Introduction to Internet of Things

Perception Layer - Part 2

RTOS fundamentals

https://www.freertos.org

Content

1. Introduction
2. Justification
3. Tasks
4. Mutual exclusion
5. Queues
6. Practice

1. Introduction

What does “Real Time” mean in RTOS?

“Real-time" refers to the system’s ability to process data and respond to events within strict, predefined time constraints (deadlines). It emphasizes determinism (predictable timing) rather than raw speed, ensuring tasks complete within milliseconds or microseconds to prevent system failure. Key aspects of real-time in an RTOS include:

- Predictability (Deterministic Behavior): The OS guarantees that critical tasks will be completed within a specific timeframe, every time.
- Deadline Management: The correctness of the system depends not only on the logical result but also on the time the result is delivered. Missing a deadline is considered a failure.
- Hard vs. Soft Real-Time:
- Hard Real-Time: Missing a deadline results in catastrophic failure (e.g., airbags, flight control).
- Soft Real-Time: Missing a deadline reduces quality but is not catastrophic (e.g., video streaming, user interface updates).
- Event-Driven & Preemptive: The system can immediately prioritize critical tasks over less important ones, pausing lower-priority tasks (preemption) to handle urgent events.
- Low Jitter: RTOS ensures consistent, precise timing with minimal variation in task execution time.
- Real-time does not necessarily mean "instantaneous," but rather that the response occurs within the time necessary to control the environment.

1. Introduction

What is a Real Time Operating System?

A Real Time Operating System is an operating system that is optimised for use in embedded/real time applications. Their primary objective is to ensure a timely and deterministic response to events. An event can be external, like a limit switch being hit, or internal like a character being received.

Using a real time operating system allows a software application to be written as a set of independent tasks. Each task is assigned a priority and it is the responsibility of the Real Time Operating System to ensure that the task with the highest priority that is able to run is the task that is running. Examples of when a task may not be able to run include when a task is waiting for an external event to occur, or when a task is waiting for a fixed time period.

1. Introduction

What is a Real Time Kernel?

A Real Time Operating System can provide many resources to application writers - including TCP/IP stacks, files systems, etc. The Kernel is the part of the operating system that is responsible for task management, and intertask communication and synchronisation. FreeRTOS is a real time kernel.

What is a Real Time Scheduler?

Real Time Scheduler and Real Time Kernel are sometimes used interchangeably. Specifically, the Real Time Scheduler is the part of the RTOS kernel that is responsible for deciding which task should be executing.

2. Justification

Why use an RTOS?

You do not need to use an RTOS to write good embedded software. At some point though, as your application grows in size or complexity, the services of an RTOS might become beneficial for one or more of the reasons listed below. These are not absolutes, but opinion. As with everything else, selecting the right tools for the job in hand is an important first step in any project.

- Abstract out timing information
- Maintainability/Extensibility
- Modularity
- Cleaner interfaces
- Easier testing (in some cases)
- Code reuse
- Improved efficiency?
- Idle time
- Flexible interrupt handling
- Mixed processing requirements
- Easier control over peripherals

3 .Tasks

Tasks are implemented as C functions. Tasks must implement the expected function prototype shown in Listing 4.1. which takes a void pointer parameter and returns void.

```
void vATaskFunction( void * pvParameters );
```

3. Tasks

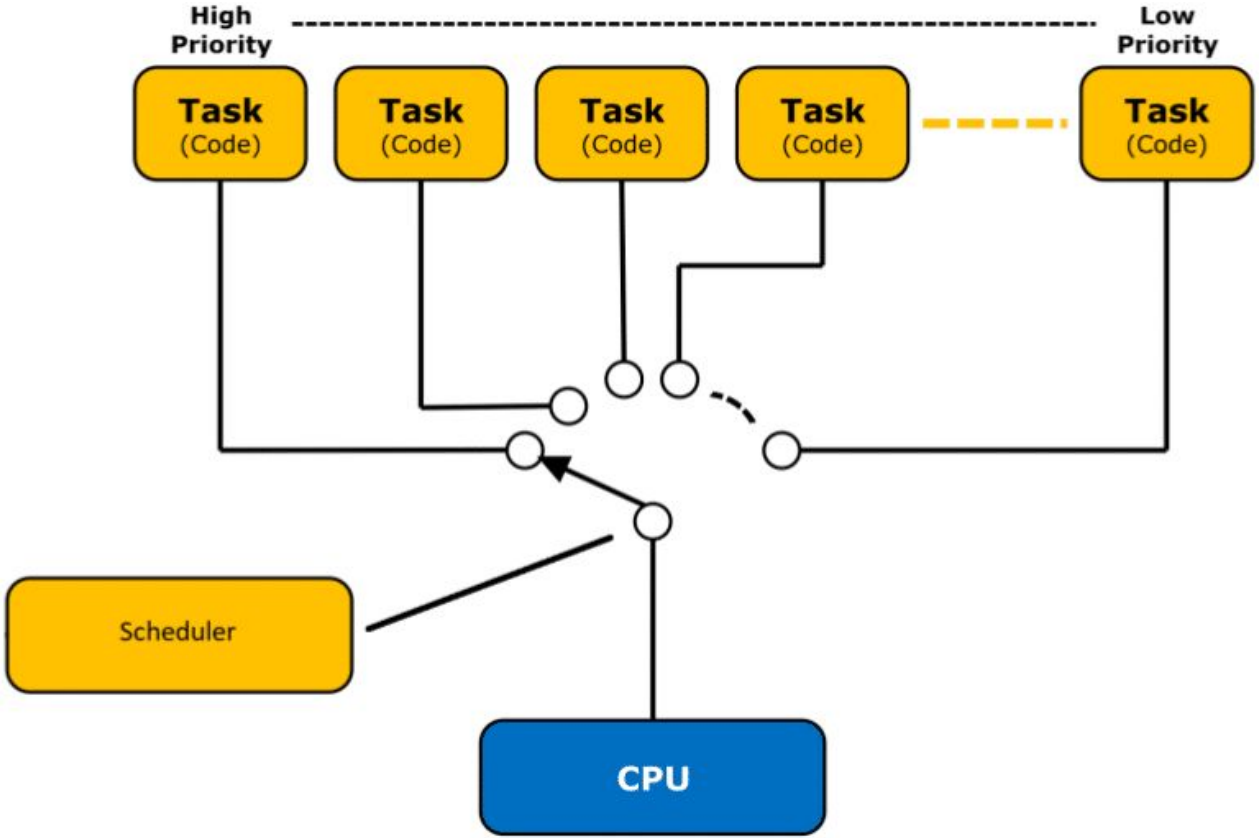
Tasks models process as an infinite loop

```
void vATaskFunction( void * pvParameters )
{
    /*
     * Stack-allocated variables can be declared normally when inside a function.
     * Each instance of a task created using this example function will have its
     * own separate instance of lStackVariable allocated on the task's stack.
     */
    long lStackVariable = 0;

    /*
     * In contrast to stack allocated variables, variables declared with the
`static`
     * keyword are allocated to a specific location in memory by the linker.
     * This means that all tasks calling vATaskFunction will share the same
     * instance of lStaticVariable.
     */
    static long lStaticVariable = 0;

    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /*
     * If the task implementation ever exits the above loop, then the task
     * must be deleted before reaching the end of its implementing function.
     * When NULL is passed as a parameter to the vTaskDelete() API function,
     * this indicates that the task to be deleted is the calling (this) task.
     */
    vTaskDelete( NULL );
}
```

3. Tasks

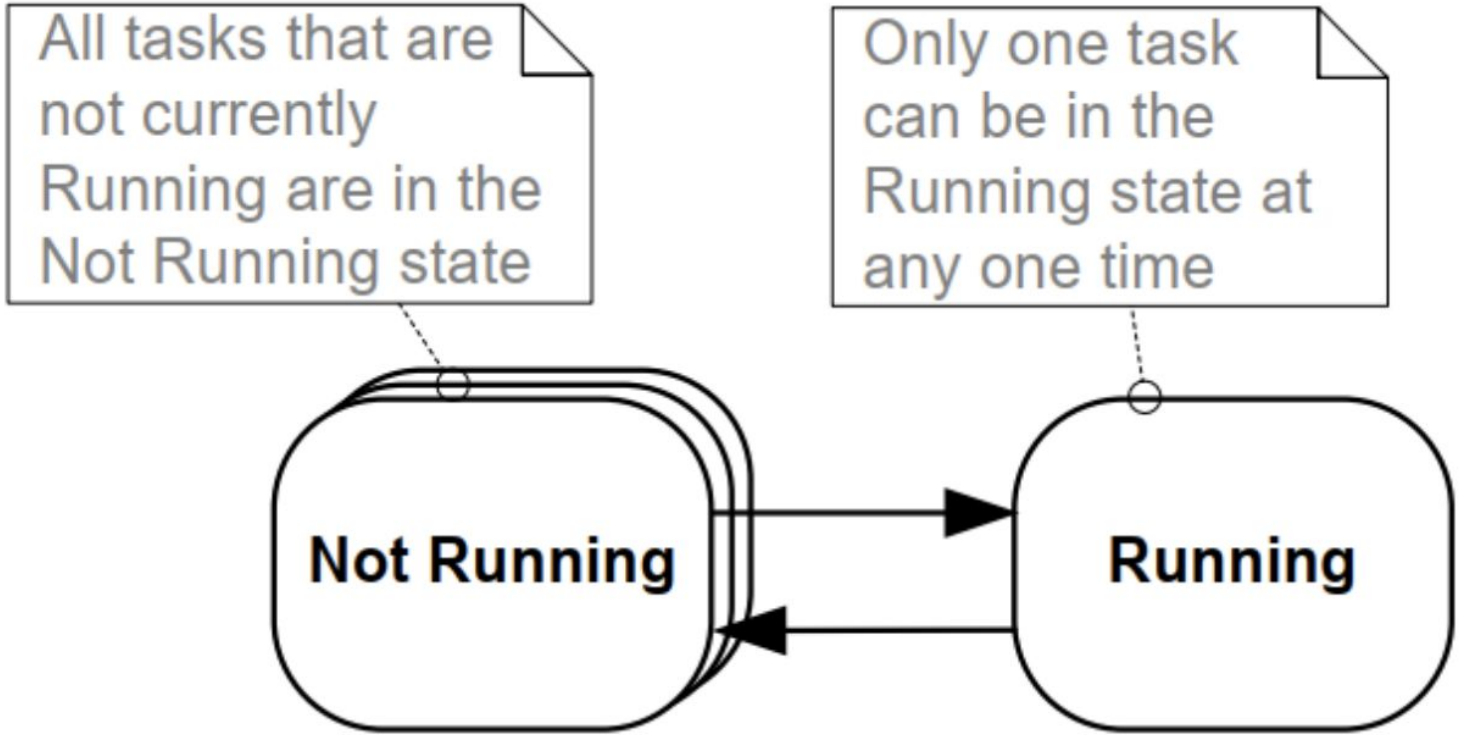
Scheduler

CPU as shared resource



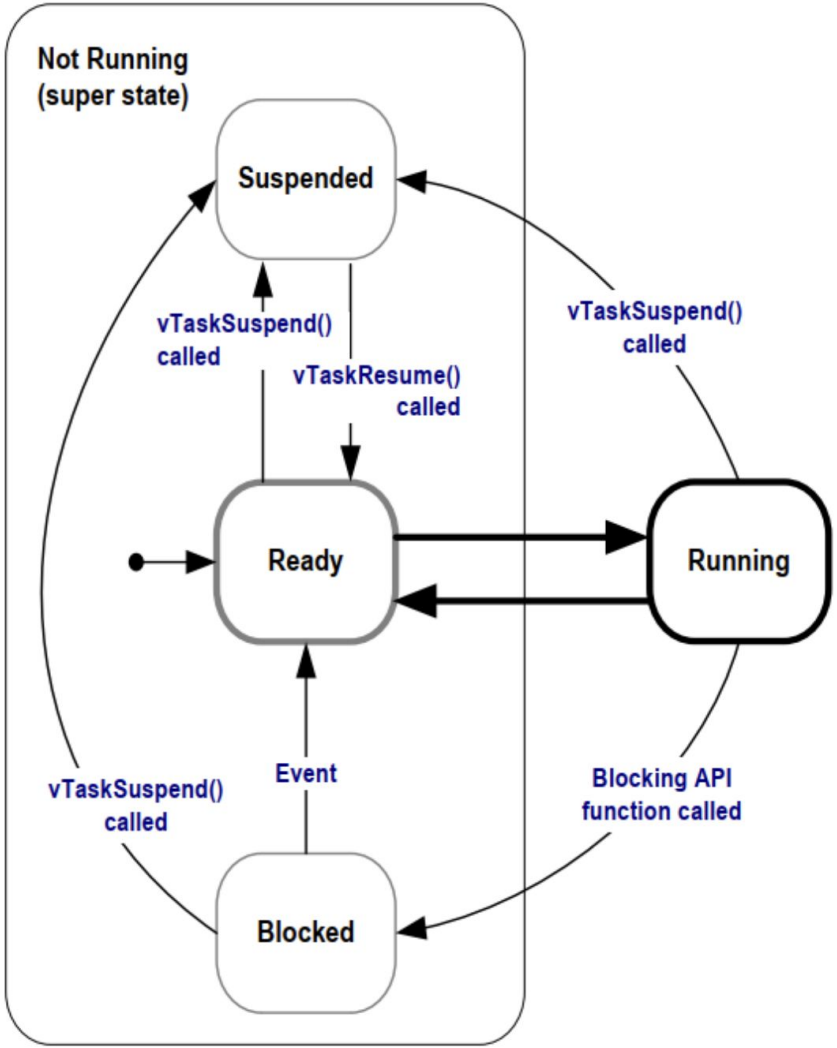
3 .Tasks

Task execution …



3. Tasks

Task execution … A more detailed scheme



3. Task

Task implementation

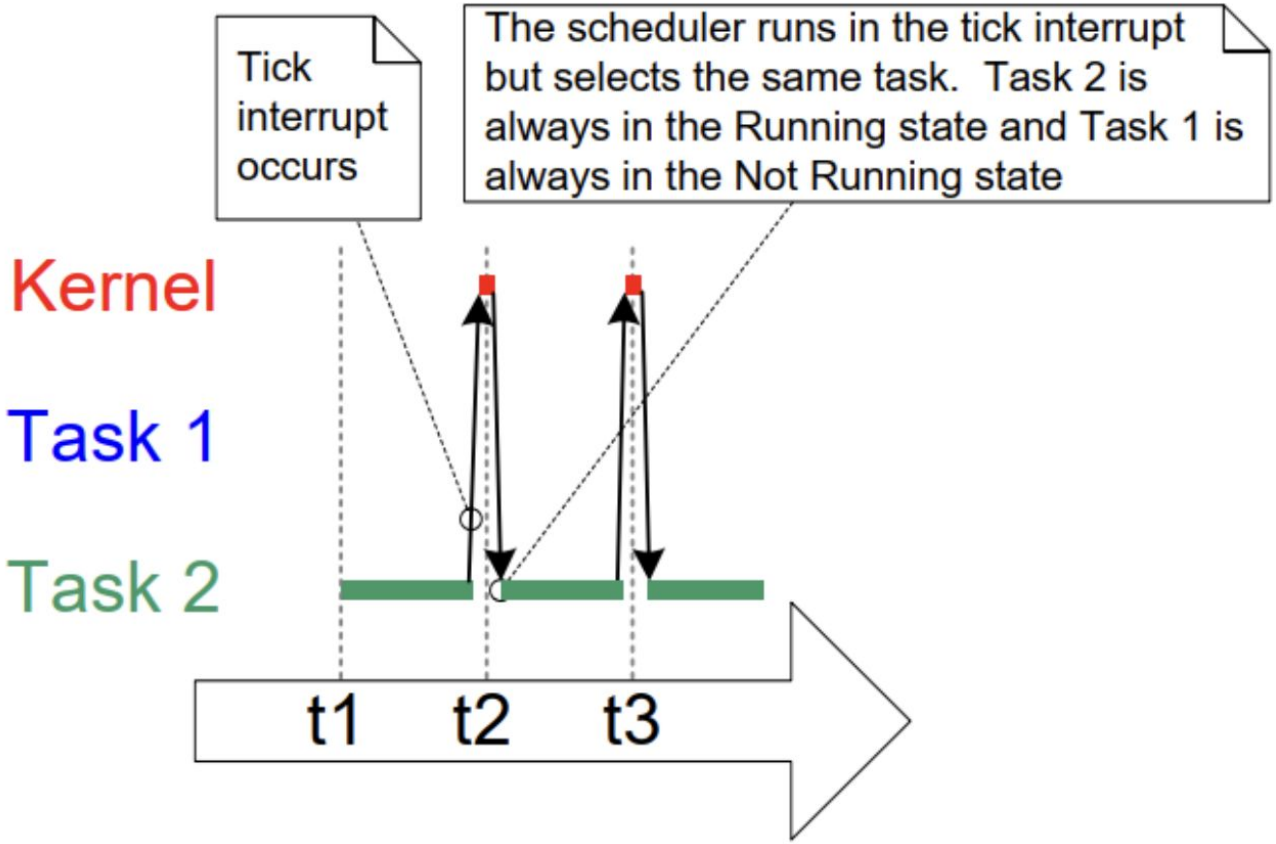
```
/*
 * Define the strings that will be passed in as the task parameters.
 * These are defined const and not on the stack to ensure they remain valid
 * when the tasks are executing.
 */
static const char * pcTextForTask1 = "Task 1 is running";
static const char * pcTextForTask2 = "Task 2 is running";

int main( void )
{
    /* Create the first task with a priority of 1. */
    xTaskCreate( vTaskFunction,             /* Task Function    */
                 "Task 1",                  /* Task Name        */
                 1000,                      /* Task Stack Depth */
                 ( void * ) pcTextForTask1, /* Task Parameter   */
                 1,                         /* Task Priority    */
                 NULL );

    /* Create the second task at a higher priority of 2. */
    xTaskCreate( vTaskFunction,             /* Task Function    */
                 "Task 2",                  /* Task Name        */
                 1000,                      /* Task Stack Depth */
                 ( void * ) pcTextForTask2, /* Task Parameter   */
                 2,                         /* Task Priority    */
                 NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

    /* Will not reach here. */
    return 0;
}
```

4. Tasks

Pay attention to priorities

Task 1: Priority 1

Task 2: Priority 2



3. Tasks

Periodic task, with “CPU deliver”

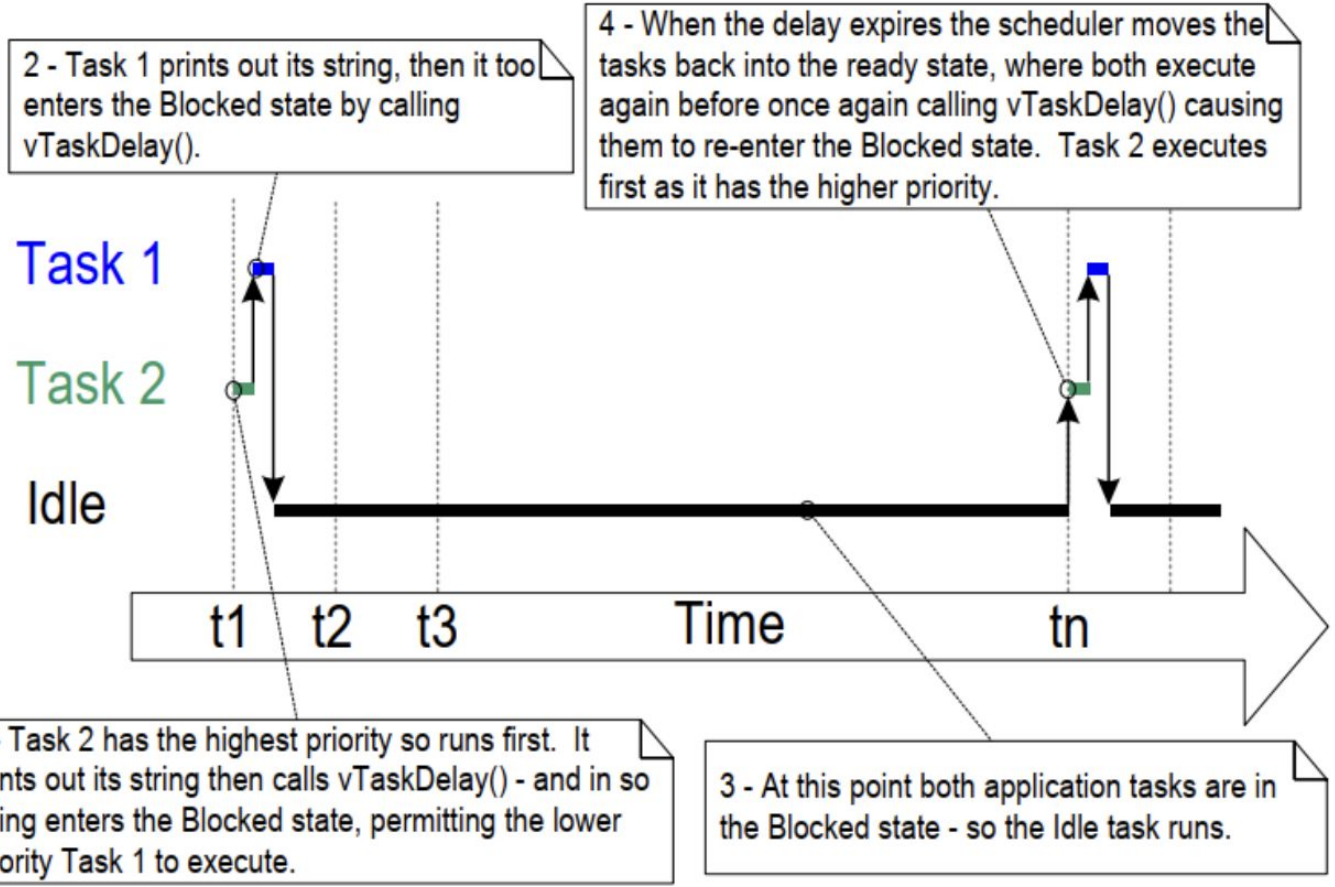
```
void vTaskFunction( void * pvParameters )
{
    char * pcTaskName;
    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /*
     * The string to print out is passed in via the parameter. Cast this to a
     * character pointer.
     */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintLine( pcTaskName );

        /*
         * Delay for a period. This time a call to vTaskDelay() is used which
         * places the task into the Blocked state until the delay period has
         * expired. The parameter takes a time specified in 'ticks', and the
         * pdMS_TO_TICKS() macro is used (where the xDelay250ms constant is
         * declared) to convert 250 milliseconds into an equivalent time in
         * ticks.
         */
        vTaskDelay( xDelay250ms );
    }
}
```

3. Tasks

Same function

2 tasks

Round robin



3. Tasks

Continuous processing tasks

```
void vContinuousProcessingTask( void * pvParameters )
{
    char * pcTaskName;

    /*
     * The string to print out is passed in via the parameter. Cast this to a
     * character pointer.
     */
    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /*
         * Print out the name of this task. This task just does this repeatedly
         * without ever blocking or delaying.
         */
        vPrintLine( pcTaskName );
    }
}
```

3. Tasks

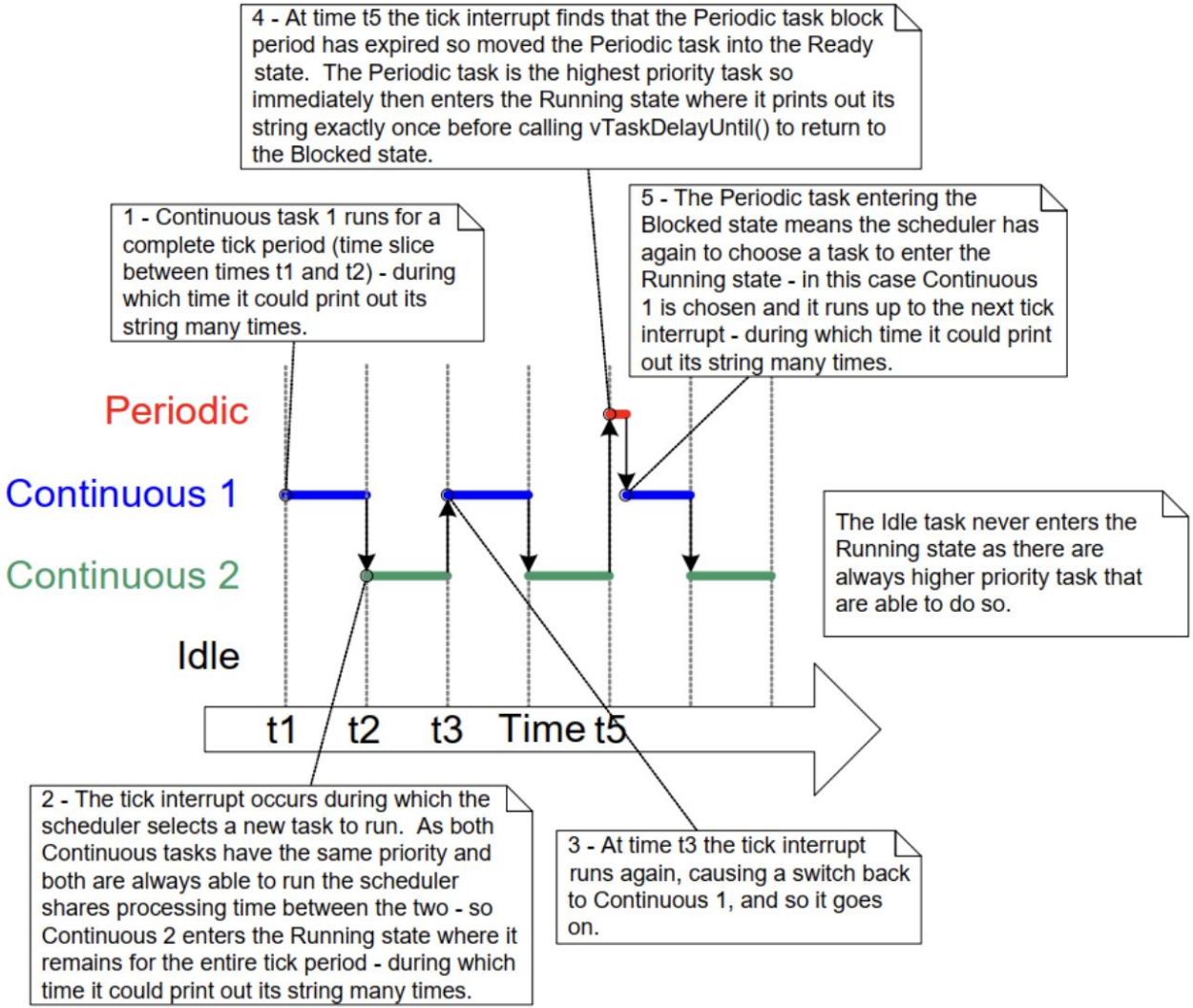
Periodic task

```
void vPeriodicTask( void * pvParameters )
{
    TickType_t xLastWakeTime;

    const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

    /*
     * The xLastWakeTime variable needs to be initialized with the current tick
     * count. Note that this is the only time the variable is explicitly
     * written to. After this xLastWakeTime is managed automatically by the
     * vTaskDelayUntil() API function.
     */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintLine( "Periodic task is running" );

        /*
         * The task should execute every 3 milliseconds exactly - see the
         * declaration of xDelay3ms in this function.
         */
        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
    }
}
```

3. Tasks

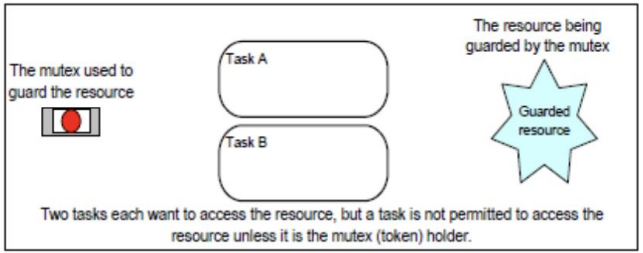
Continuous processing task and Periodic tasks execution

What about tasks priorities?

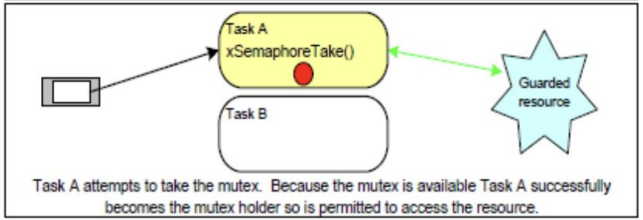


4. Mutual exclusion

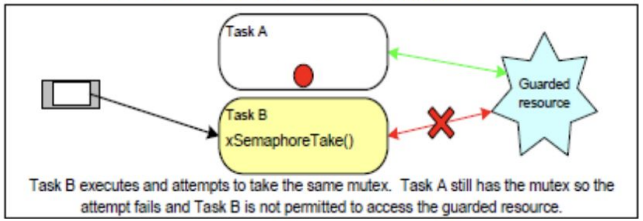
For a task to access the resource legitimately, it must first successfully 'take' the token (be the token holder). When the token holder has finished with the resource, it must 'give' the token back. Only when the token has been returned can another task successfully take the token, and then safely access the same shared resource. A task is not permitted to access the shared resource unless it holds the token.



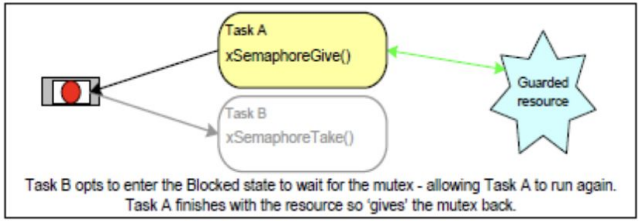
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.



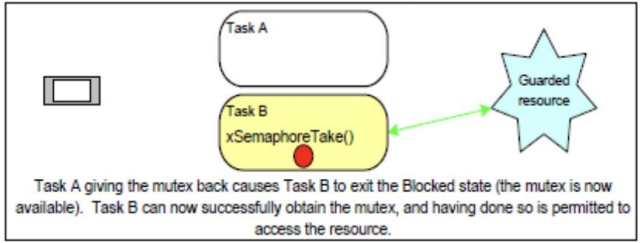
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.



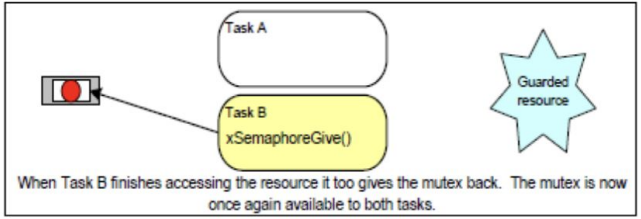
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.



Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.



Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.



When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

4. Mutex

This is a more complete example with:

- ESP32
- Platformio
- Espidf (no Arduino) style

Header part of main file

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

#define TAG "MUTEX_EXAMPLE"

SemaphoreHandle_t xMutex;

int sharedCounter = 0;
```

4. Mutex

Create “competitive” tasks

```c
void task1(void *pvParameters)
{
    while (1)
    {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE)
        {
            int local = sharedCounter;
            vTaskDelay(pdMS_TO_TICKS(100)); // Simula trabajo
            sharedCounter = local + 1;

            ESP_LOGI(TAG, "Task1 incrementó contador a %d", sharedCounter);

            xSemaphoreGive(xMutex);
        }

        vTaskDelay(pdMS_TO_TICKS(500));
    }
}
```

```c
void task2(void *pvParameters)
{
    while (1)
    {
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE)
        {
            int local = sharedCounter;
            vTaskDelay(pdMS_TO_TICKS(100)); // Simula trabajo
            sharedCounter = local + 1;

            ESP_LOGI(TAG, "Task2 incrementó contador a %d", sharedCounter);

            xSemaphoreGive(xMutex);
        }

        vTaskDelay(pdMS_TO_TICKS(700));
    }
}
```

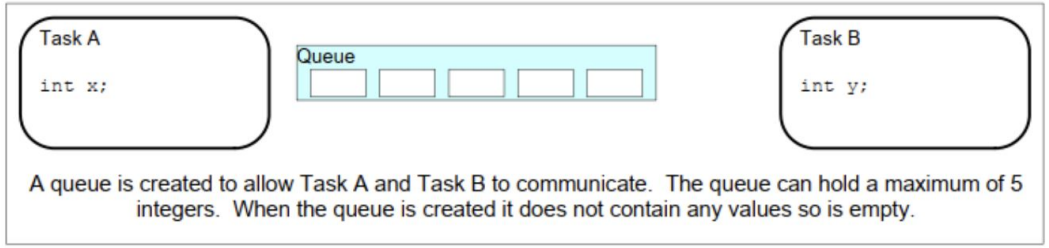
4. Mutex

main function

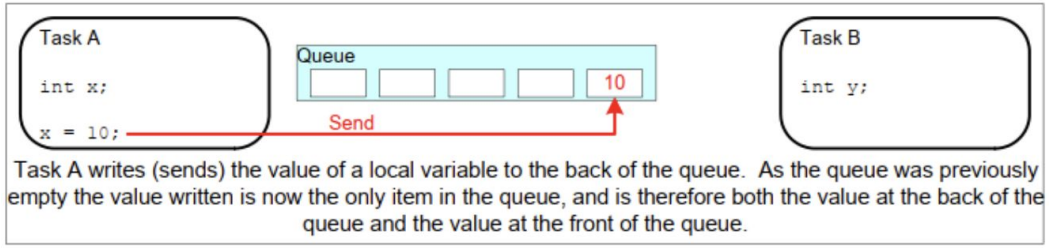
```
void app_main(void)
{
    xMutex = xSemaphoreCreateMutex();

    if (xMutex == NULL)
    {
        ESP_LOGE(TAG, "Error creando el mutex");
        return;
    }

    xTaskCreate(task1, "Task1", 2048, NULL, 2, NULL);
    xTaskCreate(task2, "Task2", 2048, NULL, 2, NULL);
}
```

5. Queues

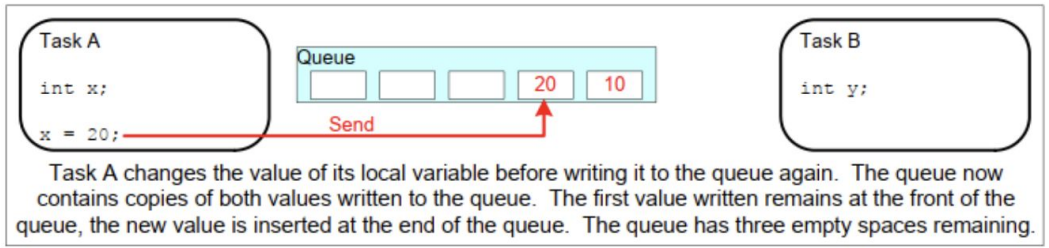
Queues are normally used as First In First Out (FIFO) buffers, where data is written to the end (tail) of the queue and removed from the front (head) of the queue. Figure 5.1 demonstrates data being written to and read from a queue that is being used as a FIFO. It is also possible to write to the front of a queue, and to overwrite data that is already at the front of a queue.



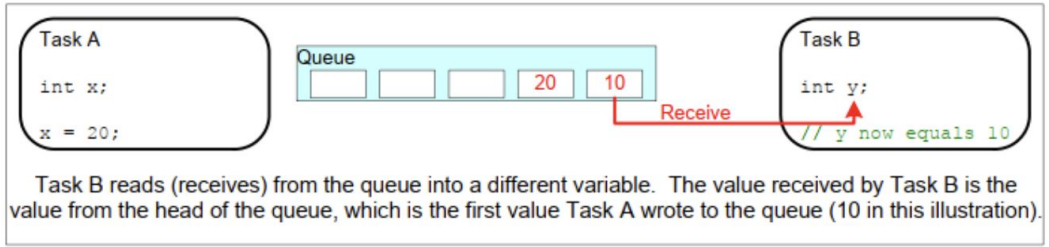
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

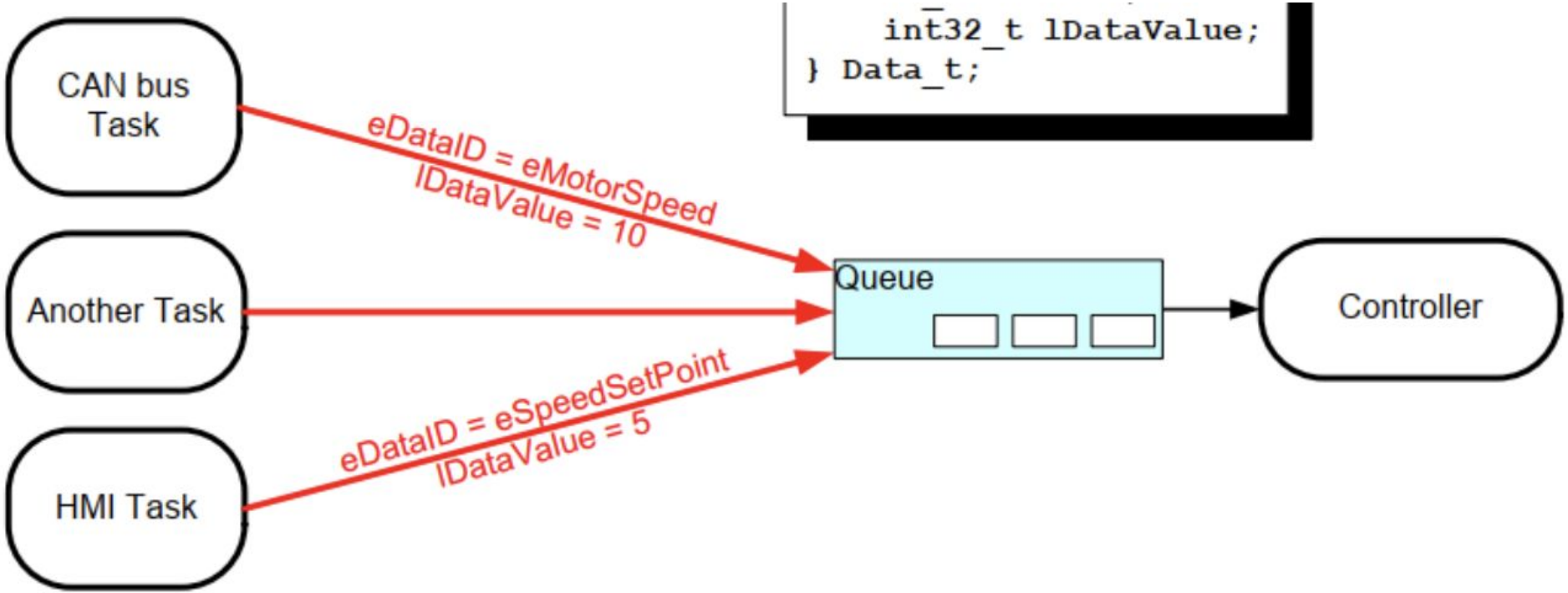


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

5. Queues


int32_t lDataValue;
} Data_t;
CAN bus Task
Another Task
HMI Task
eDataID = eMotorSpeed
lDataValue = 10
eDataID = eSpeedSetPoint
lDataValue = 5
Queue
Controller

5. Queues

This is a more complete example with:

- ESP32
- Platformio
- Espidf (no Arduino) style

Header part of main file

```c
#include <stdio.h>
#include <string.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "esp_log.h"

#define TAG "QUEUE_EXAMPLE"

#define QUEUE_LENGTH 10

QueueHandle_t dataQueue;
```

```c
typedef enum {
    DATA_INT,
    DATA_STRING
} data_type_t;

typedef struct {
    data_type_t type;
    union {
        int intValue;
        char strValue[32];
    } data;
} queue_message_t;
```

5. Queues

Create sender tasks

```c
void producerTask1(void *pvParameters)
{
    queue_message_t msg;
    int counter = 0;

    while (1)
    {
        msg.type = DATA_INT;
        msg.data.intValue = counter++;

        if (xQueueSend(dataQueue, &msg, portMAX_DELAY) == pdPASS)
        {
            ESP_LOGI(TAG, "Task1 envió entero: %d", msg.data.intValue);
        }

        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

```c
void producerTask2(void *pvParameters)
{
    queue_message_t msg;
    int counter = 0;

    while (1)
    {
        msg.type = DATA_STRING;
        snprintf(msg.data.strValue, sizeof(msg.data.strValue),
                 "Mensaje %d", counter++);

        if (xQueueSend(dataQueue, &msg, portMAX_DELAY) == pdPASS)
        {
            ESP_LOGI(TAG, "Task2 envió string: %s", msg.data.strValue);
        }

        vTaskDelay(pdMS_TO_TICKS(1500));
    }
}
```

5. Queues

Create receiver task

```
void consumerTask(void *pvParameters)
{
    queue_message_t receivedMsg;

    while (1)
    {
        if (xQueueReceive(dataQueue, &receivedMsg, portMAX_DELAY) == pdPASS)
        {
            switch (receivedMsg.type)
            {
                case DATA_INT:
                    ESP_LOGI(TAG, "Consumer recibió entero: %d",
                             receivedMsg.data.intValue);
                    break;

                case DATA_STRING:
                    ESP_LOGI(TAG, "Consumer recibió string: %s",
                             receivedMsg.data.strValue);
                    break;
            }
        }
    }
}
```

5. Queues

main function

```
void app_main(void)
{
    dataQueue = xQueueCreate(QUEUE_LENGTH, sizeof(queue_message_t));

    if (dataQueue == NULL)
    {
        ESP_LOGE(TAG, "Error creando la cola");
        return;
    }

    xTaskCreate(producerTask1, "Producer1", 2048, NULL, 2, NULL);
    xTaskCreate(producerTask2, "Producer2", 2048, NULL, 2, NULL);
    xTaskCreate(consumerTask,  "Consumer",  2048, NULL, 1, NULL);
}
```

6. Practice

- Install Platformio complement on VSCode
- Create and test Round-Robin application with 2 tasks
  - Printing messages
- Create and test Mutex application
  - Printing a counter
- Create a Queue application
  - Use the example

6. Practice

GPIO configuration

File header

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/gpio.h"
#include "esp_log.h"

#define TAG "LED_BUTTON"

#define LED_GPIO     2
#define BUTTON_GPIO  4

QueueHandle_t buttonQueue;
```

6. Practice

GPIO configuration

```
void gpio_init(void)
{
    // LED como salida
    gpio_config_t led_conf = {
        .pin_bit_mask = (1ULL << LED_GPIO),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    gpio_config(&led_conf);

    // Botón como entrada con pull-up interno
    gpio_config_t button_conf = {
        .pin_bit_mask = (1ULL << BUTTON_GPIO),
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_ENABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    gpio_config(&button_conf);
}
```

6. Practice

Task of button

```c
void taskButton(void *pvParameters)
{
    int lastState = 1;  // Pull-up → 1 = no presionado
    int currentState;

    while (1)
    {
        currentState = gpio_get_level(BUTTON_GPIO);

        // Detectar flanco de bajada
        if (lastState == 1 && currentState == 0)
        {
            ESP_LOGI(TAG, "Botón presionado");
            xQueueSend(buttonQueue, &currentState, portMAX_DELAY);

            vTaskDelay(pdMS_TO_TICKS(200)); // debounce simple
        }

        lastState = currentState;
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}
```

6. Practice

Task led

```
void taskButton(void *pvParameters)
{
    int lastState = 1;  // Pull-up → 1 = no presionado
    int currentState;

    while (1)
    {
        currentState = gpio_get_level(BUTTON_GPIO);

        // Detectar flanco de bajada
        if (lastState == 1 && currentState == 0)
        {
            ESP_LOGI(TAG, "Botón presionado");
            xQueueSend(buttonQueue, &currentState, portMAX_DELAY);

            vTaskDelay(pdMS_TO_TICKS(200)); // debounce simple
        }

        lastState = currentState;
        vTaskDelay(pdMS_TO_TICKS(50));
    }
}
```

6. Practice

main function

```
void app_main(void)
{
    gpio_init();

    buttonQueue = xQueueCreate(5, sizeof(int));

    xTaskCreate(taskButton, "TaskButton", 2048, NULL, 2, NULL);
    xTaskCreate(taskLed,    "TaskLed",    2048, NULL, 2, NULL);
}
```

6. Practice

Interrupt version

```c
#include <stdio.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/gpio.h"
#include "esp_log.h"

#define TAG "LED_ISR"

#define LED_GPIO     2
#define BUTTON_GPIO  4

QueueHandle_t buttonQueue;
```

6. Practice

Interrupt Sub-Routine

```
static void IRAM_ATTR button_isr_handler(void* arg)
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;
    int event = 1;

    xQueueSendFromISR(buttonQueue, &event, &xHigherPriorityTaskWoken);

    if (xHigherPriorityTaskWoken)
    {
        portYIELD_FROM_ISR();
    }
}
```

6. Practice

Init GPIOs

```c
void gpio_init(void)
{
    // LED salida
    gpio_config_t led_conf = {
        .pin_bit_mask = (1ULL << LED_GPIO),
        .mode = GPIO_MODE_OUTPUT,
        .pull_up_en = GPIO_PULLUP_DISABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_DISABLE
    };
    gpio_config(&led_conf);

    // Botón entrada con pull-up e interrupción
    gpio_config_t button_conf = {
        .pin_bit_mask = (1ULL << BUTTON_GPIO),
        .mode = GPIO_MODE_INPUT,
        .pull_up_en = GPIO_PULLUP_ENABLE,
        .pull_down_en = GPIO_PULLDOWN_DISABLE,
        .intr_type = GPIO_INTR_NEGEDGE  // flanco descendente
    };
    gpio_config(&button_conf);

    gpio_install_isr_service(0);
    gpio_isr_handler_add(BUTTON_GPIO, button_isr_handler, NULL);
}
```

6. Practice

LED task

```c
void taskLed(void *pvParameters)
{
    int received;
    int ledState = 0;

    while (1)
    {
        if (xQueueReceive(buttonQueue, &received, portMAX_DELAY))
        {
            ledState = !ledState;
            gpio_set_level(LED_GPIO, ledState);

            ESP_LOGI(TAG, "LED cambiado a: %d", ledState);

            // Debounce simple por software
            vTaskDelay(pdMS_TO_TICKS(200));
        }
    }
}
```

6. Practice

main function

```
void app_main(void)
{
    gpio_init();

    buttonQueue = xQueueCreate(10, sizeof(int));

    if (buttonQueue == NULL)
    {
        ESP_LOGE(TAG, "Error creando la cola");
        return;
    }

    xTaskCreate(taskLed, "TaskLed", 2048, NULL, 2, NULL);
}
```