# Apuntes Tiva C

## Índice



# Tema 2: Arquitectura de microcontroladores ARM Cortex-M

## 2.1. Inicialización y GPIO

### 2.1.1. Inicialización

Necesaria para hacer funcionar el sistema, periféricos y módulos a emplear.

Para ello usaremos la **API SysCtl**, añadimos:

- Añadimos las carpetas **driverlib/sysctl.h** y **driverlib/sysctl.c** en los proyectos.
- Y hacer un :

```
#include "driverlib/sysctl.h”
```

### 2.1.2. Reloj del sistema

Se necesitan señales de reloj para comportamientos síncronos, la primera operación debe ser inicializar y configurar el reloj del sistema. Las fuentes de reloj son:

- PIOSC (Osc. Interno precisión): 16Mhz +- 3 %
- MOSC (Main Osc.) Cristal, en Launchpad 16 MHz
- LFIOSC (Osc. Interno baja frecuencia): 30KHz +- 50%
- RTC-External (Reloj tiempo real): 32KHz solo en modo Hibernación

Reloj del Sistema SysClk, alimentado por:

- Cualquiera de las fuentes anteriores.
- PLL (400MHz) la señal de reloj referencia debe estar comprendida en el rango de 5MHz a 25MHz
- PIOSC/4 (4 MHz)
- Factor de división

Por lo tanto, para configurar el reloj del sistema usamos:

```
SysCtlClockSet()
```

Cuentas con diferentes opciones:

- Fuente de reloj:
  - SYSCTL_OSC_MAIN → MOSC
  - SYSCTL_OSC_INT → PIOSC
  - SYSCTL_OSC_INT4 → PIOSC/4
  - SYSCTL_OSC_INT30 → LFIOSC
  - SYSCTL_OSC_EXT32 → RTC-EXT
- PLL u otra fuente
- Frecuencia del MOSC (solo ciertos valores posibles)
- Factor de División:
  - SYSCTL_SYSDIV_1,...,SYSCTL_SYSDIV_64
- Deshabilitar PIOSC y/o MOSC

**NO TODAS LAS COMBINACIONES SON POSIBLES**

Algunas de las configuraciones del reloj:

```
// To clock the system from an external source (such as an external crystal oscillator)
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_MAIN);

// To clock the system from the main oscillator
SysCtlClockSet(SYSCTL_USE_OSC | SYSCTL_OSC_MAIN);

// To clock the system from the PLL and select the appropriate crystal
// with one of the SYSCTL_XTAL_xxx values.
SysCtlClockSet(SYSCTL_USE_PLL | SYSCTL_OSC_MAIN);
```

### 2.1.3. Habilitación de periféricos

Los periféricos se encuentran declarados en el fichero **sysctl.h**.

Habilitar periférico para poder usarlo:

```
SysCtlPeripheralEnable(periférico)
```

Deshabilitar periférico:

```
SysCtlPeripheralDisable(periférico)
```

En modos de bajo consumo podemos controlar que periféricos se habilitan o deshabilitan con funciones similares.

### 2.1.4. GPIO

**Figure 1-1. Tiva C Series TM4C123G LaunchPad Evaluation Board**

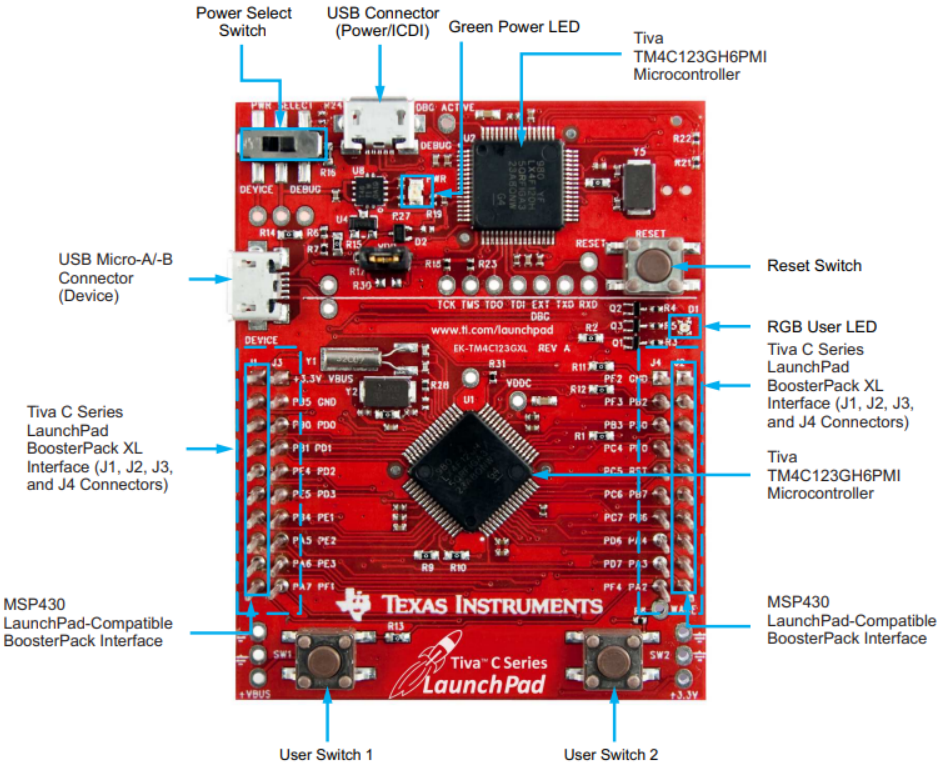



**Table 2-2. User Switches and RGB LED Signals**

| GPIO Pin | Pin Function | USB Device |
|---|---|---|
| PF4 | GPIO | SW1 |
| PF0 | GPIO | SW2 |
| PF1 | GPIO | RGB LED (Red) |
| PF2 | GPIO | RGB LED (Blue) |
| PF3 | GPIO | RGD LED (Green) |

GPIO (General Purpose Input Output Ports), son puertos de entrada/salida de 8 bits. Todos los pines admiten interrupciones, en modo salida se puede elegir la corriente (compromiso entre consumo/energía), resistencias internas de pull-up/down y acceso a través de 2 posibles buses APB o AHB, este último permite mas cambios por segundo consumiendo más.

Usaremos las funciones de la API GPIO:

- Añadir **driverlib/gpio.h** y **driverlib/gpio.c** en el proyecto.
- Y hacer un:

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

Algunas funciones de la API de GPIO:

- Configurar varios pines de un puerto para que tengan una función:

  ```
  GPIOPinType(param1,param2)
  ```

  Las diferentes funciones están en el pdf “**1-PeripheralDriverLibraryUsersGuide.pdf**”.

  El parámetro 1 es la base del puerto:

  - GPIO_PORT**X**_BASE, sustituir la X por el puerto concreto.
  - GPIO_PORT**X**_AHB_BASE si se usa el bus AHB, sustituir la X por el puerto concreto.

  El parámetro 2 es la selección de pines, por ejemplo:

  ```
  // Configurar pines 0 y 1 del puerto A como señales UART
  GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
  ```

- Si se usa una función distinta a Entrada/Salida, la función **GPIOPinType** se debe combinar con **GPIOPinConfigure**.

  ```
  // pin1 puertoF como salida de PWM
  GPIOPinTypePWM(GPIO_PORTF_BASE, GPIO_PIN_1);

  // Configura pin PF1 en su funcion especial como salida PWM del Timer0 (T0CCP1)
  GPIOPinConfigure(GPIO_PF1_T0CCP1);
  ```

  **Se debe incluir** en el proyecto el fichero “driverlib/pin_map.h y la librería:

  ```
  #include “driverlib/pin_map.h”
  ```

- Escribir los bits indicados en pines seleccionador del puerto indicado:

  ```
  // escribe un “1” lógico en los pines 0 y 2 del puerto F,
  // y un “0” lógico en el pin 1 (siempre que los pines 0,1,2 estén configurados
  // como salidas): el resto de pines no es afectado.

  // Recordar que 0x5 (hex) -> 0101, 0 en pin3, 1 en pin2, 0 en pin1 y 1 en pin0
  GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2, 0x5)
  ```

  Tener en cuenta que solo tendrán efecto los bits de las posiciones indicadas por pines, y solo si están configurados como salidas digitales.

- Leer el estado de los pines:

  ```
  // Lee el estado actual del LED y escribe el estado opuesto
  if(GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_2)){
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, 0);
  }else{
    GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_2, GPIO_PIN_2);
  }

  // También podemos leer el estado de varios pines
  // Leer el estado de los 3 pines del puerto F
  GPIOPinRead(GPIO_PORTF_BASE, GPIO_PIN_0|GPIO_PIN_1|GPIO_PIN_2)
  ```

## 2.2. Interrupciones y Timers

### 2.2.1. Interrupciones

- Admite 65 interrupciones y 7 excepciones con 8 niveles de prioridad programables:
  - 0 máxima prioridad y 7 mínima. En el arranque todas las interrupciones tienen prioridad 0. Para reprogramar la prioridad:

    ```
    IntPrioritySet (periférico, nivel);
    ```

- Si se produce una interrupción y se acepta, es procesada de forma automática sin la intervención de programas de usuario. Se almacena el contexto, se accede a la rutina de tratamiento gracias al vector de interrupciones y la salida recupera el contexto accediendo a la instrucción interrumpida.
- La tabla de vectores de interrupción se codifica como un array de punteros a funciones **g_pfnVectors[]**.
  - Inicialmente situada en la dirección 0.
  - Tabla de vectores codificada en el archivo "**tm4c123gh6pm_startup_ccs.c**". En este fichero se definen las funciones por defecto asociadas a fallo y ResetISR.
  - Reubicación de la tabla de vectores y redefinición de las rutinas de interrupción en tiempo de ejecución:

    ```
    IntRegister (periférico, dirección función)
    ```

Usaremos los API Interrupt Controller necesitamos:

- Añade driverlib/interrupt.c, with driverlib/interrupt.h en tu proyecto
- Añadir:

  ```
  #include "driverlib/interrupt.h"
  ```

Además existen también funciones del sistema asociadas a interrupciones que son propias de cada periférico y se encuentran en la API del periférico, ejemplo:

```
GPIOIntEnable();
```

### 2.2.2. Timers

- Timers "divisibles", un TimerA de 32 bits se puede dividir en subTimerA y B de 16 bits.
- 6 Timers de 32bits (TIMER) y 6 de 64 bits (WTIMER).
- Posibilidad de cuenta única o repetitiva.
- En algunos modos de trabajo (PWM, captura) solo se trabaja a nivel de subtimer

Usaremos los API Timer necesitamos:

- Añade driverlib/timer.h y driverlib/timer.c en tu proyecto
- Añadir:

  ```
  #include "driverlib/timer.h"
  ```

Para modo PWM o captura se deben configurar los pines de puertos (CCPs) asociados en sus modos de trabajo "especiales".

Algunas funciones de API de TIMERS:

- Configurar el modo de trabajo del timer, por ejemplo:

  ```
  // Configura el Timer0 para cuenta periodica de 16 bits con la mitad B del TIMER0
  // con 40MHz me da una frecuencia no visible por lo que no vere el led parpadear
  TimerConfigure(TIMER0_BASE, TIMER_CFG_SPLIT_PAIR | TIMER_CFG_B_PERIODIC);
  ```

  Mirar configuraciones "**1-PeripheralDriverLibraryUsersGuide.pdf**" en la página 539.
- Configurar el tiempo de cuenta timer/subtimer:

```
// Carga la cuenta en el Timer0B
TimerLoadSet(TIMER0_BASE, TIMER_B, ui16Period);

// Cuenta para 0.1segundos (si no hay preescalado)
TimerLoadSet(TIMER0_BASE, TIMER_A, ((SysCtlClockGet( ) / 10) -1));
```

Mirar configuraciones “**1-PeripheralDriverLibraryUsersGuide.pdf**” en la página 549.

- Para la gestión de interrupciones de TIMER:

```
// Habilitacion interrupcion TIMEOUT subtimerA del TIMER0
TimerIntEnable(TIMER0_BASE, TIMER_TIMA_TIMEOUT);
```

Mirar configuraciones “**1-PeripheralDriverLibraryUsersGuide.pdf**” existen diferentes configuraciones, buscar por **TimerInt**.

- Configurar el ciclo de trabajo en PWM, por ejemplo:

```
TimerConfigure(TIMER0_BASE, TIMER_CFG_SPLIT_PAIR|TIMER_CFG_B_PWM);
TimerLoadSet(TIMER0_BASE, TIMER_B, ui32Period);

// Ciclo de trabajo al 25% (o 75% según ciclo activo)
// Se recomienda indicar cuentaCiclo como un porcentaje
// de la cuenta indicada en TimerLoadSet
TimerMatchSet(TIMER0_BASE, TIMER_B, ui32Period/4);
```

# Tema 3: Desarrollo de aplicaciones para Sistemas Empotrados basados en microcontroladores

## 3.1. Variables

- Locales: variables temporales que deben permanecer accesibles durante el tiempo de ejecución de la función donde se definen. Ubicadas en la RAM reservada a la PILA y en registros de la CPU
  - static: deben permanecer accesibles durante todo el tiempo de ejecución del código. Ubicadas en la memoria RAM genérica.
- Globales: variables permanentes que deben permanecer accesibles durante todo el tiempo de ejecución del código. Ubicadas en la RAM genérica.
  - static: deben permanecer accesibles durante todo el tiempo de ejecución del código, pero solo son accesibles en el fichero en el que se declaró, no es accesible en otros ficheros. Ubicadas en la memoria RAM genérica.
- Dinámicas: variables creadas con funciones de gestión de memoria como “malloc”. Ubicadas en la RAM en el HEAP.
  - static: indica que las variables son de sólo lectura y no se pueden alterar mediante el código. Ubicadas en la FLASH.

Otros tipos:

- volatile Globales o Locales: indica al compilador que no optimice los accesos a este tipo de variables, ya que sus valores pueden modificarse independientemente de las instrucciones ejecutadas en el código. Ubicadas en la memoria RAM genérica (GLOBALES) o en la PILA o REGISTROS (LOCALES).
- extern Globales: indica que las variables han sido declaradas en otro fichero.
- static Funciones: Indica que a la función solo puede ser accesible en el fichero en el que está definida.
- extern Funciones: indica que la función ha sido implementada en otro fichero de código o librería.

## 3.2. Software

### 3.2.1. Tabla de vectores

Tabla de vectores en “**XXXX_startup_ccs.c**”.

Se utiliza para asignar rutinas de tratamiento de interrupción asignadas a puertos, timers ...

Hay que asegurarse:

- Colocar en la parte del documento de “startup_ccs.c” donde pone “External declaration...”:

```
extern "Nombre_RTI"(void);
```

- Meter la RTI en la tabla de vectores tal y como se ve en las imágenes siguiente.

```
#pragma DATA_SECTION(g_pfnVectors, ".intvecs")
void (* const g_pfnVectors[])(void) =
{
    (void (*)(void))((unsigned long)&__STACK_TOP),
                                    // The initial stack pointer
    ResetISR,                       // The reset handler
    NmiSR,                          // The NMI handler
    FaultISR,                       // The hard fault handler
    IntDefaultHandler,              // The MPU fault handler
    IntDefaultHandler,              // The bus fault handler
    IntDefaultHandler,              // The usage fault handler
    0,                              // Reserved
    0,                              // Reserved
    0,                              // Reserved
    0,                              // Reserved
    IntDefaultHandler,              // SVCall handler
    IntDefaultHandler,              // Debug monitor handler
    0,                              // Reserved
    IntDefaultHandler,              // The PendSV handler
    IntDefaultHandler,              // The SysTick handler
    IntDefaultHandler,              // GPIO Port A
    IntDefaultHandler,              // GPIO Port B
    IntDefaultHandler,                 ...
    IntDefaultHandler,                 ...
.........................
```

Cada entrada es un puntero a función

Manejador por defecto, permite detectar errores parando el programa de forma controlada si se produce una interrupción

- xxxx_startup_ccs.c

## Rutinas por defecto

```
void ResetISR(void)
{

    // Jump to the CCS C initialization routine.  This will enable the
    // floating-point unit as well, so that does not need to be done here.
    __asm("    .global _c_int00\n"
          "    b.w     _c_int00");
}

static void NmiSR(void)
{
    while(1){}; // Enter an infinite loop.
}

static void FaultISR(void)
{
     while(1){}; // Enter an infinite loop.
}

static void IntDefaultHandler(void)
{
     while(1){}; // Enter an infinite loop.
}
```

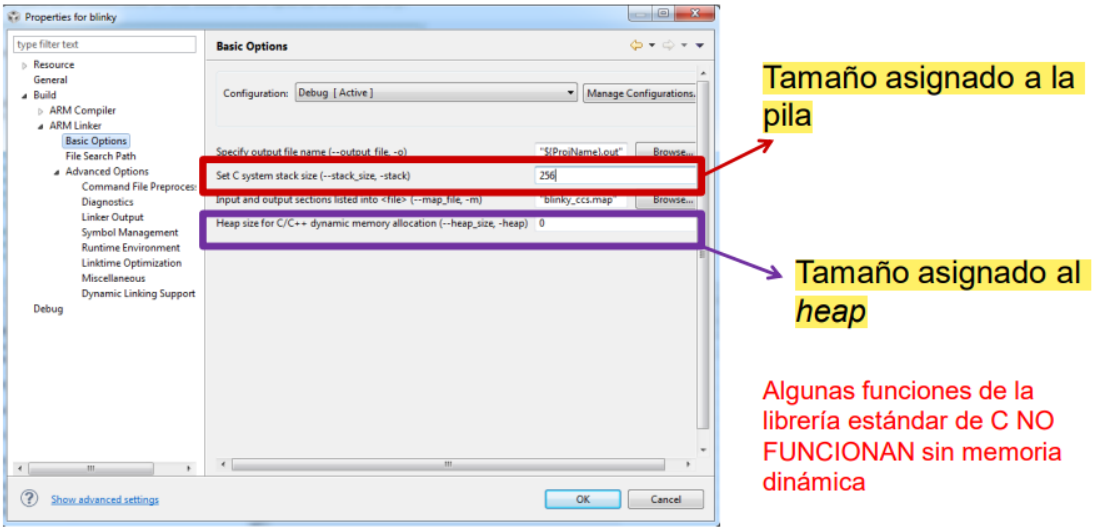
Llama a una función de la biblioteca “runtime” del compilador que inicializa las variables globales y salta a la función main();

Permiten detectar errores parando el programa de forma controlada si se produce una interrupción o una excepción

Si se me “cuelga” el programa y al pausar del depurador está detenido en una de estas funciones, es que ha ocurrido el correspondiente error

### 3.2.2. Pila

El tamaño de la pila se puede configurar, se configura entre el fichero de comandos del enlazados y las opciones básicas del enlazador en el proyecto.




Un desbordamiento de la pila es crítico. Puede provocar un comportamiento errático del sistema dependiendo de la posición en memoria de la pila.

### 3.2.3. Tabla de vectores reubicable

La tabla de vectores se sitúa por defecto en las posiciones bajas de la memoria.

La tabla de vectores en los microcontroladores Cortex M es reubicable en tiempo de ejecución:

- Su posición viene determinada por el registro VTABLE (del NVIC/System Control), que por defecto se inicializa a 0.
- Sustituir una tabla de vectores en ROM por otra tabla de vectores en ROM.
- Mover la tabla de vectores a la RAM y por tanto ser capaz de cambiar la función que gestiona una interrupción o excepción de forma dinámica.
- Empleamos para ello la función:

```
IntRegister(VECTOR_INTERRUPCION,funcionManejadora)
```

La primera vez que se ejecuta copia la tabla de vectores en ROM a la RAM, y cambia el registro VTABLE. A partir de entonces sólo cambia la posición VECTOR_INTERRUPCION del array que está almacenado en RAM.

- Tabla de vectores reubicable

```
void IntRegister(unsigned long ulInterrupt, void (*pfnHandler)(void))
{
    unsigned long ulIdx, ulValue;

    ASSERT(ulInterrupt < NUM_INTERRUPTS); // Check the arguments.
    // Make sure that the RAM vector table is correctly aligned.
    ASSERT(((unsigned long)g_pfnRAMVectors & 0x000003ff) == 0);

    // See if the RAM vector table has been initialized.
    if(HWREG(NVIC_VTABLE) != (unsigned long)g_pfnRAMVectors)
    {
        // Copy the vector table from the beginning of FLASH to the RAM vectortable.
        ulValue = HWREG(NVIC_VTABLE);
        for(ulIdx = 0; ulIdx < NUM_INTERRUPTS; ulIdx++)
        {
            g_pfnRAMVectors[ulIdx] = (void (*)(void))HWREG((ulIdx * 4) + ulValue);
        }
        // Point the NVIC at the RAM vector table.
        HWREG(NVIC_VTABLE) = (unsigned long)g_pfnRAMVectors;
    }

    // Save the interrupt handler.
    g_pfnRAMVectors[ulInterrupt] = pfnHandler;
}
```

Array de punteros a funciones definido en el mismo fichero (almacenado en RAM)

guarda la dirección donde está almacenado el array en el registro VTABLE

### 3.2.4. Excepciones y errores

¿Qué excepciones soporta el Cortex M4?

- Disparadas por errores
  - HardFault: Error durante la excepción o no se pudo gestionar de otra forma
  - Memory Fault: Relacionadas con la protección de memoria (MMU,…)
  - Bus Fault: Relacionadas con acceso incorrecto a memoria, desalineamiento,...
  - Usage Fault: Relacionadas con la ejecución (código u operandos ilegales…)
- Otras:
  - Interrupciones, SysTick
  - PendSV: Petición de servicio al S.O. activando un bit del NVIC (cambio contexto).
  - SVCall: Petición de servicio al S.O. mediante instrucción SVC.
  - Reset

¿Puedo depurar el error de alguna manera?

- Al pausar la depuración se pueden mirar los registros del NVIC/System Control, algunos de los cuales contienen información sobre el error:
  - NVIC_FAULT_ADDR
  - NVIC_MM_ADDR
  - NVIC_FAULT_STAT

## 3.3. Conceptos de FreeRTOS

### 3.3.1. Definición

Sistema operativo en tiempo real (RTOS) donde podremos crear procesos que permitan emplear “paralelización” (realmente ejecutan diferentes partes de un programa, como en concurrencia, ya que solo se tiene una CPU) y que implementaremos mediante librerías.

### 3.3.2. Documentación

Enlace: https://www.freertos.org/RTOS.html

### 3.3.3. Introducción

Las aplicaciones han de tener un bucle infinito, con una serie de flags que controlan la ejecución de distintas partes del código. Esta metodología es apropiada para aplicaciones sencillas pero no para tareas más complejas. RTOS, puede simplificar el desarrollo de las aplicaciones y sus ventajas son:

- Modularidad: dividir la aplicación en tareas de menor complejidad, facilita el desarrollo y depuración.
- Reutilización: reusar distintas tareas en otras aplicaciones.
- Escalabilidad: aumentar la funcionalidad de la aplicación añadiendo nuevas tareas sobre las ya existentes.

El uso de un RTOS implica utilizar recursos del sistema por lo que no siempre es eficiente su uso. Para el usuario, un RTOS se reduce al uso de una API para gestionar las tareas y su interacción.

### 3.3.4. Multitarea

Los recursos del sistema se reparten entre las diferentes tareas de la aplicación, la sensación del usuario es que todas las tareas se están ejecutando concurrentemente permitiendo diseños independientes y cooperativos.

**Dentro del RTOS el responsable de decidir qué tarea es la que se tiene que ejecutar en cada instante de tiempo es el panificador (scheduler)**. Utilizar un sistema de prioridades, las tareas con más prioridad tienen preferencia a la hora de ejecutarse, y para escoger dentro de las tareas de igual prioridad se suele emplear un esquema de tipo round-robin.

### 3.3.5. Cambio de contexto

La operación de desalojar una tarea de la CPU y ocuparla con otra tarea se denomina cambio de contexto. La responsabilidad del RTOS es salvar adecuadamente el contexto de cada tarea antes de desalojarla de la CPU, para posteriormente restaurar esos valores y retomar la ejecución de la misma en el punto donde se quedó antes de desalojarla de la CPU.

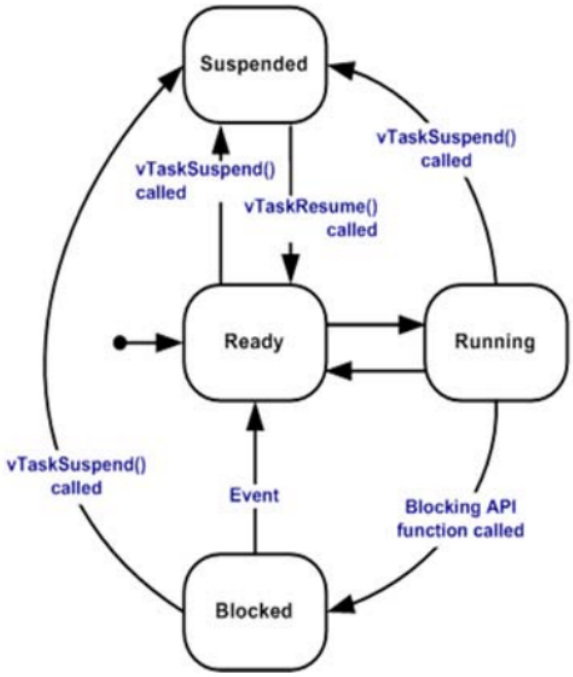
### 3.3.6. Tick del sistema

RTOS necesita un mecanismo de temporización que le permita gestionar el tiempo de ejecución de las distintas tareas, para poder repartir los recursos del sistema. Se suele implementar mediante temporizadores del sistema, generando periódicamente una interrupción (tick) que activa al planificador para cambiar la tarea en ejecución que utiliza la CPU si es necesario. El intervalo del tick que se programe determina el tiempo mínimo en el que el sistema operativo puede hacer alguna operación relacionada con la multitarea.

### 3.3.7. Tareas

Unidad básica en el desarrollo de aplicaciones en un RTOS. Se estructura como un bucle infinito en el que se realizan operaciones y el RTOS se encarga de gestionar. El bucle infinito nunca ha de finalizar, para las tareas su creación y su eliminación se deben realizar mediante las funciones que proporcione el sistema para reservar y liberar memoria adecuadamente.

Cada tarea puede encontrarse en diferentes estados utilizados por el RTOS para controlar su ejecución, permitiendo activar las tareas que se pueden ejecutar o bloquear aquellas tareas que no se pueden ejecutar por encontrarse a la espera de la llegada de algún evento. Normalmente los siguientes estados siempre se encuentran presentes en cualquier RTOS:




Donde:

- Ejecución (running): estado que tiene la tarea que está haciendo uso de la CPU. Sólo una tarea puede estar en este estado en el sistema.
- Activa (ready): estado que tienen las tareas que pueden ejecutarse cuando la tarea actualmente en ejecución abandone la CPU.
- Bloqueada (blocked): estado que tienen las tareas que no pueden ejecutarse por estar a la espera de la llegada de algún evento.

Cada tarea suele poseer una prioridad, es esta forma el planificador puede escoger la más prioritaria a la hora de conmutar la tarea en ejecución de entre las tareas activas. **Las tareas de mayor valor numérico de prioridad, serán las que tengan mayor prioridad lógica.**

### 3.3.8. Interrupciones en sistema multitarea

El RTOS ofrece servicios de un sistema multitarea. La mayoría, están basados en la ejecución de ciertas interrupciones especiales propias de los micros orientados al uso de RTOS.

En ARM Cortex M, las interrupciones son anidables por lo que pueden interrumpirse entre sí. Cada interrupción se le concede un nivel de prioridad, de manera que una interrupción solo podrá interrumpir a otra de menor prioridad. Si la prioridad de la interrupción que se produce es menor que la de la interrupción cuya rutina de atención se está ejecutando, deberá esperar a que acabe antes de ser atendida. Es muy importante el nivel de prioridad que se agina a las interrupciones.

### 3.3.9. Mecanismos de comunicación entre tareas (IPC)

Las tareas necesitan cooperar entre sí, es necesario por tanto que coordinen su operación e intercambien información entre ellas. Cualquier RTOS proporciona, para gestionar la utilización de las tareas, un conjunto de mecanismos de comunicación entre tareas.

Tenemos:

- Colas: permiten el intercambio de datos entre tareas. Formadas por un buffer que almacena ordenadamente los datos procedentes de cualquier tarea, y del cual se pueden leer ordenadamente los datos almacenados por cualquier otra tarea. Son asíncronas.

  Las rutinas de interrupción también pueden escribir o leer de las colas de mensajes para intercambiar datos con las tareas.

  Un parámetro importante de las colas, que suele se configurable, es el tamaño y que cuando se escribe un dato se emplea una nueva posición del buffer para almacenarlo y cada vez que se lee un dato se libera la posición del mismo donde estaba almacenado.

  El uso de colas es eficiente, y los RTOS suelen ofrecer funciones de bloqueos de tareas si intenta escribir en una cola llena o leer de una cola vacía.

- Semáforos: los semáforos binarios (cogen valor 0 o 1, existen los semáforos contador que tienen el mismo funcionamiento pero pueden tomar valores mayores a 1) permiten la sincronización entre tareas. Están formados por un flag que puede estar en un estado abierto o cerrado, y que cualquier tarea puede modificar y leer. Así pues, permiten que las tareas se coordinen entre sí mediante la indicación de eventos.

  Los RTOS suelen ofrecer funcionalidades para bloquear tareas si intentan capturar un semáforo cerrada o activar tareas si se abre un semáforo cerrado.

- Mutexes: en un sistema multitarea es muy frecuente que las tareas hagan uso de un mismo recurso. La mayoría de los RTOS implementan mecanismo de exclusión mutua que permiten bloquear el acceso a un determinado recurso por parte de otras tareas, hasta que la tarea que lo está utilizando lo deja libre y permite que alguna de las tareas en espera pueda hacer uso de él.

## 3.4. Proyectos basados en FreeRTOS

### 3.4.1. Organización del código de un proyecto FreeRTOS

- Demo: directorio que contiene programas de ejemplo para cada port.
- Source: directorio que contiene el código fuente de FreeRTOS, en el deben existir SIEMPRE tres ficheros imprescindibles: : list.c, tasks.c, y queue.c. Además, se añadirán más o menos ficheros adicionales.

### 3.4.2. Ficheros necesarios para FreeRTOS

Para usar FreeRTOS en cualquier aplicación es necesario compilar con el mismo código fuente para el port del compilador y del microcontrolador deseado. Dicho código está contenido en los siguientes directorios y archivos, todos ellos incluidos en el subdirectorio Source:

- include: directorio que contiene los archivos de definición.
- portable: directorio que contiene el código fuente a utilizar para el puerto deseado:
  - MemMang: contiene los archivos utilizados para la gestión de la memoria dinámica del microcontrolador de cualquier puerto. Tiene 3 esquemas de gestión:
    - heap_1.c: permite la asignación de memoria dinámica pero no la liberación de la misma. Sencillo y más eficiente. Sólo se permite la creación de tareas, colas y semáforos, pero no su eliminación por lo que es válido para aplicaciones que utilizan un número fijo de tareas, colas y semáforos y donde su creación se realiza antes de ejecutar el RTO.
    - heap_2.c: permite la asignación y la liberación de memoria dinámica. Más avanzado pero no tan eficiente. Se permite la creación y eliminación de tareas, colas y semáforos. Válido en aquellas aplicaciones que itlizan un número variable de tareas, colas y semáforos y su creación y eliminación se puede realizar en cualquier momento.
    - heap_3.c: no implementa la gestión de la memoria dinámica, utiliza las herramientas del compilador para realizar la gestión.
    - heap_4.c: esquema similar al heap_2 pero permitiendo fusionar bloques de memoria liberados adyacentes empleando más tiempo. → **En la mayoría de casos será la que usemos para la asignatura.**
    - heap_5.c: parecido al heap_4, pero puede gestionar memoria alojada en zonas de memoria diferentes.
  - CCS: directorio que contiene el código fuente para el compilador Code Composer Studio.
    - ARM_CM4F: directorio que contiene el código fuente para los microcontroladores Cortex-M4.
    - list.c, queue.c, tasks.c: ficheros que implementan el core de FreeRTOS y otros ficheros.

### 3.4.3. Recursos utilizados/reservados por FreeRTOS

- Módulo Systick: utilizado como temporizador para generar la base de tiempos (ticks) del sistema. Es posible configurar el intervalo de tiempo de tick.

- Interrupciones SystIck, PendSV, y SVCCall: Estas interrupciones asociadas a FreeRTOS, por lo que se fija su ISR asociada en la tabla de vectores a las funciones de RTOS xPortSysTickHandler, xPortPendSVHandler, y vPortSVCHandler. **Por este motivo, estas funciones deben estar incluidas en la tabla de vectores de interrupción, asociadas a las interrupciones/entradas SysTick handler, PendSV handler, y SVCall handler,** respectivamente.

**Abre el fichero startup_ccs.c, desde el explorador de proyectos, y asegúrate de que las funciones indicadas en el párrafo anterior están definidas como externas, y en las entradas indicadas de la tabla de vectores**. Si decides configurar un proyecto basado en FreeRTOS desde 0, será tu responsabilidad incluir estas funciones en este fichero (CCS no las incluye automáticamente).

### 3.4.4. Fichero de configuración de FreeRTOS

Prácticamente cualquier RTOS posee un mecanismo de configuración que permite adaptar su funcionamiento a la aplicación particular que se esté desarrollando.

FreeRTOS cuenta con un archivo de configuración “FreeRTOSConfig.h” que se debe incluir en cada aplicación que se desarrolle. Algunos de los parámetros más relevantes son los siguientes:

- configUSE_PREEMPTION: determina si el comportamiento del sistema es expropiativo o colaborativo.
- configUSE_IDLE_HOOK: flag que indica si se debe utilizar una función de usuario para la tarea IDLE. Si se encuentra activo es necesario definir una función de usuario para la tarea IDLE.
- configCPU_CLOCK_HZ: valor que contiene la frecuencia de operación del microcontrolador, parámetro necesario, NO sirve para fijar la frecuencia de operación del microcontrolador, sino para indicarle a FreeRTOS cuál es el valor de dicha frecuencia. Debe ser la aplicación la que fije la frecuencia de operación del microcontrolador.
- configTICK_RATE_HZ: valor que contiene la frecuencia de los ticks del sistema que se desea generar. A mayor frecuencia mayor es la concurrencia alcanzada en el sistema, pero así mismo mayor es la sobrecarga de ejecución de FreeRTOS para gestionar las tareas. **Las tareas tienen una prioridad comprendida entre 0 (más baja) y “configMAX_PRIORITIES-1” (prioridad más alta).**
- configMINIMAL_STACK_SIZE: valor que contiene la cantidad de memoria reservada en el sistema para la pila de cada tarea. Puesto que la memoria de un microcontrolador suele ser muy limitada, es MUY IMPORTANTE no definir un número excesivo de variables locales en una tarea, siendo preferible en caso de ser necesario definir variables globales.
- configTOTAL_HEAP_SIZE: valor que contiene la cantidad de memoria reservada en el sistema para la operación de FreeRTOS, necesario para la gestión de las tareas, colas, semáforos...

En “FreeRTOSConfig.h” la eliminación de determinados mecanismos permite el ahorro de memoria RAM y de código cuando la aplicación no requiera de tales mecanismos.

### 3.4.5. Programa principal, main, de FreeRTOS

El elemento fundamental de una aplicación FreeRTOS es el programa principal (main).

Ejemplos:

```
int main(void){

  // Set the clocking to run at 40 MHz from the PLL.
  // Observa  que  la  etiqueta  configCPU_CLOCK_HZ definida  en
  // FreeRTOSConfig.h, coincide con este valor. No olvides modificar esta etiqueta SIEMPRE
  // para que coincida con el valor de reloj configurado  (o, alternativamente,
  // definir la etiqueta como una llamada a SysCtlClockGet(…)
  MAP_SysCtlClockSet(SYSCTL_SYSDIV_5|SYSCTL_USE_PLL|SYSCTL_XTAL_16MHZ|SYSCTL_OSC_MAIN);

  // Initialize the UART de los pines del puerto A and configure it for 115,200, 8-N-1 operation.
  // se usa para mandar mensajes por el puerto serie
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_UART0);
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOA);
  MAP_GPIOPinConfigure(GPIO_PA0_U0RX);
  MAP_GPIOPinConfigure(GPIO_PA1_U0TX);
  MAP_GPIOPinTypeUART(GPIO_PORTA_BASE, GPIO_PIN_0 | GPIO_PIN_1);
  UARTStdioConfig(0, 115200, SysCtlClockGet());

  //Inicializa el puerto F (LEDs)
  MAP_SysCtlPeripheralEnable(SYSCTL_PERIPH_GPIOF);
  MAP_GPIOPinTypeGPIOOutput(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3);
  MAP_GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3, 0); //LEDS APAGADOS

  // Crear tareas
  if((xTaskCreate(LEDTask, "Led1", LED1TASKSTACKSIZE,pui32parametros1,
     tskIDLE_PRIORITY + LED1TASKPRIO, NULL) != pdPASS){

      while(1){

      }
  }

  // ARRANCAR EL PLANIFICADOR
  // siempre tras crear las tareas y recursos. La función vTaskStartScheduler
  // es un bucle infinito del que nunca se vuelve
```

```
  vTaskStartScheduler();

  // Bucle infinito → si el arranque del planificador ha sido correcto,
  // nunca llegará a ejecutarse este código, es buena idea poner aquí un bucle
  // infinito para “atrapar” un posible mal funcionamiento del sistema
  while(1){
  }
}
```

### 3.4.6. Tareas en FreeRTOS

Cada tarea FreeRTOS tiene su propio contexto y pila consumiendo RAM. Por lo general, las tareas conviven en un sistema expropiativo, es decir, siempre se ejecuta la tarea con mayor prioridad de entre las que estén preparadas y si hay varias con la misma prioridad, se reparten el tiempo de CPU.

Una tarea se crea en FreeRTOS mediante:

```
xTaskCreate();
```

Sus parámetros son:

- pxTaskCode: puntero al código de la tarea.
- pcName: nombre que se asignará a la tarea.
- usStackDepth: tamaño de la pila de la tarea.
- pvParameters: puntero a parámetros que se pueden pasar a la tarea en su creación. Si se van a usar a lo largo de la vida de la tarea, se deben crear en el programa como globales.
- uxPriority: prioridad de la tarea. Debe estar entre 0 y configMAX_PRIORITIES.
- pxCreatedTask: puntero a la tarea creada, si es NULL representa que no guardamos el punto en ninguna parte.

Por ejemplo:

```
// La llamada devuelve "pdPASS" si todo ok.
// De los contrario se atrapa el error en el while
if(xTaskCreate(LEDTask, "Led1", LED1TASKSTACKSIZE, pui32Parametros1,
  tskIDLE_PRIORITY + LED1TASKPRIO, NULL)!= pdPASS){
  while(1){}
}
```

El código de una tarea se define a partir de la macro:

```
static portTASK_FUNCTION(vFunction,pvParameters){
  ...
}
```

El código de la tarea NUNCA DEBE RETORNAR dentro de ella se ejecuta un bucle infinito. Por ejemplo:

```
static portTASK_FUNCTION(LEDTask,pvParameters){
  int32_t i32Estado_led=0;
  uint32_t ui32Puerto=((uint32_t *)pvParameters)[1];
  uint32_t ui32Intervalo=((uint32_t *)pvParameters)[0];

  // Loop forever.
  while(1){
    MAP_SysCtlDelay(MAP_SysCtlClockGet()/(3*1000))*ui32Intervalo);//Espera ineficiente (bucle)
    i32Estado_led=!i32Estado_led;

    if(i32Estado_led){
      MAP_GPIOPinWrite(GPIO_PORTF_BASE, ui32Puerto , ui32Puerto);
    }else{
      MAP_GPIOPinWrite(GPIO_PORTF_BASE, ui32Puerto,0);
    }
  }
}
```

OJO:

- static portTASK_FUNCTION(LEDTask,pvParameters) es igual que static void LEDTask portTASK_FUNCTION(void * pvParameters)

Para la creación de varios hilos o tareas que utilizan la misma función que implementa su comportamiento es posible siempre que dicha función no tenga variables estáticas en sucesivas llamadas, ni retorna un puntero a datos estáticos, ni llama a subrutinas que tengan estas características.

Al crear una tarea con xTaskCreate, se le asigna una prioridad a través del penúltimo parámetro de la llamada. Las tareas comparten el uso de la CPU, solo la tarea activa con la máxima prioridad la que tome el control de la CPU y ejecute su programa. Si hay varias tareas activas que tienen la misma prioridad el SO repartirá el uso de la CPU, dándole a cada una un porcentaje de tiempo para ejecutarse.

El TICK del sistema marca el tiempo mínimo en el que el RTOS es capaz de detectar que ha pasado algo,  o hacer algo. Se implementa mediante una interrupción que permite actualizar el valor de cuenta de ticks, determinar posibles cambios de contexto y chequear si hay que realizar alguna actividad al haber alcanzado un cierto número de ticks. El valor de TICK se define en el fichero “FreeRTOSConfig”con la clave:

```
configTICK_RATE_HZ
```

El principal inconveniente de la implementación de tarea actual es que el bucle de espera se basa en una espera activa que tarda más tiempoa. Además, si una tarea activa es más prioritaria que las otras, impide que las demás se ejecuten.

Una alternativa mejor es usar funciones que calculen el tiempo de espera medio en ticks. Se deben usar funciones

```
void vTaskDelay(const TickType_t xTicksToDelay);
void vTaskDelayUntil(TickType_t *pxPreviousWakeTime, const TickType_t xTimeIncrement);
```

- xTicksToDelay: número de ticks que la tarea permanecerá suspendida.
- pxPreviousWakeTime: puntero a una variable con el instante en ticks en la que la tarea se activó por última vez. Esta variable se actualiza automáticamente tras cada llamada a la función.
- XtimeIncrement: número de ticks para la reactivación de la tarea.

vTaskDelay es más conveniente para retardos únicos, no es aconsejable para cuentas periódicas.

vTaskDelayUntil realiza una cuenta absoluta de ticks, por lo que se consigue mejor precisión en el control de tareas periódicas.

Por ejemplo:

```
static portTASK_FUNCTION(LEDTask,pvParameters){

    int32_t i32Estado_led=0;

    // Variable para la recogida de los ticks
    TickType_t ui32LastTime;

    //pvParameters es un puntero que se le pasa a la funcion vTaskCreate.
    uint32_t ui32Puerto=((uint32_t *)pvParameters)[1];
    uint32_t ui32Intervalo=((uint32_t *)pvParameters)[0];

    UARTprintf("\n\nWelcome to the TIVA EK-TM4C123GXL FreeRTOS Demo!\n");
    // Get the current tick count.
    ui32LastTime = xTaskGetTickCount();


    // Loop forever.
    while(1){

        // Como se trata de controlar una tarea periódica, por ello
        // usaremos vTaskDelayUntil
        // Suspende la tarea un tiempo y pasado un tiempo la vuelven a arrancar
        //Espera del RTOS (eficiente, no gasta CPU)
        vTaskDelayUntil(&ui32LastTime, ui32Intervalo);
        i32Estado_led=!i32Estado_led;

        if(i32Estado_led){
          GPIOPinWrite(GPIO_PORTF_BASE, ui32Puerto , ui32Puerto);
        }else{
          GPIOPinWrite(GPIO_PORTF_BASE,  ui32Puerto,0);
        }
    }
}
```

Para convertir la cuenta en TICKS a milisegundos:

```
portTICK_PERIOD_MS
```

Por ejemplo, para establecer un retardo de 500 ms con la función:

```
vTaskDelay(500/portTICK_PERIOD_MS)
```

En FreeRTOS existe una tarea especial predefinida llamada IDLE. Esta tarea es la que se ejecuta cuando ninguna otra tarea está activa y tiene el nivel de prioridad más bajo (0) definido por la constante tskIDLE_PRIORITY. Definida por FreeRTOS.

La implementación de la tarea IDLE es código interno de FreeRTOS que no se debe modificar, para ello, el usuario puede definir una función “gancho”, vApplicationIdleHook() que se ejecutaría en cada iteración de la tarea IDLE. Para habilitar esta funcionalidad es necesario en el fichero “FreeRTOSConfig.h” poner a 1 la etiqueta:

```
configUSE_IDLE_HOOK
```

Y definir en el proyecto una función con el siguiente nombre y prototipo:

```
void vApplicationIdleHook(void){
  // Codigo de usuario asociado a la tarea IDLE;
  // Se suelen implementar funciones del micro de bajo consumo
  SysCtlSleep();
}
```

También es posible asociar una función de usuario a la interrupción de TICK, para ello, poner 1 en “FreeRTOSConfig.h”:

```
configUSE_TICK_HOOK
```

### 3.4.7. Gestión de Memoria

Cada tarea tiene una pila propia, además, el kernel del SO necesita usar memoria dinámica o heap, para la creación de recursos IPC.

Heap, la cantidad de memoria reservada se define en bytes en FreeRTOSConfig.h:

```
configTOTAL_HEAP_SIZE
```

Se puede asociar una función MALLOC_FAILED_HOOK, para que se ejecute cuando se produzca un fallo en la adquisición de memoria heap. Para ello se debe definir a 1 la constante de FreeRTOSConfig.h:

```
configUSE_MALLOC_FAILED_HOOK
```

La función hook tendrá la siguiente estructura:

```
void vApplicationMallocFailedHook(void){
  // Codigo de usuario asociado a un fallo de heap;
}
```

Cada tarea se le asocia una pila propia de un determinado tamaño. La tarea usa la pila para almacenar variables locales, anidamiento de llamadas y salvar el contexto de la tarea al realizar el cambio de contexto.

La pila es utilizada para almacenar las variables locales de la tarea, así como el árbol de llamadas a funciones, almacenar las variables locales de las funciones llamadas por la tarea o de las subfunciones llamadas por estas. La cantidad de memoria de pila depende de a

cuántas subfunciones se llamen de forma anidada y del tamaño de las variables locales que dichas funciones. Lo normal y recomendable suele ser realizar una estimación más o menor conservadores y ajustarla durante la depuración.

FreeRTOS implementa mecanismos para detectar posibles desbordamientos de pila para ello poner a 1 o 2 en el fichero “FreeRTOSConfig.h”:

```
configCHECK_FOR_STACK_OVERFLOW
```

En la mayoría de casos dejarlo a 2.

Cuando se activan los mecanismos de comprobación de pila, se debe definir una función:

```
void vApplicationStackOverflowHook( TaskHandle_t xTask, signed char *pcTaskName ) {
  // Codigo de usuario asociado a un fallo de pila;
}
```

### 3.4.8. Interrupciones en el Port FreeRTOS para el Cortex-M4

En ARM se permite que las interrupciones sean anidables, es decir, pueden actuar incluso cuando se está atendiendo a otra interrupción, siempre y cuando la interrupción “entrante” tenga mayor prioridad que la interrupción que se está atendiendo en ese momento.

En el Cortex-M la prioridad de una interrupción es mayor cuanto menor es su valor numérico. (justo al contrario que las prioridades de las tareas en FreeRTOS): la prioridad máxima de interrupción será la 0; la mínima 255. Ten en cuenta además que todas las interrupciones son más prioritarias que todas las tareas y podrían interrumpirlas si están habilitadas.

Para asegurar que una ISR no es interrumpida por otra durante una sección crítica, FreeRTOS utiliza la máscara de interrupción, e inhibe todas las interrupciones con prioridad inferior a una dada, determinada por la constante de configuración:

```
configMAX_SYSCALL _INTERRUPT_PRIORITY
```

Cuando la sección crítica finaliza, FreeRTOS restablece la máscara de interrupción al nivel de la interrupción que se estaba cursando, permitiendo que interrupciones más prioritarias que ella la interrumpan.

A todas las interrupciones de los programas de aplicación se les deberá asignar una prioridad. Según la prioridad asignada, formará parte de uno de entre dos subgrupos de niveles:

- aquellas que serán enmascaradas cuando se estén ejecutando ciertas funcionalidades de FreeRTOS.
- aquellas que estarán habilitadas en todo momento. Del fichero “FreeRTOSConfig.h”, la constante:

  ```
  configMAX_SYSCALL _INTERRUPT_PRIORITY
  ```

  determina el nivel de prioridad frontera de estos dos subgrupos. Se debe implementar definiendo en “FreeRTOSConfig.h” una constante de la forma:

  ```
  /* Priority 5, or 0xA0 as only the top three bits are implemented. */
  #define configMAX_SYSCALL_INTERRUPT_PRIORITY ( 5 << 5 )
  ```

FreeRTOS utiliza, en el “port” para la arquitectura ARM Cortex M, tres rutinas de interrupción:

- Systick: controla el tick del RTOS.
- La excepción software SVC para el arranque del sistema.
- La excepción pendSV para cambio de contexto.

Todas se configuran en un parámetro de “FreeRTOSConfig.h”:

```
/* Priority 7, or 0xE0 as only the top
three bits are implemented. This is the lowest priority. */
define configKERNEL_INTERRUPT_PRIORITY ( 7 << 5 )
```

Tener en cuenta que el nivel de prioridad de todas las interrupciones de ser mayor o igual (por lo tanto, han de tener un valor menor o igual) a:

```
configKERNEL_INTERRUPT_PRIORITY
```

Por defecto, las interrupciones Cortex-M tienen por defecto nivel de prioridad 0, que es la máxima.

### 3.4.9. Mecanismos de comunicación entre procesos (IPC) en FreeRTOS

Tenemos los semáforos binarios, su lógica:

- La tarea que quedará suspendida esperando, dentro del bucle infinito, a que se abra el semáforo.
- La interrupción asociada al botón abrirá el semáforo.
- La tarea, al abrirse el semáforo, pasará y volverá a cerrarlo.

Este ciclo se repetirá indefinidamente.

La tarea se suspende mientras el semáforo está cerrado, por lo que no participa del uso de la CPU y permite a otras tareas, incluso de menor prioridad, tomar el control de la misma.

Para crear el sistema de coordinación basado en un semáforo binario:

- Añadir:

  ```
  #include "semphr.h"
  ```

- Añadir una variable global:

  ```
  SemaphoreHandle_t nombre_semaforo;
  ```

- Añadir la creación del semáforo binario antes de poner en marcha el planificador usando xSemaphoreCreateBinary.

  ```
  int main(void){

    semaforo_freertos1 = xSemaphoreCreateBinary();
    if((semaforo_freertos1 == NULL)){
      while(1); //No hay memoria para los semaforo
    }

    // Start the scheduler. This should not return.
    vTaskStartScheduler();
  }
  ```

La apertura de semáforos binarios se realiza mediante:

- xSemaphoreGive (para tarea)
- xSemaphoreGiveFromISR (para interrupción)

Para obtener el semáforo binario:

- xSemaphoreTake (tarea), tiene un segundo argumento xTicksToWait que es el tiempo máximo (en ticks) que se quedará en espera de que alguien abra el semáforo, este tiempo oscila entre 0 a infinito (representado por el valor portMAX_DELAY)
- xSemaphoreTakeFromISR (interrupción), no tienen tiempo de espera.

Decir que si el semáforo binario ya ha sido ocupado la tarea que intenta cogerlo se quedará en espera, las funciones que puedan llamarse en rutinas de interrupción no son bloqueantes. Tener en cuenta que las llamadas devuelven pdTrue si todo ok y pdFalse si no.

Ya que los semáforos binarios se crean cerrados por defecto, podemos añadir al código de la tarea una llamada a xSemaphoreTake previa a la entrada en el bucle infinito para asegurar que el semáforo queda cerrado al inicio:

```
static portTASK_FUNCTION(Switch1Task,pvParameters){
  xSemaphoreTake(nombre_semaforo_creado,0);
  //...
  while(1){
    //...
  }
}
```

También existen semáforos contadores capaces de tomar valores mayores de 1.  Mismo funcionamiento que los binarios pero se debe emplear xSemaphoreCreateCounting:

```
SemaphoreHandle_t xSemaphoreCreateCounting(unsigned BaseType_t uxMaxCount,
                                           unsigned BaseType_t uxInitialCount);
```

Los parámetros son la cuenta máxima del semáforo y su valor inicial. Para utilizar semáforos contadores poner en “FreeRTOSConfig.h” a 1:

```
configUSE_COUNTING_SEMAPHORES
```

FreeRTOS implementa mecanismos para el intercambio de información entre tareas mediante colas, para ello, se deben incluir los ficheros “queue.h” y “queue.c” bajo la carpeta FreeRTOS/Source además añadir:

```
#include "queue.h"
```

Definimos una variable global:

```
QueueHandle_t cola;
```

Y creamos la cola antes de poner en marcha el planificador:

```
int main(void){

  cola_freertos=xQueueCreate(16,sizeof(uint32_t));

  if (NULL==cola_freertos){
    while(1);
  }

  // Start the scheduler. This should not return.
  vTaskStartScheduler();
}
```

Los parámetros de la función xQueueCreate son el número de datos que caben en la cola y el tamaño de cada dato.

Por ejemplo:

```
// Escribe en la cola freeRTOS
// Para el caso de una rutina de tratamiento
xQueueSendFromISR(cola_freertos,&i32PinStatus,&higherPriorityTaskWoken);

// Recogida de datos
// Definida en otra parte del programa
// Esta tarea se quedará suspendida en la instrucción xQueueReceive, mientras la cola de mensajes esté vacía.
static portTASK_FUNCTION(Switch2Task,pvParameters){
  uint32_t ui32Status;

  // Loop forever.
  while(1){
    if(xQueueReceive(cola_freertos,&ui32Status,portMAX_DELAY)==pdTRUE){
      // Decodifica lo recogido en ui32Status y enciende los LEDs correspondientes

    }else{ //Por esta rama nunca llega excepto que la espera no sea portMAX_DELAY
           // Como no leemos el valor de status, apagamos los LEDs
        GPIOPinWrite(GPIO_PORTF_BASE, GPIO_PIN_1|GPIO_PIN_2|GPIO_PIN_3,0);
    }
  }
}
```

También contamos con Mutexes. Para utilizar un mutex:

```
#include "semphr.h"
```

Definir una variable global tipo semáforo:

```
SemaphoreHandle_t g_pUARTSemaphore;
```

Crear un mutex a partir de ella en el main y antes de poner en marcha el planificador:

```
int main(void){
  g_pUARTSemaphore = xSemaphoreCreateMutex();
}
```

La forma de utilizar el mutex es capturarlo antes de utilizar el recurso compartido y liberarlo tras utilizarlo. Si una tarea A intenta capturar el mutex y éste está ocupado, significa que hay otra tarea B utilizando el recuro compartido y que dicho recurso está ocupado, por lo que la tarea A se bloquea hasta que la otra tarea B libere el recurso y suelte el mútex.

Ejemplo:

```
xSemaphoreTake(g_pUARTSemaphore,portMAX_DELAY);
UARTprintf("\n Exclusion mutua");
xSemaphoreGive(g_pUARTSemaphore);
```

Los flags de eventos son otro tipo de IPC para la sincronización entre tareas que se pueden manipular por las tareas como por las rutina de interrupción.

Se debe habilitar en el fichero de configuración “FreeRTOSConfig.h”:

```
#define INCLUDE_xEventGroupSetBitFromISR 1
#define INCLUDE_xTimerPendFunctionCall 1
```

Incluir:

```
#include "event_groups.h"
```

Definir una variable manejadora de tipo EventGroupHandle_t y llamar a una función de creación es la estructura antes de usarlos:

```
FlagsEventos = xEventGroupCreate();
if( FlagsEventos == NULL ){
  while(1);
}
```

La activación de los bits del grupo de  flags se puede hacer, bien desde una  tarea con la  función xEventGroupSetBits, bien desde una interrupción, con xEventGroupSetBitsFromISR.

```
BaseType_t xEventGroupSetBitsFromISR(EventGroupHandle_t xEventGroup,const EventBits_t uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken );
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet );
```

Además del grupo de evento afectado (primer parámetro) se indica como segundo parámetro la máscara de bits a activar. La función llamada desde ISR añade un parámetro extra para “despertar” a una tarea como resultado de la activación de nuevos flags.
Las funciones correspondientes para desactivar flags serían:

```
EventBits_t xEventGroupClearBitsFromISR(EventGroupHandle_t xEventGroup,const EventBits_t uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken );
EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet );
```

Leer el estado de los flags del grupo de eventos:

```
EventBits_t xEventGroupGetBitsFromISR(EventGroupHandle_t xEventGroup);
EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup);
```

La “espera” hasta la activación de unos flags en concreto del grupo, se consigue con:

```
EventBits_t xEventGroupWaitBits(const EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToWaitFor,
                                const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits, TickType_t xTicksToWait );
```

Parámetros:

- El primero (xEventGroup) es el manejador del grupo de eventos, al igual que en otras funciones.
- El segundo, uxBitsToWaitFor, es una máscara que indica los bits cuya activación estamos esperando.
- El tercero, xClearOnExit, es un booleano (pdTRUE o pdFALSE) para indicar si queremos borrar o no los flags.
- El cuarto,xWaitForAllBits, es otro booleano que indica si queremos esparar a que cualquiera de los bits indicados por la máscara se active (pdFALSE), o si queremos esperar a que todos los bits indicados por la máscara se active (pdTRUE).
- El último parámetro,xTicksToWait es el número de ticks que queremos esperar a que se de la condición

Por último tenemos los conjunto de colas, otro mecanismo que permite que una tarea atienda a la vez a varias colas y/o semáforos. Permiten agrupar varias colas y/o semáforos de forma que una tarea se bloquee hasta que ocurra algo en uno de ellos.

Para utilizar los Queue Sets habilitar la definición en “FreeRTOSConfig.h”:

```
#define configUSE_QUEUE_SETS
```

Incluir:

```
#include "queue.h"
```

Definir una estructura QueueSetHandle_t mediante la función xQueueCreateSet().

Se deben crear los semáforos y colas que serán los miembros del conjunto, para añadir estos elementos al conjunto deben estar vacíos, por lo que el mejor momento para añadirlos es nada mas crearlos.

Para añadir un nuevo miembro al conjunto de colas se usa la función:

```
BaseType_t xQueueAddToSet (QueueSetMemberHandle_t xQueueOrSemaphore,QueueSetHandle_t xQueueSet );
```

Eliminar un miembro del conjunto:

```
BaseType_t xQueueRemoveFromSet (QueueSetMemberHandle_t xQueueOrSemaphore,QueueSetHandle_t xQueueSet );
```

Para bloquear una tarea ante cualquiera de los elementos de un conjunto de colas:

```
QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet,const TickType_t xTicksToWait);
QueueSetMemberHandle_t xQueueSelectFromSetFromISR (QueueSetHandle_t xQueueSet);
```

La llamada devuelve un puntero al elemento (semáforo o cola) que ha permitido el desbloqueo y poder realizar una acción diferente en cada caso. Si expira el tiempo de espera, la llamada devuelve NULL. En la versión de interrupción, simplemente se leerá el estado de todos los miembros y se devolverá NULL inmediatamente, si ninguno permite el desbloqueo.

En el caso de la cola, podemos leer el dato recibido mediante la función xQueueReceive(), de lo contrario el dato permanecerá almacenado en la cola.

### 3.4.10. Timers Software

En aplicaciones resulta frecuente tener que realizar determinadas acciones de forma periódica, podríamos utilizar los timers hardaware e interrupciones del microcontrolador. No obstante, son escasos, hay que planificar muy bien su uso.

FreeRTOS ofrece un mecanismo de timers software que se integran mejor en los proyectos construidos a partir de FreeRTOS con un menor impacto sobre la ejecución del resto de tareas.

Incluir los ficheros timers.h y timers.c en la carpeta FreeRTOS/Source y añadir:

```
#include "timers.h"
```

Y habilitar la constante del fichero “FreeRTOSConfig.h”:

```
configUSE_TIMERS
```

Crear timers:

```
// Recordar que portTICK_PERIOD_MS permite convertir ms en ticks del sistema
xTimer = xTimerCreate(“Timer”,duracion/portTICK_PERIOD_MS,pdTRUE,pvTimerID ,vTimerCallback);
if( xTimer == NULL ){
  while(1);
}
```

El primer parámetro es el nombre del timer y se usa para depurar, el segundo es el intervalo de cuenta en ticks, el tercero indica si el timer se autorecargará cuando llegue a su valor final, la cuenta será periódica, el cuarto es un ID del timer que permitiría identificarlo en caso de ser necesario y el quinto es el nombre de la función callback de usuario que se ejecutará cuando el timer llegue a la cuenta indicada.

Arrancar el timer:

```
if( xTimerStart( xTimer, 0 ) != pdPASS ){
  /* The timer could not be set into the Active state. */
}
```

Devuelve el valor pdPASS.

Si el timer se arranca desde una rutina de interrupción se utiliza la versión:

```
xTimerStartFromISR( xTimer, &higherPriorityTaskWoken)
```

El timer se puede detener en cualquier momento usando las llamadas  xTimerReset o xTimerStartFromISR. Mismo parámetros que xTimerStart.

Si se ha creado un timer en modo repetitivo, cambiar el periodo indicado en la llamada de creación del timer, usando las funciones xTimerChangePeriod, o xTimerChangePeriodFromISR, tarea o desde una rutina de interrupción. Un nuevo parámetro para representar el nuevo periodo del timer.

Cuando el timer llega al número de ticks estipulado para el timer, se ejecuta una función callback definida por el usuario:

```
void vTimerCallback(TimerHandle_t pxTimer)
```

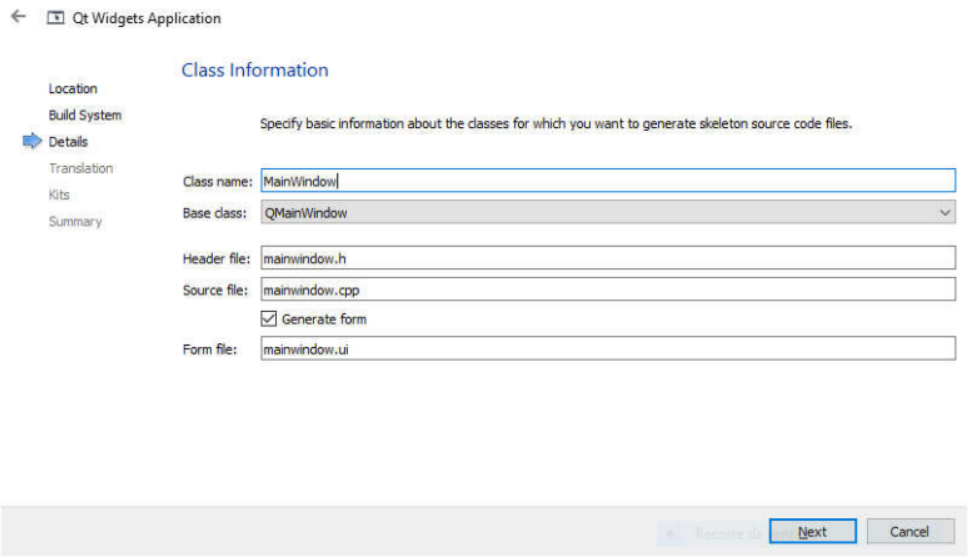
Se debe evitar que una función callback de un timer FreeRTOS se quede bloqueada indefinidamente o por mucho tiempo.

## 3.5. Introducción a QT

### 3.5.1. Crear un entorno en QT

Pasos a seguir para crear un entorno de QT:

- Iniciar el programa QTCreator. Pulsar en File → New File or Project ó Project → New. Si tenemos un proyecto ya existente abrimos el fichero .pro de la carpeta.
- Nosotros trabajaremos principalmente en el desarrollo de aplicciones gráficas utilizando C++, por lo que vamos a seleccionar: Application y QT Widgets Application como opción de desarrollo y pulsaremos el botón Choose...
- Configuramos el nombre del proyecto y la carpeta que lo va a contener y seleccionamos el sistema de construcción de la aplicación QT. Seleccionamos qmake.
- Posteriormente se abrirá una ventana similar a la siguiente:




QTCreator nos creará automáticamente una clase llamada MainWindow, que deriva de la clase QMainWindow de la biblioteca QT, y cuyo código quedará almacenado en los ficheros mainwindow.cpp (implementación) y mainwindow.h (declaración). Además, nos creará un fichero de interfaz de usuario mainwindow.ui donde podremos editar el tamaño y aspecto de nuestra ventana principal y añadirle botones, controles e indicadores entre otros.

- Posteriormente seleccionamos el idioma Spanish(Spain).
- Seleccionamos la versión de QT y compilador que vamos a utilizar en el proyecto, en nuestro caso, “Desktop Qt 6.2.3 MinGW 64-bit”.
- Finalmente, nos aparecerá un diálogo con el resumen de opciones elegidas para el proyecto, finalizamos.

### 3.5.2. Opciones de compilación para todas las bibliotecas adicionales

Conforme se van añadiendo opciones al fichero de proyecto (.pro) para añadir soporte a los componentes QT adicionales en bibliotecas como analogwidgets y qwt (y más adelante para MQTT) se deberán incluir las siguientes opciones en el fichero .pro:

```
QT += core gui widgets svg charts network serialport
CONFIG += qwt analogwidgets qmqtt colorwidgets embeddeduma
```

### 3.5.3. Desactivación del plugin “Clang-code-model”

Este plugin realiza un análisis sintáctico y semántico del código en tiempo real y va mostrando consejos en pantalla. El principal motivo para desactivarlo es que afecta al autocompletado del código. Para desactivarlo:

help → About plugins → Buscar en la sección C++ buscar “ClangCodeModel”, desmarcamos la pestaña y cerramos el configurador de plugins y reiniciamos QT.

Esto es necesario hacerlo sólo una vez.