

Arquitecturas SW y RTOS

Javier Valls

Índice

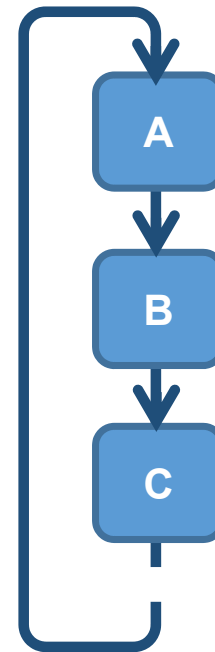
- Arquitecturas SW
 - Round-robin
 - Round-robin con interrupciones
 - Sistema operativo en tiempo real
- FreeRTOS
 - Introducción RTOS
 - Tareas, sus estados y el planificador de tareas
 - Semáforo binario: sincronización de tareas
 - Semáforo binario: sincronización entre ISR y tarea
 - Semáforo mutex: acceso a datos compartidos
 - Colas: comunicación entre tareas

Arquitecturas SW

- Sistemas embebidos
 - Interactúan con el entorno a través de sensores y actuadores
 - Realizan control, procesamiento, comunicación,...
 - Gestionan múltiples tareas con diferentes (prioridades) restricciones de tiempo de respuesta
 - responder a eventos en un determinado tiempo
 - “Hard real-time deadline”: el sistema falla si no responde al evento en el tiempo especificado
- Arquitectura SW: organización del sistema
 - ¿Cómo se detectan los eventos?
 - ¿Cómo se accede a la rutina que responde al evento?
- Selección de la arquitectura SW
 - ¿Cuánto control quiero tener sobre la respuesta del sistema a un evento?

Arquitecturas SW: Round-robin

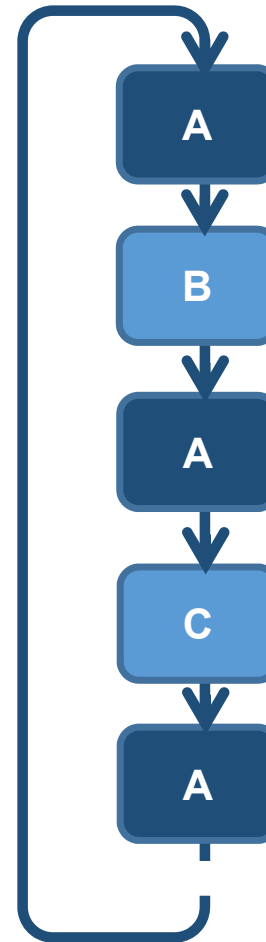
- Arquitectura simple válida para aplicaciones sencillas
- Todos los eventos tienen la misma prioridad
- Ejecución por consulta (“polling”)
 - No hay interrupciones
- Peor caso de tiempo de ejecución:
 - Suma del tiempo de todos los procesos
- Ampliar el sistema incluyendo más eventos cambia el tiempo de respuesta



```
while (TRUE)
{
    if (eventoA)
        procesaEventoA();
    if (eventoB)
        procesaEventoB();
    if (eventoC)
        procesaEventoC();
    ...
}
```

Arquitecturas SW: Round-robin

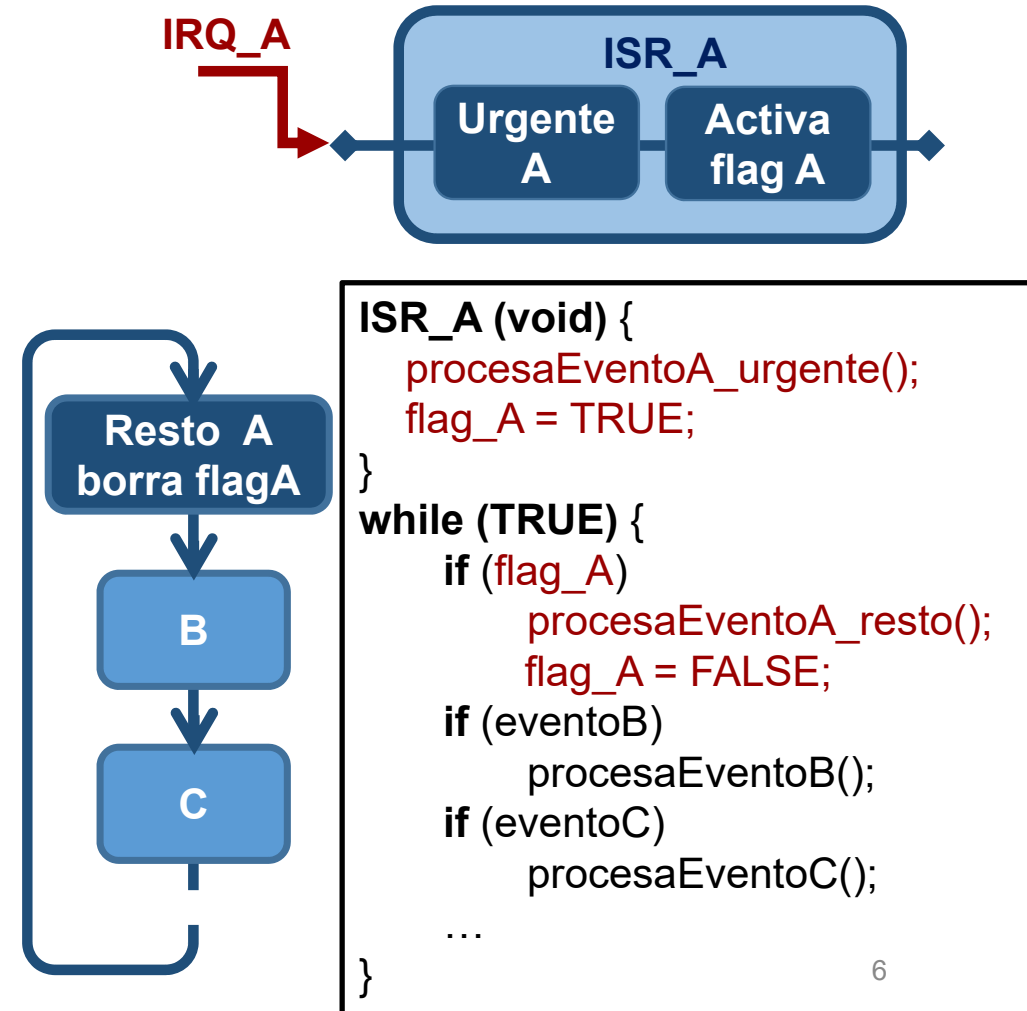
- Se puede mejorar el tiempo de respuesta de un evento concreto (darle más prioridad) evaluándolo más veces.



```
while (TRUE)
{
    if (eventoA)
        procesaEventoA();
    if (eventoB)
        procesaEventoB();
    if (eventoA)
        procesaEventoA();
    if (eventoC)
        procesaEventoC();
    if (eventoA)
        procesaEventoA();
    ...
}
```

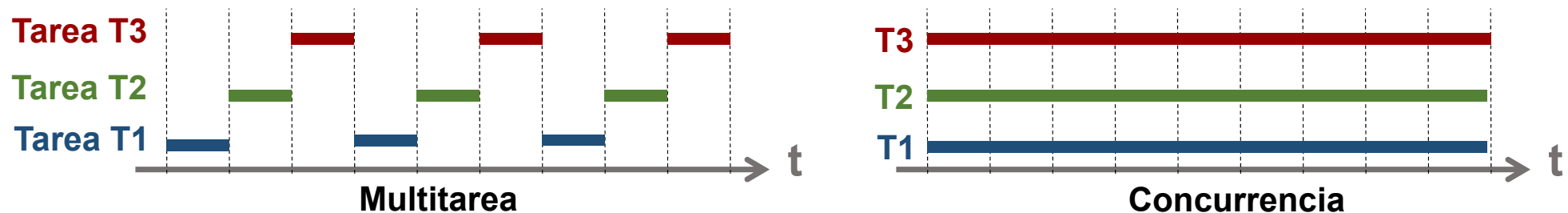
Arq. SW: Round-robin con interrupciones

- Mejora el tiempo de respuesta
- Las tareas urgentes se procesan en la rutina de atención a la interrupción (ISR)
 - Se usa un flag para activar el resto del procesado, no urgente, mediante Round-robin
- Se usa Round-robin para el resto de eventos
- Cuidado con los datos compartidos entre la ISR y el resto del procesado
 - Deshabilitar las interrupciones en las secciones críticas



Arq. SW: Real-Time Operating System (RTOS)

- El RTOS realiza procesamiento **multitarea**
 - Apariencia de concurrencia (procesado simultáneo de las tareas)



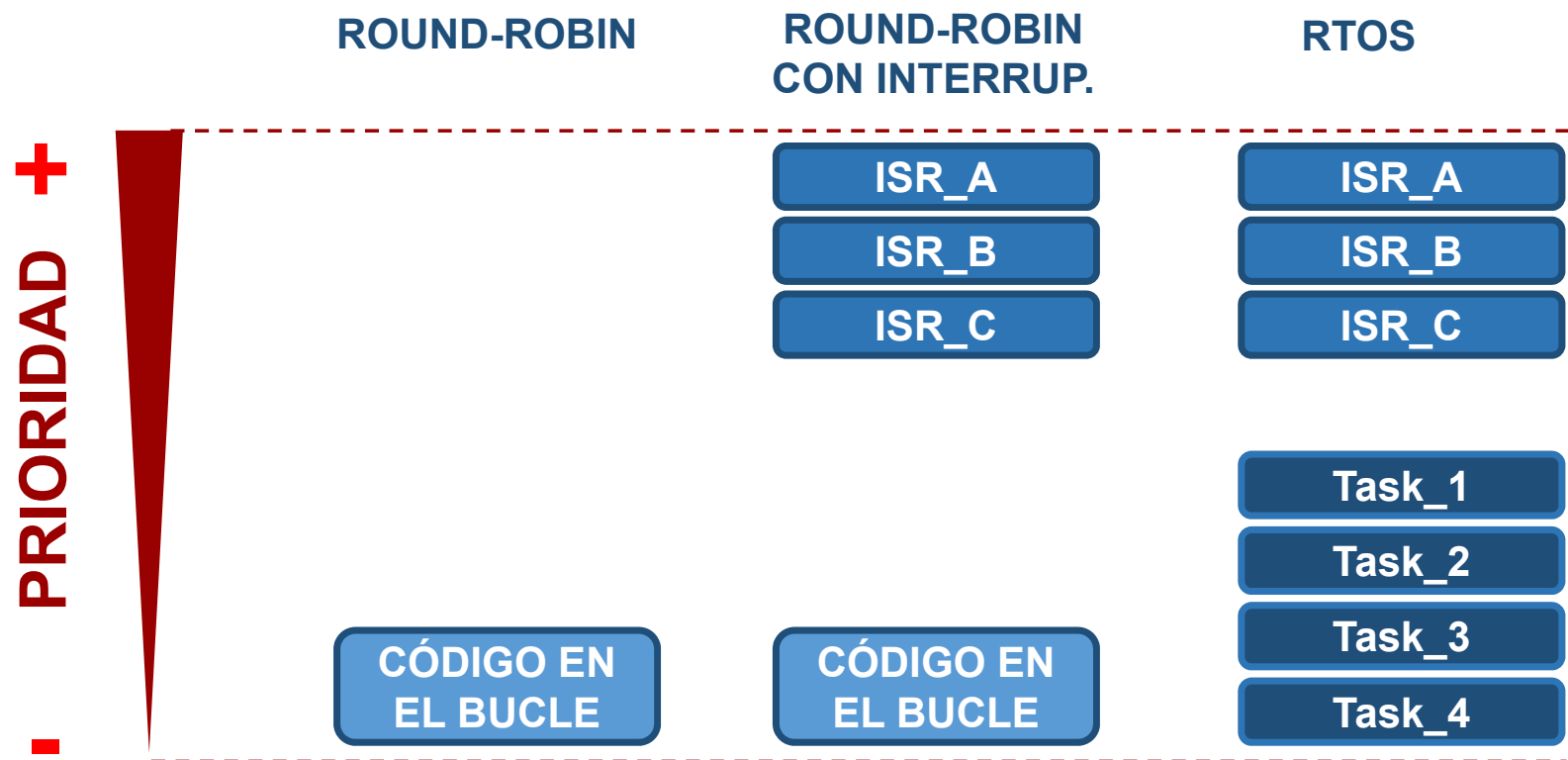
- La partición del sistema en tareas puede simplificar el diseño de sistemas complejos.
 - Facilita modificación del sistema y la depuración y reúso de tareas
- El RTOS gestiona la secuencialización de tareas (guiado por el programador de la aplicación)

Arq. SW: Real-Time Operating System (RTOS)

- Procesado dividido entre rutinas de atención a interrupciones y tareas
- Las rutinas de atención a las interrupciones procesan las operaciones más urgentes y activan “señales” para desbloquear tareas
- Las tareas no están en un “loop” ordenadas, se les asigna prioridades
- El RTOS decide qué tarea activar en función de las “señales” de las interrupciones y las prioridades de las tareas
- El RTOS puede suspender una tarea a mitad de su ejecución para activar otra más prioritaria

```
void ISR_A (void) {  
    procesaEventoA_urgente();  
    signal_A ;  
}  
void ISR_B (void) {  
    procesaEventoA_urgente();  
    signal_B ;  
}  
void vTask_1 (void) {  
    while (TRUE) {  
        Wait for signal_A;  
        procesaEventoA_resto();  
    }  
}  
void vTask_2 (void) {  
    while (TRUE) {  
        Wait for signal_B;  
        procesaEventoB_resto();  
    }  
}
```


Arquitecturas SW



RTOS

- Sistema operativo (OS)
 - Conjunto de programas que gestionan los recursos HW
 - Proporciona servicios a las aplicaciones SW
- Real-Time OS (RTOS)
 - Proporciona las herramientas para dar respuestas en un determinado tiempo a los eventos
 - La respuesta debe ser determinista: siempre el mismo tiempo de respuesta (latencia en el peor caso)
 - El programador es el que debe gestionar esos recursos para cumplir con las especificaciones temporales



RTOS

- RTOS vs. OS convencional
 - Aplicación enlazada con el RTOS
 - Solo se cargan los servicios del RTOS que requiere la aplicación
 - Al arrancar el sistema la aplicación toma el control y arranca el RTOS
- Ejemplos de RTOS disponibles:
 - **FreeRTOS**, μ C/OS, eCos, QNX, Nucleus, VxWorks, LynxOS, ...
- FreeRTOS: open-source desarrollado por Real Time Engineers Ltd
 - <https://www.freertos.org/RTOS.html>
- ESP32 y FreeRTOS
 - Incluido en el entorno de desarrollo ESP-IDF y en Arduino



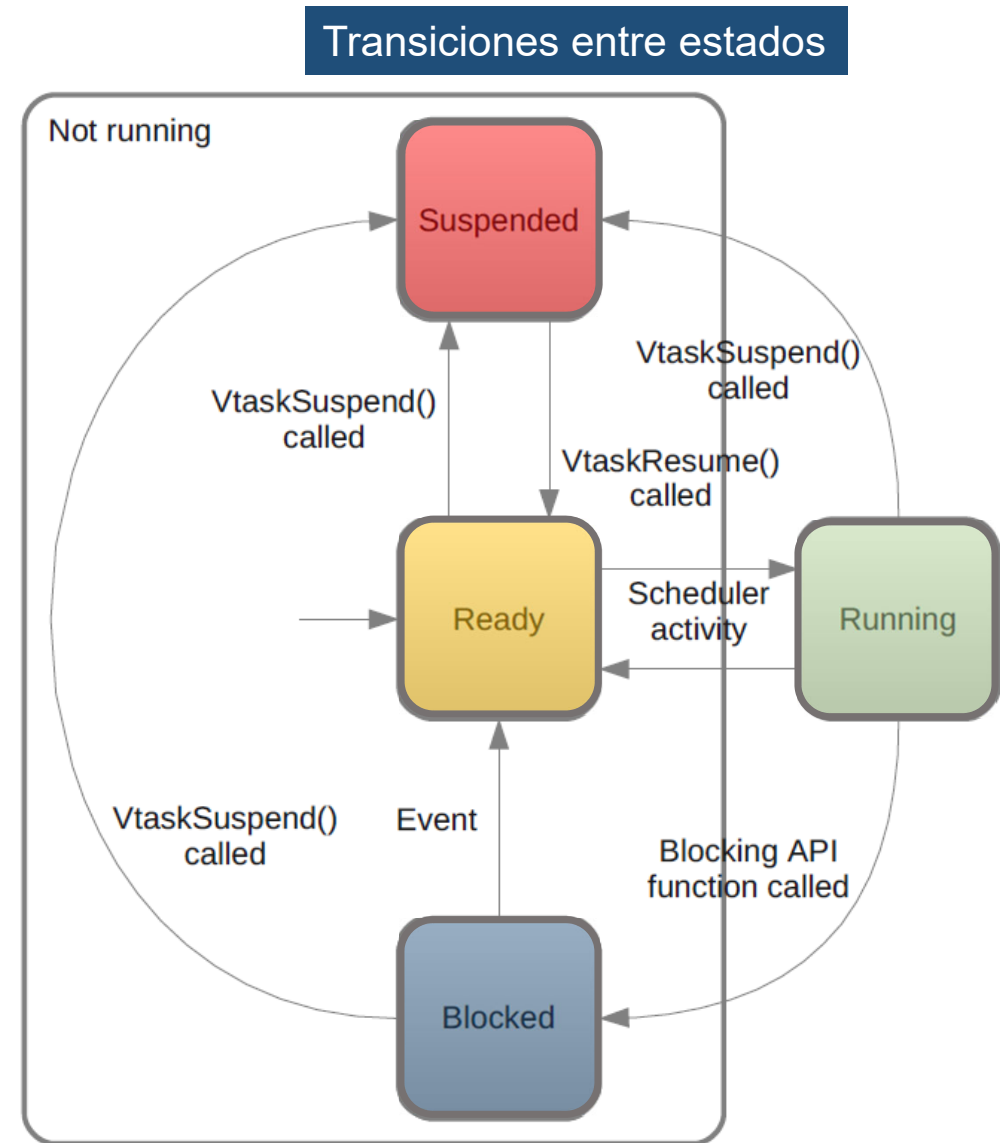
Tareas y sus estados

- La tarea es el bloque básico del RTOS
 - Es una subrutina sin retorno (bucle infinito)
 - Puede haber tantas como la memoria lo permita
- Las tareas se crean indicando (entre otras cosas):
 - su nombre y el puntero a la función que la implementa,
 - el tamaño de memoria (pila) que necesita
 - su prioridad
- Las tareas pueden estar en uno de estos **4 estados**:
 - **Running**: está siendo ejecutada por la CPU
 - **Ready**: está preparada para ejecutar (pero la CPU está ejecutando otra tarea)
 - **Blocked**: no hace nada ahora, está esperando a algún tipo de evento y no puede ejecutarse
 - **Suspended**: está inactiva y no se activa por eventos, se activa y desactiva “manualmente”

```
void vTask_1 (void * pvParameters)
{
    for( ;; ) {
        /* código de la tarea */
    }
}
```

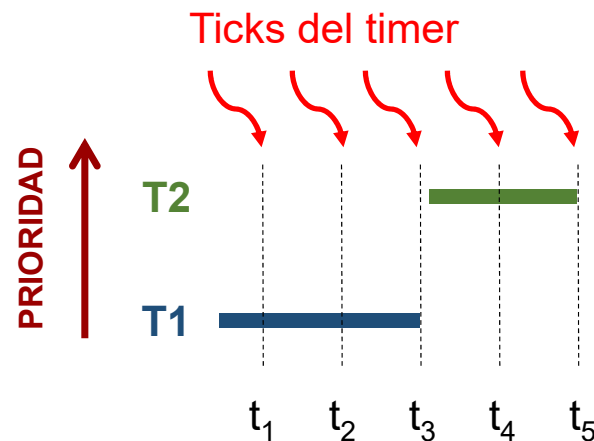
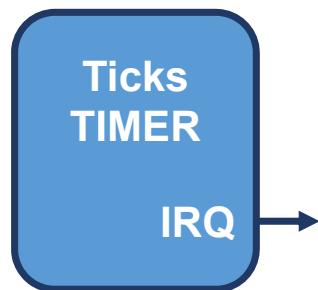
Tareas y sus estados

- En un microprocesador con 1 núcleo:
 - Solo una tarea puede estar en estado *running*
 - El resto de tareas están en uno de los otros estados (*ready*, *blocked* o *suspended*)
- Una tarea solo se puede bloquear por decisión propia (ya ha terminado de ejecutarse o necesita esperar a un evento)
- Una tarea solo sale del estado *blocked* por la llegada de algún evento desde una ISR u otra tarea del sistema
- El **planificador de tareas (Scheduler)** decide qué tarea pasa a estado *running*



El planificador de tareas (scheduler)

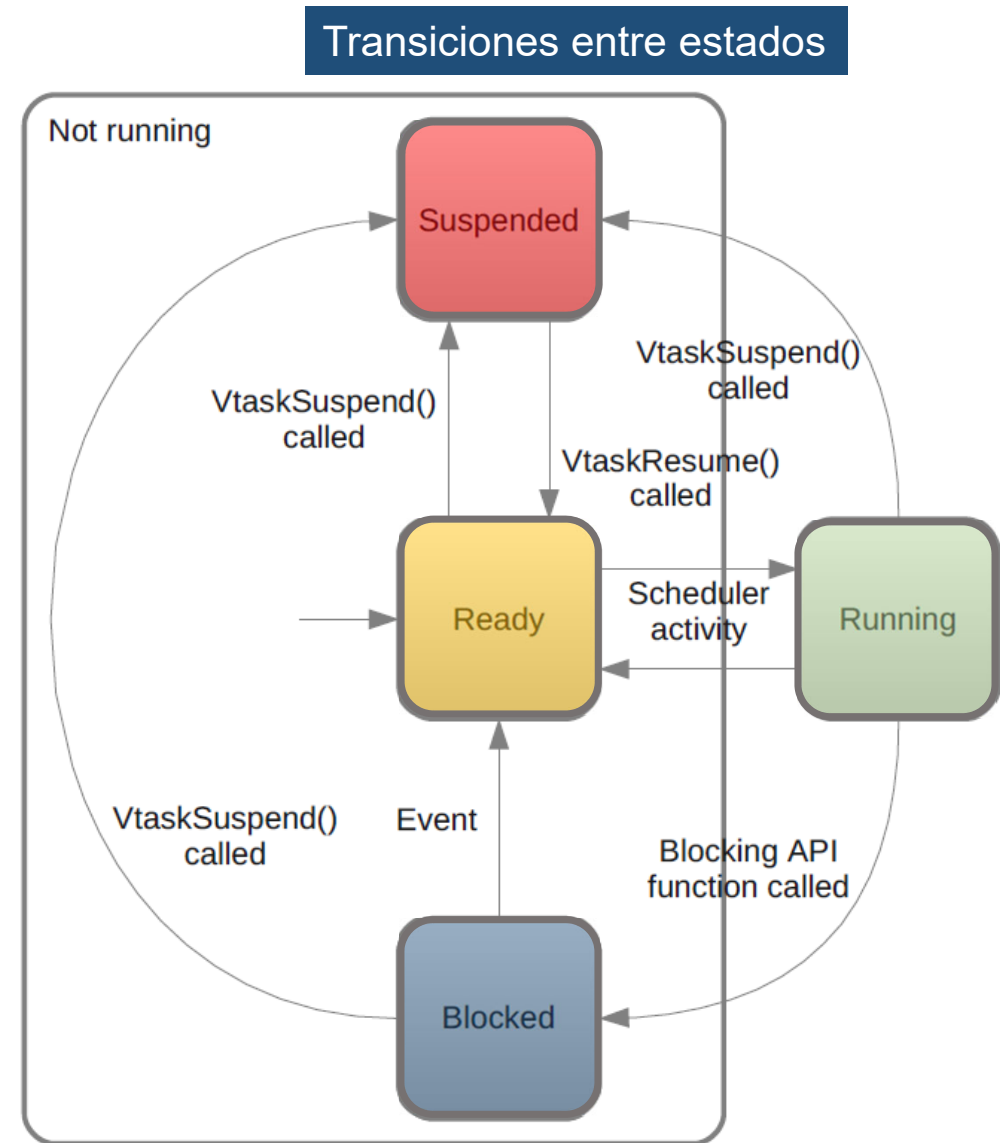
- El **tick** es la medida de tiempo en el RTOS
 - Un timer genera interrupciones periódicas
 - Se puede configurar la frecuencia de los ticks
- En la rutina de atención a la interrupción de ese timer el **planificador de tareas** (***Scheduler***) comprueba cuales están bloqueadas y pasa a estado running la tarea no bloqueada más prioritaria



```
Tick_ISR ()  
{  
    Increment tick count  
    Select execution context  
    Return form ISR  
}
```

Planificador de tareas

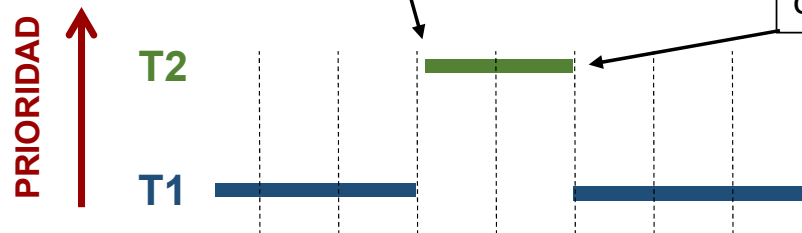
- El **planificador de tareas** (*Scheduler*) decide qué tarea pasa a estado *running*
 - Selecciona la tarea con mayor prioridad que esté en estado *ready*
 - Tareas de poca prioridad podrían no ejecutarse nunca (*starving*)
 - Es responsabilidad del diseñador que esto no ocurra
 - Si hay tareas en estado *ready* con la misma prioridad
 - El *scheduler* las alterna en el tiempo
 - Si todas las tareas están en estado *blocked*
 - El *scheduler* espera a que algún evento pase alguna a estado *ready*



Planificador de tareas

- El **planificador de tareas** (*Scheduler*) decide qué tarea pasa a estado *running*
 - Si la tarea de menor prioridad está ejecutándose y se desbloquea una de mayor prioridad
 - FreeRTOS es expropiativo (*preemptive*)
 - Detiene la tarea de menor prioridad para ejecutar la de mayor prioridad, cuando ésta se bloquea sigue ejecutando la de menor prioridad

T2 se desbloquea y pasa a ready, el scheduler detiene la tarea T1 y ejecuta la T2

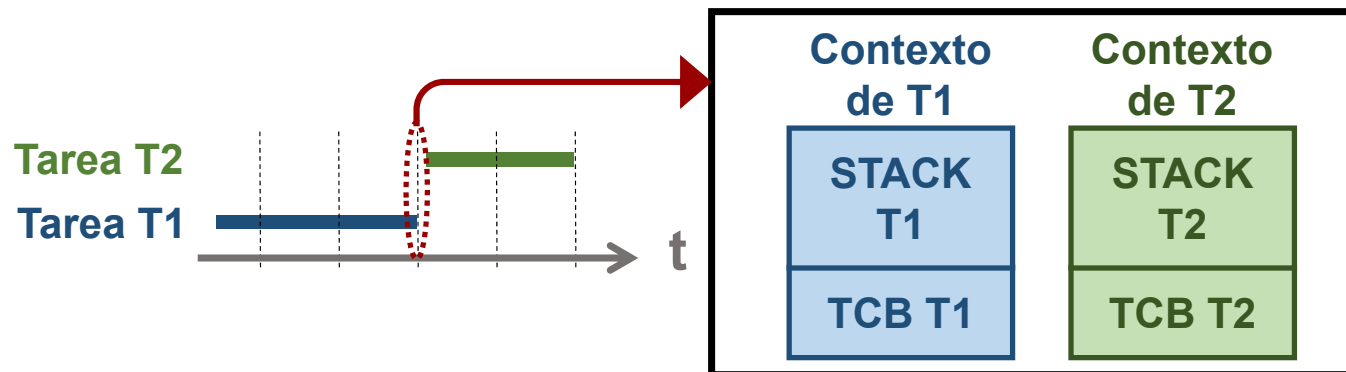


Bloqueo de T2
El scheduler decide continuar con la ejecución de T1

Cambio de (contexto) tarea

- Cambios de tarea:
 - 1º) Se guarda el contexto de la tarea bloqueada
 - 2º) Se carga el contexto de la tarea a ejecutar
- El **contexto**: información necesaria para continuar con la ejecución de la tarea, en el mismo estado en el que estaba cuando se bloqueó:
 - TCB (Task Control Block): contiene la información necesaria para identificar y describir el estado de la tarea
 - Stack: guarda los contenidos de los registros de la CPU, PC, ...

Task Control Block	
Top of Stack	Pointer to last item placed on the stack for this task
Task State	List item that puts the TCB in the ready or blocked queues
Event List	List item used to place the TCB in event lists.
Priority	Task priority (0 = lowest)
Stack Start	Pointer to the start of the process stack
TCB Number	A debugging and tracing field.
Task Name	A task name
Stack Depth	Total depth of the stack in variables (not bytes)



Implementación de tareas con FreeRTOS

- Definición de la tarea:

```
void vATaskFunction( void *pvParameters )
{
    for( ;; )
    {
        -- Task application code here. --
    }
}
```

En Arduino

- Las tareas se escriben fuera del setup y del loop
- Se crean en el setup

- Declarar un manejador para la tarea: `TaskHandle_t xTask;`

- Crear tareas:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *xTask
                    );
```

Puntero a la tarea

Nombre de la tarea

Tamaño de la pila en words

Paso de parámetros o **NULL**

Prioridad de la tarea

NULL o pasarle un manejador

- Eliminar tarea: `void vTaskDelete(TaskHandle_t xTask);`

<https://www.freertos.org/RTOS.html>

Implementación de tareas con FreeRTOS

- Bloquear una tarea durante un número de ticks:

```
void vTaskDelay( const TickType_t xTicksToDelay );
```



Tiempo en ticks que está bloqueada:

0 → no espera, no bloquea la tarea, es como si estuviera haciendo "polling"

2000/portTICK_RATE_MS → espera 2 seg

portMAX_DELAY → espera de forma indefinida

<https://www.freertos.org/RTOS.html>

Ejercicio 1: Tareas, estados y prioridades

Programar 4 tareas que se activen cada 5 segundos y escriban el nombre de la tarea en el monitor serie. Inicialmente asigne una prioridad distinta (de 1 a 4) a cada tarea.

Utilizar

- la función `xTaskCreate` en el **setup** para crear las tareas y asígnele a la pila de cada tarea un tamaño de 1024 words;
- la función `vTaskDelay` para bloquear cada tarea

Compruebe e interprete cómo afecta:

- a) la modificación de las prioridades en las tareas
- b) la asignación de la misma prioridad a todas las tareas
- c) la reducción del tamaño de la pila en una tarea (bájelo en la tarea 1 a 256 words)
- d) eliminar la función de bloqueo en una tarea (por ejemplo en la tarea con la prioridad 3 comente la función `vTaskDelay` e incluya un retardo sin usar “delay”)

Arduino: FreeRTOS con una única tarea

```
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "Arduino.h"

#if CONFIG_AUTOSTART_ARDUINO

#if CONFIG_FREERTOS_UNICORE
#define ARDUINO_RUNNING_CORE 0
#else
#define ARDUINO_RUNNING_CORE 1
#endif

void loopTask(void *pvParameters)
{
    setup();
    for(;;) {
        micros(); //update overflow
        loop();
    }
}

extern "C" void app_main()
{
    initArduino();
    xTaskCreatePinnedToCore(loopTask, "loopTask", 8192, NULL, 1, NULL, ARDUINO_RUNNING_CORE);
}


#endif
```

Tarea loopTask



- Código main.cpp disponible en \cores\esp32

Crea la tarea loopTask



La tarea “idle” y la función “idle hook”

- FreeRTOS siempre debe tener al menos una tarea en estado running
- La tarea “idle” se crea automáticamente cuando se activa el scheduler
 - Tiene la prioridad más baja (0): no bloquea a ninguna tarea del usuario
 - Está en estado running cuando todas las tareas están bloqueadas
 - Es la encargada de liberar la memoria asignada a las tareas eliminadas
- Se puede **incluir una función para** que sea ejecutada en **la tarea idle**
 - Útil para ejecutar algo en background (ej. forzar el modo de bajo consumo cuando no hay tareas ejecutándose)
 - ¿Cómo hacerlo en el ESP32?
 - Usar la función **esp_register_freertos_idle_hook(vApplicationIdleHook)** declarada en `esp_freertos_hooks.h`
 - Definir la función: **bool vApplicationIdleHook(void);**

Ejercicio 2: “Idle task hook function”

Renombre el programa del ejercicio anterior y añada una función para ejecutarla en la tarea idle. Esta función realizará únicamente el incremento de un contador asociado a una variable global. Por otro lado, la tarea 1 escribirá en el monitor serie, además de su mensaje, el valor de la cuenta. Para realizar este ejercicio debe bloquear la tarea en la que se ejecuta la función loop de Arduino mediante `vTaskDelay`.

Funciones a utilizar:

- `esp_register_freertos_idle_hookvTaskDelay`

No olvide añadir:

- `#include "esp_freertos_hooks.h"`

Semáforos binarios: sincronización de tareas

- Hasta ahora la sincronización de rutinas se ha hecho utilizando **flags**
 - La tarea a sincronizar está siempre activa en el bucle comprobando (polling) si el flag se activa

```
ISR_A (void) {  
    procesaEventoA_urgente();  
    flag_A = TRUE;  
}  
  
while (TRUE) {  
    if (flag_A) ←  
        procesaEventoA_resto();  
        flag_A = FALSE;  
    ...  
}
```

Sincronización entre ISR
y rutina en el bucle

- **Semáforo binario**: recurso para la sincronización de tareas
 - La tarea a sincronizar está bloqueada hasta que recibe el semáforo

```
ISR_A (void) {  
    procesaEventoA_urgente();  
    DarSemaforo(S1);  
}  
  
void vTask_A (void) {  
    for ( , , ) {  
        if (RecibirSemaforo(S1)) ←  
            procesaEventoA_resto();  
        ...  
    }  
}
```

Sincronización entre ISR y tarea

Semáforos binarios: funciones del FreeRTOS

- Declarar manejador del semáforo: SemaphoreHandle_t **xSemaphore** = NULL;
 - Crear un semáforo binario:
 - **xSemaphore** = xSemaphoreCreateBinary(void);
 - Manejador para utilizar el semáforo, debe estar declarado previamente
 - Dar y recibir un semáforo en una tarea:
 - xSemaphoreGive(SemaphoreHandle_t **xSemaphore**);
 - xSemaphoreTake(SemaphoreHandle_t **xSemaphore**, TickType_t xTicksToWait);
 - Devuelve pdTRUE si se recibe el semáforo
 - Dar semáforo en una rutina de atención a una interrupción:
 - xSemaphoreGiveFromISR (SemaphoreHandle_t **xSemaphore**, signed BaseType_t *pxHigherPriorityTaskWoken);
 - NULL
- Tiempo en ticks que espera el semáforo para estar disponible. Si es portMAX_DELAY, espera de forma indefinida

Ejercicio 3: Sincronización de tareas

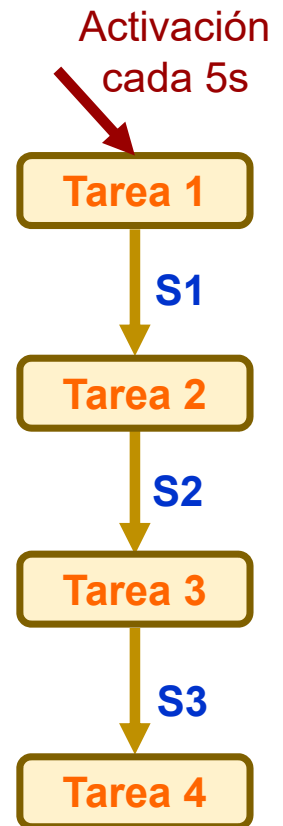
Programar 4 tareas de la misma prioridad que escriban el nombre de la tarea en el monitor serie y que se sincronicen con la **Tarea 1** a través de 3 semáforos binarios **S1**, **S2** y **S3**.

La **Tarea 1** se activará cada 5s y enviará **S1** para activar la **Tarea 2**. La **Tarea 2** se activará al recibir **S1** y enviará **S2**. La **Tarea 3** se activará al recibir **S2** y enviará **S3**. La **Tarea 4** se activará al recibir **S3**.

Utilice las funciones `xTaskCreate`, `vTaskDelay`, `xSemaphoreCreateBinary`, `xSemaphoreGive` y `xSemaphoreTake` y declare el semáforo con el tipo `SemaphoreHandle_t`.

En la función setup:

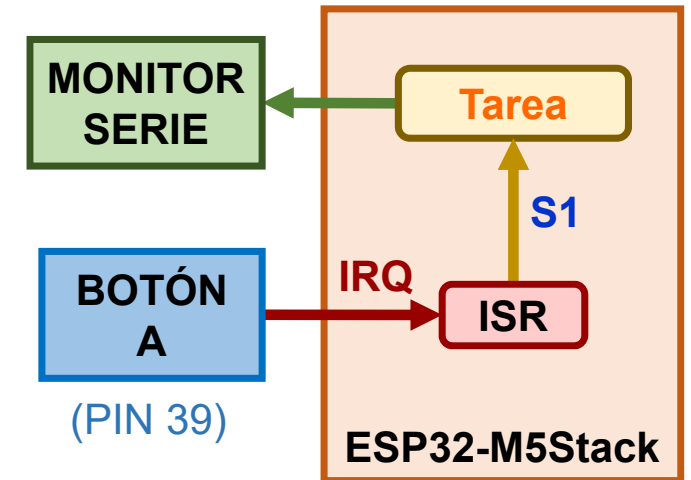
- 1º) Crear los semáforos binarios
- 2º) Crear las tareas



Ejercicio 4: Sincronización ISR y tarea

Complete el programa `Ejer_4_sincr_ISR_tarea.ino` para generar una interrupción al pulsar el botón A del M5Stack y que la rutina de atención a la interrupción active un semáforo binario (`S1`) para sincronizar una tarea que escribirá en el monitor serie el mensaje “Botón A pulsado”.

Utilice las funciones `xSemaphoreCreateBinary`, `xSemaphoreGiveFromISR`, `xSemaphoreTake`, `xTaskCreate` y declare el semáforo con el tipo `SemaphoreHandle_t`.



Datos compartidos entre ISRs o sus tareas

- **Sección crítica:** parte del programa que accede a datos compartidos entre rutinas
 - Es susceptible de ser computado con errores si mientras se está computando la sección crítica es interrumpida por otra rutina de prioridad superior que accede a los mismos datos
- Si las **secciones críticas se ejecutan por la llegada de interrupciones**, es necesario desactivar las interrupciones o subir su prioridad durante la ejecución de la sección crítica
 - Las secciones críticas deben ser lo más corta posible para no mermar el tiempo de respuesta a las interrupciones
- Funciones del FreeRTOS para proteger la sección crítica en tareas
 - void `taskENTER_CRITICAL`(void);
 - void `taskEXIT_CRITICAL`(void);
- Funciones del FreeRTOS para proteger la sección crítica en rutinas de atención a las interrupciones (ISR)
 - void `taskENTER_CRITICAL_FROM_ISR` (void);
 - void `taskEXIT_CRITICAL_FROM_ISR` (void);

Datos compartidos entre ISRs o sus tareas

Dos tareas sincronizadas con sendas interrupciones comparten datos

```
static int cEvents;  
  
void vTask_1 (void) {  
    int MyNewEvents = 0;  
    ...  
    cEvents += cMyNewEvents;  
    ...  
}  
  
void vTask_2 (void) {  
    int MyNewEvents = 0;  
    ...  
    cEvents += cMyNewEvents;  
    ...  
}
```

Operación **no atómica**:
puede interrumpirse a
mitad de ejecución

Desactivación de
interrupciones

Protección secciones críticas

```
static int cEvents;  
  
void vTask_1 (void) {  
    int MyNewEvents = 0;  
    ...  
    taskENTER_CRITICAL();  
    cEvents += cMyNewEvents;  
    taskEXIT_CRITICAL ();  
    ...  
}  
  
void vTask_2 (void) {  
    int MyNewEvents = 0;  
    ...  
    taskENTER_CRITICAL();  
    cEvents += cMyNewEvents;  
    taskEXIT_CRITICAL ();  
    ...  
}
```

Datos compartidos entre ISRs o sus tareas

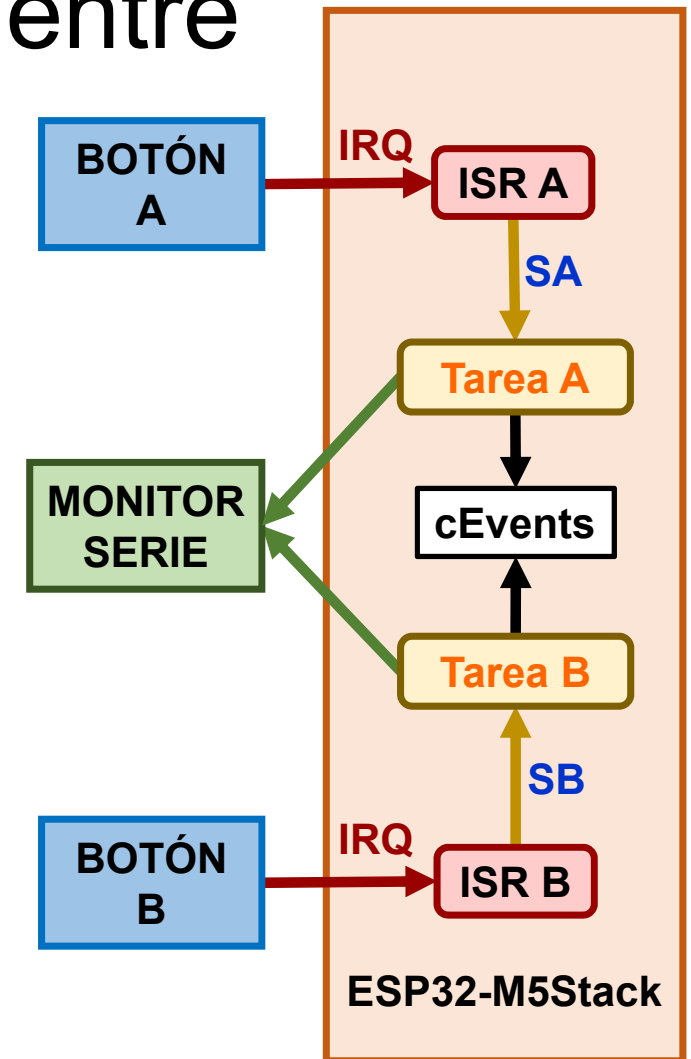
- El **ESP32** es un procesador con **doble núcleo**: puede ejecutar 2 tareas a la vez
- Las funciones `taskENTER_CRITICAL...` solo son válidas para procesadores con un núcleo
- Es necesario declarar una variable del tipo `portMUX_TYPE` para deshabilitar a la vez las interrupciones de los dos núcleos
 - `portMUX_TYPE mux = portMUX_INITIALIZER_UNLOCKED;`
- Funciones para proteger en el ESP32 la sección crítica en tareas
 - `void portENTER_CRITICAL(&mux);`
 - `void portEXIT_CRITICAL(&mux);`
- Funciones para proteger en el ESP32 la sección crítica en las ISRs
 - `void portENTER_CRITICAL_FROM_ISR (&mux);`
 - `void portEXIT_CRITICAL_FROM_ISR (&mux);`

Ejercicio 5: Datos compartidos entre ISRs o sus tareas

Dos tareas (**Tarea A** y **Tarea B**) están sincronizadas con dos rutinas de atención a las interrupciones generadas con los botones A y B (**ISR A** e **ISR B**). Las tareas cuentan las veces que se han pulsado los botones A y B utilizando la variable compartida **cEvents**.

Use el fichero **Ejer_5_int_sec critica.ino**

- Lea e interprete el código del programa
- Compruebe el funcionamiento del programa pulsando los botones A y B a la vez
- Utilice `portENTER_CRITICAL(&mux)` y `portEXIT_CRITICAL(&mux)` para proteger la sección crítica de las tareas

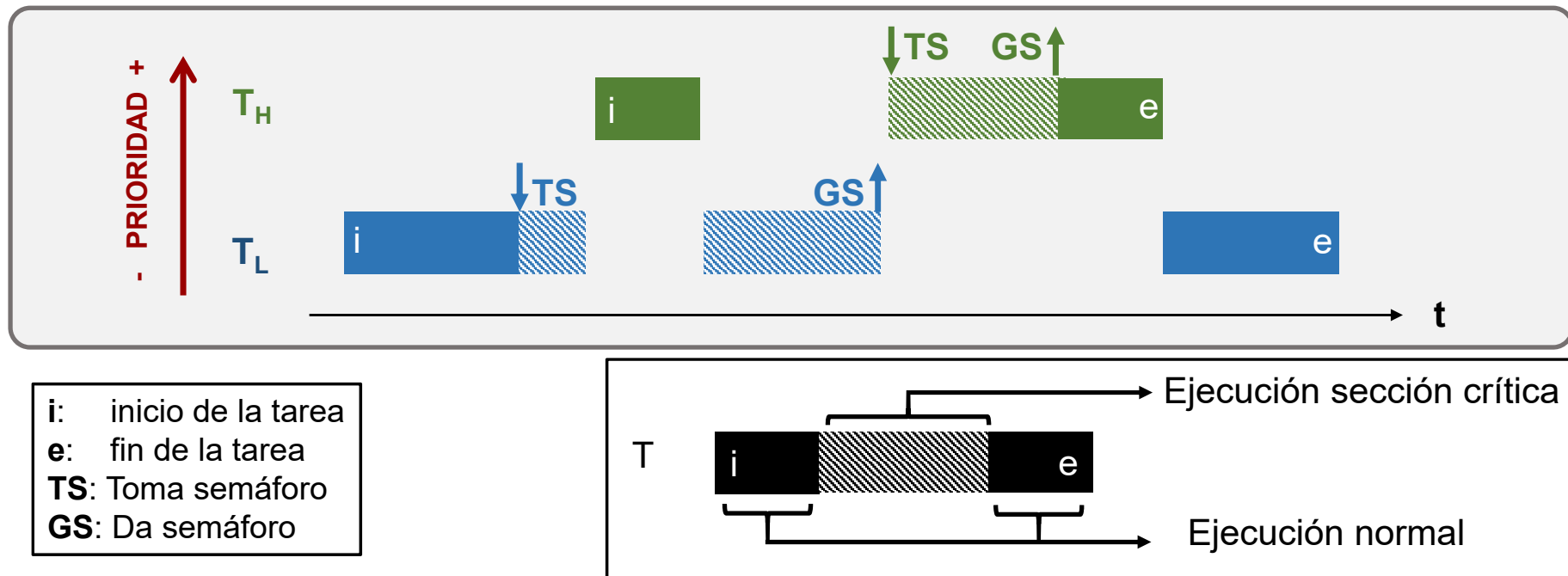


Datos compartidos entre tareas

- Si las **secciones críticas** están **en tareas que no están sincronizadas con interrupciones** la solución anterior no es válida
 - El planificador de tareas puede dar paso a la ejecución de una tarea de mayor prioridad cuando está a mitad de la ejecución de una sección crítica
 - Se pueden utilizar semáforos binarios para proteger la sección crítica
 - Puede surgir el **problema de inversión de prioridades**
- Se requiere el uso de otro tipo de semáforos: **semáforo Mutex**
 - **Mutex** \Rightarrow **Mutual exclusion**
 - Es como el semáforo binario pero incluye el mecanismo para heredar la prioridad de otras tareas
 - Si otra tarea se bloquea al intentar tomar el semáforo, la tarea de menor prioridad que tiene el semáforo aumenta temporalmente su prioridad a la de la tarea bloqueada
 - Sirve para asegurar que la tarea de mayor prioridad que se ha bloqueado al solicitar el semáforo, esté en ese estado el menor tiempo posible
 - Minimizar el problema de la inversión de prioridades

Datos compartidos entre tareas

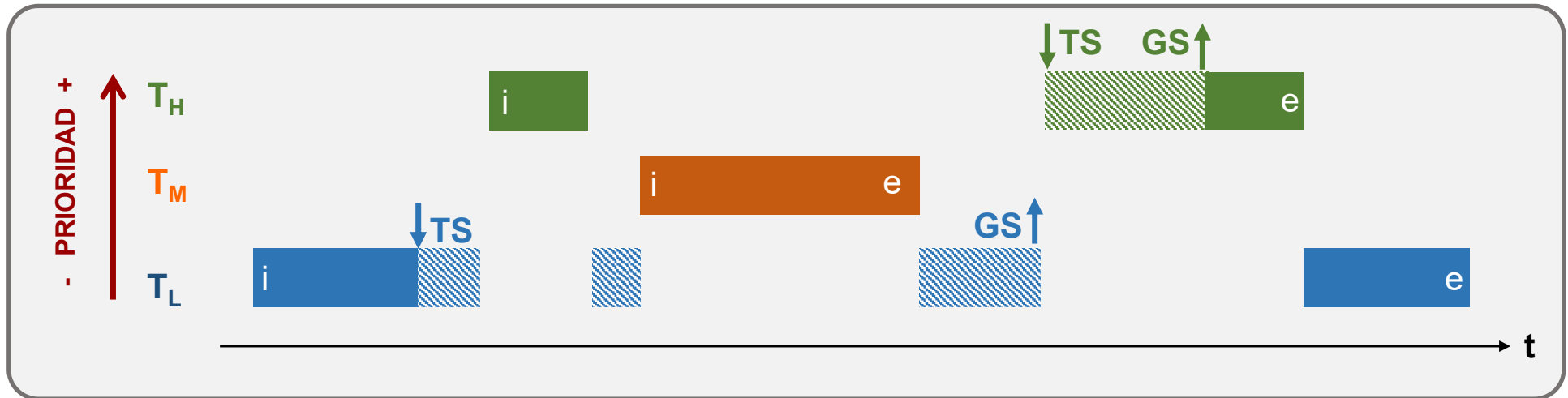
Ejemplo 1: se utilizan **semáforos binarios** para proteger la sección crítica de 2 tareas con diferentes prioridades ($T_H > T_L$) que comparten datos



- La **inversión de prioridades** de este ejemplo es inevitable
- La sección crítica debe ser lo más corta posible para no agravar el problema de la inversión

Datos compartidos entre tareas

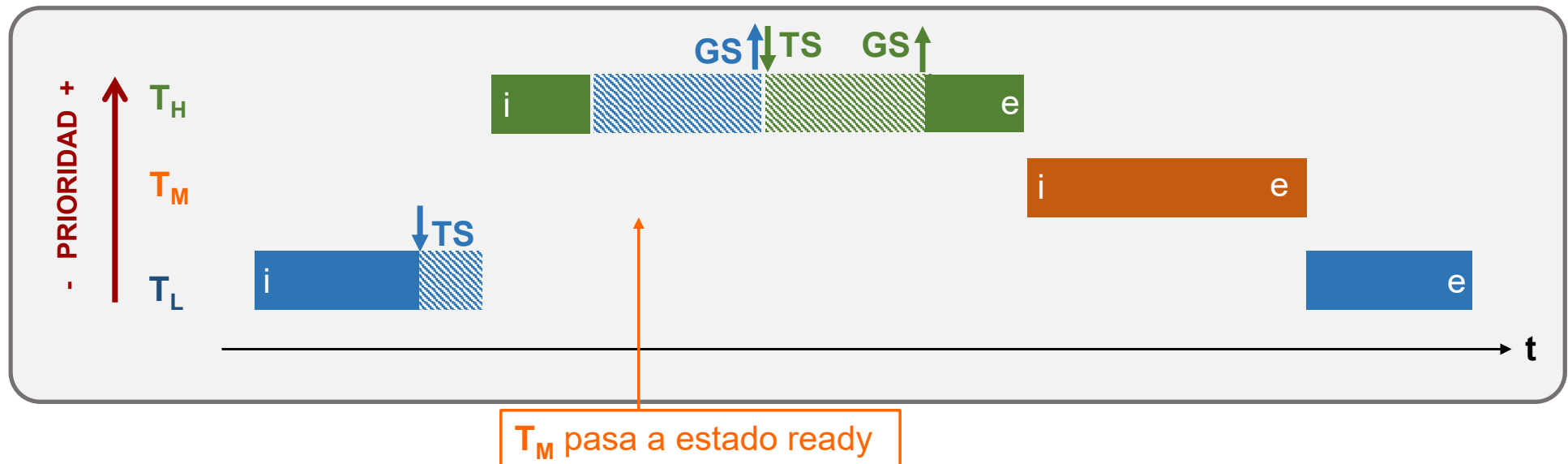
Ejemplo 2: se utilizan **semáforos binarios** para proteger la sección crítica de 2 tareas con diferentes prioridades ($T_H > T_L$) que comparten datos. Una tercera tarea de prioridad media ($T_H > T_M > T_L$) pasa a estado *ready* mientras se ejecuta la sección crítica de T_L .



- Cuando T_M pasa a estado *ready*, el *scheduler* la ejecuta porque su prioridad es mayor que T_L y retrasa la ejecución de T_H , que tiene mayor prioridad que las anteriores
- El aumento del tiempo que dura la inversión de prioridades debido a T_M es evitable si se incluye “**herencia de prioridades**” en los semáforos, con los **mutex**

Datos compartidos entre tareas

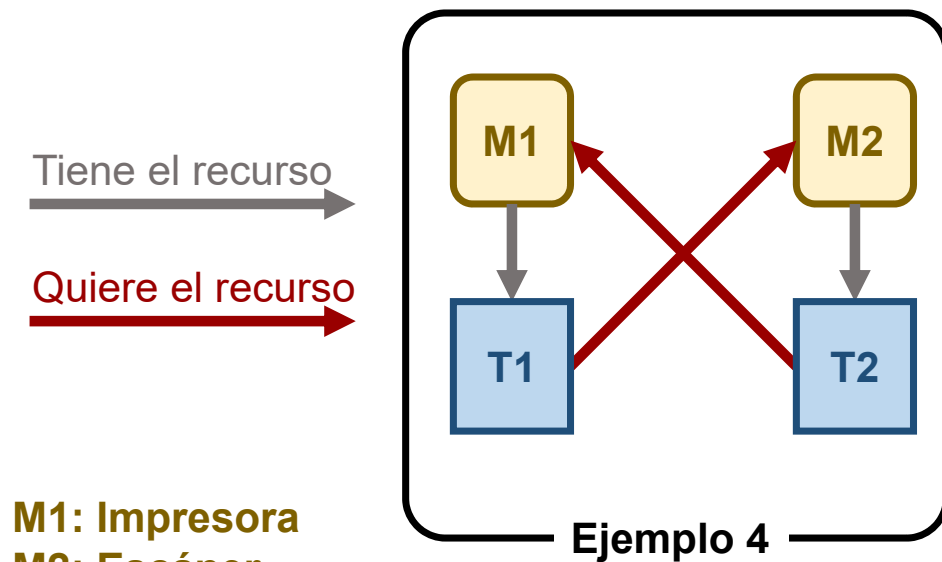
Ejemplo 3: se utilizan **semáforos mutex** para proteger la sección crítica de 2 tareas con diferentes prioridades ($T_H > T_L$) que comparten datos. Una tercera tarea de prioridad media ($T_H > T_M > T_L$) pasa a estado *ready* mientras se ejecuta la sección crítica de T_L .



- Cuando se bloquea T_H al solicitar el mutex, T_L hereda su nivel de prioridad
- Cuando T_M pasa al estado *ready*, el *scheduler* la deja en ese estado porque su prioridad es menor que la prioridad actual de T_L ($T_H = T_L > T_M$)

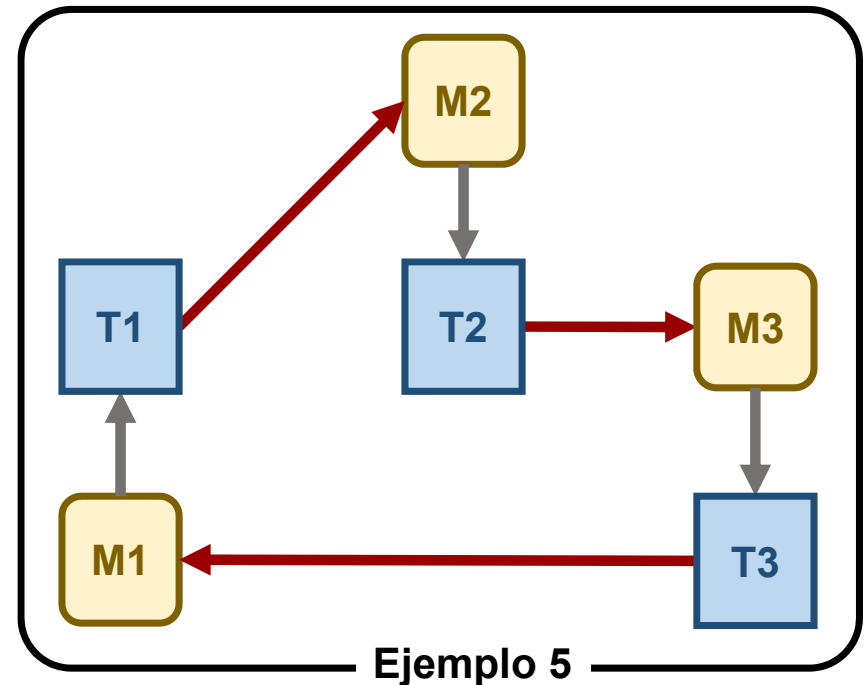
Datos compartidos entre tareas

- Mal uso de los semáforos “**Deadlock**”
 - Varias tareas se bloquean permanentemente porque nunca se cumplen las condiciones para conseguir el recurso protegido con un semáforo



M1: Impresora
M2: Escáner
M3: Buffer memoria


Ejemplo 4




Ejemplo 5

Semáforos Mutex: funciones del FreeRTOS

- Declarar manejador del semáforo: SemaphoreHandle_t **xSemaphore** = NULL;
- Crear un semáforo mutex:
 - **xSemaphore** = xSemaphoreCreateMutex(void);
- Dar y recibir un semáforo en una tarea:
 - xSemaphoreGive(SemaphoreHandle_t **xSemaphore**);
 - xSemaphoreTake(SemaphoreHandle_t **xSemaphore**, TickType_t xTicksToWait);

 Manejador para utilizar el semáforo, debe estar declarado previamente

 Devuelve pdTRUE si se recibe el semáforo

 Tiempo en ticks que espera el semáforo para estar disponible.
Si es portMAX_DELAY, espera de forma indefinida
Si es 0, es equivalente al “polling”

Mutexes: protección de datos compartidos entre tareas

Dos tareas comparten datos

```
static int cEvents;  
  
void vTask_1 (void) {  
    int MyNewEvents = 0;  
    ...  
    cEvents += cMyNewEvents;  
    ...  
}  
  
void vTask_2 (void) {  
    int MyNewEvents = 0;  
    ...  
    cEvents += cMyNewEvents;  
    ...  
}
```

Operación **no atómica**:
puede interrumpirse a
mitad de ejecución

Exclusión mútua:
Solo una de las dos tareas
puede estar accediendo a
la sección crítica en un
instante dado

Protección secciones críticas

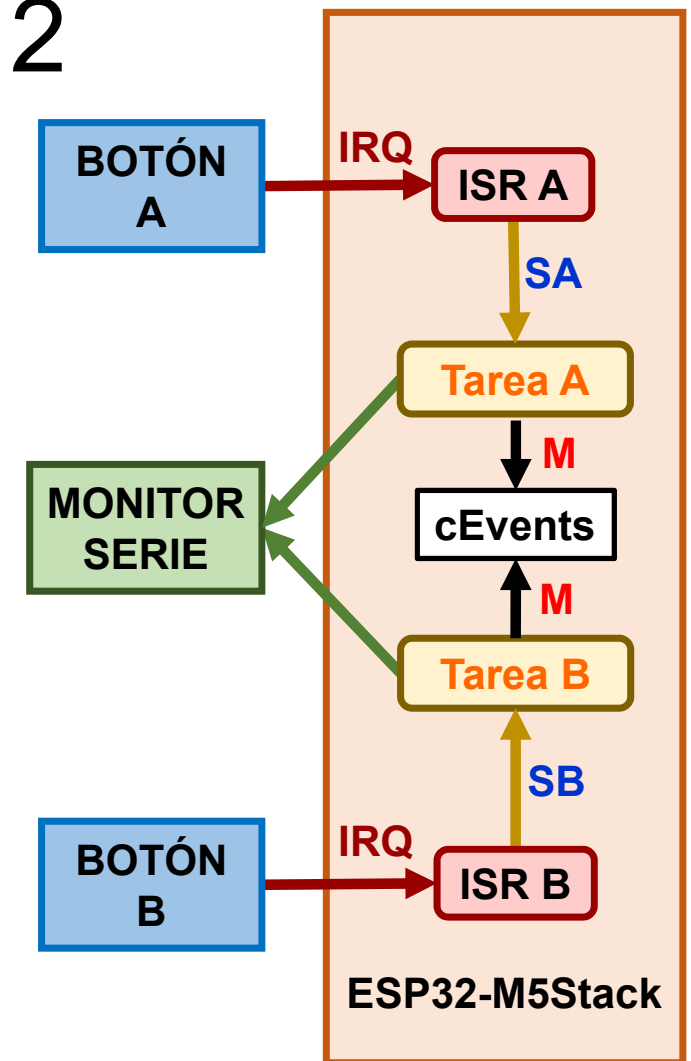
```
static int cEvents;  
  
void vTask_1 (void) {  
    int MyNewEvents = 0;  
    ...  
    if (xSemaphoreTake ()) {  
        cEvents += cMyNewEvents;};  
        xSemaphoreGive ();  
    ...  
}  
  
void vTask_2 (void) {  
    int MyNewEvents = 0;  
    ...  
    if (xSemaphoreTake ()) {  
        cEvents += cMyNewEvents;};  
        xSemaphoreGive ();    ...  
}
```

Ejercicio 6: Datos compartidos 2

Dos tareas (**Tarea A** y **Tarea B**) están sincronizadas con dos rutinas de atención a las interrupciones generadas con los botones A y B (**ISR A** e **ISR B**). Las tareas cuentan las veces que se han pulsado los botones A y B utilizando la variable compartida **cEvents**.

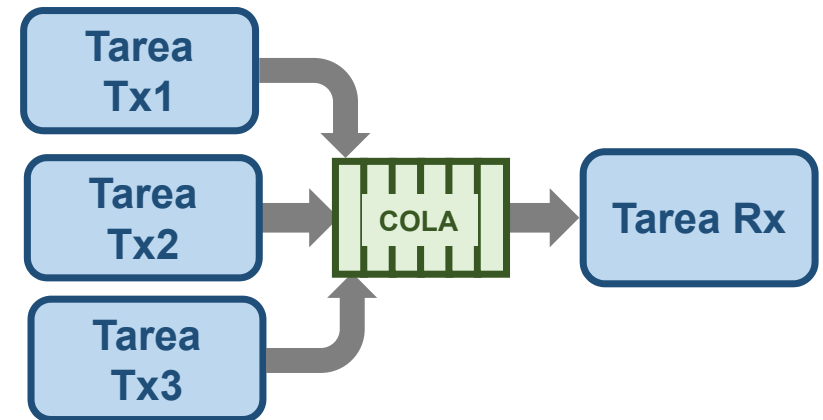
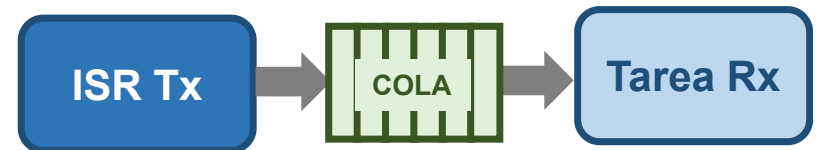
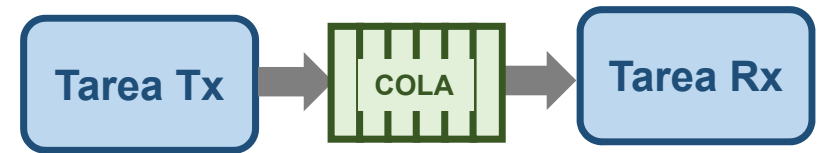
Use el fichero **Ejer_6_tarea_secritica.ino**

- Lea e interprete el código del programa
- Compruebe el funcionamiento del programa pulsando los botones A y B a la vez
- Utilice un semáforo mutex (**M**) para proteger la sección crítica de las tareas. Utilice las funciones: `xSemaphoreCreateMutex`, `xSemaphoreGive` y `xSemaphoreTake`. Declare el mutex con el tipo `SemaphoreHandle_t`.



Colas: comunicación entre tareas

- Mecanismo de almacenamiento de datos para la comunicación entre tareas y entre ISR y tareas
- Se suelen utilizar como FIFOs
 - existen funciones para cambiar el comportamiento FIFO
- El dato enviado a la cola se copia en la cola
- Múltiples tareas pueden acceder a una cola
- La cola incluye mecanismos de bloqueo de tareas:
 - Bloqueo en escritura si la cola está llena
 - Bloqueo en lectura si la cola está vacía



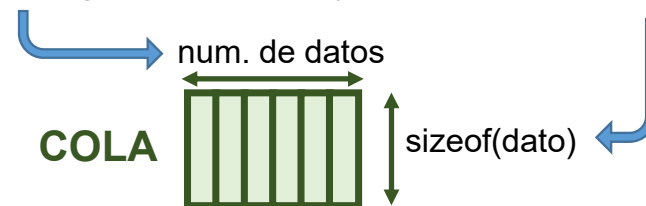
Colas: funciones del FreeRTOS

- xQueueHandle **xQueue**;

- Crear una cola:

- **xQueue** = xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);

Manejador para utilizar la cola,
debe estar declarado previamente



- Enviar/recibir datos a/de una cola:

- xQueueSend(QueueHandle_t **xQueue**, const void * pvItemToQueue, TickType_t xTicksToWait);

- xQueueReceive(QueueHandle_t **xQueue**, void *pvBuffer, TickType_t xTicksToWait);

Devuelven pdTRUE si la operación
se realiza correctamente

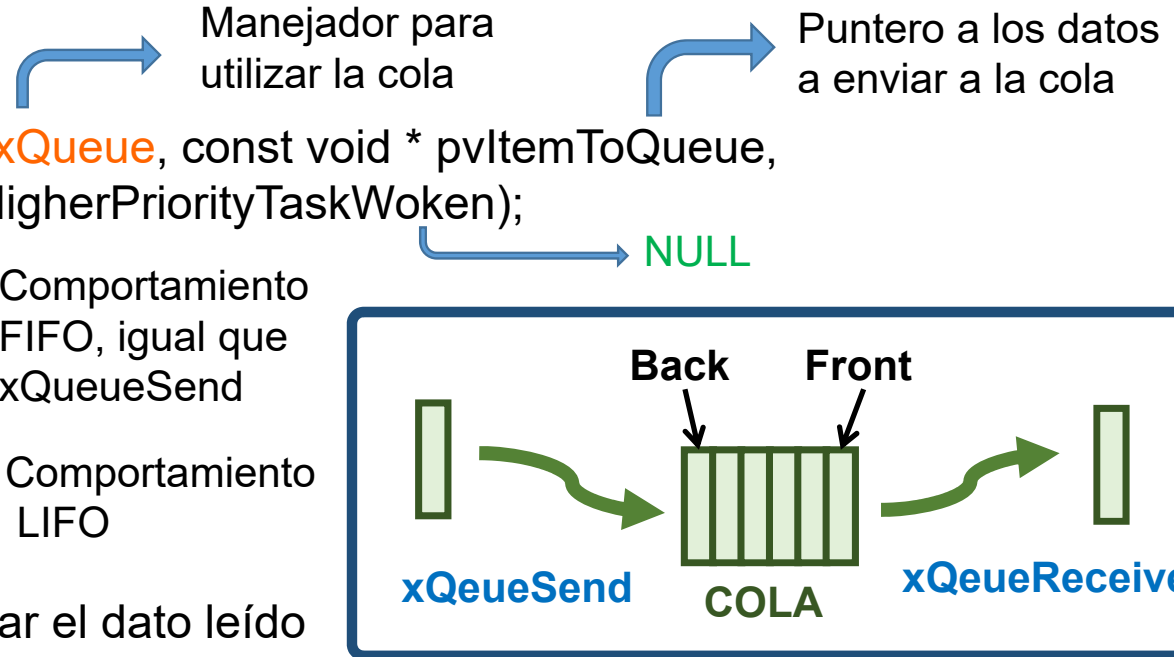
Puntero del buffer en el que se
reciben los datos de la cola

Puntero a los datos
a enviar a la cola

Tiempo en ticks que espera la
cola para estar disponible. Si es
portMAX_DELAY, espera de
forma indefinida
Si es 0, es equivalente al “polling”

Colas: funciones del FreeRTOS

- Enviar datos desde una ISR:
 - `xQueueSendFromISR`(QueueHandle_t **xQueue**, const void * pvltemToQueue, BaseType_t *pxHigherPriorityTaskWoken);
- Alterar el funcionamiento como FIFO
 - `xQueueSendToBack()`
 - `xQueueSendToBackFromISR()`
 - `xQueueSendToFront()`
 - `xQueueSendToFrontFromISR()`
 - `xQueuePeek()` : leer de la cola sin quitar el dato leído
- Otras funciones
 - UBaseType_t `uxQueueMessagesWaiting`(QueueHandle_t xQueue);
 - UBaseType_t `uxQueueSpacesAvailable`(QueueHandle_t xQueue);

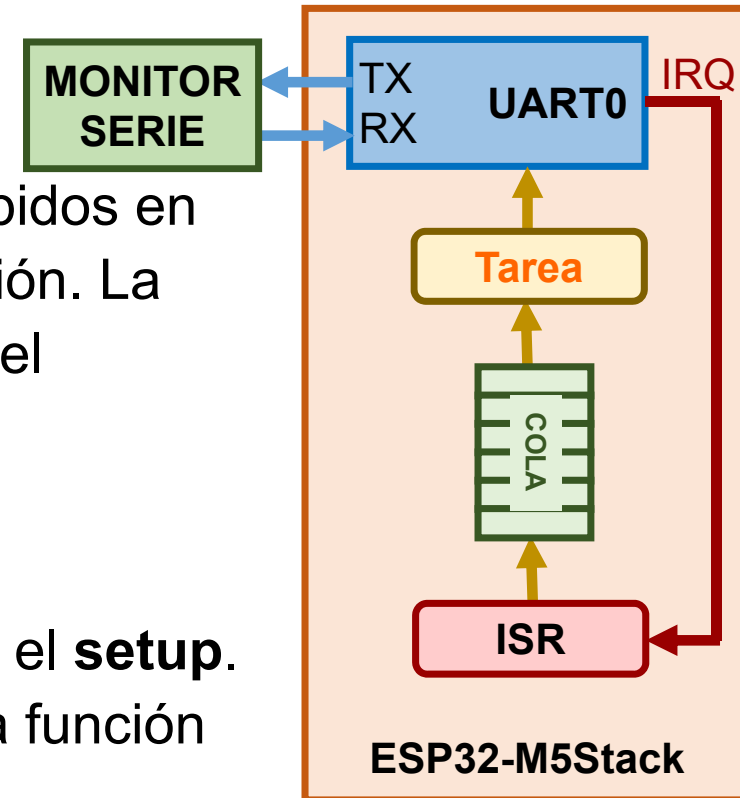


Ejercicio 7: Comunicación entre ISR y una tarea

Complete el programa del fichero `Ejer_7_int_cola.ino` para enviar a una tarea mediante una cola los bytes recibidos en la UART0 y leídos en su rutina de atención a la interrupción. La tarea los enviará de nuevo a través de la UART0. Utilice el monitor serie para enviar y visualizarlos los datos.

Acciones a realizar:

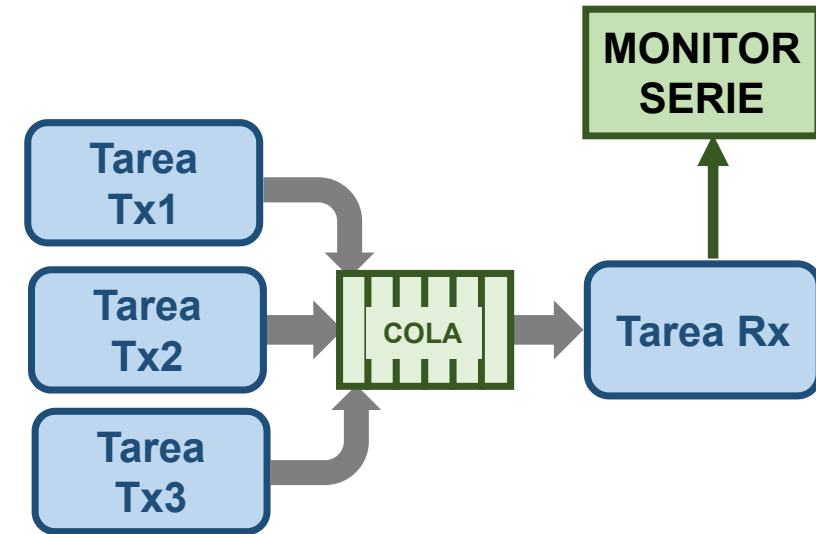
1. Declarar el manejador de la cola
2. Crear una cola de TAM_COLA=10 x TAM_MSG=1 en el **setup**.
3. Enviar a la cola los bytes leídos en la ISR mediante la función `xQueueSendFromISR`
4. Leer los datos de la cola en la tarea `lee_cola`, mediante la función `xQueueReceive` y enviarlos al monitor mediante `uart_write_bytes`



Ejercicio 8: Comunicación de 3 tareas con 1 tarea

Complete el programa `Ejer_8_tareas_aCola.ino` para enviar datos a través de una cola desde tres tareas transmisoras a una receptora. La tarea receptora enviará un mensaje al monitor serie identificando la tarea transmisora y escribiendo el dato.

Los datos se envían mediante la estructura `type_msg`, con la que se identificará al transmisor.



Identificación del Tx:

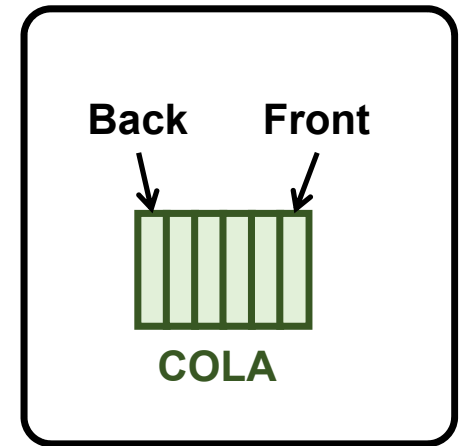
- Estructura a enviar en la cola:

```
typedef struct{
    uint8_t id_sender;
    uint8_t data;
} type_msg;
```

Ejercicio 9: Colas FIFO y LIFO

Abra el programa [Ejer_9_FIFO_LIFO.ino](#)

1. Lea e interprete el código ¿Qué uso tienen los parámetros `LENGTH_QUEUE` y `NUM_DATA_TO_QUEUE`?
2. Compruebe el efecto de enviar los datos a la cola con las funciones [xQueueSendToBack](#) y [xQueueSendToFront](#)
3. Compruebe el efecto que tiene el enviar un número mayor de datos que el tamaño de la cola. Modifique el valor de `NUM_DATA_TO_QUEUE`



Referencias

- **D.E. Simon, An Embedded Software Primer, Pearson 2006:**
 - **Tema 5: Survey of software architectures**
 - **Tema 6: Introduction to real-time operating Systems**
 - **Tema 7: More operating system services**
- *Nicolas Melot, Study of an operating system: FreeRTOS,* http://wiki.csie.ncku.edu.tw/embedded/FreeRTOS_Melot.pdf
- **Tutorial Arduino ESP32 FreeRTOS 1: How to create a task:** <http://www.iotsharing.com/2017/06/how-to-apply-freertos-in-arduino-esp32.html>
- **Tutorial Arduino ESP32 FreeRTOS 4: How to use Binary Semaphore - Mutex - Counting semaphore - Critical section for resources management**
<http://www.iotsharing.com/search?q=Freertos+2&updated-max=2017-06-13T09:18:00-07:00&max-results=20&start=6&by-date=false>
- **Tutorial ESP32 Arduino: FreeRTOS Queues,** <https://techtutorialsx.com/2017/08/20/esp32-arduino-freertos-queues/>