ESP32 GPIO and Interrupts

GPIO - overview

The ESP32 chip features 40 physical GPIO pads.

Some GPIO pads cannot be used or do not have the corresponding pin on the chip package; each pad can be used as a general purpose I/O or can be connected to an internal peripheral signal.

- GPIO6-11 are usually used for SPI flash;
- GPIO34-39 can only be set as input mode and do not have software pullup or pulldown functions.

To access GPIO pins functionality, include the file:

```
#include "driver/gpio.h"
```

GPIO – pin direction

To configure GPIO's direction, such as *output_only*, *input_only*, *output_and_input*, use the function:

```
esp_err_t gpio_set_direction(gpio_num_t gpio_num, gpio_mode_t mode)
```

the returned value can be:

- ESP_OK Success
- ESP_ERR_INVALID_ARG GPIO error

configure GPIO pins number, for example GPIO_NUM_2 (2), which corresponds to the blue led on doit-ESP32 board

defines GPIO direction. Possible values are:

- GPIO_MODE_DISABLE: disable input and output;
- GPIO_MODE_INPUT: input only;
- GPIO_MODE_OUTPUT : output only;
- GPIO_MODE_OUTPUT_OD: output only with open-drain mode;
- GPIO_MODE_INPUT_OUTPUT_OD : output and input with open-drain mode;
- GPIO_MODE_INPUT_OUTPUT : output and input mode

GPIO – pin level

To control GPIO's value, following functions are availables:

- `int` **`gpio_get_level`**`(gpio_num_t gpio_num)` - get input level of `gpio_num` GPIO.

  Possible returned values are:

  - 0 – if the GPIO input level is 0;
  - 1 - if the GPIO input level is 1;

  → please note that (if the pad is not configured for input (or input and output) the returned value is always 0.)

- `esp_err_t` **`gpio_set_level`**`(gpio_num_t gpio_num, uint32_t level)` – set value of `gpio_num` GPIO to `level` (0: low ; 1: high).

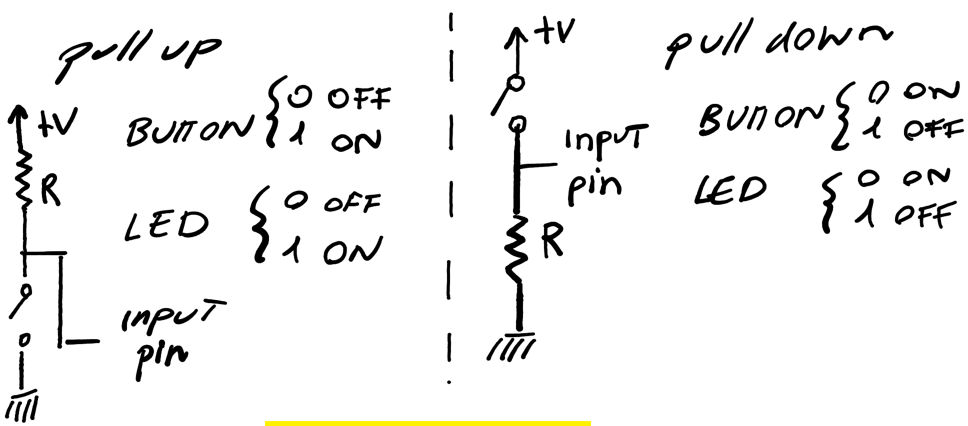
  Possible returned values are:

  - ESP_OK – in case of success;
  - ESP_ERR_INVALID_ARG – in case of GPIO number error;

GPIO – pull-up and pull-down resistors

GPIO have pull-up and pull-down configurable functions (actually, only pins that support both input & output have integrated pull-up and pull-down resistors, while input-only GPIOs 34-39 do not). To activate such resistors, use:

- `esp_err_t` **`gpio_set_pull_mode`**`(gpio_num_t gpio_num, gpio_pull_mode_t pull)` – configure GPIO pull-up/pull-down resistors.



possible pull parameter values are:

- GPIO_PULLUP_ONLY: pad pull up;
- GPIO_PULLDOWN_ONLY: pad pull down;
- GPIO_PULLUP_PULLDOWN: pad pull up + pull down;
- GPIO_FLOATING: pad floating

Possible returned values are:

- ESP_OK – in case of success;
- ESP_ERR_INVALID_ARG – in case of parameter error;

GPIO – multiple configuration

It’s possible to apply the same configuration to more than one GPIO using:

- `esp_err_t gpio_config(const gpio_config_t *pGPIOConfig)`

`pGPIOConfig` is a pointer to `gpio_config_t` struct which has the following parameters:

- *uint64_t* **pin_bit_mask**: set with bit mask, each bit maps to a GPIO;
- *gpio_mode_t* **mode**: set input/output mode;
- *gpio_pullup_t* **pull_up_en**: GPIO pull-up;
- *gpio_pulldown_t* **pull_down_en**: GPIO pull-down;
- *gpio_int_type_t* **intr_type**: GPIO interrupt type

Possible returned values are:

- ESP_OK – in case of success;
- ESP_ERR_INVALID_ARG – in case of parameter error;

GPIO – multiple configuration

Here is an example of multiple GPIO configuration:

```
…
//declare gpio_config_t struct;
gpio_config_t IO_config;

//select GPIO pin 6 and GPIO pin 8
IO_config.pin_bit_mask = ((1ULL<<GPIO_NUM_6) | (1ULL<<GPIO_NUM_8));

//define pin mode (INPUT)
IO_config.mode = GPIO_MODE_INPUT;

//enable pull-up resistors
IO_config.pull_up_en = GPIO_PULLUP_ENABLE;

//disable pull-down resistors
IO_config.pull_down_en = GPIO_PULLDOWN_DISABLE;

//disable interrupt
IO_config.intr_type = GPIO_INTR_DISABLE;

//apply configuration
gpio_config(&IO_config);
```

GPIO – events and interrupts

Embedded real-time systems have to take actions in response to events that originate from the environment and which will have different processing overhead and response time requirements.

Interrupts are normally used to detect hardware events and ***Interrupt Service Routines*** (*ISR*) are functions used to process such events. Actually, although written in software, an ISR is a hardware feature because is the hardware that controls which ISR will run, and when it will run.

For this reason, software defined tasks, which are unrelated to the hardware, will only run when there are no ISRs running; in other words the lowest priority interrupt will always interrupt the highest priority task, and there is no way for a task to pre-empt an ISR.

FreeRTOS provides separates API to be used from ISR: having a separate API for use in interrupts allows task code to be more efficient, ISR code to be more efficient, and interrupt entry to be simpler.

GPIO – interrupt configuration

It's possible to select GPIO interrupt type using:

```
esp_err_t gpio_set_intr_type(gpio_num_t gpio_num, gpio_int_type_t intr_type)
```

`intr_type` is a `gpio_int_type_t` enum which can assume one of the following possible values:

- GPIO_INTR_DISABLE (0): disable GPIO interrupt;
- GPIO_INTR_POSEDGE (1): interrupt on rising edge;
- GPIO_INTR_NEGEDGE (2): interrupt on falling edge;
- GPIO_INTR_ANYEDGE (3): interrupt both on rising and falling edge;
- GPIO_INTR_LOW_LEVEL (4): interrupt on input low level trigger;
- GPIO_INTR_HIGH_LEVEL (5): interrupt on input high level trigger.

Possible returned values are:

- ESP_OK – in case of success;
- ESP_ERR_INVALID_ARG – in case of parameter error;

GPIO – interrupt configuration

To install the driver's GPIO ISR handler service, which allows per-pin GPIO interrupt handlers, use the function:

esp_err_t **gpio_install_isr_service**(*int* intr_alloc_flags)

This function is incompatible with the alternative **gpio_isr_register()** function: if this last function is used, a single global ISR is registered for all GPIO interrupts, while if **gpio_install_isr_service()** is used, the ISR service provides a global GPIO ISR and individual pin handlers are registered via the **gpio_isr_handler_add()** function.

Possible returned values are:

- ESP_OK – in case of success;
- ESP_ERR_NO_MEM – if there is no memory to install this service;
- ESP_ERR_INVALID_STATE – if ISR service is already installed;
- ESP_ERR_NOT_FOUND – if no free interrupt is found with the specified flags;
- ESP_ERR_INVALID_ARG – in case of GPIO error.

(The function *void* **gpio_uninstall_isr_service**(*void*) uninstall the driver's GPIO ISR service) freeing related resources.

GPIO – interrupt configuration

Once the driver’s GPIO ISR handler service is installed, an ISR handler can be associated to a GPIO pin, using the function:

```
esp_err_t gpio_isr_handler_add(gpio_num_t gpio_num, gpio_isr_t
isr_handler, void *args)
```

`args` is (`void *`) to the parameter for ISR handler

`isr_handler` is the function that will be called on selected interrupt event (I.e. the Interrupt Service Routine) and which generally has the following prototype:

```
void isr_handler(void *args);
```

Possible returned values are:

- ESP_OK – in case of success;
- ESP_ERR_INVALID_STATE - wrong state, the ISR service has not been initialized;
- ESP_ERR_INVALID_ARG – in case of parameter error;

GPIO – deferred interrupt processing

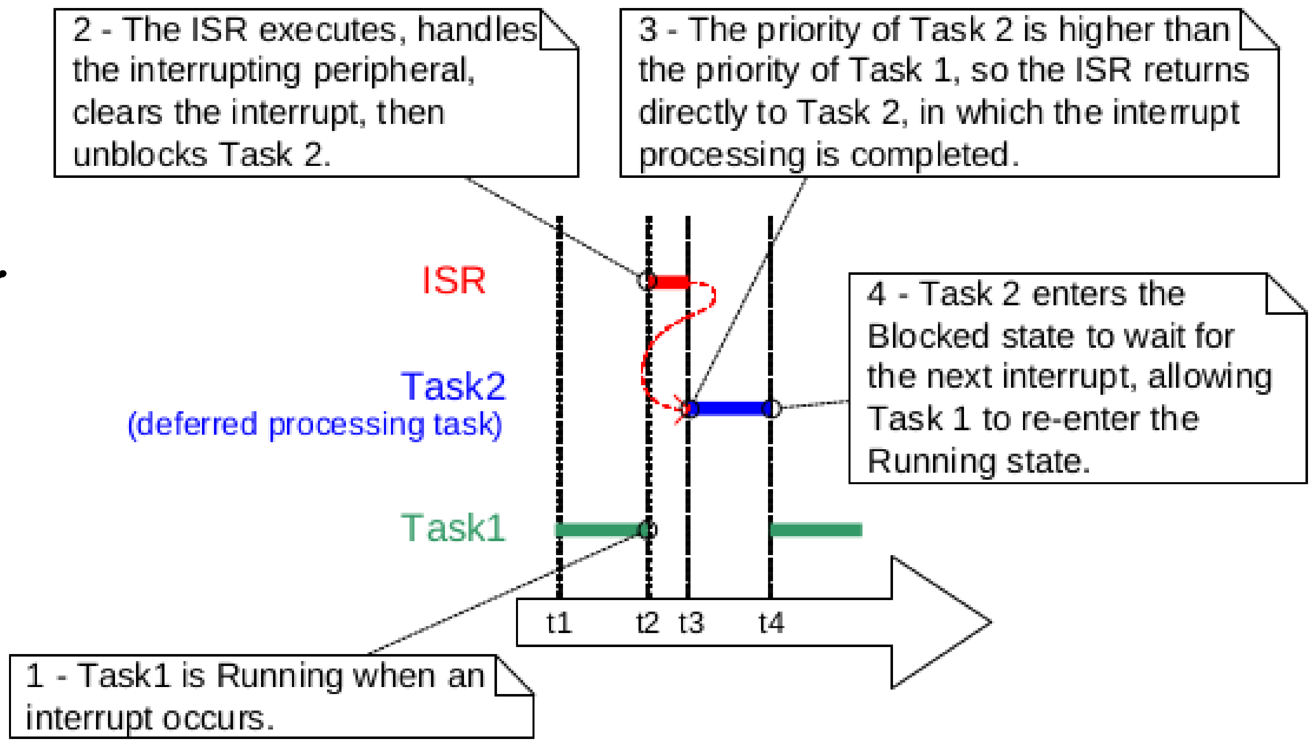
It is normally considered best practice to keep ISRs as short as possible. Reasons for this include:

- even if tasks have been assigned a very high priority, they will only run if no interrupts are being serviced by the hardware;
- ISRs can disrupt (add ‘jitter’ to) both the start time, and the execution time, of a task;
- depending on the architecture on which FreeRTOS is running, it might not be possible to accept any new interrupts while an ISR is executing but even if interrupts are allowed to nest, this increase complexity and reduce predictability;
- the application writer needs to consider the consequences of resources such as variables, peripherals, and memory buffers being accessed by a task and an ISR at the same time.

⇒ an ISR have just to record the cause of the interrupt, and clear the interrupt; any other processing necessitated by the interrupt can often be performed in a task, allowing the ISR to exit as quickly as is practical. This is called **deferred interrupt processing**, because the processing necessitated by the interrupt is ‘deferred’ from the ISR to a task.

GPIO – deferred interrupt processing

If the priority of the task to which interrupt processing is deferred is above the priority of any other task, then the processing will be performed immediately, just as if the processing had been performed in the ISR itself:



GPIO – deferred interrupt processing

There is no absolute rule as to when it is best to perform all processing necessitated by an interrupt in the ISR, and when it is best to defer part of the processing to a task. Deferring processing to a task is most useful when:

(banale)

- the processing necessitated by the interrupt is not trivial: for example, if the interrupt is just storing the result of an analog to digital conversion, then it is almost certain this is best performed inside the ISR, but if result of the conversion must also be passed through a software filter, then it may be best to execute the filter in a task;

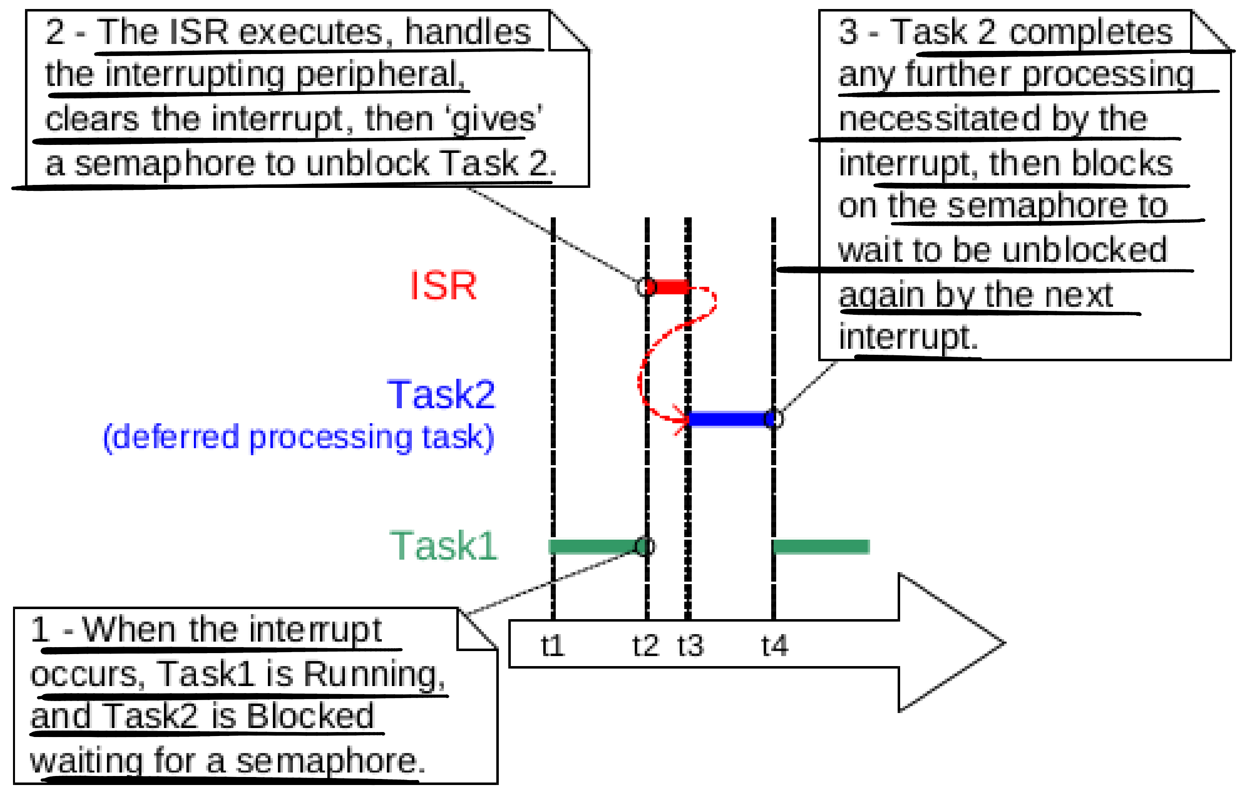
- it is convenient for the interrupt processing to perform an action that cannot be performed inside an ISR, such as write to a console, or allocate memory;

- the interrupt processing is not deterministic—meaning it is not known in advance how long the processing will take.

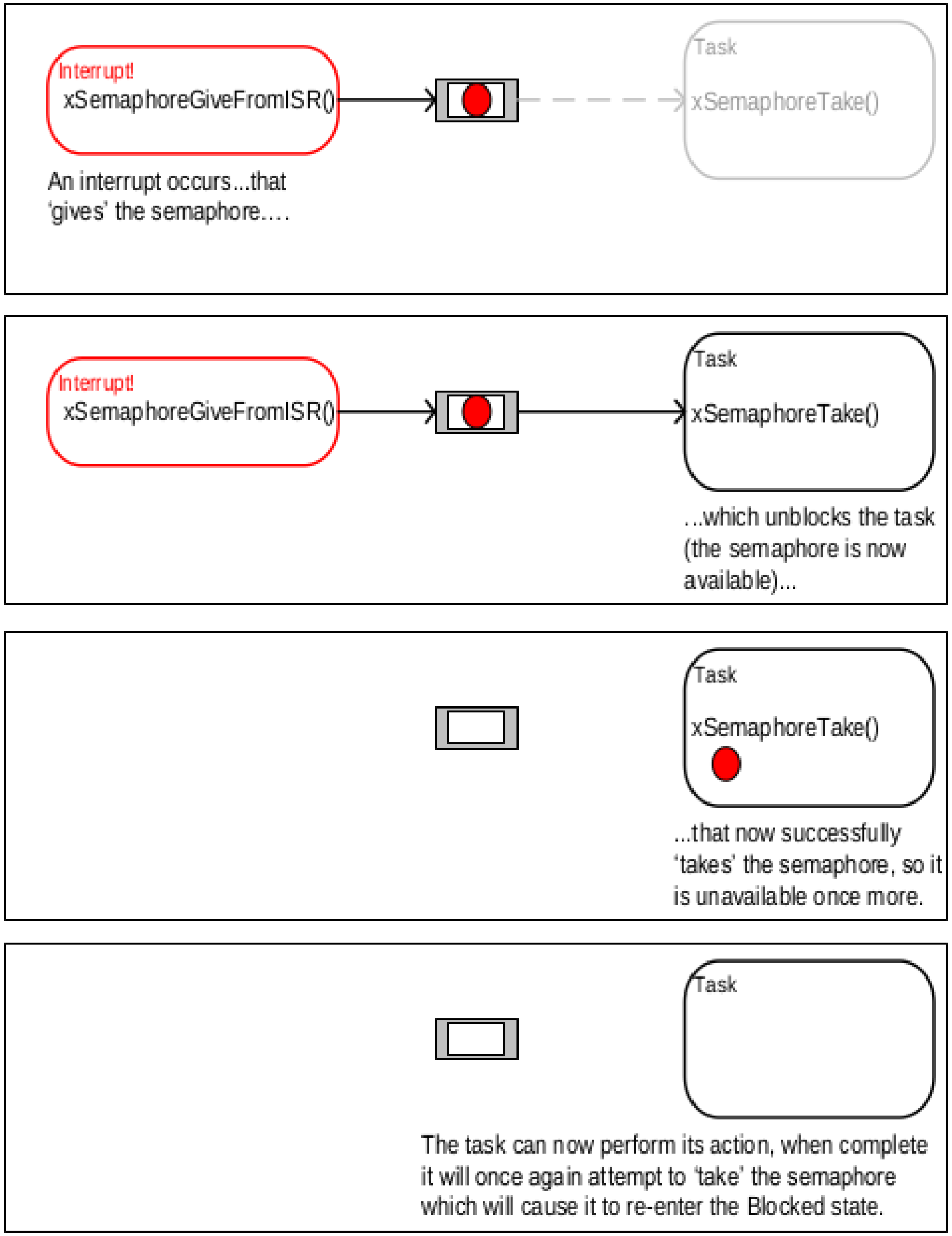
GPIO – binary semaphores

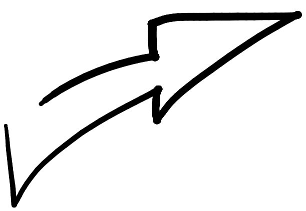
In order to implement deferred interrupt processing, binary semaphores can be used. A **binary semaphore** is a binary (i.e. two states) object that can be used to synchronize tasks:

- the deferred processing task uses a blocking ‘**take**’ call to a semaphore as a means of entering the Blocked state to wait for the event to occur;
- when the event occurs, the ISR uses a ‘**give**’ operation on the same semaphore to unblock the task so that the required event processing can proceed;



GPIO – binary semaphores





GPIO – binary semaphores

The functions to be used to manage binary semaphores are:

- `SemaphoreHandle_t` **`xSemaphoreCreateBinary()`** **-** this function creates a binary semaphore and returns an handle to it;
- **`xSemaphoreTake`**`(`*`SemaphoreHandle_t`*` xSemaphore, `*`TickType_t`*` xTicksToWait)`

`xTicksToWait` is the time in ticks to wait for the semaphore to become available. A block time of portMAX_DELAY can be used to block indefinitely.

`xSemaphore` is the bynary semaphore handler(the semaphore must be explicitly created before it can be used)

Taking the semaphore means to ‘obtain’ or ‘receive’ it, if available. The xSemaphoreTake() function must not be used from an interrupt service routine.

GPIO – binary semaphores

- **`xSemaphoreGiveFromISR`**`(`*`SemaphoreHandle_t`*` xSemaphore, `*`BaseType_t`*` pxHigherPriorityTaskWoken)`

it is possible that a single semaphore will have one or more tasks blocked on it waiting for the semaphore to become available.
Calling xSemaphoreGiveFromISR() can make the semaphore available, and so cause a task that was waiting for the semaphore to leave the Blocked state.
If calling xSemaphoreGiveFromISR() causes a task to leave the Blocked state, and the unblocked task has a priority higher than the currently executing task (the task that was interrupted), then, internally, xSemaphoreGiveFromISR() will set pxHigherPriorityTaskWoken to pdTRUE.
If xSemaphoreGiveFromISR() sets this value to pdTRUE, then normally a context switch should be performed before the interrupt is exited. This will ensure that the interrupt returns directly to the highest priority Ready state task.

(The binary semaphore can be 'given' using the xSemaphoreGiveFromISR() function (which is the interrupt safe version of xSemaphoreGive() function).)

GPIO – deferred interrupt example

```
#include <stdio.h>
#include "driver/gpio.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/semphr.h"

#define ESP_INTR_FLAG_DEFAULT 0

#define blueLed_GPIO 2
#define bootButton_GPIO 0

SemaphoreHandle_t xSemaphore = NULL;
uint32_t pause = 100;

// interrupt service routine, called when the button is pressed
void button_isr_handler(void* arg) {
    // notify the button task
    xSemaphoreGiveFromISR(xSemaphore, NULL);
}
...
```

sblocca il semaforo

GPIO – deferred interrupt example

```
...
// task that will react to button clicks
void button_task(void *arg) {
    while(1) {
        // wait for the notification from the ISR
        if(xSemaphoreTake(xSemaphore, portMAX_DELAY) == pdTRUE) {
            printf("Button pressed!\n");
            if(pause == 100)
                pause = 1000;
            else
                pause = 100;
        }
    }
}

void blink_task(void *arg) {
    while(1) {
        gpio_set_level(blueLed_GPIO, 1);
        vTaskDelay(pause / portTICK_RATE_MS);
        gpio_set_level(blueLed_GPIO, 0);
        vTaskDelay(pause / portTICK_RATE_MS);
    }
}
...
```

GPIO – deferred interrupt example

```
...
void app_main()
{
    // create the binary semaphore
    xSemaphore = xSemaphoreCreateBinary();

    // set the GPIO direction
    gpio_set_direction(bootButton_GPIO, GPIO_MODE_INPUT);
   gpio_set_direction(blueLed_GPIO, GPIO_MODE_OUTPUT);

    // enable interrupt on falling (1->0) edge for button pin
    gpio_set_intr_type(bootButton_GPIO, GPIO_INTR_NEGEDGE);

    // install ISR service with default configuration
    gpio_install_isr_service(ESP_INTR_FLAG_DEFAULT);

    // attach the interrupt service routine
    gpio_isr_handler_add(bootButton_GPIO, button_isr_handler, NULL);

    // start the task that will handle the button and the blink task
    xTaskCreate(button_task, "button_task", 2048, NULL, 2, NULL);
    xTaskCreate(blink_task, "blink_task", 2048, NULL, 2, NULL);
}
```