## Sistemas Embebidos II - IMT 322

# Laboratorio 1: Threads en FreeRTOS

**Deadline:** 24/02/2023

## Introducción

Este laboratorio introduce el concepto de tasks en un sistema operativo de tiempo real (RTOS). Para este propósito, se uso FreeRTOS, que está diseñado para ejecutar aplicaciones con requisitos de tiempo real suaves en sistemas embebidos. Permite que una aplicación se divida en tareas independientes, lo cual es importante cuando se trata de diferentes requisitos en tiempo real. Conceptualmente, FreeRTOS es un kernel en tiempo real que se ocupa del manejo de tareas y permite que las aplicaciones integradas usen multitarea. A diferencia de las plataformas multiprocesador, los sistemas embebidos a menudo contienen solo un procesador que puede ejecutar solo una tarea a la vez. En FreeRTOS, esta tarea se define en estado Running, cómo se ilustra en la Imagen 1. Todas las demás tareas están en estado No running. El kernel decide cuándo y qué tarea debe pasar al estado Running y, en consecuencia, determina un cronograma para la aplicación.

Se realizara una aplicación básica con múltiples tareas y se presentara los conceptos de tasks y cronogramas. Se recomienda usar la documentación. El libro 'Mastering the FreeRTOS Real Time Kernel' y el reference manual del sistema operativo utilizado 'FreeRTOS Reference Manual' para entender mejor cada una de las funciones que se esta usando.

## Objetivos del Lab

- Introduccir a RTOS
- Aprender sobre tasks, threads
- Entender el proceso de creacion de tasks
- Entender el rol del kernel 'FreeRTOS'
- Crear su primer programa en FreeRTOS

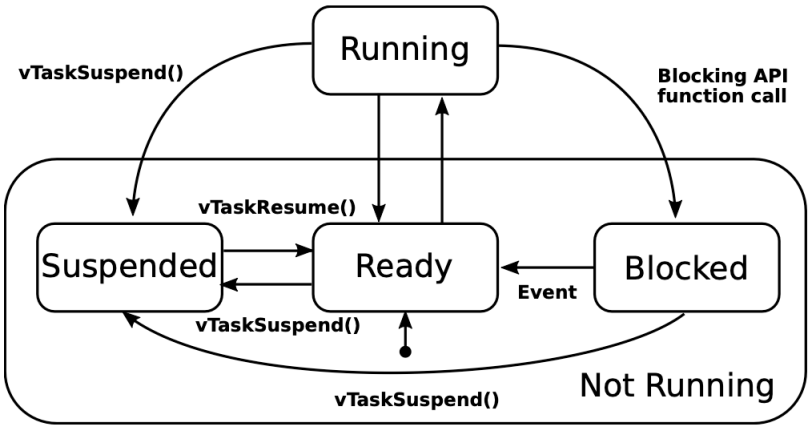


Imagen 1: Estados de una tarea en FreeRTOS

## Introuduccion a FreeRTOS

En FreeRTOS, las tareas pueden ser creadas usando la función xTaskCreate(). La declaración se muestra en el Codigo 1. Como puede ver esta función tiene varios parámetros. Revisemos que quiere decir cada uno de estos.

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,      // Pointer to the task function
                        const char * const pcName,      // Descriptive name of the task
                        uint16_t usStackDepth,     // Stack size allocated for the task
                         void *pvParameters ,          // Task specific parameters
                         UBaseType_t uxPriority ,      // Priority of the task
                         TaskHandle_t *pxCreatedTask      // Handle to the task
                      );
```

Codigo 1: Declaracion de xTaskCreate()

- pvTaskCode - este argumento apunta a una función que contiene la lógica de la tarea. Esta función nunca debe terminarse. Por lo tanto, la funcionalidad de la tarea debe encapsularse en un bucle infinito. El Código 2 muestra un ejemplo de una función de tarea de este tipo.
- pcName - este argumento debe elegirse para que sea un nombre descriptivo para la tarea. FreeRTOS no utiliza este nombre, pero facilita el proceso de depuración.
- usStackDepth - para cada tarea, se preasignará un espacio de memoria. El programador debe especificar el tamaño del stack según los requisitos de memoria de la tarea. En este laboratorio, configuramos usStackDepth en un valor predeterminado de 1000. En práctica, el tamaño de pila requerido se puede determinar con las herramientas de FreeRTOS y luego se puede configurar en un valor razonablemente pequeño.
- pvParameters - este argumento permite a los usuarios pasar información adicional a la función de la tarea para la inicialización, como parámetros que describen un comportamiento más detallado de la lógica de la tarea. Para un uso genérico, esto se logra mediante un puntero. Por lo tanto, cualquier argumento debe pasarse a través de un puntero a un objeto vacío.
- uxPriority - este argumento se puede utilizar para asignar una prioridad a la tarea. El papel de las prioridades se explicará en el Ejercicio.
- pxCreatedTask - este argumento permite al usuario recibir un identificador de la tarea creada. Algunas funciones de FreeRTOS requieren un identificador para realizar operaciones en una tarea específica durante el tiempo de ejecución, por ejemplo, para cambiar la prioridad de una tarea o eliminar una tarea.

```
void vTaskFunction(void *pvParameters)
{
// Init can be here
  while(1)
  {
    // Task functionality
  }
}
```

Codigo 2: Declaracion funcion del task

El cronograma puede ser iniciado con la funcion vTaskStartScheduler(), este se llama dentro de main(). Normalmente la función no retornara nada y nunca terminara. Por este motivo, toda configuración y creación de Tareas deberá ser realizada antes de llamas vTaskStartScheduler().

### Importante

Algunas veces hay que hacer conversiones de tipo en C. Por ejemplo, cuando el tipo de argumento de una función no debe estar predefinido para permitir un uso genérico de la función. En C, una string se realiza con un puntero a una matriz de chars. El siguiente fragmento de código muestra cómo convertimos un puntero char a un puntero vacío y viceversa.

```
1 char* string = "Una frase que hay que mandar";
2 void* pvPointer = (void*)string;
3 char* pcPointer = (char*)pvPointer;
```

Codigo 3: Declaracion funcion del task

Tenga en cuenta que no convertimos nada en el sentido de manipular el contenido de la cadena. Cambiamos el tipo de datos, que solo determina cómo el sistema debe interpretar la matriz de memoria, pero no cambiamos los valores en la memoria. Como resultado, después de que volvimos a cambiar el tipo de datos de (void*) a (char*), podemos trabajar con pcPointer de la misma manera que lo hacemos con string.

## Ejercicio 1: Creacion de proyecto

1. Cree un nuevo proyecto Keil
2. Seleccione su plataforma y todas las dependencias adecuadas. En el apartado RTOS asegurese de seleccionar FreeRTOS y no RTOS-API.
3. Cree un prototipo de una funcion para la creacion de un Task.
4. Cree la implementacion de una funcion como se indica en el Codigo 2.
5. Dentro de la funcion main configure su primer Task utilizando su funcion creada previamente.
6. Una vez configurado el Task inicie el cronograma con la funcion vTaskStartScheduler()
7. Compile el codigo y asegurese de que no tenga ninguna error (Puede ignorar las Warnings)
8. Cargue su codigo a la plataforma. Por ahora, este codigo no tiene ninguna funcionalidad.

## Ejercicio 2: Agregando funcionalidades

1. Cree un nuevo prototipo de funcion
2. Implemente la funcion tal como se indica en el Codigo 2
3. Utilizando esta nueva funcion configure un nuevo Task. Por ahora, para ambos Tasks coloque la prioridad de 1. usStackDepth de 1000. pvParameters - NULL. No se olvide de crear un Handle para cada Task.
4. Antes de la creacion del cronograma configure tres pins (a su eleccion) para prender y apagar tres LEDs.
5. Una vez estos esten configurados como salida a un LED mande un voltaje alto y a los otros dos un voltaje bajo. Asegurese que esto se realice antes de la llamada a vTaskStartScheduler().
6. Compile y cargue su codigo a la plataforma. Los Tasks no hacen nada aun. Asegurese de que su codigo se comporte como espera. Es decir, dos LEDs se encuentra apagado y uno prendido. Asegurese de configurar sus pines adecuadamente para tener este comportamiento.

7. Un Task se encargara de intercalar (Toggle) los valores el LED1, otro Task se encargara de intercalar los valores del LED2. Y el LED3 debera mantenerse prendido y ningun Task puede modificarlo.
8. Compile y cargue su codigo. Que puede observar? El LED1 y el LED2 intercalan valores? Se puede apreciar el blinking de los LEDs?
9. Agregue un delay en cada Task de al menos 250 ms. Compile y cargue su codigo. Que puede observar?

## Ejercicio 3: Prioridades

1. Busque en el documento FreeRTOSConfig.h cual es la prioridad maxima.
2. Cambie la prioridad del Task1 a la prioridad maxima.
3. Compile y caargue su codigo.
4. Que ocurre con el Task1? Que ocurre con el Task2?