Les interruptions

Une interruption est un déclenchement qui arrête l'exécution d'un programme ou d'un bout de code pour exécuter une autre fonction, puis d'en reprendre l'exécution à partir de l'instant où il a été stoppé.
En d'autres termes, dès qu'une interruption est détectée par le processeur, celui-ci sauve son état d'exécution, exécute une portion de code liée à l'interruption (Interrupt Service Routine), et revient ensuite à son état avant l'interruption pour continuer ce qu'il faisait.
Il est très important que le programme d'interruption aie un temps d'exécution le plus court possible. On ne fera donc aucun calcul compliqué et aucun appel à des fonctions longues.

INTERRUPTIONS DANS UN MICROCONTRÔLEUR

Les interruptions sont utilisées pour gérer les événements qui ne se produisent pas pendant l'exécution séquentielle d'un programme. Par exemple, on veut effectuer certaines tâches et ces tâches s'exécutent séquentiellement dans le programme Arduino. Mais il existe peu de tâches qui ne s'exécutent que lorsqu'un événement spécial se produit, tel qu'un signal de déclenchement externe vers la broche d'entrée numérique d'un microcontrôleur.

Interruption externe

Une interruption externe ou une « interruption matérielle » est causée par le module matériel externe. Par exemple, il existe une interruption qui se produit lorsqu'on appuie sur un bouton (une interruption GPIO lorsqu'un bouton est enfoncé). Avec l'interruption, on n'a pas besoin de vérifier en permanence l'état de la broche d'entrée numérique. Lorsqu'une interruption survient (un changement est détecté), le processeur arrête l'exécution du programme principal et une fonction est appelée ISR ou Interrupt Service Routine. Le processeur travaille alors temporairement sur une tâche différente (ISR) puis revient au programme principal après la fin de la routine de traitement.

Arduino :

Cartes	Pins numériques utilisés pour les interruptions
Uno, Nano, Mini	2, 3
Mega	2, 3, 18, 19, 20, 21
Micro, Leonardo	0, 1, 2, 3, 7
Zero	Tous les pins numériques sauf le 4
DUE	Tous les pins numériques

Création d'une interruption

La fonction à utiliser est la suivante :

```
attachInterrupt(interrupt, function, mode)
```

- `interrupt` : numéro de l'interruption (0 pour le pin 2 et 1 pour le pin 3)
- `function` : la fonction ou l'ISR à appeler. La fonction ne doit pas prendre de paramètre et ne doit rien renvoyer. Comme le processeur est « gelé » à ce moment-là, on doit s 'assurer d'avoir une fonction la plus courte possible. Au lieu d'appeler d'autres fonctions lourdes, on active un flag qui sera lu dans le code général et on sort de l'interruption.
- `mode` : type de déclenchement (LOW // FALLING // RISING // CHANGE).

- LOW : l'interruption est déclenchée quand la broche concernée est LOW. Comme il s'agit d'un état et non d'un événement, l'interruption sera déclenchée tant que la broche est LOW. Par conséquent, dès que l'ISR aura terminé son exécution, elle la recommencera. Pendant ce temps, loop() ne sera pas exécuté.
- CHANGE : l'interruption est déclenchée quand la broche concernée change d'état, c'est à dire passe de LOW à HIGH ou bien de HIGH à LOW. Il s'agit d'un événement.
- RISING : l'interruption est déclenchée quand la broche concernée passe de LOW à HIGH. Il s'agit également d'un événement.
- FALLING : l'interruption est déclenchée quand la broche concernée passe de HIGH à LOW. Il s'agit encore d'un événement.

Utilisation

On veut que le programme se stoppe lorsque l'on appuie sur un bouton.
On va appeler une fonction `Pause()` lorsque l'utilisateur appuie sur un bouton branché sur le pin 2.
Ainsi, on aura la syntaxe suivante :

```
attachInterrupt(0, Pause, FALLING);
```

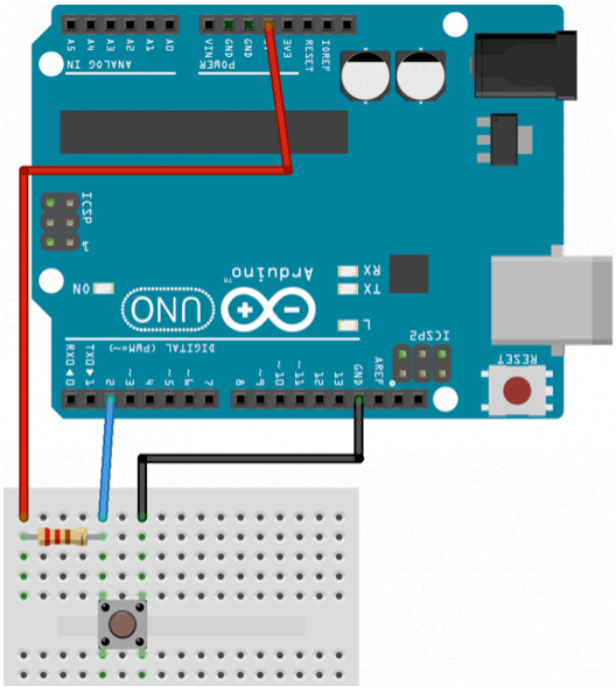
On note l'absence des parenthèse après l'appel de fonction `Pause`. On peut coder par exemple la fonction `Pause()` un peu plus loin dans le programme.

Lorsqu'on utilise une interruption, tout le programme se met en « pause » et la fonction appelée dans l'interruption prend le relais. Ainsi, à l'intérieur de la routine d'interruption, certaines fonctions sont inutilisables (**delay** ne fonctionne pas, **millis** renverra toujours la même valeur, les données séries reçues seront perdues et les signaux PWM sont affectés).

Toute valeur modifiée à l'intérieur de la routine d'interruption devra être déclarée comme **volatile**, afin que le processeur aille chercher la valeur en mémoire et ne se fie pas à ce qui se trouve dans ses registres qui étaient gelés au moment de l'interruption.

- Si une fonction **attachInterrupts** assigne une routine à une pin, celle-ci écrase la configuration précédente.
- La fonction detachInterrupt(pin) permet de retirer l'assignation d'une interruption à une pin.
- Les fonctions interrupts() et noInterrupts() permettent respectivement d'activer ou de désactiver les interruptions. Celles-ci sont activées par défaut.

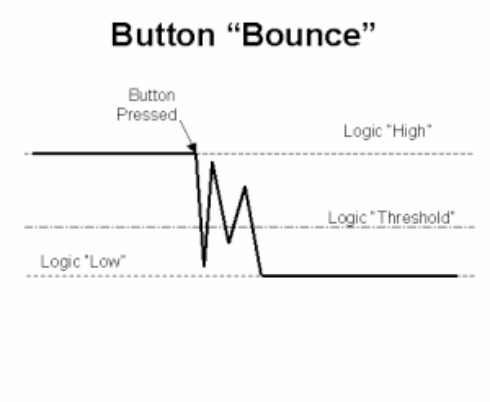
Exemple



```
int pin = 13;
volatile int state = LOW;  // Déclaration d'une variable volatile

void Setup() {
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, CHANGE);  // Attache l'interruption externe
n°0 à la fonction "blink"
}

void Loop() { // La fonction appelée par l'interruption externe n°0
  digitalWrite(pin, state);  // La LED reflète l'état de la variable
}

void blink() {
  state = !state;  // Inverse l'état de la variable
}
```

Exemple : Interruption évoluée

Après avoir câblé le tout et uploadé le code, on remarque que celui-ci ne fonctionne pas exactement comme il devrait. Ceci est dû au bouton qui crée un effet « rebond (bounce) » lorsqu'il est pressé.



Pour contrer cet effet, il existe plusieurs méthodes: hardware (avec un condensateur en parallèle du bouton pour le transformer en circuit RC) ou software.

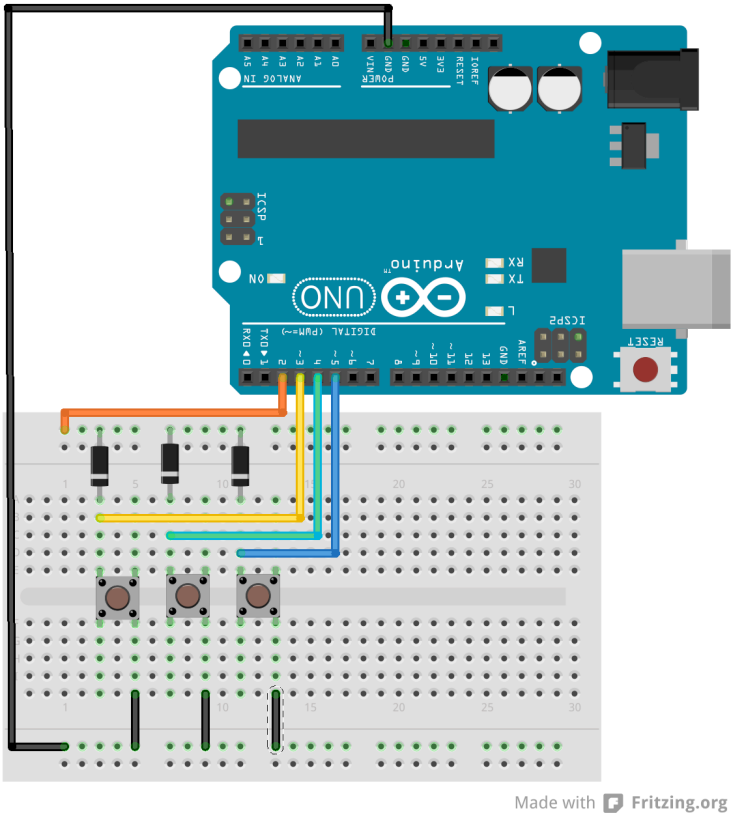
On va introduire un temps de debounce, ce qui permet, suite à une interruption, d'ignorer les interruptions suivantes si elles se trouvent dans cette fenêtre de temps. On prend pour l'exemple un temps de 250 ms, ce qui permet de contrer l'effet bounce de la majorité des boutons. On peut essayer avec différents boutons de diminuer ce délai jusqu'à ce que les effets néfastes réapparaissent et on remarquera que les boutons ne sont pas tous égaux, certains étant plus performants (et généralement plus bruyants). Dans notre cas, on utilisera un fil.

```
int pin = 13;

volatile unsigned long button_time = 0;
volatile unsigned long last_button_time = 0;
int debounce = 250;          // Debounce latency en ms
volatile int state = LOW;  // Déclartion d'une variable volatile

void setup() {
  pinMode(pin, OUTPUT);
  attachInterrupt(0, blink, FALLING);  // Attache l'interruption externe
n°1 à la fonction blink
}

void loop() {
  digitalWrite(pin, state);  // La LED reflète l'état de la variable
}

void blink() {  // La fonction appelée par l'interruption externe n°0
  button_time = millis();
  if (button_time - last_button_time > debounce) {
    state = !state;  // Inverse l'état de la variable
    last_button_time = button_time;
  }
}
```

Interruptions multiples

Si vous utilise un Arduino UNO avec plusieurs boutons déclenchant des interruptions, on risque d'être limité par les 2 seules interruptions disponibles.

Grâce à quelques diodes et un code un peu plus élaboré, on va pouvoir étendre les possibilités:




```
#include <lib_serial.h>

#define inter 2    //pull-up IN
#define button1 3  //pull-up IN
#define button2 4  //pull-up IN
#define button3 5  //pull-up IN

volatile unsigned long button_time = 0;
int debounce = 50;  // Debounce latency en ms

volatile int trigger = 0;
int total = 0;
int inta = 0;
int intb = 0;
int intc = 0;

void setup() {
  initPins();
  initSER();
  Serial.println("Démarrage…");
```

```
  attachInterrupt(0, detect, FALLING);
}

void loop() {
  if (trigger != 0) {
    total++;
    if (trigger == 1)
      inta++;
    if (trigger == 2)
      intb++;
    if (trigger == 3)
      intc++;
    Serial.print(total);
    Serial.print(" || ");
    Serial.print(inta);
    Serial.print(" | ");
    Serial.print(intb);
    Serial.print(" | ");
    Serial.println(intc);
    trigger = 0;
  }
}

void detect() {
  if (button_time > millis())
    return;
  delayMicroseconds(32000);
  if (!digitalRead(button1))
    trigger = 1;
  if (!digitalRead(button2))
    trigger = 2;
  if (!digitalRead(button3))
    trigger = 3;
  button_time = millis() + debounce;
}
// ______
void initPins() {
  pinMode(button1, INPUT);
  pinMode(button2, INPUT);
  pinMode(button3, INPUT);
  pinMode(inter, INPUT);
  pinMode(led, OUTPUT);
  digitalWrite(button1, HIGH);
  digitalWrite(button2, HIGH);
  digitalWrite(button3, HIGH);
  digitalWrite(inter, HIGH);
  digitalWrite(led, LOW);
}

void initSER() {
  Serial.begin(19200);
  while (!Serial) {
    ;  // wait for serial port to connect. Needed for Leonardo only
  }
}
```

On note que la routine d'interruption a été légèrement modifiée, et qu'elle incorpore un **delayMicroseconds()**.

Contrairement au **delay()**, la fonction **delayMicroseconds** peut être utilisée dans une routine d'interruption car elle ne se base pas sur des interruptions mais sur des NOP (1 cycle de processeur pendant lequel il ne fait rien). Ce qui nous permet d'ajouter 0.030 seconde (la fonction prend un **int** en paramètre, ce qui nous limite sur un Arduino UNO à environ 32.000, arrondi à 30.000). Bien que négligeable, il s'avère que ce délai est d'une aide précieuse pour la lecture et l'identification du pin qui a déclenché l'interruption. Avant de l'utiliser, l'interruption configurée sur FALLING, donc sur front descendant, se déclenchait lorsqu'on appuyait sur le bouton, mais également lorsqu'on le relâchait. Ce qui n'est plus le cas avec ce micro-délai. Par ailleurs, le debounce s'en retrouve amélioré et le debounce time peut être diminué de manière très conséquente. Dans le cadre de cette application, on a même pu supprimer les mécanismes de debounce pour ne laisser que le **delayMicroseconds()**.

Exemple :

Comptage des impulsions d'un bouton poussoir relié à D2 malgré une tempo de 5s dans le programme principal.

```
int boutonPin = 2;
int ledPin = 3;
int compteur = 0;

/*  Pour qu'on puisse changer la valeur d'une variable depuis une fonction de
gestion d'interruption, il faut que la variable soit déclarée "volatile" */

volatile boolean changed = false;

void setup() {
  Serial.begin(9600);
  pinMode(boutonPin, INPUT);
  pinMode(ledPin, OUTPUT);
  // Attacher la fonction ISR, detection de front descendant (5v vers 0v)
  attachInterrupt(0, doContact, FALLING);
}

void loop() {
  if (changed) {
    changed = false;
    Serial.print("Action sur le BP ");
    Serial.print(compteur);
    Serial.println(" fois");
    // On allume la LED pendant 1 seconde
    digitalWrite(ledPin, HIGH);
    delay(1000);
    digitalWrite(ledPin, LOW);
  }
  delay(5000);
}

void doContact() {
  changed = true;
  compteur++;
}
```

ESP32

On veut allumer une LED lorsqu'on appuie sur un bouton qui est relié à un pin GPIO de l'ESP32. **Le plus simple** est de regarder en permanence dans la fonction `loop()` si on a appuyé sur le bouton :

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

// Etat du bouton poussoir
int buttonState = 0;

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

  //Configuration du pin en entrée 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);
  }
}
```

Le problème est que le processeur du microcontrôleur est totalement occupé par cette tâche. Alors on peut dire au microcontrôleur de faire d'autres tâches dans la `loop()` , mais dans ce cas le microcontrôleur ne regardera l'état du bouton qu'une seule fois à chaque itération de la fonction `loop()` . Il se peut qu'on manque un évènement. On ne peut pas traiter en temps réel des évènements extérieurs. Les interruptions permettent de détecter un évènement en temps réel tout en laissant le processeur du microcontrôleur faire d'autres tâches. Ainsi le fonctionnement d'une interruption est le suivant :

Détection d'un évènement → Interruption du programme principal → Exécution du code de l'interruption → Le processeur reprend là où il s'est arrêté.

Note : Avec les interruptions, il n'y a plus besoin de regarder en permanence la valeur d'un pin : lorsqu'un changement est détecté, une fonction est exécutée.

Les modes de détection

La détection d'un événement est basée sur l'allure du signal qui arrive au pin.

On peut choisir le mode de détection de l'interruption :

- `LOW` : Déclenche l'interruption dès que le signal est à 0V

- `HIGH` : Déclenche l'interruption dès que le signal est à 3.3V
- `RISING` : Déclenche l'interruption dès que le signal passe de `LOW` à `HIGH` (0 à 3.3V)
- `FALLING` : Déclenche l'interruption dès que le signal passe de `HIGH` à `LOW` (3.3V à 0)
- `CHANGE` : Déclenche l'interruption dès que le signal passe de `LOW` à `HIGH` ou de `HIGH` à `LOW` .

Les modes `RISING` et `FALLING` sont les plus utilisés. A noter que si on utilise les modes `LOW` et `HIGH` , l'interruption se déclenchera en boucle tant que le signal ne change pas d'état.

Utilisation sur l'ESP32

L'utilisation des interruptions sur l'ESP32 est similaire à celle sur l'Arduino avec la fonction `attachInterrupt()` . **N'importe quel pin GPIO peut être utilisé pour les interruptions, à l'exception de GPIO6, GPIO7, GPIO8, GPIO9, GPIO10 et GPIO11.**

```
attachInterrupt(digitalPinToInterrupt(pin), fonction_ISR, mode)
```

La fonction attachInterrupt() prend trois arguments :

- digitalPinToInterrupt(pin): Il s'agit d'une fonction qui prend la broche GPIO de la carte ESP32 comme paramètre à l'intérieur. La broche indique le GPIO associé à la broche qui provoquera une interruption. Par exemple, si on définit GPIO2 comme broche d'interruption, la fonction sera spécifiée comme digitalPinToInterrupt(2).
- ISR : il s'agit du deuxième argument utilisé pour configurer une interruption. Il s'agit d'un type spécial de fonction connue sous le nom de routine de service d'interruption qui ne prend aucun paramètre et ne renvoie rien. Chaque fois que l'interruption se produira, cette fonction sera appelée.
- mode : indique l'action de déclenchement pour que l'interruption se produise. Les quatre paramètres suivants sont utilisés pour spécifier le mode :
  - LOW : Ceci est utilisé pour déclencher l'interruption lorsque la broche est à l'état bas.
  - CHANGE: Ceci est utilisé pour déclencher l'interruption lorsque la broche change d'état (HIGH-LOW ou LOW-HIGH)
  - RISING : Ceci est utilisé pour déclencher l'interruption lorsque la broche passe de LOW à HIGH.
  - FALLING : Ceci est utilisé pour déclencher l'interruption lorsque la broche passe de HIGH à LOW.

```
void IRAM_ATTR fonction_ISR() {
  // Contenu de la fonction
}
```

Il est conseillé d'ajouter le flag `IRAM_ATTR` pour que le code de la fonction soit stocké dans la RAM (et non pas dans la Flash), afin que la fonction s'exécute plus rapidement.

Le code entier sera de la forme :

```
void IRAM_ATTR fonction_ISR() {
  // Code de la fonction
}

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

void loop() {
}
```

- Dès que la tension passera de 3.3V à 0V, la fonction `fonction_ISR()` sera exécutée. On peut ensuite faire d'autres tâches dans la fonction `loop()` .

Il faut garder en tête que la fonction d'une interruption doit s'exécuter le plus rapidement possible pour ne pas perturber le programme principal. Le code doit être le plus concis possible et il est déconseillé de dialoguer par SPI, I2C, UART depuis une interruption.

On ne peut pas utiliser la fonction `delay()` ni `Serial.println()` avec une interruption. On peut néanmoins afficher des messages dans le moniteur série en remplaçant `Serial.println()` par `ets_printf()` qui est compatible avec les interruptions.

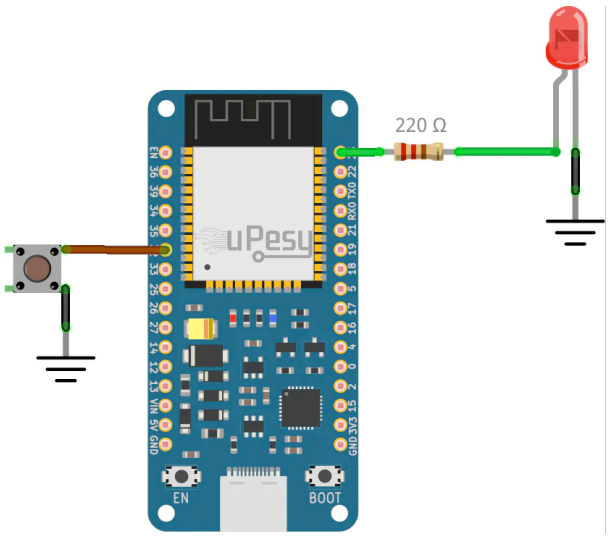
Le code ci-dessous affiche *« Boutton pressé »* lorsqu'on presse sur un bouton relié au pin 33.

```
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() {
}
```

Exemple

On va modifier le code de base qui permet d'allumer une LED si on appuie sur un bouton.

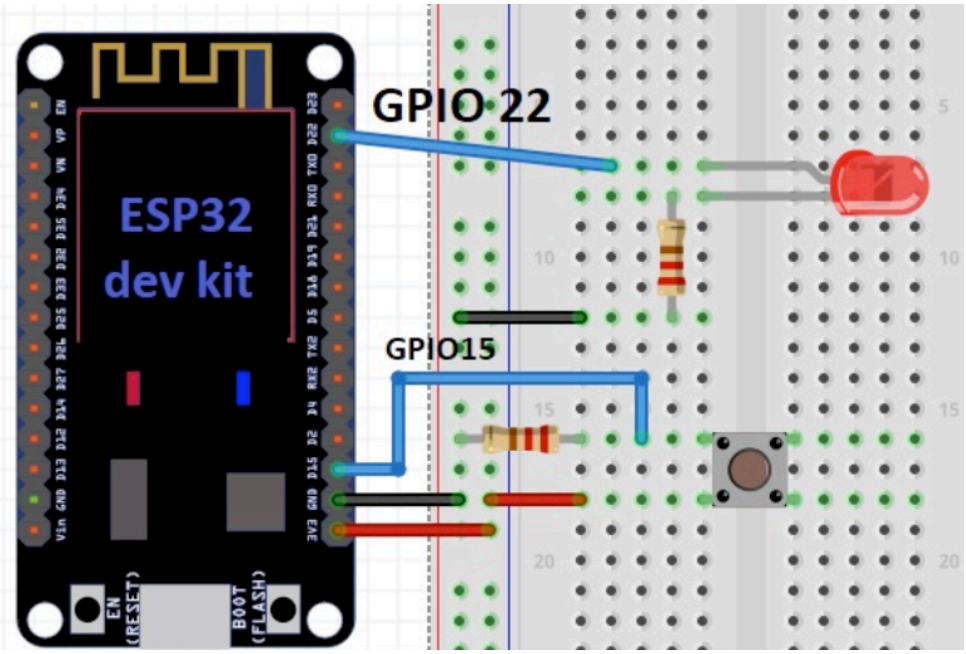
```
const int buttonPin = 32;
const int ledPin = 23;

// Etat du bouton poussoir
int buttonState = 0;

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

  //Configuration du pin en entrée pullup
  pinMode(buttonPin, INPUT_PULLUP);
  pinMode(ledPin, OUTPUT);
}

void loop() {

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

  if (buttonState == LOW) {
    // Allume la led
    digitalWrite(ledPin, HIGH);
  } else {
    // Eteins la led
    digitalWrite(ledPin, LOW);
  }
}
```



On va utiliser les interruptions pour gérer l'événement et libérer le processeur pour qu'il puisse faire d'autres tâches.

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

void IRAM_ATTR fonction_ISR() {
  if (millis() - lastMillis > 10) {  // Software debouncing button
    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, fonction_ISR, CHANGE);
  digitalWrite(ledPin, buttonState);
}

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

Exemple : V2



Le bouton-poussoir sera connecté à une broche d'interruption de l'ESP32 et configuré comme une entrée. Alors que la LED sera configurée comme une sortie numérique. La LED basculera à chaque front montant.

Une résistance de 220 ohms
Une résistance de 10k ohms

Le bouton-poussoir a quatre bornes. Une borne est alimentée par 3,3 volts à partir d'ESP32 et l'autre borne est connectée par GPIO15 et la résistance de 10k ohms qui agit comme une résistance pull-down. L'autre extrémité de la résistance est reliée à la masse commune. Lorsque le bouton-poussoir n'est pas enfoncé, un niveau logique bas apparaît sur GPIO15, ou l'état du bouton-poussoir est bas et lorsque le bouton-poussoir est enfoncé, un niveau logique haut apparaît sur GPIO15. Cela signifie qu'un front montant se produit lorsqu'un bouton-poussoir est enfoncé. On peut détecter ce front montant à l'aide des broches d'interruption de l'ESP32.

Le bouton-poussoir agit comme une source pour l'interruption externe. Cela signifie qu'on connecte la sortie du bouton-poussoir avec la broche GPIO de l'ESP32. De plus, on attache l'interruption déclenchée par le front montant à cette broche GPIO. Cela signifie que cette broche GPIO déclenchera l'interruption chaque fois qu'elle détectera un front montant sur son entrée.

Lorsque le bouton-poussoir sera enfoncé, une interruption externe est provoquée, la LED basculera. Si elle était initialement éteinte, elle s'allumera et vice versa.

```
#define pushButton_pin 15
#define LED_pin 22

void IRAM_ATTR toggleLED() {
  digitalWrite(LED_pin, !digitalRead(LED_pin));
}

void setup() {
```

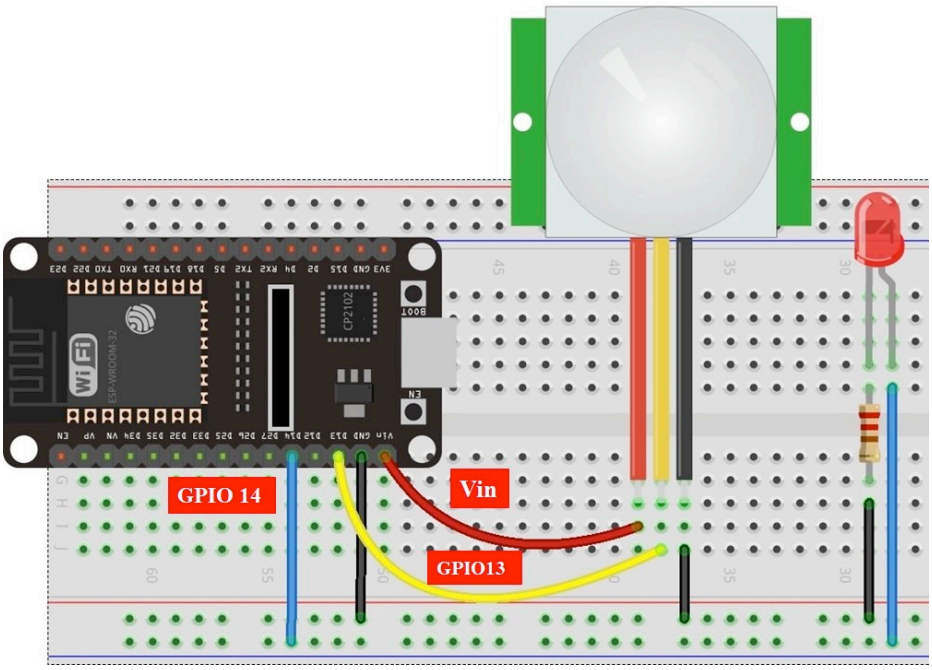
```
  pinMode(LED_pin, OUTPUT);
  pinMode(pushButton_pin, INPUT);
  attachInterrupt(pushButton_pin, toggleLED, RISING);
}

void loop() {
}
```

- Tout d'abord, on définit la broche GPIO à travers laquelle la LED est connectée. Il s'agit de GPIO22.
- On définit également la broche GPIO à travers laquelle le bouton-poussoir est connecté. Il s'agit de GPIO15.

Exemple :

Capteur de mouvement PIR avec ESP32

Lorsqu'un capteur PIR détectera un mouvement, on allumera la LED pendant 5 secondes.



GPIO14 est connecté à la broche d'anode de la LED et que la broche de cathode est connectée à la masse commune via la résistance de 220 ohms.

La broche centrale du capteur PIR est la broche de sortie qui fournit une impulsion haute chaque fois qu'un mouvement est détecté. Sinon, cette broche reste active au niveau bas. Cela signifie qu'un front montant se produit lorsqu'un capteur PIR détecte un mouvement. On peut détecter ce front montant à l'aide des broches d'interruption de l'ESP32. Ici, on a connecté la broche de sortie du capteur avec GPIO13.

Le capteur PIR agit comme une source pour l'interruption externe. Cela signifie qu'on connecte la sortie du capteur PIR avec la broche GPIO de ESP32. De plus, on attache l'interruption déclenchée

par le front montant à cette broche GPIO. Cela signifie que cette broche GPIO déclenchera l'interruption chaque fois qu'elle détectera un front montant sur son entrée.

Lorsqu'un capteur PIR détecte un mouvement, une interruption externe est provoquée, la LED reste allumée pendant 5 secondes, puis s'éteint pendant 5 secondes, et le processus se répète.

```
const int led_pin = 14;
const int sensor_pin = 13;

const long interval = 5000;
unsigned long current_time = millis();
unsigned long last_trigger = 0;
boolean timer_on = false;

void IRAM_ATTR movement_detection() {
  Serial.println("Mouvement détecté");
  digitalWrite(led_pin, HIGH);
  timer_on = true;
  last_trigger = millis();
}

void setup() {

  Serial.begin(115200);

  pinMode(sensor_pin, INPUT_PULLUP);
  attachInterrupt(digitalPinToInterrupt(sensor_pin), movement_detection,
RISING);
  pinMode(led_pin, OUTPUT);
  digitalWrite(led_pin, LOW);
}

void loop() {

  current_time = millis();
  if (timer_on && (current_time - last_trigger > interval)) {
    Serial.println("Mouvement arrêté");
    digitalWrite(led_pin, LOW);
    timer_on = false;
  }
}
```