial.println("°F");
  // Transmits the measurements received in the serial monitor
}
```

Note

The `DHT.h` at the beginning of the program includes the DHT Sensor library.

This is what you get in the console:

Note

You can blow on the DHT11 sensor like a cold window to clear the fog. You should see the humidity level skyrocket.

It is not possible to have more measurements per second. This sensor is not very suitable for quickly detecting a sudden temperature change.

Making asynchronous measurements with the DHT11 sensor

You will notice that the measurement is blocked: the ESP32 cannot do anything else during the measurements. An obvious solution is to replace the `delay()` with the function `millis()` in a condition `if` . But despite this, the exchange of data with the DHT11 will permanently be blocked: your program will be blocked by jerks

The best solution is to use a library that works with timers and interrupts so that the measurements are done in the background:

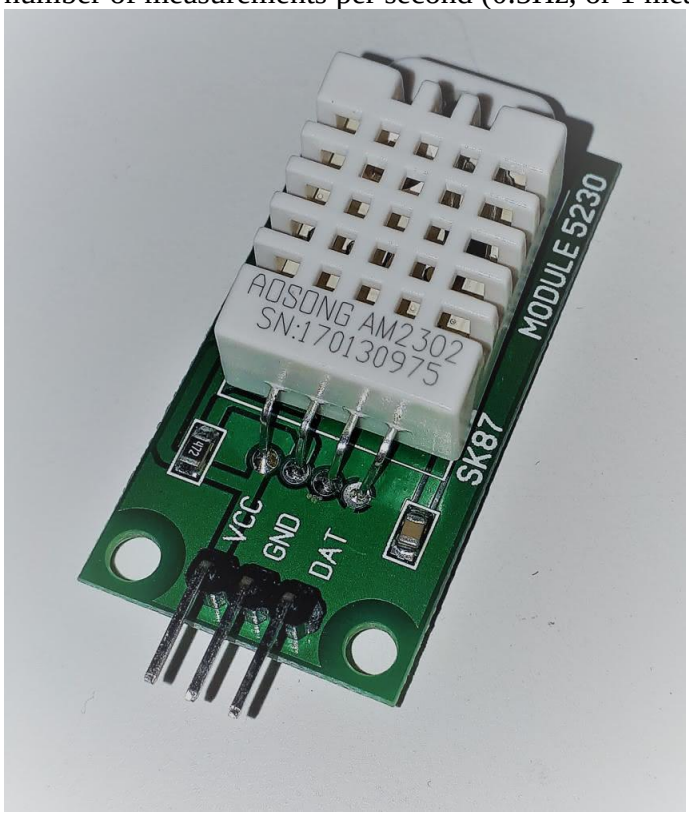
DHT22: Measuring temperature and humidity with the ESP32 in Arduino code

(Updated at 01/06/2023)

The DHT22 is the big brother of the DHT11, the low-cost temperature and humidity sensor for DIY projects.

Warning

It offers better measurement accuracy but has the same limitation as the DHT11: a low number of measurements per second (0.5Hz, or 1 measurement every 2 seconds)!



The DHT22 sensor with its characteristic white protection

Getting started with the DHT22 sensor

The DHT22 is part of the DHTxx family. A comparison between the DHT11 and the DHT22 is available to see their differences better. The DTH22 has a white plastic housing,

while the DHT11 has a blue one. If your sensor is blue, I recommend reading the DHT11 tutorial .

Note

Sometimes the **DHT22** is designated by the reference **AM2302.**

Characteristics of the DHT22 sensor

Technical characteristics of the 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 more suitable than the DHT11 for measuring temperatures outdoors, as it can measure temperatures down to -40°C. For an IoT greenhouse, for example, the DHT22 is recommended!

The performance of the DHT22 is really basic and less reliable than the BMExxx sensors from the manufacturer Bosh. In the end, it all depends on your application. 😉

Connections of the DHT22 sensor

The DHT22 sensor has between 3 and 4 pins depending on your module. The sensor has 4 pins, but since one is useless, the modules that integrate it only expose the 3 useful pins. Beware because there are several variations of the pin-out of the DHT22 module, depending on the manufacturer.

The numbering is done from the left when you hold the sensor facing you (the part with the grid in front of you). The power supply is always the first pin; the others may vary.

 3 pins

Pin Wiring

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

 3 pins (bis)  4 pins (DHT22 only)

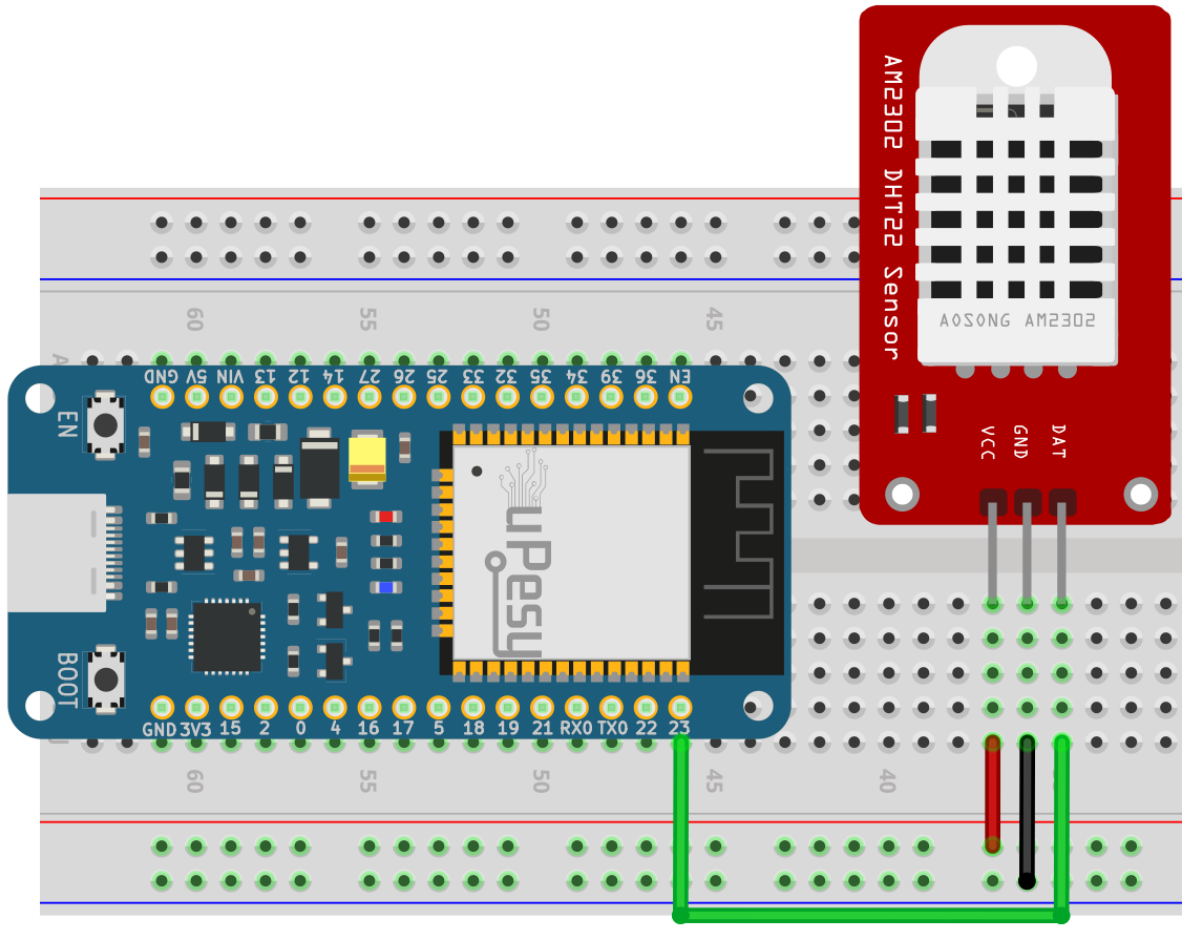
Apart from the power supply pins, the DHT22 has a single pin to transmit sensor data.

Wiring diagram for using the DHT22 module with an ESP32

If the module does not contain a pull-up resistor, you must add one between 4.7kΩ to 10kΩ between the pins `3V3` and the signal (`GPIO23`). You can always add one more; it won't hurt 😉

Any output pin can be used. Here we use the pin `GPIO23` . Don't forget to adapt the circuit to your model if it is slightly different. 🙂

- 3 pins

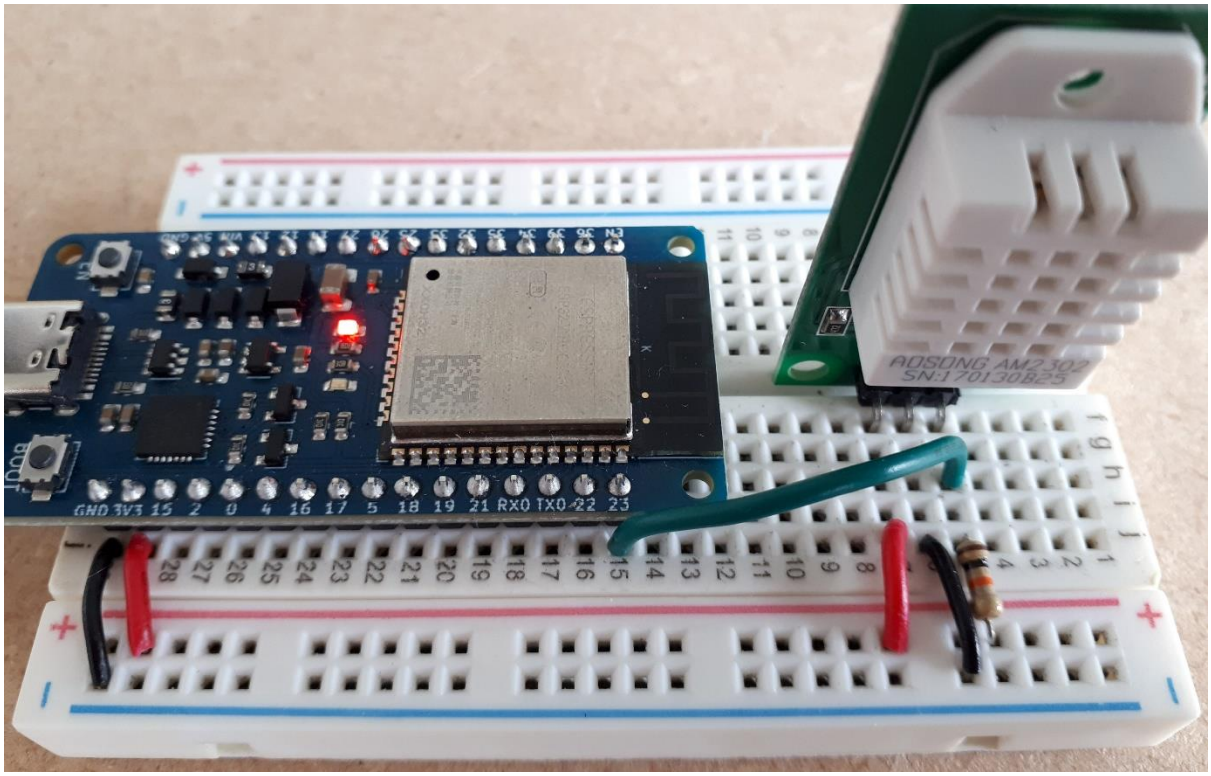


Pin Wiring

- 3 pins (bis)
- 4 pins (DHT22 Only)

Warning

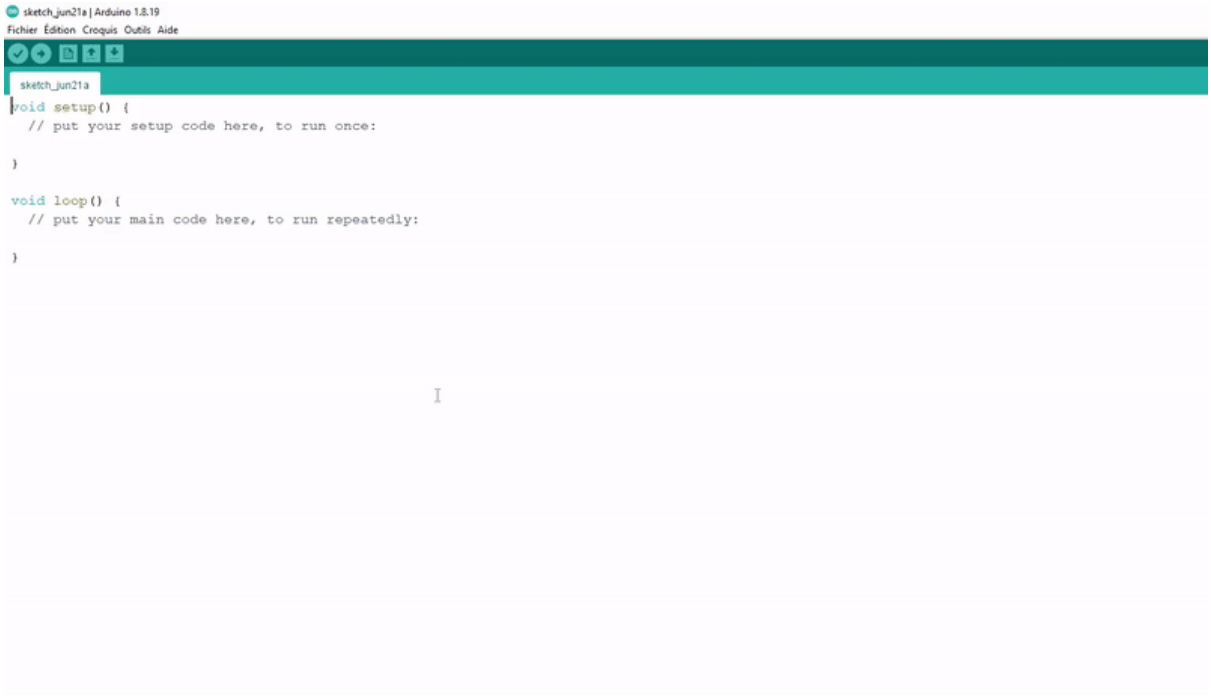
Supply the module with 3.3V and not 5V to have a 3.3V data signal.



Wiring the DHT22 with a uPesy ESP32 Wroom DevKit board

Measuring the temperature and humidity on the DHT22 module with Arduino code

It would help if you used a library to communicate with the DHT22, which uses a proprietary protocol. I recommend the dHT Sensor library from Adafruit , which is compatible with ESP32.



DHT22 library installation on Arduino IDE

Once installed, the following code can retrieve temperature and humidity from the DHT22.

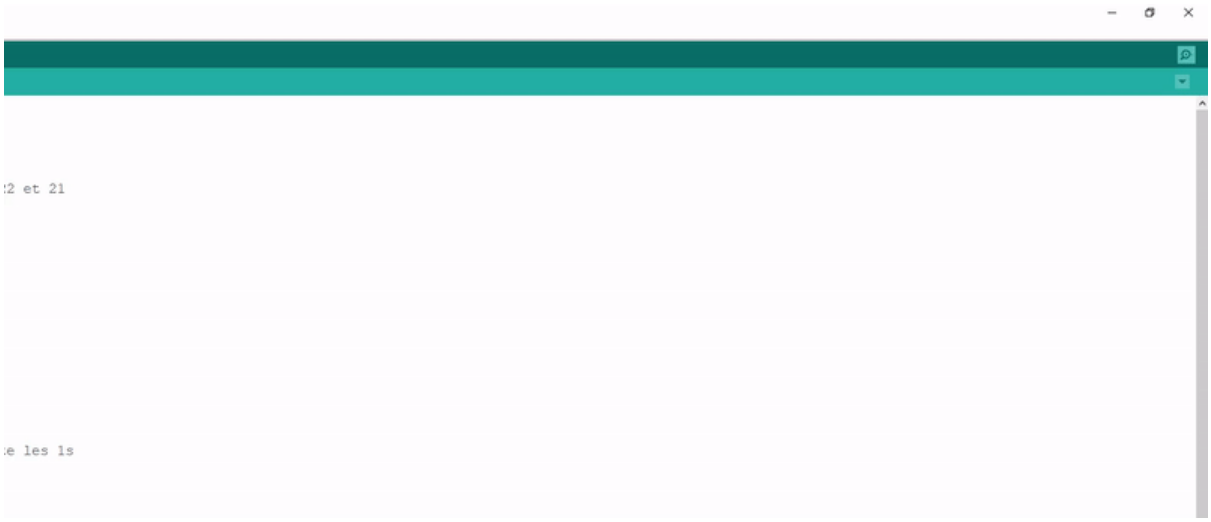
```
#include "DHT.h"

#define DHTPIN 23
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);

void setup() {
  Serial.begin(115200);
  dht.begin();
}

void loop() {
  delay(2000);
  // The DHT22 returns at most one measurement every 2s
  float h = dht.readHumidity();
  // Reads the humidity in %
  float t = dht.readTemperature();
  // Reads the temperature in degrees Celsius
  float f = dht.readTemperature(true);
  // true returns the temperature in Fahrenheit

  if (isnan(h) || isnan(t) || isnan(f)) {
    Serial.println(F("Failed reception"));
    return;
    // Returns an error if the ESP32 does not receive any measurements
  }

  Serial.print("Humidite: ");
  Serial.print(h);
  Serial.print("%  Temperature: ");
  Serial.print(t);
```

```
  Serial.print("°C, ");
  Serial.print(f);
  Serial.println("°F");
  // Transmits the received measurements to the serial terminal via USB
}
```

Note

The `DHT.h` at the beginning of the program includes the library **DHT Sensor** .

This is what you get in the console:



Note

You can blow on the DHT22 sensor like a frost-covered window. You should see a rapid increase in humidity.

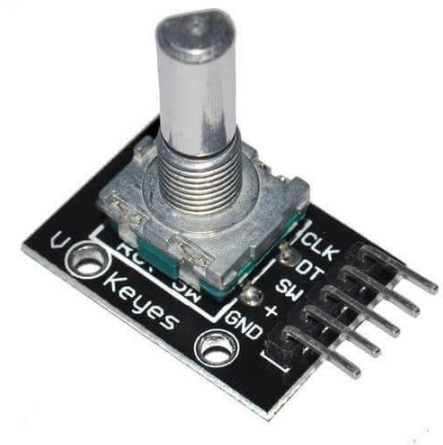
Calculate the heat index | Feel temperature

To calculate this felt temperature, we can use the heat index . This is a value in °C based on the air's temperature and relative humidity. The higher the humidity in the air at a given temperature, the higher the temperature felt. The DHT Sensor library provides a built-in function to calculate this, called `computeHeatIndex()` .

```
#include "DHT.h"

#define DHTPIN 23
#define DHTTYPE DHT22

DHT dht(DHTPIN, DHTTYPE);
```

Using a rotary encoder with Arduino Code with an ESP32

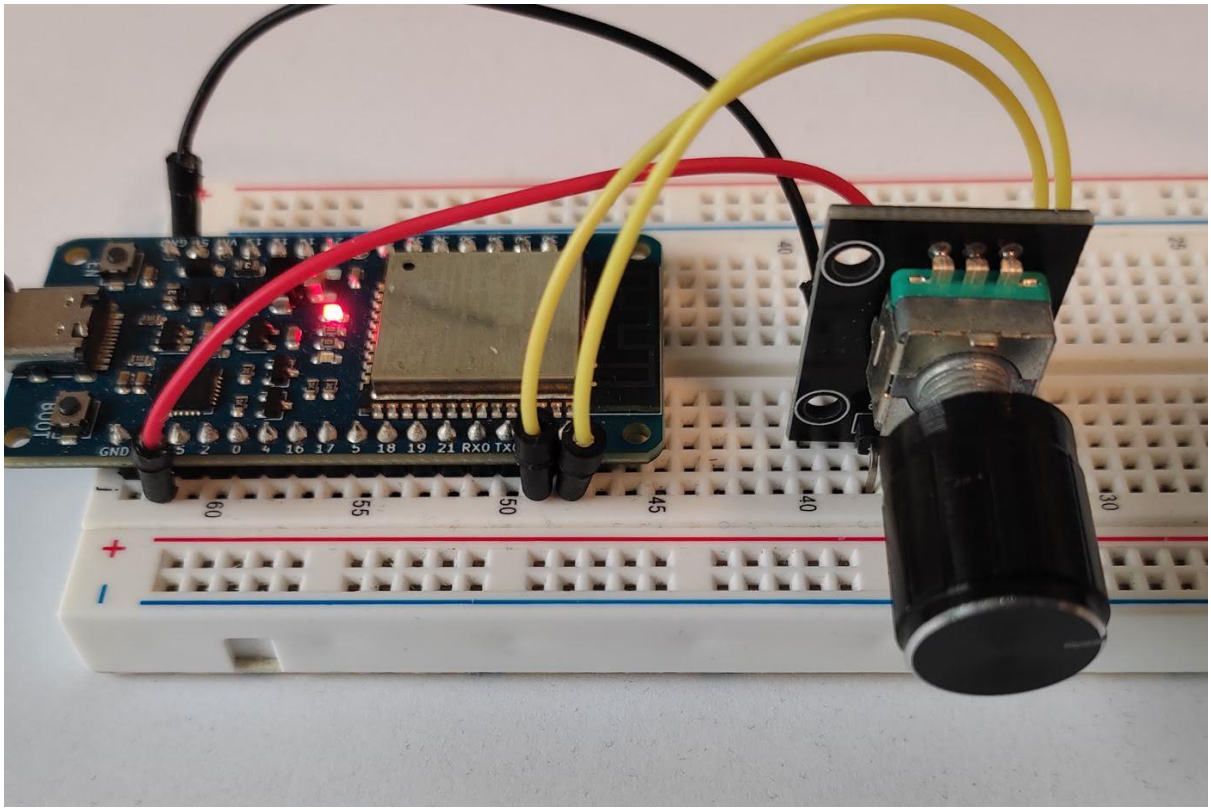
(Updated at 01/20/2023)



Rotary encoders are position sensors that measure an axis's angular position (or rotation). They're mostly used in motors for PID control and user interfaces as an alternative to potentiometers. Furthermore, many encoders used in DIY kits have a push button built-in.

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

Getting started with a rotary encoder: the KY-040



This tutorial will explore utilizing a **KY-040 rotary encoder** module, a common component in DIY kits. We can use this module to create a user interface for our program; for example, to select options from a menu or change the value of a variable.

Difference between rotary encoder and potentiometer

At first, the rotary encoder `KY-040` may look like a potentiometer. However, a potentiometer is an analog sensor while the rotary encoder is digital. A potentiometer changes the value of a resistor, but its range is limited.

A rotary encoder can detect a specific number of “steps” for each revolution and sends a signal at each step. It also can turn endlessly. You can physically feel the difference between the two; a potentiometer rotates smoothly, while a rotary encoder turns jerkily.

A rotary encoder is a device that can be easily used with a microcontroller, as it sends digital signals. The precision of the encoder is based on the number of steps per rotation. At the same time, the accuracy of a potentiometer is determined by the resolution of the ADC that is used to measure its position.

Note

A potentiometer is a device utilized to modify analog settings, such as the volume of an amplifier. In contrast, a rotary encoder is employed to get an angular direction and position accurately.

How works a rotary encoder?

This guide provides an overview of rotary encoders but does not dive into technical details.🤓 To gain a deeper understanding of the physical processes, different technologies, and classic topologies (such as the operation of an optical quadrature phase rotary encoder 😨), please refer to the theoretical presentation of a rotary encoder.

Connections of the KY-040 rotary encoder to the ESP32

This rotary encoder has 2 signals to know the position: `CLK` and `DT` . The pin `SW` is connected to the integrated push button (SWitch).

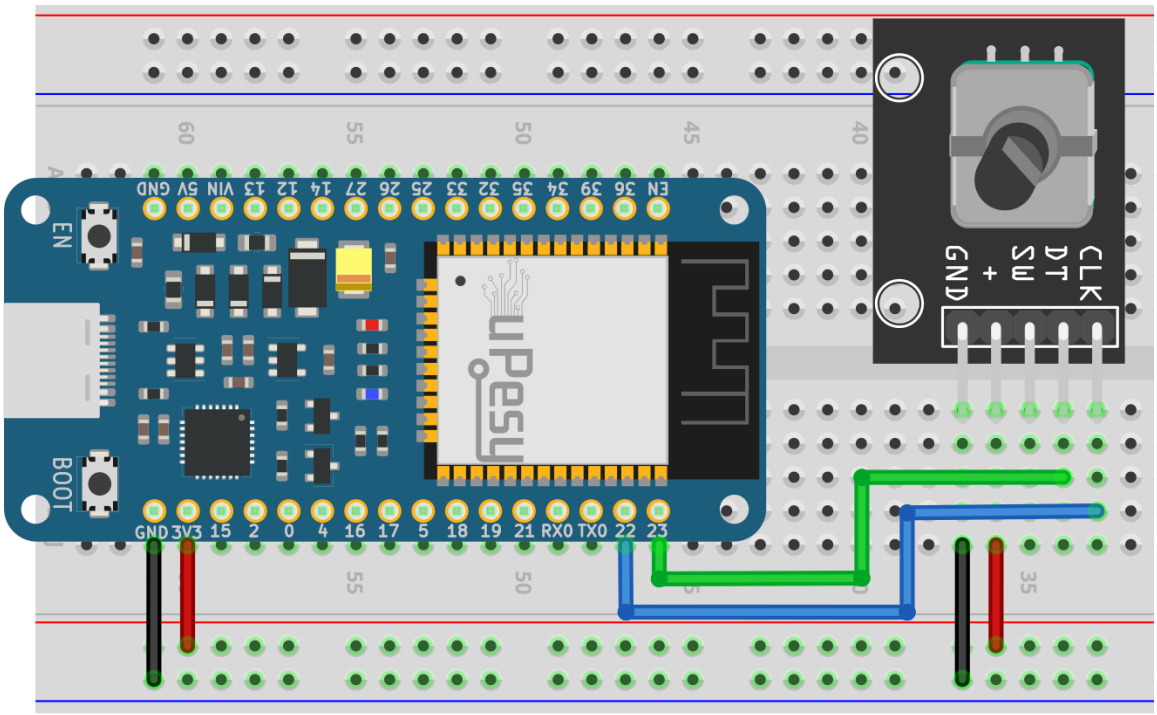
Warning

If you use a raw rotary encoder, not a breakout module, you must add pull-up resistors to the 3 logic pins. This is because these resistors must distinguish between the different logic levels as with the standard push-button circuit.

Here is a proposal for a connection to an ESP32 board :

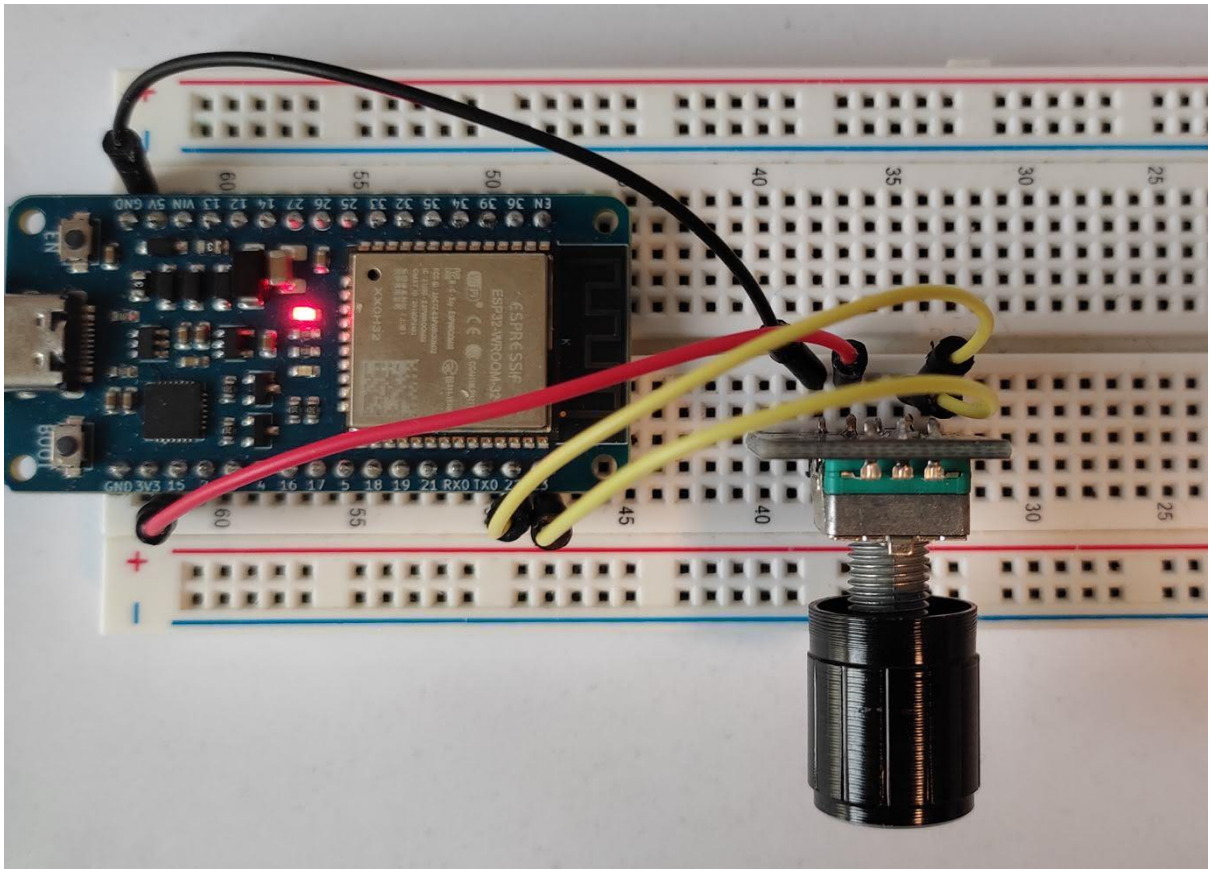
Rotary Encoder	ESP32
CLK	GPIO22
DT	GPIO23
SW	GPIO21
+	3V3
GND	GND

The circuit to be done is as follows:



Electronic circuit to be replicated

And here is an example of breadboard mounting:



Get the angular position of the KY-040 rotary encoder in Arduino code

Incremental encoders are used to measure the angular position in degrees based on the number of increments made by the user. Instead of providing an exact angular position, this type of encoder returns the number of increments done. To get the angle, the value of the increment must be determined in the program.

Finding the amount of steps an encoder has made is possible through various methods. It is best to use a reliable way to get steps without blocking the program. A basic approach where a loop looks for logical signals received from the encoder carries the risk of missing steps if the encoder is rotated fast.

Note

Programs can sometimes struggle with tasks and count in the wrong direction. (A traditional bug)

Many code sources are available online if you want to use rotary encoders. However, not all of them are the same! The best choice is to trigger hardware interrupts when a logical level change is detected. These interrupts are independent of the CPU and won’t stop the code from running.

Note

The ESP32 can attach an interrupt to any output pins, which is impossible when using an Arduino, as only a few pins can have interrupts.

Some Arduino libraries compatible with the ESP32

It's often better to use libraries rather than write your code. Two libraries that are well-suited for ESP32 are:

- The AiEsp32RotaryEncoder library is an excellent choice for creating user interfaces, like menus, or for adjusting parameters. It utilizes hardware interrupts for incrementing calculations and is specifically adapted for rotary encoders with integrated push buttons, such as the KY-040 encoder.
- The ESP32Encoder library uses the PCNT peripheral on the ESP32 to perform counting operations. The advantage of this approach is that it doesn't require any CPU processing, making it very performant. However, its functionality is limited compared to other libraries; there are no interrupts for each increment and no push button functionality. It is recommended to use this library when measuring encoder values from a motor, where there will be hundreds of increments per second. The current increment can be read easily at any time with the simple function provided by the library.

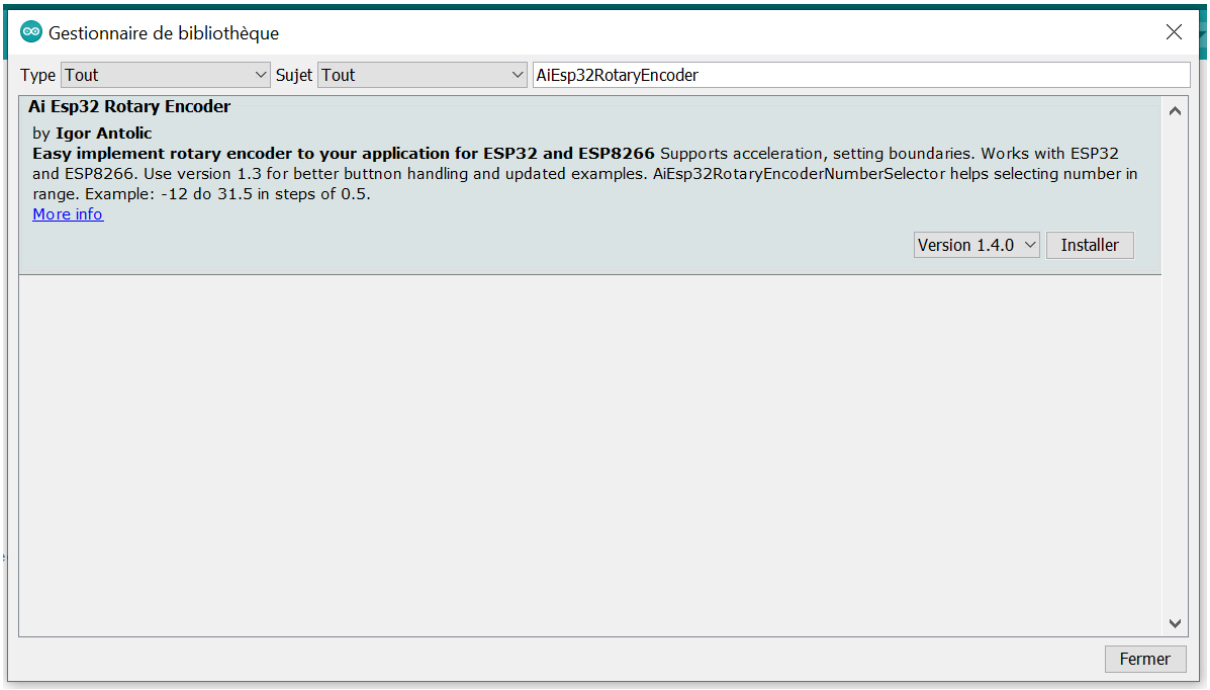
  Note

  The ESP32's PCNT hardware controls up to 8 rotary encoders simultaneously.

The 2 libraries are easily installed from the Arduino IDE :

Use a rotary encoder for a user interface with the library `AiEsp32RotaryEncoder`

The library is installed from the library manager in the Arduino IDE.

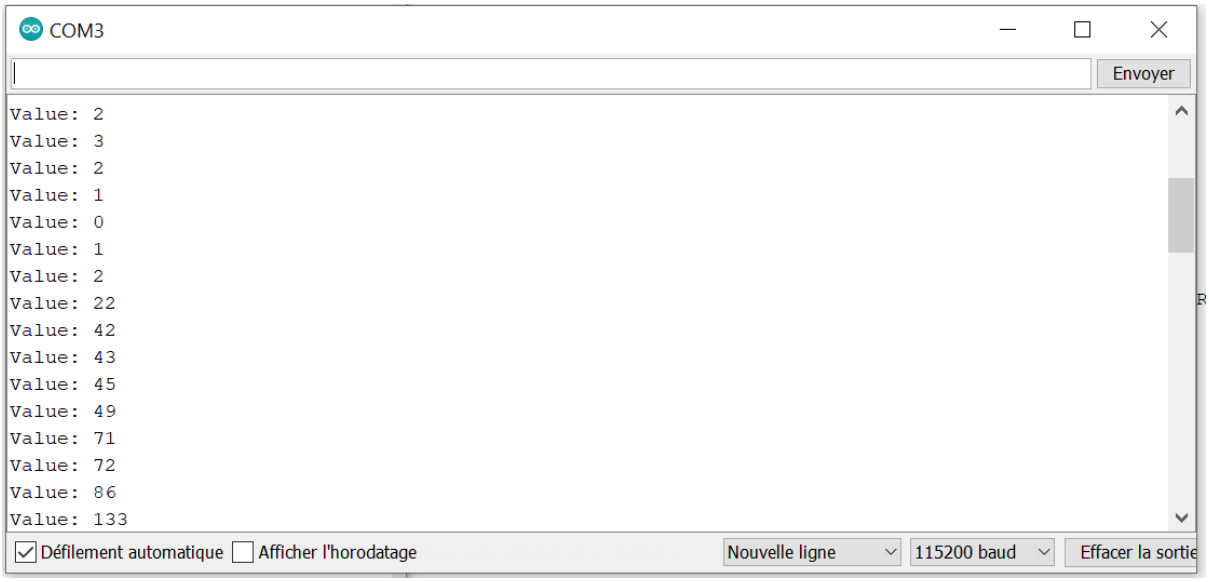


Here is a skeleton code to show the possibilities of this library:

```
#include "AiEsp32RotaryEncoder.h"
#include "Arduino.h"

#define ROTARY_ENCODER_A_PIN 23
#define ROTARY_ENCODER_B_PIN 22
#define ROTARY_ENCODER_BUTTON_PIN 21
#define ROTARY_ENCODER_VCC_PIN -1
#define ROTARY_ENCODER_STEPS 4

//instead of changing here, rather change numbers above
AiEsp32RotaryEncoder rotaryEncoder =
AiEsp32RotaryEncoder(ROTARY_ENCODER_A_PIN, ROTARY_ENCODER_B_PIN,
ROTARY_ENCODER_BUTTON_PIN, ROTARY_ENCODER_VCC_PIN, ROTARY_ENCODER_STEPS);

void rotary_onButtonClick()
{
    static unsigned long lastTimePressed = 0; // Soft debouncing
    if (millis() - lastTimePressed < 500)
    {
            return;
    }
    lastTimePressed = millis();
    Serial.print("button pressed ");
    Serial.print(millis());
    Serial.println(" milliseconds after restart");
}

void rotary_loop()
{
    //dont print anything unless value changed
    if (rotaryEncoder.encoderChanged())
    {
            Serial.print("Value: ");
            Serial.println(rotaryEncoder.readEncoder());
    }
```

```
    if (rotaryEncoder.isEncoderButtonClicked())
    {
            rotary_onButtonClick();
    }
}

void IRAM_ATTR readEncoderISR()
{
    rotaryEncoder.readEncoder_ISR();
}

void setup()
{
    Serial.begin(115200);

    //we must initialize rotary encoder
    rotaryEncoder.begin();
    rotaryEncoder.setup(readEncoderISR);
    //set boundaries and if values should cycle or not
    //in this example we will set possible values between 0 and 1000;
    bool circleValues = false;
    rotaryEncoder.setBoundaries(0, 1000, circleValues); //minValue,
maxValue, circleValues true|false (when max go to min and vice versa)

    /*Rotary acceleration introduced 25.2.2021.
   * in case range to select is huge, for example - select a value between
0 and 1000 and we want 785
   * without acceleerateion you need long time to get to that number
   * Using acceleration, faster you turn, faster will the value raise.
   * For fine tuning slow down.
   */
    //rotaryEncoder.disableAcceleration(); //acceleration is now enabled by
default - disable if you dont need it
    rotaryEncoder.setAcceleration(250); //or set the value - larger number
= more accelearation; 0 or 1 means disabled acceleration
}

void loop()
{
    //in loop call your custom function which will process rotary encoder
values
    rotary_loop();
    delay(50); //or do whatever you need to do...
}
```

This is what you get in the serial monitor when you turn the encoder:

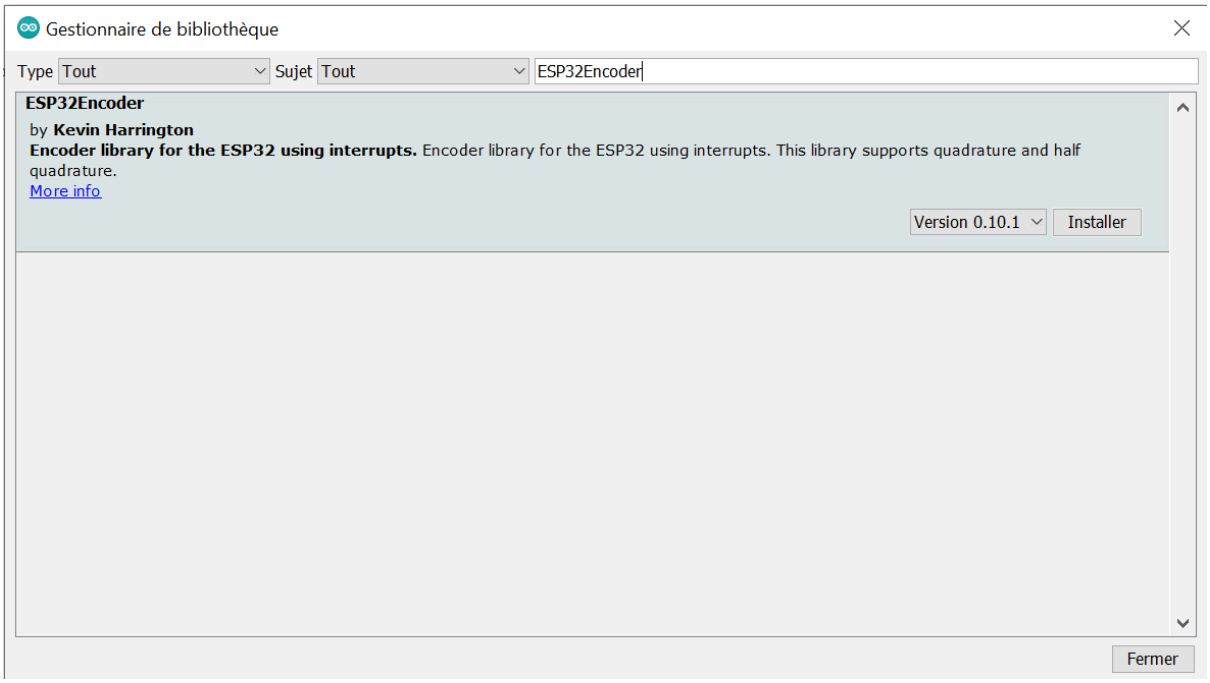


When you turn the encoder, the values don't always increase linearly. That's because the "Acceleration" mode is enabled, which increases the value quicker when you turn the encoder quickly. This is very helpful for adjusting a parameter that has huge values. If you want to turn off the acceleration mode, use the command `rotaryEncoder.disableAcceleration()` . You can also change the acceleration behavior with `rotaryEncoder.setAcceleration(0-1000)` .

This library can count in the background, even if you take the `rotary_loop()` out of the `loop()` . The code above can be used as a starting point for your project. To find out all the features this library has to offer, please visit the GitHub repository for the documentation.

Get the angular position of a rotary encoder using the ESP32 hardware counter (PCNT) with the lib `ESP32Encoder`

The library `ESP32Encoder` is installed directly from the Arduino IDE :



The code is much simpler than the previous library, but at the same time, you can only get the value of the encoder increment.

```
#include <ESP32Encoder.h>

#define CLK 22 // CLK ENCODER
#define DT 23 // DT ENCODER

ESP32Encoder encoder;

void setup () {
  encoder.attachHalfQuad(DT, CLK);
  encoder.setCount(0);
  Serial.begin ( 115200 );
}

void loop () {
  long newPosition = encoder.getCount();
  Serial.println(newPosition);
  delay(25);
}
```

The count will still be accurate even when a 500 ms interval is set in the `loop()` function since the counting is achieved by the PCNT hardware device. We can also get the value incrementing gradually through the serial terminal. `getCount()` is the function that retrieves the value from the hardware counter register, the one doing the counting.

Note

If you've missed some steps, you can use the `setFilter()` function to filter the signal input at the hardware counter (PCNT) as much as possible.

If you’re new to the library, you can find the raw documentation to learn about its functions.

Using a servomotor with an ESP32 in Arduino code

(Updated at 01/04/2023)

Servo motors are a special form of motor that can be fixed at a precise angular position and held until a new instruction is given. TowerPro’s popular SG90 blue servo has a 1.5 kg/cm torque for only 9g. Its low cost and ease of control from an ESP32 make it a trendy choice for makers!

Note

Servos are very commonly used in RC cars, RC planes, robotics and the industry.

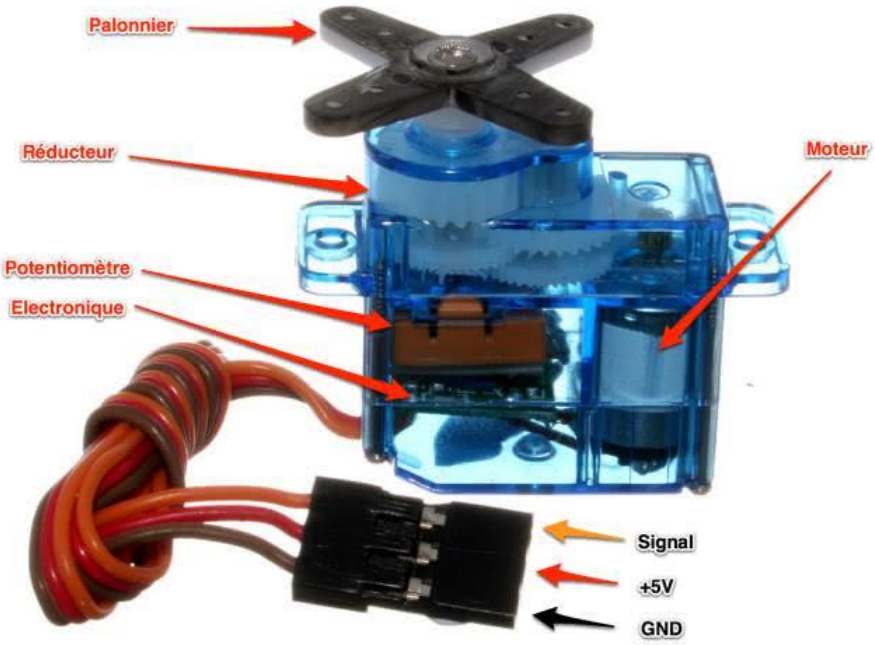
Getting started with the SG90 Servo

Unlike a conventional motor, most basic servo motors can only rotate between 0 and + 180 degrees. They consist of a DC motor, various gears to increase torque (and reduce speed) and a servo system, all packed into a tiny housing.

Warning

The plastic stop of the actuator, which limits the rotation, is relatively fragile. I advise you not to manually turn the axis of the servomotor to avoid forcing the stop.

How a servomotor works



This actuator can be controlled using a 50 Hz pulse width modulated (PWM) signal, which produces a pulse every 20ms. The position of the actuator can be adjusted by changing the pulse duration between 1 ms and 2 ms.

Note

The servo’s position can only be controlled with a PWM signal!

How to power the actuator with an ESP32

According to the datasheet of the SG90 servo, the optimal voltage is 5V. However, it also works with 3.3V (when idle).

Note

I advise you to use 5V (even if the example is with 3.3V) to have a more reactive servo!

Servo motors consume much power, especially when subjected to high torque. Since the pin `5V` of most ESP32 boards is powered directly by the USB bus, the computer will limit it to around 500mA. With 1 or 2 servo motors connected, the USB supply should hold the load.

Once you exceed two servo motors, it is best to use a separate power supply. Be sure to connect a pin `GND` from the board to the negative terminal of the actuator power supply.

Warning

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

Connecting the SG90 Servo to an ESP32

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

Pin Wiring

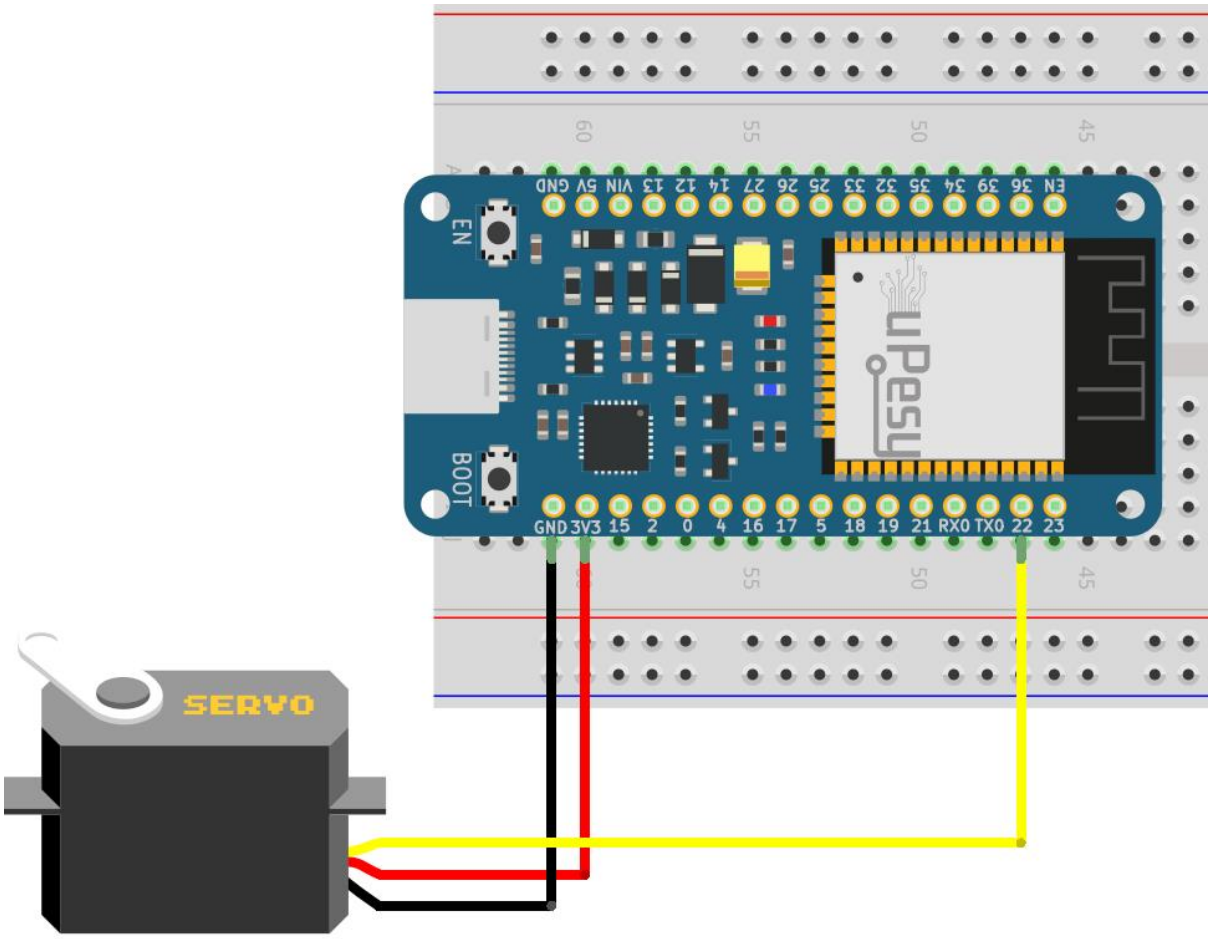
ESP32	**Wire Color**	**Servomotor SG90**
`GND`	Maroon	`GND`
`5V` ou `3V3`	Red	`5V`
`GPIO22`	Orange	Signal `PWM`

Note

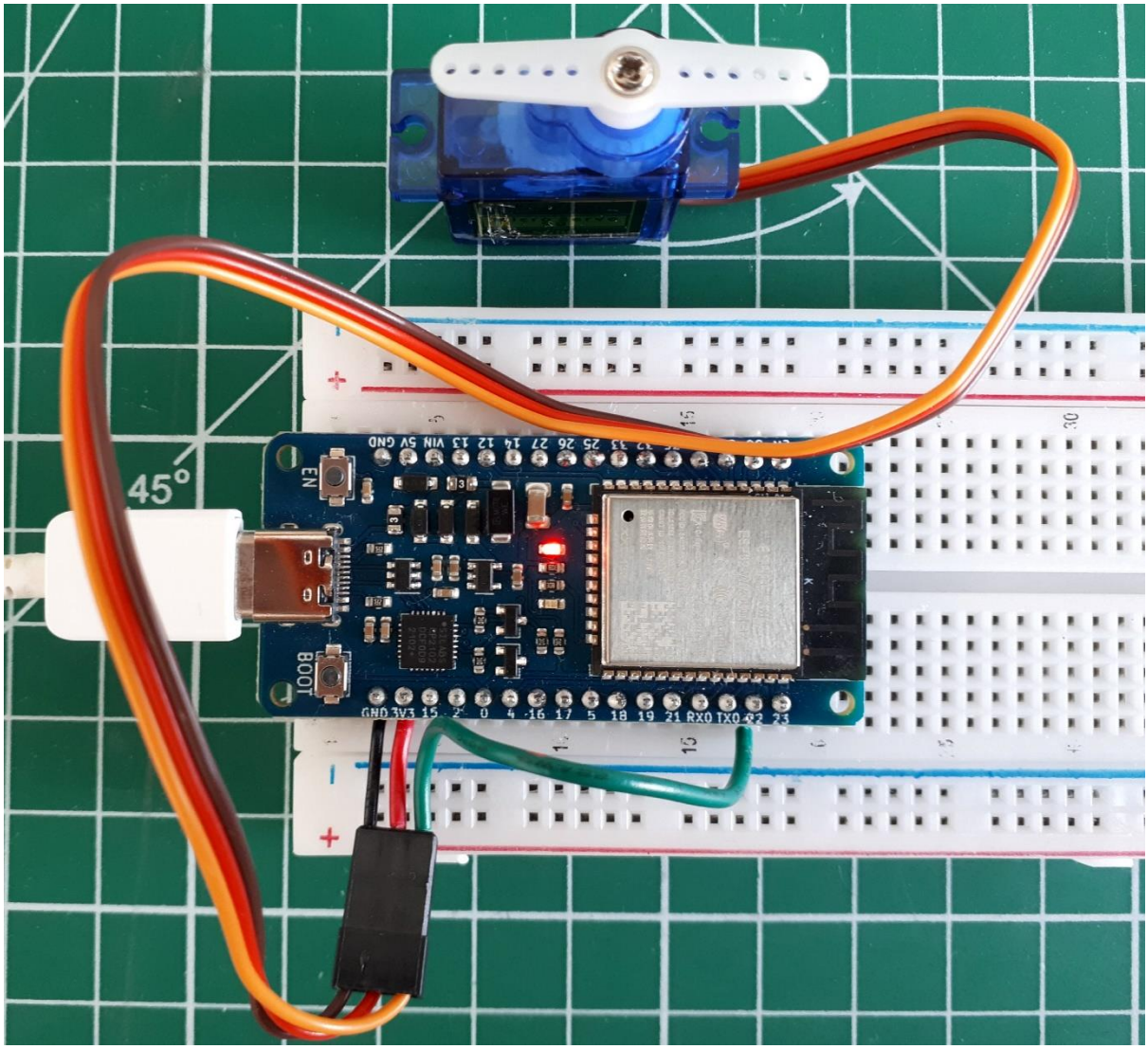
On some actuator versions, the color of the wire `PWM` is yellow or white instead of orange.

All output pins on the ESP32 can be used to control the servo motor. You can have 16 outputs `PWM` completely independent on the ESP32, which is the control of 16 servo motors! That should do it. 😊

Circuit for driving a servomotor with an ESP32



Here is an example on a breadboard with the uPesy ESP32 Wroom DevKit board, where the servo is powered with 3.3V.



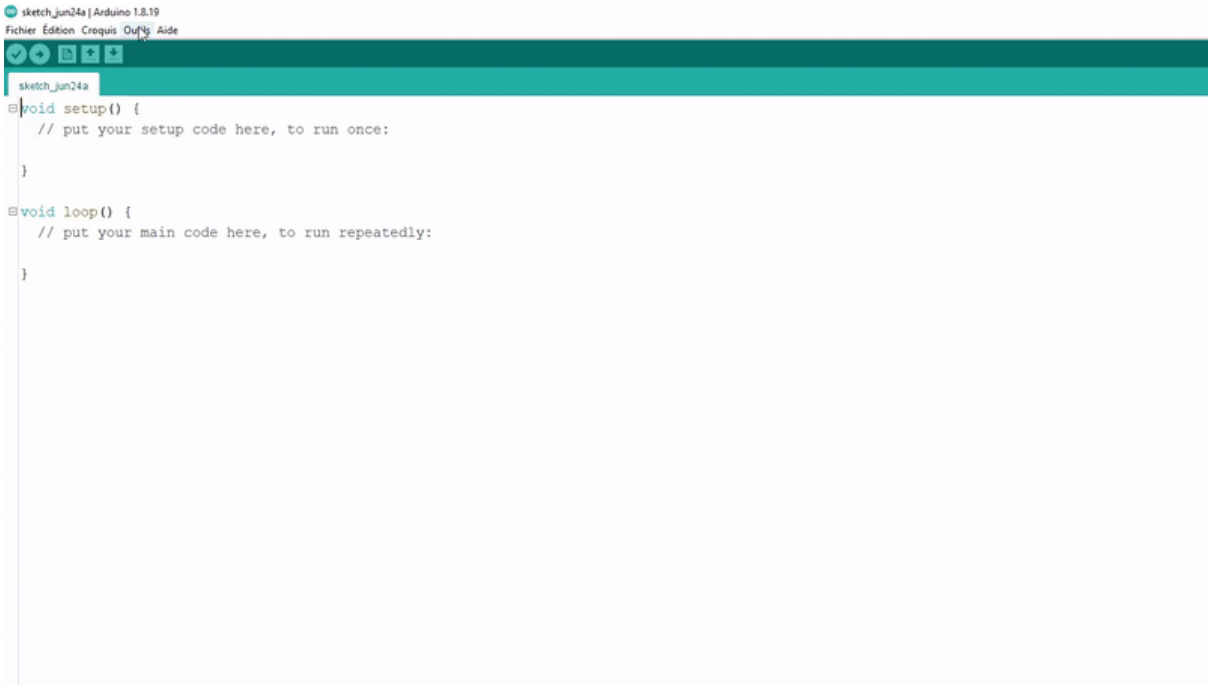
Drive a servo motor from the ESP32 with Arduino code

We could dispense with using libraries to drive servomotors, but why reinvent the wheel when there are well-made and widely used libraries?

I propose you two different libraries which work correctly on the ESP32 :

- ESP32Servo library , which works exactly as the `Servo` for an Arduino board. Use this one if you already have a lot of Arduino code that you would like to reuse as is with an ESP32 board.
- ESP32-ESP32S2-AnalogWrite is a library that allows using all the features offered by the ESP32 boards at the PWM level, with practical support to drive servos. It also works with more recent ESP32 models (ESP32S2, ESP32C3…)

They are both available from the installer integrated into the Arduino IDE (*Tools → Manage libraries*). Here is the procedure for the library `ESP32Servo` :



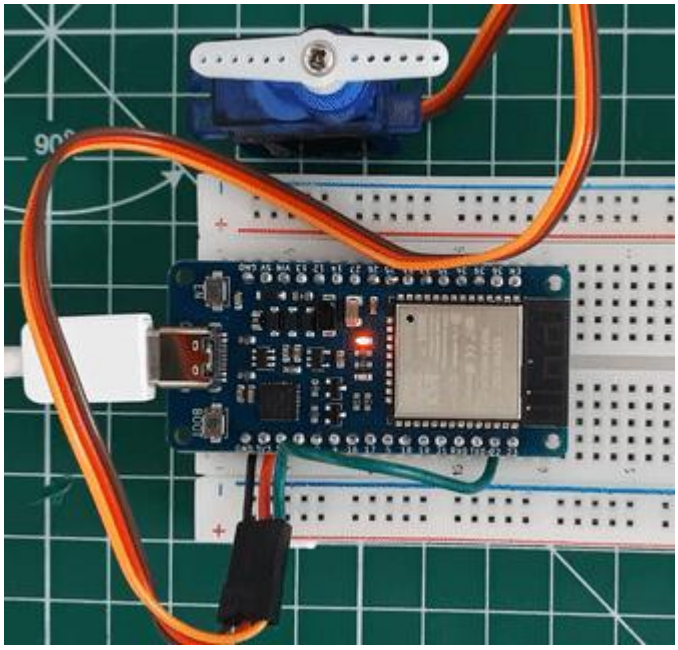
Installation of the ESP32Servo library from the Arduino IDE

Drive the servo with the library `ESP32Servo` in Arduino code

Once the library is installed, we use the following code to drive the SG90 from 0° to 180° with the ESP32.

```
#include <ESP32Servo.h>
#define PIN_SG90 22 // Output pin used

Servo sg90;

void setup() {
  sg90.setPeriodHertz(50); // PWM frequency for SG90
  sg90.attach(PIN_SG90, 500, 2400); // Minimum and maximum pulse width (in
us) to go from 0° to 180
 }
void loop() {
 //rotation from 0 to 180°
  for (int pos = 0; pos <= 180; pos += 1) {
    sg90.write(pos);
    delay(10);
  }
 // Rotation from 180° to 0
  for (int pos = 180; pos >= 0; pos -= 1) {
    sg90.write(pos);
    delay(10);
  }
}
```

Here is a demonstration video with the above code:



Control of the SG90 servo with the ESP32

Control the servo with the Arduino library `ESP32-ESP32S2-AnalogWrite`

The code is very similar to the other library.

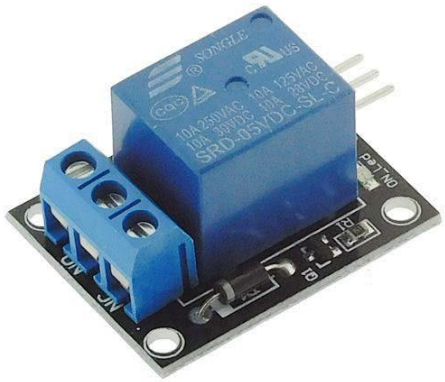
Warning

Even if the library is called `ESP32-ESP32S2-AnalogWrite` you need to import the header `pwmWrite.h` .

```
#include <pwmWrite.h>
#define PIN_SG90 22 // Output pin used

Pwm pwm = Pwm();

void setup() {
}

void loop() {
//rotation from 0 to 180°
  for (int pos = 0; pos <= 180; pos++) {  // go from 0-180 degrees
    pwm.writeServo(PIN_SG90, pos);
    delay(10);
  }
// Rotation from 180° to 0
  for (int pos = 180; pos >= 0; pos--) {
    pwm.writeServo(PIN_SG90, pos);
    delay(10);
  }
}
```

Control electrical devices with a Relay and the ESP32 with Arduino code

(Updated at 01/09/2023)



If you want to do home automation, the relay is the essential module. Indeed, coupled with an ESP32, you can directly control household appliances on your phone, your computer, or anywhere in the world from the Internet if you wish.

Warning

Handling line voltages are dangerous. If you have to use them, it must be done with the utmost caution. For permanent use, I strongly advise you to use good quality professional relay modules and to have your installation checked by a professional before any use.

Getting started with a Relay

A relay is very easy to use, primarily when they are sold in the form of a module that contains an elementary circuit to use it.

Note

I strongly encourage you to take a ready-made module that integrates the relay with its minimal circuit rather than using 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 put on it anyway).

Presentation and functioning of a Relay

Suppose you have never used a relay before or are curious to understand how it works, its use cases and its limitations. In that case, you should consult a:ref:` more theoretical article on how a relay works <8ae9b35e6bf743569833746474be4657>` .

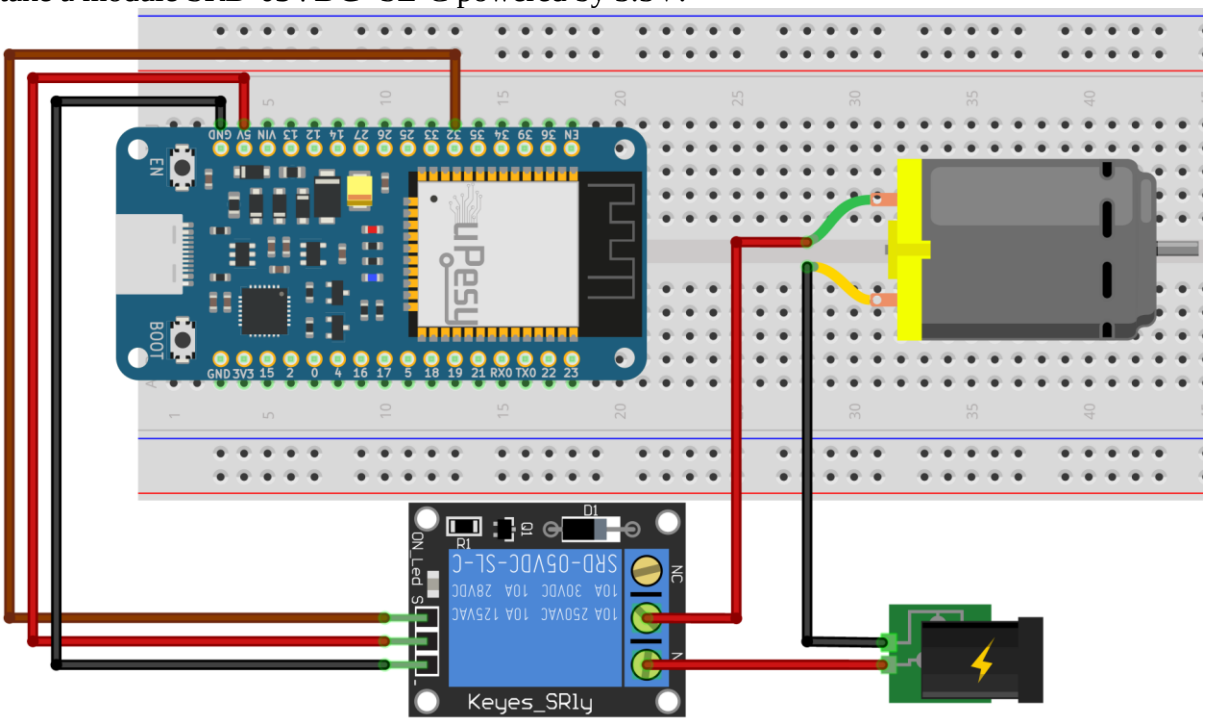
Very briefly, it's a mechanical switch that you control via the ESP32 to turn on/off a strong DC electrical circuit (RGB led strips, pumps) or a device connected to the 220V mains (fans, lights, heater, motor…).

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

In most Arduino kits, the modules use the **SRD-05VDC-SL-C** from the manufacturer **Songle** . Generally, they come in 2 categories: a module with a single relay and another with several relays simultaneously. So if you plan to drive several power circuits separately in one project, it is more interesting to take a multi-relay module.

Note

This relay is made to be powered in 5V: SRD-**05VDC** -SL-C. If you use an ESP32 on a LiPo battery, the available voltage will be between 3.3V and 4.2V. In this case, it would be wise to take a module SRD-**03VDC** -SL-C powered by 3.3V.



Electrical circuit to be made

Warning

Contrary to the diagram with a transistor, the ground of the logic part and the power part is completely separated. You must not connect them, especially if it is 220V AC or there is no ground!

Pin Wiring

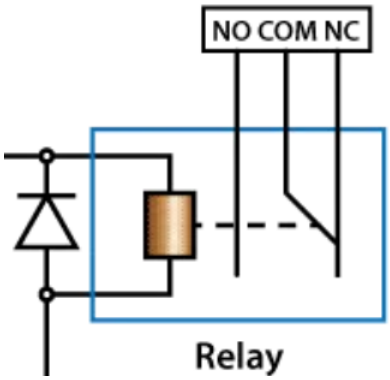
Relay Module	ESP32
S	32
VCC	5V
GND	GND

On the model **SRD-05VDC-SL-C,** there are three pins to drive the relay:

- The pin `S` , on the left, is connected to a pin of the ESP32 (here `GPIO32`). It allows sending a signal to drive the relay.
- The middle pin is the power supply connected to the 5V of the ESP32 (3V3 if it is the SRD- module **03VDC** -SL-C).
- The last pin, represented by `-` , is the ground connected to a pin `GND` .

The relay has a 3-pin terminal block on 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

The device in the circuit will be connected to the terminal `COM` and `NO` or `NC` depending on your application. By choosing `NO` , the relay will be open by default (the electrical circuit will not be closed). The circuit will be closed only when a signal is sent to the relay.

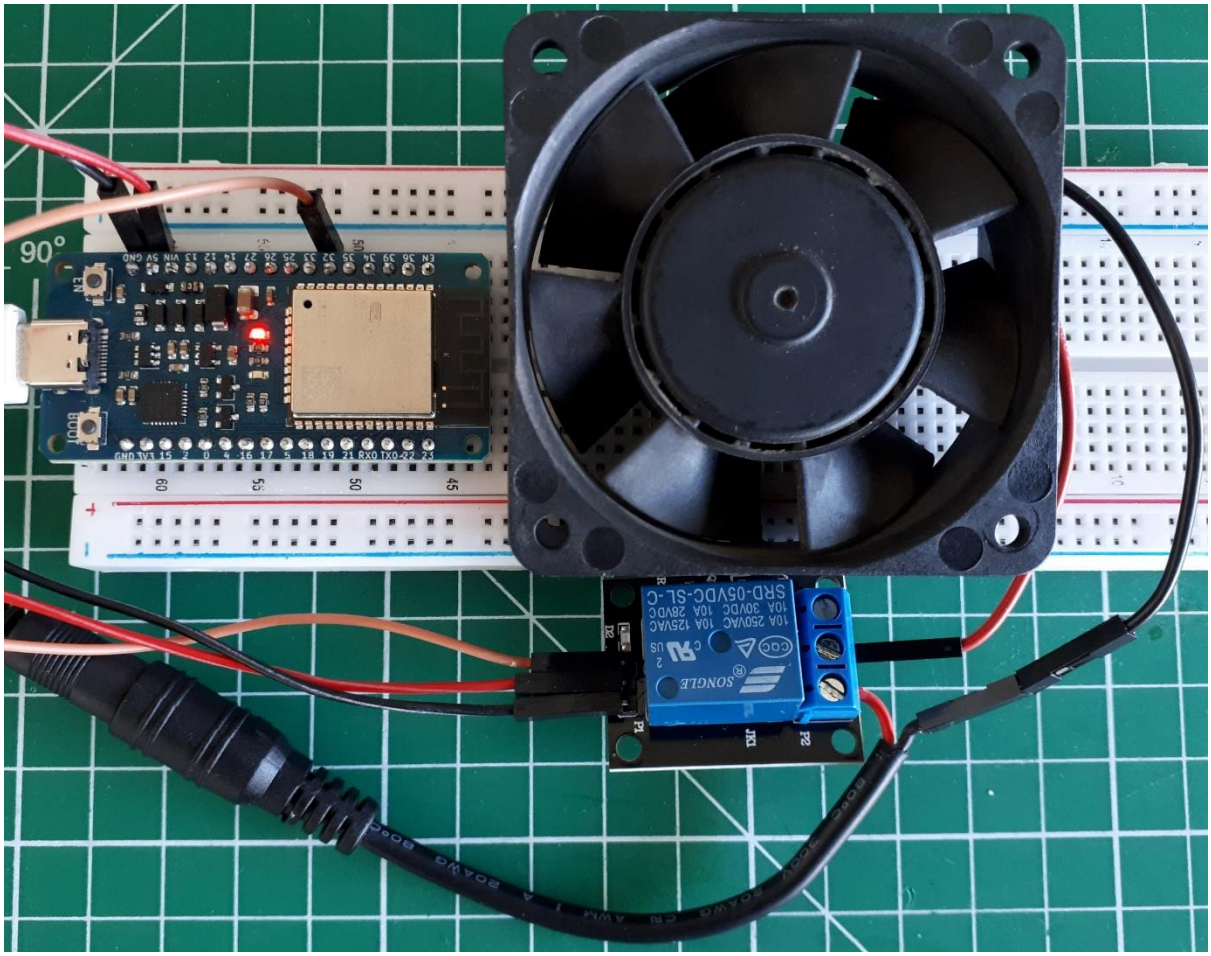
On the other hand, by choosing the terminals `COM` and `NC` , the circuit will be closed by default (when the relay is not activated): the device is switched on. When the relay is activated, the circuit will open, and the connected device will no longer be powered.

Choosing the least dangerous mode is safer in case of a control failure (or if the relay is out of service). For example, if it is to supply a heating resistor, it is better to have the circuit open if the relay does not work correctly anymore.

Note

I recommend you to use the combination `COM` and `NO` if it is to turn on a device (lamp, motor, pump).

Here is an example of a relay that turns on a large computer fan:



The NO and COM terminals are used for this configuration

Control the Relay with Arduino code on the ESP32

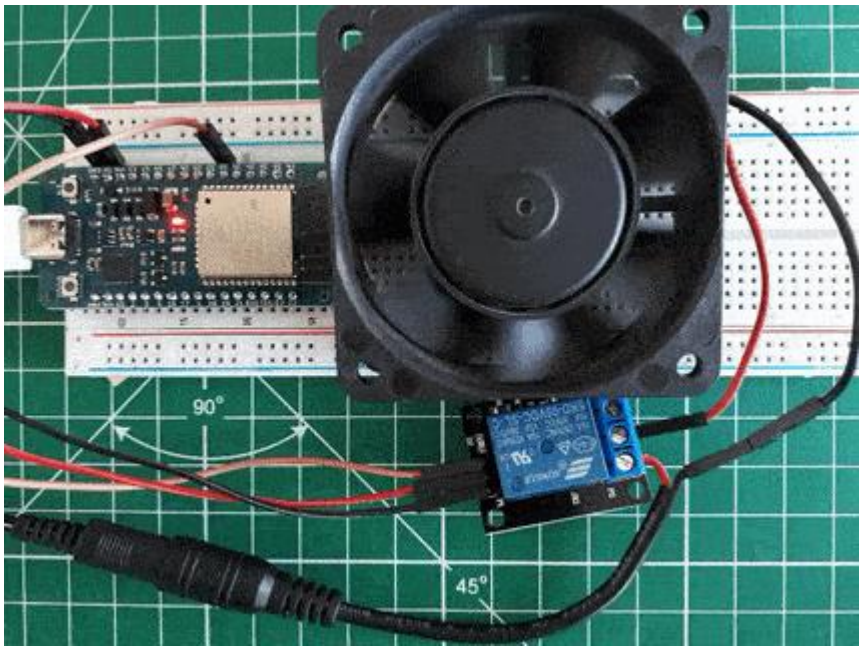
For the time being, the Arduino code is straightforward; it's like turning on and off an LED. The code is identical to the famous sketch `blink` . There is no need to install third-party libraries.

```
#define RELAIS 32

void setup() {
  pinMode(RELAIS,OUTPUT);
}

void loop() {
  digitalWrite(RELAIS, HIGH); // The circuit is closed for 200ms
  delay(200);
  digitalWrite(RELAIS,LOW); // The circuit is open for 5s
  delay(5000);
}
```

Note

The relay is controlled in all or nothing (0V or 5V) and not with a variable voltage PWM signal.

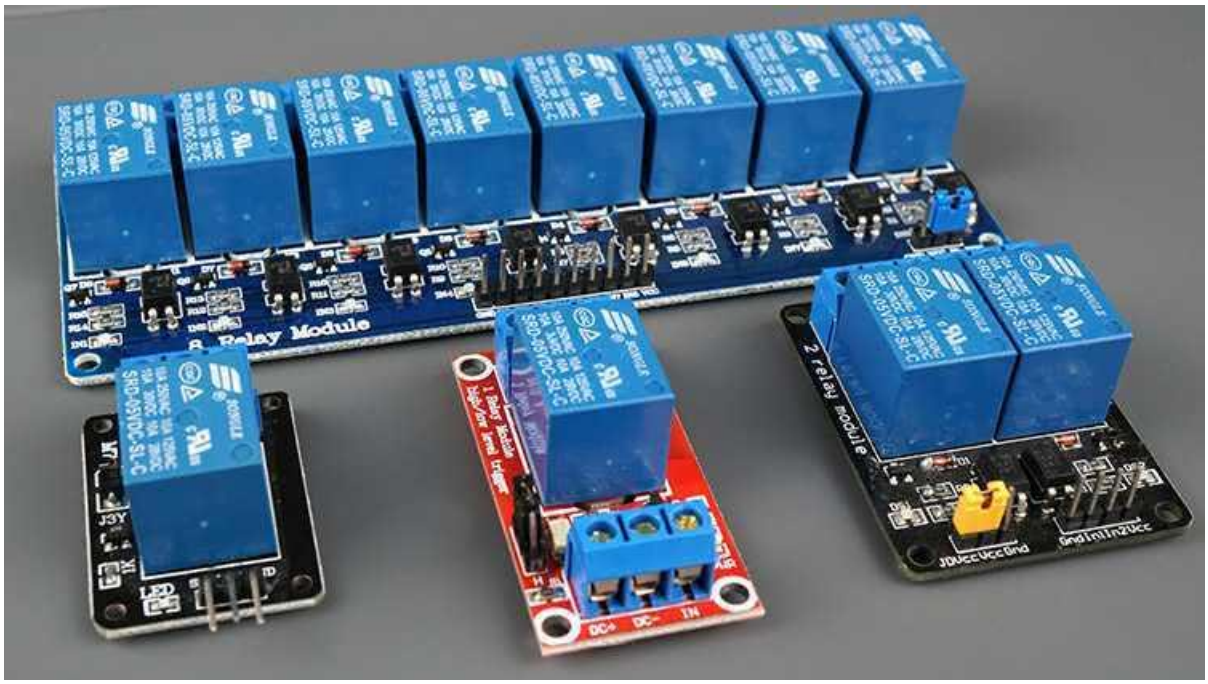
And here is what it gives with the proposed circuit:



Note

A transistor circuit would be more appropriate to switch a motor on/off very frequently.

Des modules relais avancés multicanaux



There are a variety of relay modules available, ranging from 1 to 8 channels. For a home automation system, this is very convenient. Each of them can control a corresponding number of outputs. They usually use the same blue SRD-05VDC-SL-C relay.

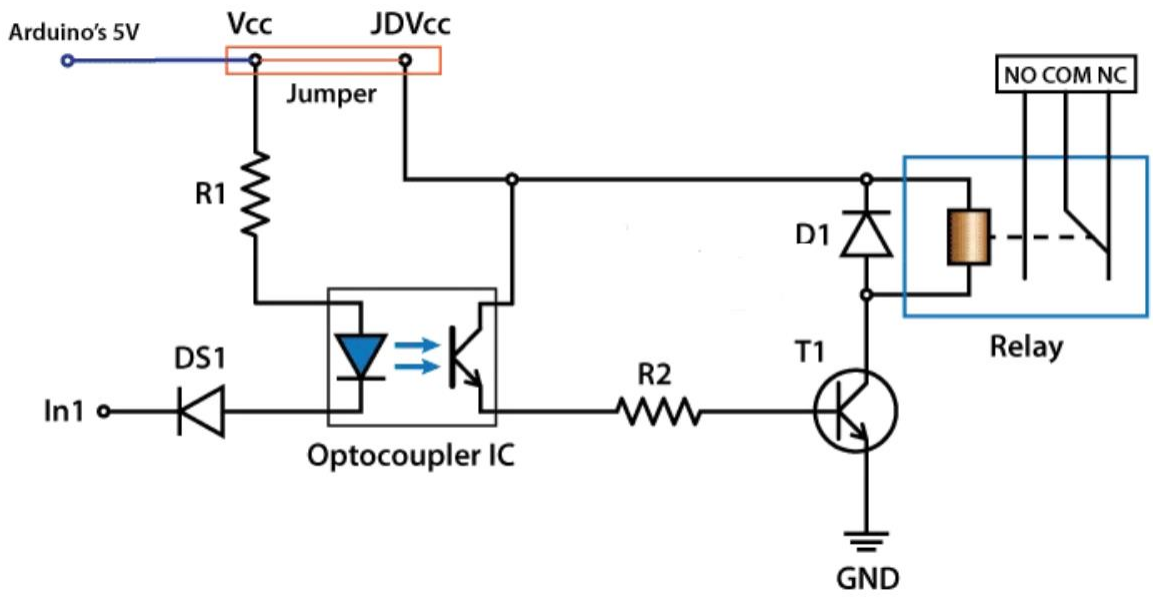
Separate the ESP32 from the Relay with the Optocouplers

There is usually a built-in optocoupler on models with fewer channels to isolate the relay from your ESP32.

Note

An optocoupler completely isolates a logic signal with a light-sensitive transmitter and receiver.

For this protection layer to be beneficial, the relay circuit must be **powered by an external 5V voltage source** on the pin `JD-VCC` . You will notice that on these models, there are usually additional pins with a jumper `JD-VCC` , `VCC` and `GND` .



By leaving the jumper, the optocoupler is no longer helpful.😕 It can be useful to use an external power supply, not coming from the ESP32, if the relays consume too much current to power the electromagnet (if there is an 8 on a module, for example).

Note

In practice, the jumper is set to the correct configuration by default; you can leave it like that.

Use multi-channel Relay modules

The operation is the same as with a single module. There are just several pins, `VIN` `VIN1` , `VIN2` , and `VIN3` … to control the relays independently.

```
#define RELAIS_0 32
#define RELAIS_1 33

void setup() {
  pinMode(RELAIS_0 ,OUTPUT);
    pinMode(RELAIS_1 ,OUTPUT);
}

void loop() {
  digitalWrite(RELAIS_0 , HIGH); // Relay 0 is activated for 5s
  digitalWrite(RELAIS_1 , HIGH); // Relay 1 is activated for 5s
  delay(5000);
  digitalWrite(RELAIS_0 ,LOW); // Relay 0 is deactivated for 5s
  digitalWrite(RELAIS_1 ,LOW); // Relay 1 is deactivated for 5s
  delay(5000);
}
```

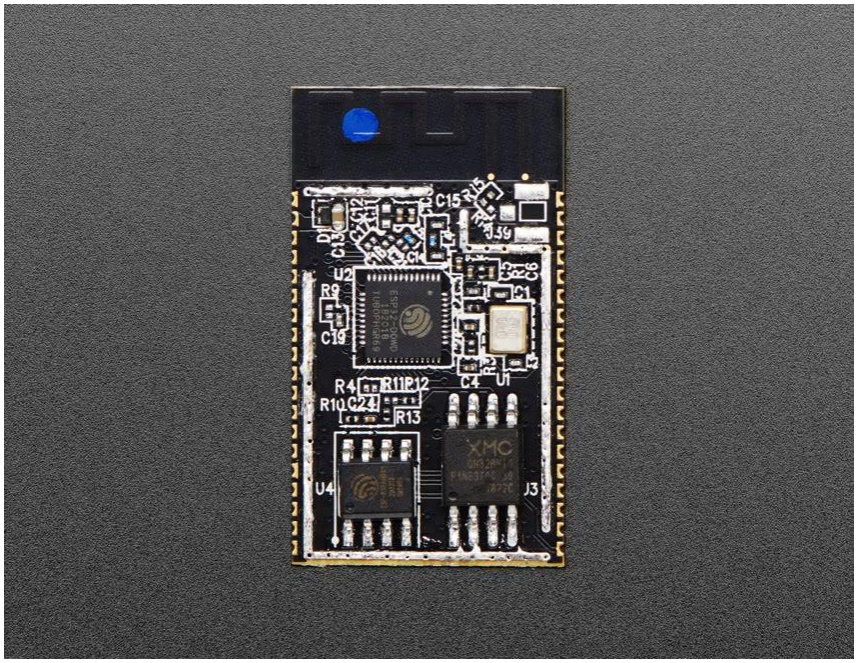
Overview and usefulness

Since the internal RAM of microcontrollers is relatively low, one can add extra external RAM to have even more. Even if the ESP32 is a microcontroller with a lot of RAM, it may not be enough, especially when you want to manipulate large files (JSON, HTML …)

PSRAM is an additional external 4 MB RAM that is present in some ESP32 modules:

- ESP32-WROVER-B
- ESP32-WROVER-I

The ESP32 communicates with the PSRAM by the SPI protocol. This is why it is also called SPI RAM. The 3rd SPI bus of the ESP32 is used to communicate with the flash memory (which contains the program) and with the PSRAM.



Internal overview of the ESP-WROVER-B module

Note

There is an ambiguity about the actual amount of memory in the PSRAM. The actual capacity of the PSRAM (on the silicon chip) is not 4 MB but 8 MB. However, only 4MB can be easily accessed by the software. Using the 8 MB of PSRAM is quite complex but possible . This said, 4 MB of additional RAM is already massive (8 times the internal RAM)!

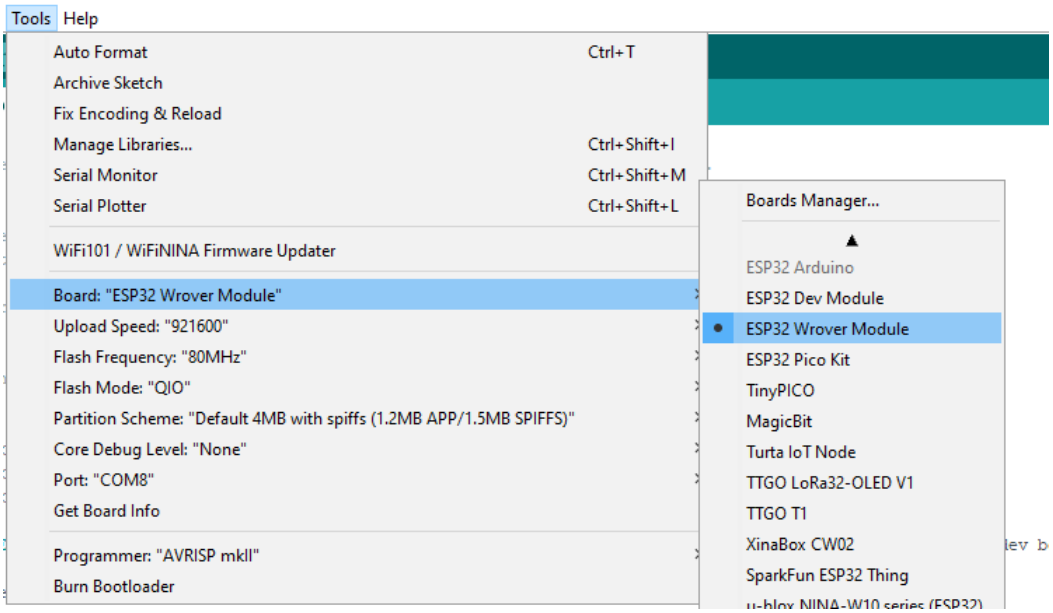
The advantage of an external RAM is relieving the internal RAM for the temporary storage of essential data.

For example, we can with the PSRAM:

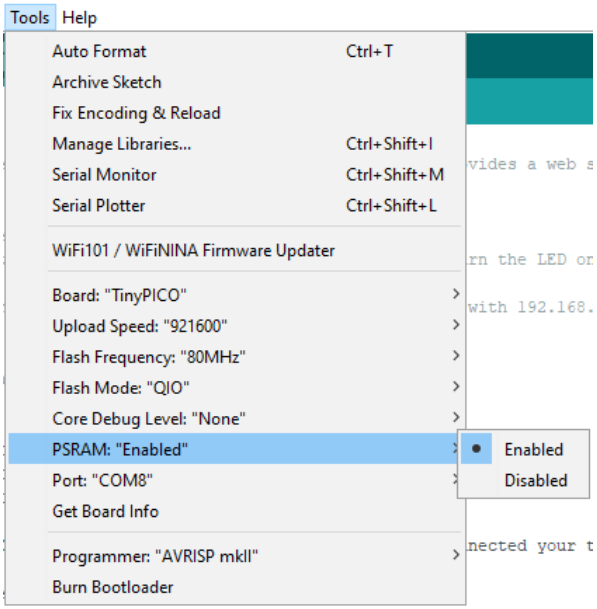
- Handle entire web pages
- Handling large JSON files
- Making a powerful web server
- Create and manipulate huge arrays
- Read and handle large files from the SPIFFS or SD card

How to use it?

On the Arduino IDE, to use the PSRAM, you have to select a compatible board, for example, the ESP32 Wrover Module, which will work for all ESP32 boards with a PSRAM.



If you are using a board in the list, for example, the [TinyPICO board](), check that the PSRAM is activated.



Warning

The PSRAM does not appear in the ESP32 memory table, so it is expected that the size of the RAM indicated on the Arduino IDE or PlateformIO is always 327 Kb. There is a clear separation between the internal RAM of the ESP32 (327 KB) and the external RAM (4 MB).

```
Sketch uses 685426 bytes (52%) of program storage space. Maximum is 1310720 bytes.
Global variables use 38832 bytes (11%) of dynamic memory, leaving 288848 bytes for local variables. Maximum is 327680 bytes.
```

To use PSRAM, you can either use libraries that support PSRAM or do dynamic allocations yourself to create arrays, and buffers, download HTLM files, JSON …

Arduino / ESP32 libraries

To avoid the headache of dynamic allocations, some Arduino libraries (compatible with ESP32) support PSRAM. The best known is ArduinoJson , which allows us to manipulate JSON files on Arduino and ESP32 easily.

ArduinoJSON

If you don't know ArduinoJson yet, read this tutorial to install and learn how to use the ArduinoJson library . To use the PSRAM with ArduinoJson, you have to create a specific JsonDocument that you place before the `setup()` function :

```
struct SpiRamAllocator {
        void* allocate(size_t size) {
                return ps_malloc(size);

        }
        void deallocate(void* pointer) {
                free(pointer);
        }
};

using SpiRamJsonDocument = BasicJsonDocument<SpiRamAllocator>;
```

Then we use the `SpiRamJsonDocument` as a standard `DynamicJsonDocument` :

```
SpiRamJsonDocument doc(100000); //100 KB here
deserializeJson(doc, input);
```

Here is an example that downloads a 65 KB JSON file. The JSON contains the results of the search "ESP32 JSON" on Google.

Exemple

```
#include <WiFi.h>
#include <HTTPClient.h>
#define ARDUINOJSON_DECODE_UNICODE 1
#include <ArduinoJson.h>

//WiFi
const char *ssid = "WiFi name";
const char *password = "WiFi password";

const char *url =
"https://raw.githubusercontent.com/uPesy/ESP32_Tutorials/master/JSON/bigJso
nExample.json";

struct SpiRamAllocator {
```

```
        void* allocate(size_t size) {
                return ps_malloc(size);

        }
        void deallocate(void* pointer) {
                free(pointer);
        }
};

using SpiRamJsonDocument = BasicJsonDocument<SpiRamAllocator>;


void setup(){
        delay(500);
        psramInit();
        Serial.begin(115200);
        Serial.println((String)"Memory available in PSRAM : "
+ESP.getFreePsram());

        //Connect to WiFi

        WiFi.begin(ssid, password);
        while (WiFi.status() != WL_CONNECTED){
                delay(100);
                Serial.print(".");
        }

        Serial.print("\nWiFi connected with IP : ");
        Serial.println(WiFi.localIP());

        Serial.println("Downloading JSON");
        HTTPClient http;
        http.useHTTP10(true);
        http.begin(url);
        http.GET();

        SpiRamJsonDocument doc(100000); //Create a JSON document of 100 KB
        DeserializationError error = deserializeJson(doc,
http.getStream());
        if(error){
                Serial.print("deserializeJson() failed: ");
                Serial.println(error.c_str());
        }

        http.end();
        Serial.println((String)"JsonDocument Usage Memory: " +
doc.memoryUsage());


        for (int i=0; i!=10;i++){
                Serial.println("\n[+]");
                Serial.println(doc["items"][i]["title"].as<String>());
                Serial.println("------------------------------");
                Serial.println(doc["items"][i]["snippet"].as<String>());
                Serial.println("------------------------------");
                Serial.println((String) "URL : " +
doc["items"][i]["link"].as<String>());

        }
}
```

```
void loop(){
}
```

Serial monitor:

```
Memory available in PSRAM: 4194252
..............
WiFi connected with IP: 192.168.43.7
Downloading JSON
JsonDocument Usage Memory: 62515

[+]
Installing ESP32 in Arduino IDE (Windows, Mac OS X, Linux ...
------------------------------
Installing ESP32 Add-on in Arduino IDE Windows, Mac OS X, Linux open. Enter
https://dl.espressif.com/dl/package_esp32_index.json into the "Additional
Board ...
------------------------------
URL : https://randomnerdtutorials.com/installing-the-esp32-board-in-
arduino-ide-windows-instructions/

[+]
ESP32: Parsing JSON - techtutorialsx
------------------------------
Apr 26, 2017 ... The objective of this post is to explain how to parse JSON
messages with the
ESP32 and the ArduinoJson library. Introduction The objective of ...
------------------------------
URL : https://techtutorialsx.com/2017/04/26/esp32-parsing-json/

...
```

See my Instructable, which explains how you use Google Search on an ESP32 and display the result on the serial monitor.

Instructables to do Google searches on ESP32

See also

[How to use the external RAM on the ESP32](#) on the official ArduinoJson website

Arduino Framework

To store variables in the PSRAM, it is necessary to make dynamic allocations with specific functions for the PSRAM of the ESP32.

Note

It is not necessary to have advanced knowledge of C / C ++ to use PSRAM! Knowledge of pointers and dynamic allocation functions like `malloc()` or `calloc()` is a plus, but the proposed examples are sufficient on their own to use PSRAM in electronic projects.

The main functions to be used are the following `psramInit()` , `ESP.getFreePsram()` , `ps_malloc()` ou `ps_calloc()` et `free()` .

The `psramInit()` function is used to initialize the PSRAM, the `ESP.getFreePsram()` function returns the amount of memory available in the PSRAM. The three other functions are used for dynamic allocation.

Examples

This section provides examples of how to use PSRAM. They allow you to understand how to use the functions mentioned above. It is not necessary to understand everything to be able to use PSRAM. You must modify the values in the examples to use them in your projects.

Variables Array String Html/JSON

Variables

- This example shows how to create variables that are stored in PSRAM:

```
//Create an integer number
int *var_int = (int *) ps_malloc(sizeof(int));
*var_int = 42;

//Create a float number
float *var_float = (float *) ps_malloc(sizeof(float));
*var_float = 42.42;
```

Here's the full sketch:

Example

```
void setup() {
Serial.begin(115200);
//PSRAM Initialisation
if(psramInit()){
```

```
        Serial.println("\nThe PSRAM is correctly initialized");
}else{
        Serial.println("\nPSRAM does not work");
}

//Create an integer
int *var_int = (int *) ps_malloc(sizeof(int));
*var_int = 42;

//Create a float
float *var_float = (float *) ps_malloc(sizeof(float));
*var_float = 42.42;

Serial.println((String)"var_int = " + *var_int);
Serial.print("var_float = ");
Serial.println(*var_float);
}
```

Terminal output:

```
The PSRAM is correctly initialized
var_int = 42
var_float = 42.42
```

Arrays

For arrays, the syntax is very similar:

```
//Create an array of 1000 integers
int n_elements = 1000;
int *int_array = (int *) ps_malloc(n_elements * sizeof(int));
//We access array values like a classic array
int_array[0] = 42;
int_array[42] = 42;
int_array[999] = 42;
```

- Here is the complete sketch to create an integer array of 1000 elements stored in the PSRAM:

Example

```
int n_elements = 1000;

void setup(){
        Serial.begin(115200);
        //Init
        if(psramInit()){
        Serial.println("\nPSRAM is correctly initialized");
        }else{
        Serial.println("PSRAM not available");
        }
```

```
        //Create an array of n_elements
        int available_PSRAM_size = ESP.getFreePsram();
        Serial.println((String)"PSRAM Size available (bytes): " +
available_PSRAM_size);

        int *array_int = (int *) ps_malloc(n_elements * sizeof(int));
//Create an integer array of n_elements
        array_int[0] = 42;
        array_int[999] = 42; //We access array values like classic array

        int available_PSRAM_size_after = ESP.getFreePsram();
        Serial.println((String)"PSRAM Size available (bytes): " +
available_PSRAM_size_after); // Free memory space has decreased
        int array_size = available_PSRAM_size - available_PSRAM_size_after;
        Serial.println((String)"Array size in PSRAM in bytes: " +
array_size);

        //Delete array
        free(array_int); //The allocated memory is freed.
        Serial.println((String)"PSRAM Size available (bytes): "
+ESP.getFreePsram());
}

void loop() {
}
```

Terminal output:

```
PSRAM is correctly initialized
PSRAM Size available (bytes): 4194252
PSRAM Size available (bytes): 4190236
Array size in PSRAM in bytes: 4016
PSRAM Size available (bytes): 4194252
```

Note

The size in bytes of the array is 4016 for an array of 1000 integers (int). The `int` type is coded on 4 bytes. So there are 4 * 1000 bytes occupied by the array, and the 16 remaining bytes contain information about the memory block (size, flags).

- For tables of other types of variables :

```
char * array1 = (char *) ps_malloc (n_elements * sizeof (char)); // Create
an empty array of n_elements characters
float * array = (float *) ps_malloc (n_elements * sizeof (float)); //
Create an array of n_elements float number
```

- We can also create arrays that are filled with zeros with `ps_calloc()` :

```
int n_elements = 20;
```

```
Serial.println((String)"PSRAM Size available (bytes): "
+ESP.getFreePsram());
Serial.println("Array of integers initialized to 0");
int *array = (int *) ps_calloc(n_elements, sizeof(int));
Serial.print("[array] : ");
for(int i=0; i!= n_elements;i++){
        Serial.print((String)array[0] + " ");
}
Serial.println((String)"\nPSRAM Size available (bytes): "
+ESP.getFreePsram());
```

Terminal output:

```
PSRAM Size available (bytes): 4194252
Array of integers initialized to 0
[array]: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
PSRAM Size available (bytes): 4194156
```

- Here are 2 ways to create 2-dimensional tables:

Example

```
int n_rows = 10;
int n_columns = 20;


void setup(){
        Serial.begin(115200);
        //Initialisation
        if(psramInit()){
                Serial.println("\nPSRAM is correctly initialized");
        }else{
                Serial.println("PSRAM not available");
        }

        //Create an array of n_elements
        int initialPSRAMSize = ESP.getFreePsram();
        Serial.println((String)"PSRAM Size available (bytes): " +
initialPSRAMSize);
        //Creating a two-dimensional array with a one-dimensional array
        int *array2D = (int *) ps_malloc(n_rows * n_columns * sizeof(int));
// Create an array of n_rows x n_columns
        //To access the value located in row 5 to column 10
        int row_i = 5;
        int column_j = 10;
        array2D[column_j * n_columns + row_i] = 42;

        //Creating a two-dimensional zero-filled array with an array of
pointers
        int **array2Dbis = (int **) ps_calloc(n_rows, sizeof(int *));
        for (int i =0; i!= n_rows;i++){
                array2Dbis[i] = (int *) ps_calloc(n_columns , sizeof(int));
        }
        //To access the value located in row 5 to column 10
        array2Dbis[row_i][column_j] = 42;
```

```
        int PSRAMSize = ESP.getFreePsram();
        Serial.println((String)"PSRAM Size available (bytes): " +
PSRAMSize);
        int array2D_size = initialPSRAMSize - PSRAMSize;
        Serial.println((String)"Size of the array in PSRAM in bytes: " +
array2D_size);

        //Delete 2D arrays
        free(array2D); //Allocated memory is freed.
        free(array2Dbis);
        Serial.println((String)"PSRAM Size available (bytes): "
+ESP.getFreePsram());
}

void loop() {
}
```

Strings

Since a String is a chain of characters, it is just an array of type `char` :

```
int n_elements = 20;
char *str = (char *) ps_calloc(n_elements, sizeof(char)); //Create an array
of n_elements null characters ('\0')
str[0] = '4';
str[1] = '2';
```

Note

The use of `ps_calloc()` is more interesting because it allows the ESP32 to detect the end of the string directly.

Here is an example sketch:

```
void setup() {
        Serial.begin(115200);
        if(psramInit()){
                Serial.println("\nThe PSRAM is correctly initialized");
        }else{
                Serial.println("\nPSRAM does not work");
        }

        int n_elements = 20;
        char *str = (char *) ps_calloc(n_elements, sizeof(char)); //Create
an array of n_elements null characters ('\0')
        for(int i = 0; i < 10;i+=3){
                str[i] = '4';
                str[i+1] = '2';
                str[i+2] = '_';
        }
        Serial.println(str);
}
```

Terminal output:

```
The PSRAM is correctly initialized
42_42_42_42_
```

Html/Json

Here is an example that allows you to download any text file, such as HTML pages or JSON files (under 4 MB) and store it in PSRAM.

```
#include <WiFi.h>
#include <HTTPClient.h>
#define TIMEOUT 5000

const char *ssid = "yourNetworkName";
const char *password = "yourNetworkPassword";
const char *url = "https://en.wikipedia.org/wiki/ESP32";

void setup() {
    Serial.begin(115200);

    // Connect to WiFi
    WiFi.begin(ssid, password);
    while (WiFi.status() != WL_CONNECTED){
        delay(100);
        Serial.print(".");
    }
    Serial.print("\nConnected to WiFi with IP :");
    Serial.println(WiFi.localIP());

    char *page = (char *) ps_malloc(sizeof(char));
    int lengthHtml = 0;
    HTTPClient http;
    http.begin(url);
    http.useHTTP10(true); // Allow to try to having a Content-Length in the
header using HTTP1

    Serial.println((String) "Try GET "+ url);
    int httpCode = http.GET();
    if (httpCode > 0){
            // Displays the response HTTP code of the GET request
            Serial.printf("[HTTP] GET... code: %d\n", httpCode);

            // If the file exists (HTML, JSON, ...)
            if (httpCode == HTTP_CODE_OK){
                // Try to get the file size ( -1 if there is no Content-
Length in the header)
                int tempLength = http.getSize();
                Serial.println((String) "Content Length :" + tempLength);

                // Storing the page in PSRAM
                Serial.printf("Adresse mémoire de la page : %p \n", page);

                // Get the TCP stream
```

```
                WiFiClient *stream = http.getStreamPtr();

                //Initialize buffer position in PSRAM
                int position = 0;

                uint32_t currentTime = 0;
                uint8_t timeoutArboted = 1;

                // Retrieves all data from the file
                while(http.connected() && (tempLength > 0 || tempLength ==
-1)){

                        // Retrieves available data (data arrives packet by
packet)
                        size_t size = stream->available();
                        if (size){
                                page = (char*) ps_realloc(page, position +
size + 1);
                                stream->readBytes(page+position, size);
                                position += size;

                                if (tempLength > 0){
                                        tempLength -= size;
                                }

                                timeoutArboted = 1;
                                }else{
                                        //If we don't know the size of the
file, we assume that all data are received before the timeout
                                        if(timeoutArboted){
                                                //Launches the timer
                                                currentTime = millis();
                                                timeoutArboted = 0;
                                        }else{
                                                if(millis()-currentTime >
TIMEOUT){
                                                        //We have reached
the Timeout

Serial.println("Timeout reached");
                                                                break;
                                                }
                                        }
                                }
                        }

                        *(page+position) = '\0';

                        lengthHtml = position;
                        Serial.println((String)"Downloaded " + lengthHtml +
" Octets");
                }
        }
        else{
                Serial.printf("[HTTP] GET... failed, error: %s\n",
http.errorToString(httpCode).c_str());
        }
        //Closes the HTTP client
        http.end();
        delay(1500);
        Serial.println(page); // To display the entire HTML page
```

```
}
```

Terminal output:

```
.....
WiFi connected with IP : 192.168.43.7
Try GET https://en.wikipedia.org/wiki/ESP32
[HTTP] GET... code: 200
Content Length :117615
Memory address of HTML page in PSRAM : 0x3f800024
Downloaded 117615 Octets
<!DOCTYPE html>
<html class="client-nojs" lang="en" dir="ltr">
<head>
<meta charset="UTF-8"/>
<title>ESP32 - Wikipedia</title>
...
...
```