

Introducción a Sistemas Operativos de Tiempo Real

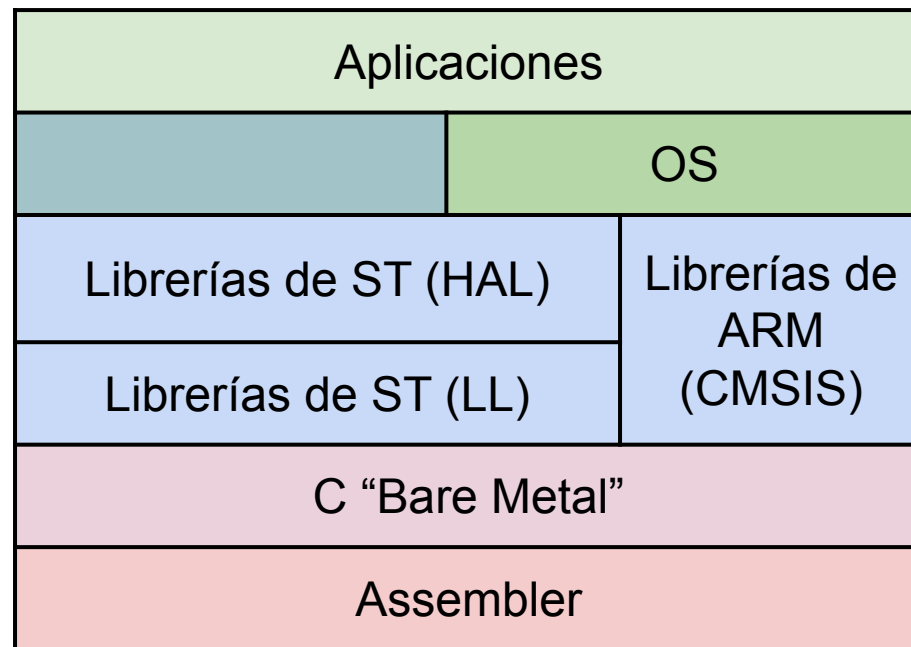
Contenido de la clase

- Generalidades de RTOS Embebidos
- Sistemas Multitarea
- Especificaciones de tiempo real
- Kernel
- Kernel: Scheduler
- Kernel: Dispatcher
- Factor de utilización
- FreeRTOS
- Convivencia entre FreeRTOS “Bare” y HAL
- Características: Tarea IDLE, Prioridades, vTaskDelayUntil
- Debug de tareas: Trace Hooks
- FreeRTOS y CMSIS

RTOS Embebidos

Embedded RTOS

- Un “sistema operativo” nos da una abstracción de más alto nivel sobre la cual podemos programar “aplicaciones”
- Se incrementa la modularidad y mantenibilidad del código, resuelve mucho “andamiaje”
- Pero, habrá código corriendo en el microcontrolador que se ejecutará para administrar las funciones que programamos: insumirá tiempo de procesamiento y recursos
- Trateremos **sistemas operativos de tiempo real embebidos**

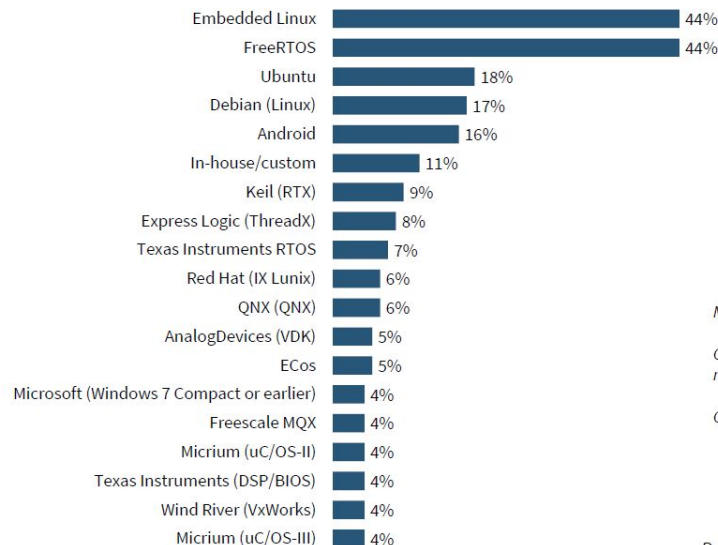


OSs más utilizados

Most popular embedded OSs – Embedded Linux, FreeRTOS and Ubuntu

Top 3 OSs are especially popular in APAC, while Embedded Linux is used more in the Americas

- **Deos** (DDC-I)
- **embOS** (SEGGER)
- **FreeRTOS** (Amazon)
- **Integrity** (Green Hills Software)
- **Keil RTX** (ARM)
- **LynxOS** (Lynx Software Technologies)
- **MQX** (Philips NXP / Freescale)
- **Nucleus** (Mentor Graphics)
- **Neutrino** (BlackBerry)
- **PikeOS** (Sysgo)
- **SafeRTOS** (Wittenstein)
- **ThreadX** (Microsoft Express Logic)
- **μC/OS** (Micrium)
- **VxWorks** (Wind River)
- **Zephyr** (Linux Foundation)



Multiple responses allowed

Only those with 4% or more total mentions shown

Other = 7%

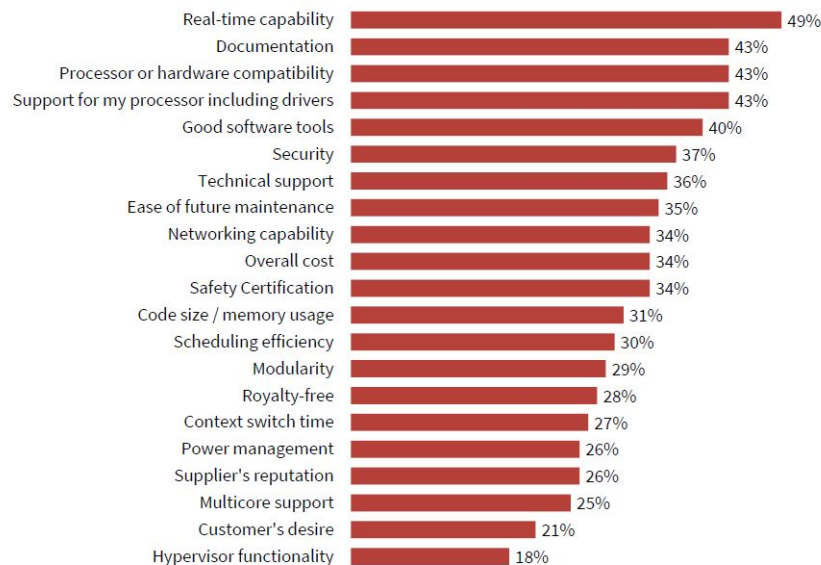


Base = Those who will use an OS (566)

Factores de importancia al elegir un OS

Those using a commercial OS look for documentation, hardware compatibility and support to complement real-time capabilities

Large OEMs and APAC developers put particular emphasis on most commercial OS capabilities



**'Very Important'
Summary**

Multiple responses allowed



Base = Those using commercial OS (200)

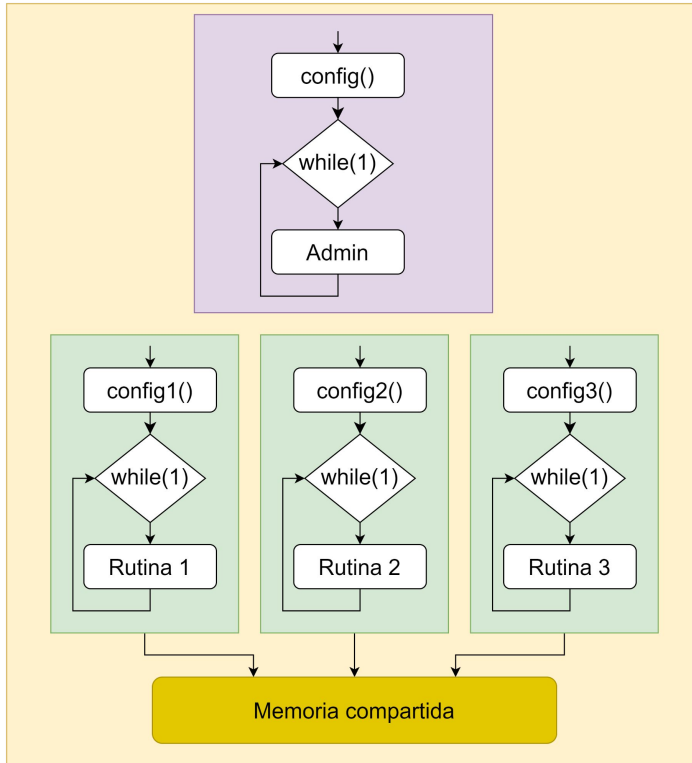
SO Multitarea

Sistema operativo Multitarea

- El paradigma de programación será **concurrente**
 - Programación lineal o secuencial
 - **Programación concurrente: Se resuelve un problema a partir de tareas individuales que se ejecutan simultáneamente**
 - Programación paralela: Un caso de concurrencia donde la ejecución es realmente en simultáneo, con varios núcleos
 - Programación distribuida: La ejecución ocurre en distintos nodos
- Si bien en un procesador con un sólo núcleo se corre un único programa por vez, programaremos subrutinas pensando que corren independientemente en su propio entorno
- Esto es una “ilusión” ya que existe un único procesador
 - Pero reduce el nivel de complejidad de las soluciones
 - Agregando posibles problemas por falta de determinismo y acceso a recursos compartidos



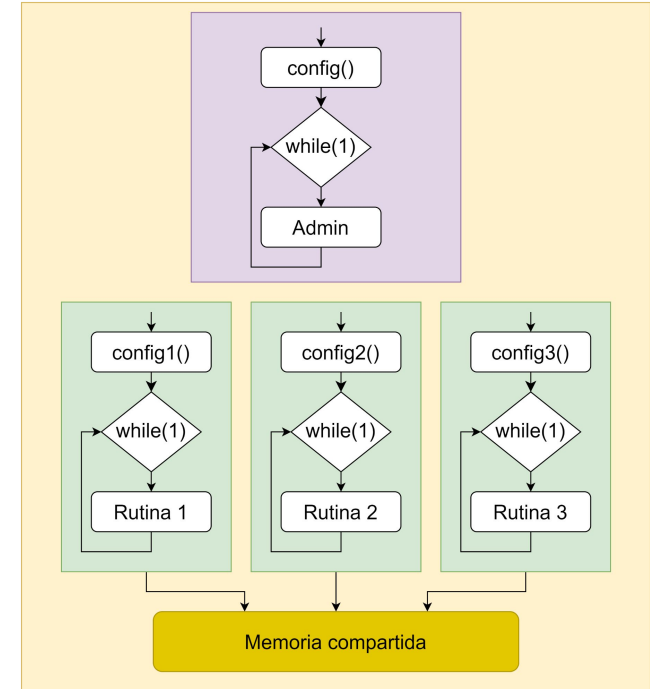
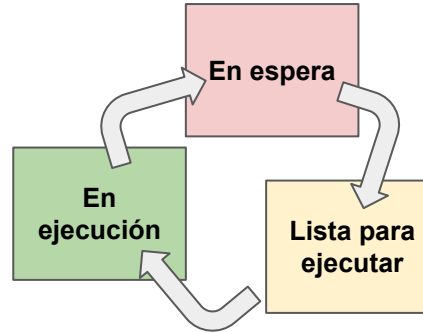
Procesos, Hilos o Tareas



- Las soluciones serán en base a **procesos/hilos/tareas**
- **Una tarea es una función en la que ejecutaremos un algoritmo “independiente”**. Le daremos su propio bucle infinito y pensaremos en ella como si fuese lo único ejecutándose.
- **Ejemplo:** Un adquiredor de datos debe medir una tensión con un convertidor analógico/digital y enviar las muestras a una PC por puerto serie. Además, tiene una interfaz física con botones y leds para configurar, iniciar y detener la adquisición.
 - Tarea 1: Tomar muestras del ADC y guardarlas en un buffer
 - Tarea 2: Enviar datos por UART a la PC, tomando los datos de a bloques de un buffer
 - Tarea 3: Administrar la interfaz: señalizar con distintas luces los estados del adquiredor y capturar presiones de botones para cambiar de estado.

Ciclo de vida de las Tareas/Hilos

- Las particularidades dependen de los detalles de implementación de cada OS
- Pero, a grandes rasgos las tareas están en uno de 3 estados:
 - En ejecución
 - En espera (o bloqueada)
 - Lista para ejecutar
- El trabajo del núcleo o “Kernel” del SO es decidir cuál tarea tiene que correr en cada momento (Scheduling) y ejecutar los cambios de contexto (Dispatcher)



Tiempo Real

Definición de “tiempo real”

- Un OS de tiempo real permite tener un **control determinístico** sobre la temporización de los eventos
- No sólo tiene la restricción de ser lógicamente correcto como un OS de propósito general, sino que además tiene **restricciones de tiempo**.
- Según la consecuencia de violar las restricciones temporales se clasifica:

- Tiempo real duro (“**hard real time**”)
 - Las acciones deben ejecutarse en un tiempo definido (cumplir con un *deadline*) y si esto no ocurre se considera una falla catastrófica (e.g. muestreo)
- Tiempo real firme (“**firm real time**”)
 - Las acciones deben ejecutarse en un tiempo definido y si no ocurre se produce una falla (e.g. transmisión multimedia)
- Tiempo real blando (“**soft real time**”)
 - Las tareas se ejecutarán lo más cerca posible de un tiempo definido, con la consecuencia de la degradación de la performance del sistema en caso de no lograrlo, pero no la falla (e.g. interfaz de usuario)

Restricciones de tiempo real: latencia

- El control determinístico sobre la temporización de los eventos se define en base a dos tipos de especificaciones o restricciones que se debe cumplir:
 - **Evento-respuesta**

Se mide a partir de la **latencia**: **el tiempo que se tarda en cumplir** con una tarea respecto del tiempo definido. Queremos que sea baja, pero en verdad lo más importante es que sea **determinista**.



Si un evento se dispara en tiempos E_i y es atendido en tiempos T_i la latencia es:

$$\Delta T_i = T_{ei} - E_{ai}$$

Si el sistema operativo es de tiempo real, se cumplirá para una cota k que

$$\Delta T_i < k \quad \forall i$$

Restricciones de tiempo real: Jitter

○ Restricciones sobre tiempos preprogramados

Se mide a partir del **jitter**: Se define para tareas periódicas y surge de la necesidad de que los eventos se temporicen con precisión; es decir que **no deben tardar más ni menos de lo estipulado**. Si necesitamos que un evento se ejecute a una tasa $f_s = 1/T_s$ necesitaremos que la tasa T_s se mantenga constante entre los tiempos programados T_{pi} .



El período real entre tiempos será $T_i = T_{pi} - T_{pi-1}$ y podemos definir el jitter para cada período como

$$\delta t_i = T_i - T_s$$

En un RTOS tendremos que $-k < \delta t_i < k \quad \forall i$, y si estamos interesados en el peor caso como sucede con un OS de tiempo real duro, tendremos el Jitter

$$\delta t = \max(\delta t_i) - \min(\delta t_i)$$

Kernel

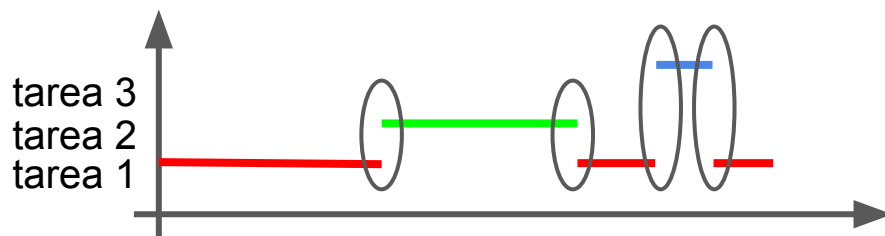
Funciones del OS (componentes del “Kernel”)

A partir del esquema multitarea podemos definir lo que forma parte de un SO:

- **Administración de memoria**
 - Definición y administración de sectores de la ram que se usarán para las tareas
- **Drivers de hardware**
 - Provee una capa de abstracción para manejar periféricos.
- **Scheduler**
 - Es una tarea que se encarga de decidir qué tarea ejecutar a continuación.
- **Dispatcher**
 - La función encargada de comenzar una tarea recuperando los recursos que necesita y resguardando los de la tarea previa.
- **Comunicación**
 - Las tareas deben poder compartir información entre sí para llevar adelante la funcionalidad del SE. Debido al modelo de programación concurrente, surgen muchos problemas por el acceso compartido a los recursos y el OS debe proveer los mecanismos para ordenarlo.

Scheduler

- Define el comportamiento más relevante a la hora de pensar cómo lograr la funcionalidad deseada y la “clasificación” del OS



- Existen distintas **estrategias de scheduling** que son los mecanismos para decidir cuál tarea será la próxima a ejecutarse y cuánto tiempo ocupará.
- Podemos “clasificar” los OS según sus estrategias de scheduling pero en general no son mutuamente excluyentes sino que **se utilizan combinaciones** de estrategias.

Estrategias de scheduling

- **Cooperativo**

- Cada tarea decide cuándo terminar o pausar su ejecución y ceder el control al OS. Las tareas deben incluir código para realizar el cambio de contexto en momentos puntuales y el scheduler es más sencillo.

- **Apropiativo o “Preemptive”**

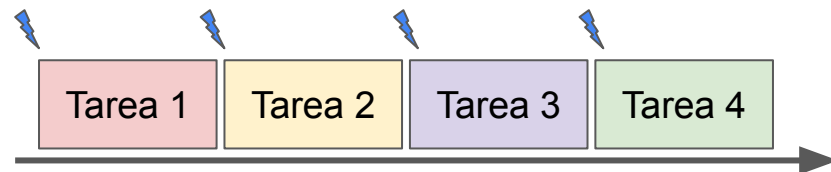
- Los eventos o interrupciones generan el cambio de contexto interrumpiendo a la tarea actual y la tarea ignora completamente que será o ha sido interrumpida.

- **Preemptive basados en prioridades**

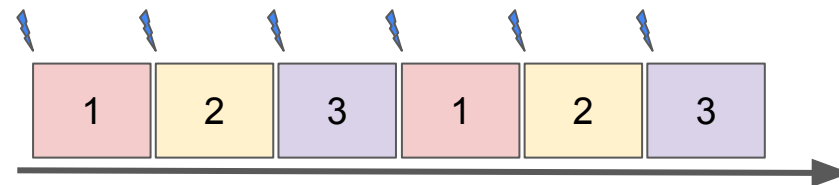
- Cuando surge una necesidad de obtener tiempo a partir de algún evento, la tarea con mayor prioridad interrumpe a las demás y se ejecuta.

Estrategias de scheduling

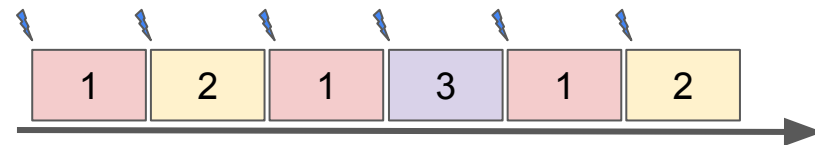
- **Time slice**
 - Existe una interrupción periódica que detiene la tarea en ejecución para ceder la ejecución a otra.
- **Round robin**
 - Se recorre una lista de tareas en orden y se les permite ejecutarse durante un cierto tiempo a cada una
 - También pueden asignarse slices según prioridades.
- **Y muchos otros...**



Scheduling usando Time slices



Scheduling usando Time slices y Round Robin



Scheduling usando Time slices y Round Robin con prioridades

Scheduling y Foreground/Background

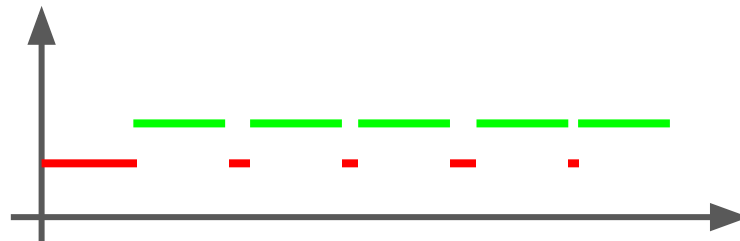
Una combinación de las estrategias de scheduling y el concepto de fg/bg nos lleva a una implementación conveniente:

- Tareas de background
 - Usan un esquema de time slice con alguna estrategia como round robin
 - Se crean al iniciar el OS, pueden ser extensas y se ejecutan en bucles infinitos y definen estados de espera
- Tareas de foreground
 - Usan un esquema apropiativo basados en prioridades
 - Se disparan por eventos, tienen una duración definida y corta.

Este esquema permite responder rápido (con baja latencia) a los eventos que lo requieran, manteniendo la ilusión del paralelismo y la simplicidad en la programación de tareas. Si las tareas apropiativas del foreground no son acotadas y rápidas, las tareas de fondo no se ejecutarán...

- El RTOS depende de que quien programe **lo haga bien!**

tarea 1 (prioridad alta)
tarea 2 (prioridad baja)



Métrica para el scheduling: Factor de utilización

- El factor de utilización es una medida del uso del procesador para cualquier actividad que no sea la “tarea inactiva” (idle)
- Si una tarea necesita un tiempo de ejecución e_i cada período p_i su factor de utilización es $u_i = e_i/p_i$
- Y en un sistema con n tareas periódicas, cada una con factor de utilización u_i el factor de utilización total es $U = \sum u_i$

TABLE 1.3. CPU Utilization (%) Zones

| Utilization (%) | Zone Type | Typical Application |
|-----------------|--------------------|-----------------------------|
| <26 | Unnecessarily safe | Various |
| 26–50 | Very safe | Various |
| 51–68 | Safe | Various |
| 69 | Theoretical limit | Embedded systems |
| 70–82 | Questionable | Embedded systems |
| 83–99 | Dangerous | Embedded systems |
| 100 | Critical | Marginally stressed systems |
| >100 | Overloaded | Stressed systems |

Asignación de prioridades para el scheduling: RMA

- Algoritmo “Rate monotonic”
- Suponemos: un esquema preemptive y prioridades fijas
- **Algoritmo RMA:**
 - **“Asignar mayor prioridad a las tareas con períodos más cortos.”**
- Si es posible encontrar una solución para un conjunto dado de tareas, el algoritmo RM permite implementar una.
 - Dicho de otra manera: si un conjunto de tareas no se puede programar con el RMA no se puede con ningún esquema fijo.

Asignación de prioridades para el scheduling: Cota del RMA

- El RMA permite definir una cota para garantizar que un sistema será programable.
- Cualquier conjunto de n tareas periódicas (bajo condiciones “ideales”) tiene garantía de ser programable si el factor de utilización U no excede el valor $U_{\max} = n(2^{1/n}-1)$ que para $n \rightarrow \infty$ resulta $U_{\max} = 0.69$
- Esto no significa que si se excede la cota no se pueda programar, y existen cotas menos pesimistas

TABLE 1.3. CPU Utilization (%) Zones

| Utilization (%) | Zone Type | Typical Application |
|-----------------|--------------------|-----------------------------|
| <26 | Unnecessarily safe | Various |
| 26–50 | Very safe | Various |
| 51–68 | Safe | Various |
| 69 | Theoretical limit | Embedded systems |
| 70–82 | Questionable | Embedded systems |
| 83–99 | Dangerous | Embedded systems |
| 100 | Critical | Marginally stressed systems |
| >100 | Overloaded | Stressed systems |

Asignación de prioridades para el scheduling

- Algoritmo “Earlier Dealine First”
 - Se trata de un algoritmo que asigna prioridades en forma dinámica.
 - En cada momento, a la tarea con el deadline más cercano se le asigna la máxima prioridad.
 - Si tomamos el deadline de cada tarea como su período también puede encontrarse una cota:

$$U = \sum_{i=1}^n \frac{e_i}{p_i} < 1$$

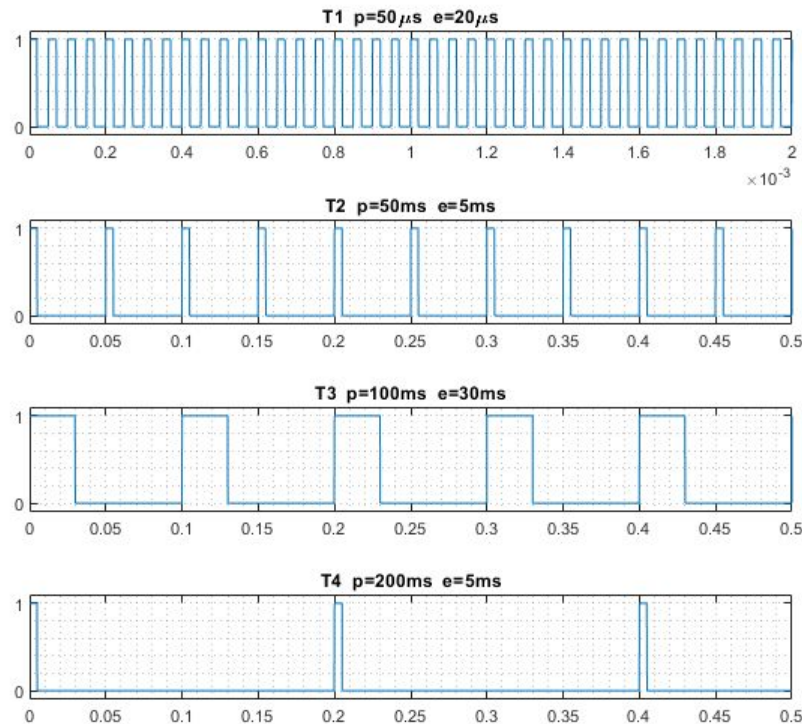
Un ejemplo

Un sistema debe:

- Muestrear una señal a una tasa de 20 kHz (el muestreo demora 20 μ s)
- Procesarla en paquetes de 1000 muestras para determinar el nivel de un componente de la señal (el procesamiento demora 5 ms)
- Refrescar una pantalla (demora 30ms en reescribir la pantalla) con la información obtenida a una tasa de 10 veces por segundo
- Responder al pedido de datos de otro sistema con una latencia menor a 10 ms (en el peor caso se encuesta cada 200 ms y la transmisión toma 1 ms).

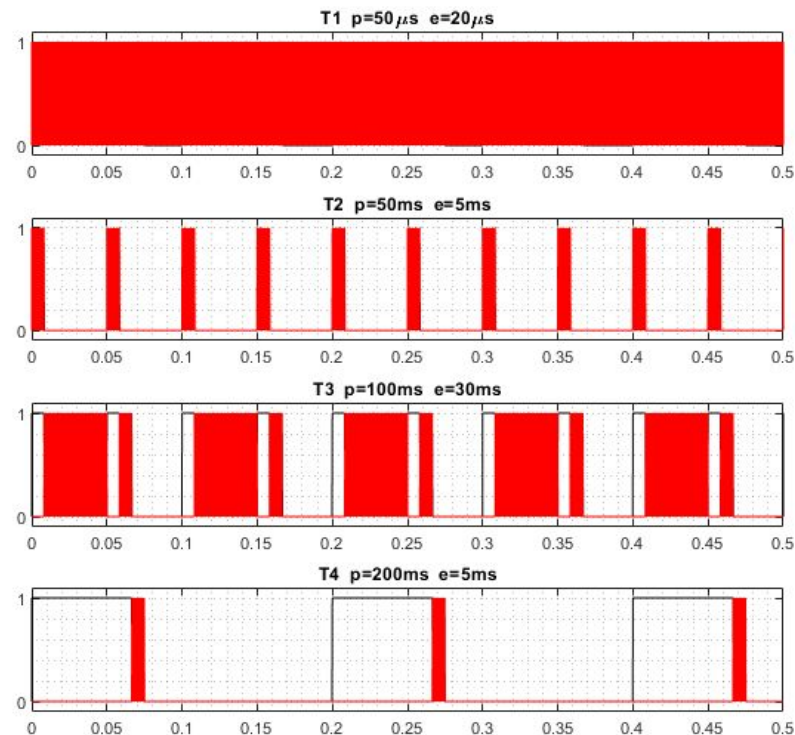
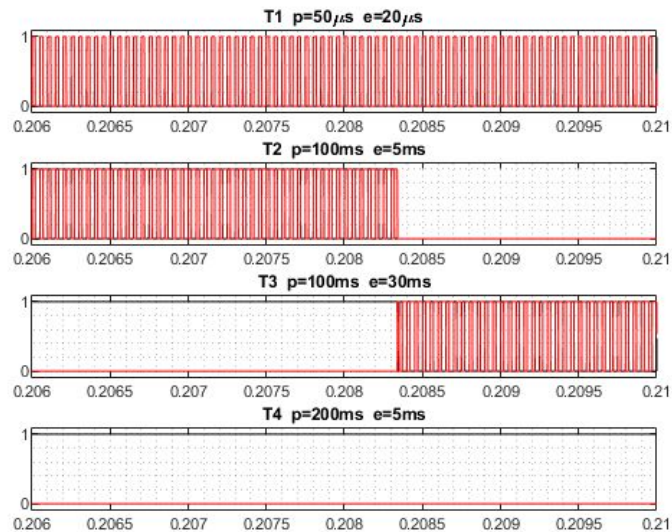
Ejemplo - Visualización de tareas

- En la imagen se representan las tareas individualmente
- ¿Cuál es el factor de utilización total?
- ¿En qué rango se encuentra según la tabla de Lapante?
- ¿Cuál es la cota real para este número de tareas?
- ¿RMA garantiza la temporización para este ejemplo?



Ejemplo - Priorización según RMA

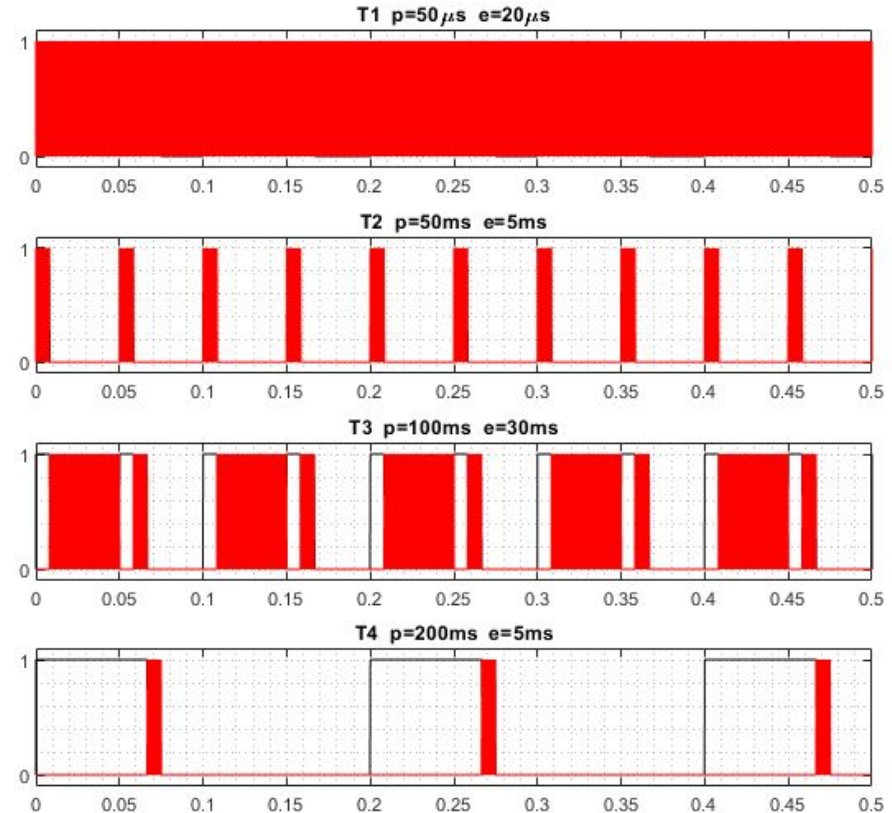
- La figura muestra un ejemplo de ejecución programando la prioridad según RMA
- Ninguna tarea excede su período



Ejemplo - Priorización según RMA

- ¿Se cumplen todas las restricciones de tiempo real?

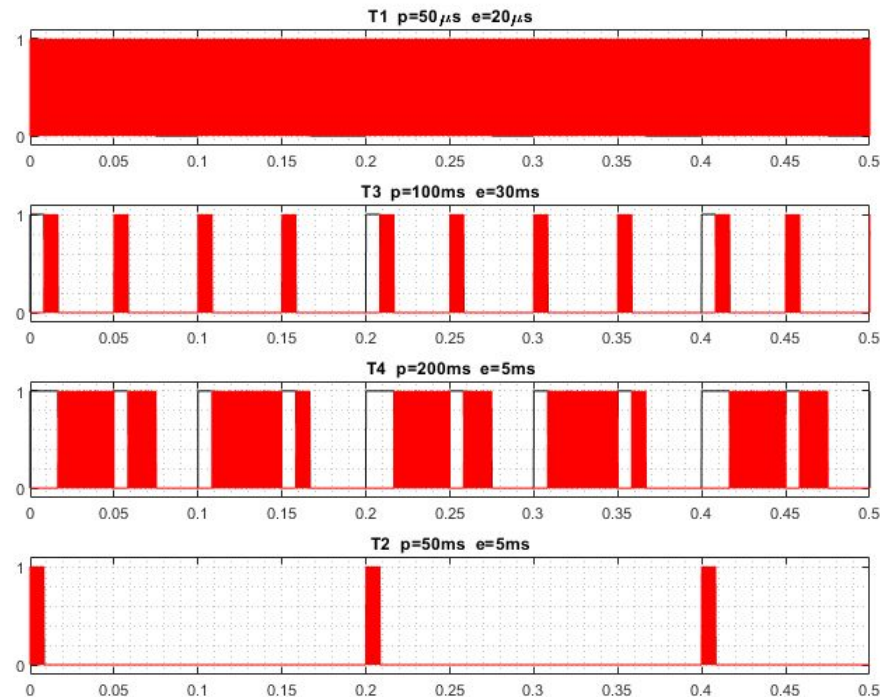
- T1: Muestreo 20 kHz
- T2: Procesamiento de 5 ms c/ 50 ms
- T3: Pantalla 30 ms c/ 100 ms
- T4: Responder datos latencia < 10 ms c/200 ms



Ejemplo - Cambio de prioridades

- Se cambiaron las prioridades: ahora T4 tienen mayor prioridad que T2 y T3

- T1: Muestreo 20 kHz
- T2: Procesamiento de 5 ms c/ 50 ms
- T3: Pantalla 30 ms c/ 100 ms
- T4: Responder datos latencia < 10 ms c/ 200 ms



Dispatcher

- El dispatcher se encarga de cambiar de una tarea a la otra.
- Recordemos que las tareas son funciones de C en un bucle infinito

```
void TareaA() {  
    initA();  
    while(1) {  
        ...  
    }  
}
```

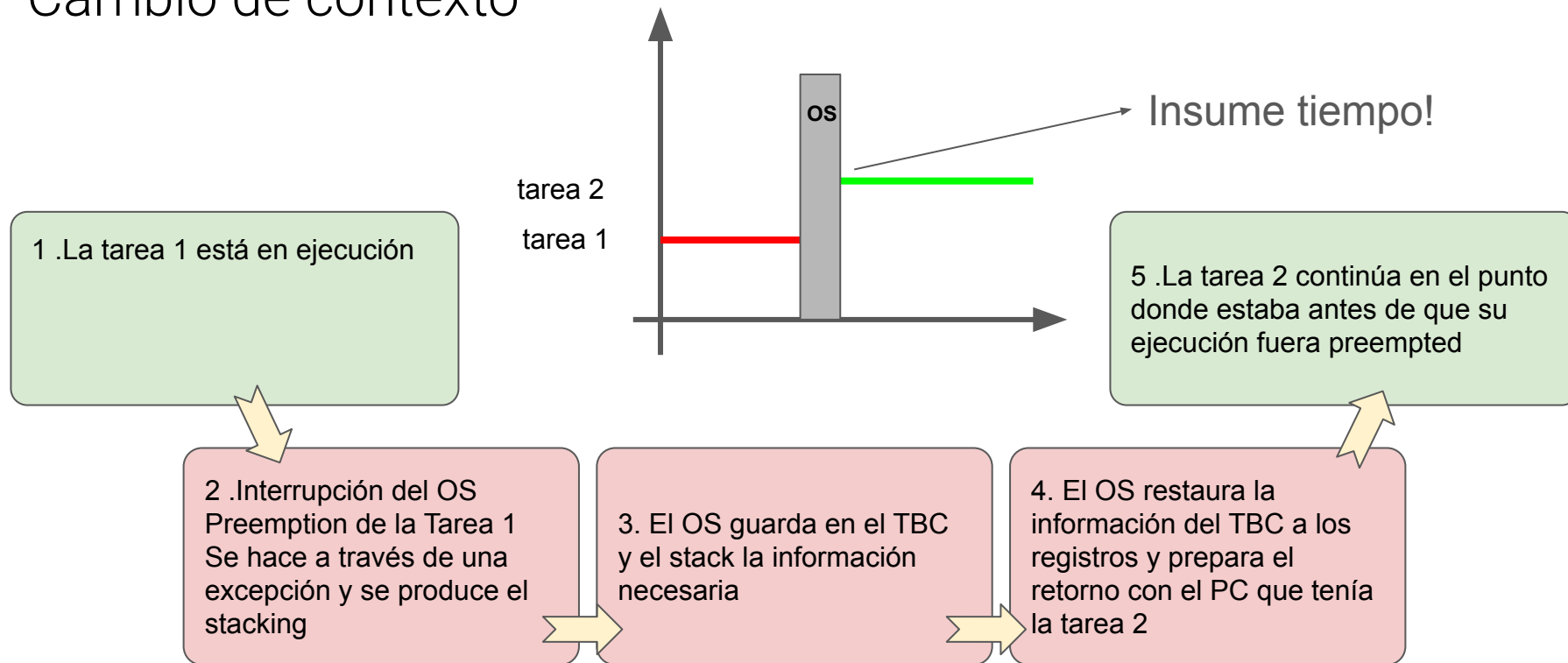
```
void TareaB() {  
    initB();  
    while(1) {  
        ...  
    }  
}
```

- ¿Qué sucede cuando se debe “cambiar de una tarea a otra”?
- Conocemos normalmente el pase de una función a otra
 - Pero al hacer esto, se “anidan” funciones, no podemos anidar infinitamente las tareas
 - Y las tareas no “retornan”
- Entonces, ¿cómo se logra? -> Cambio de contexto

Cambio de contexto - TBC

- Para cambiar de contexto el OS debe
 - Guardar el estado de ejecución de la tarea que abandona y cargar el estado de la tarea a la que ingresa
 - Registros
 - Puntero al stack
 - Señalar el PC al punto de ingreso.
- Para ello necesita retener la información en algún lado
- Utiliza una estructura de información para cada tarea: el **Thread (o task) control block**
- Deberá contener: Registros que no se guardan automáticamente durante el stacking, puntero al stack, estado del PC (si esto no se guarda en el stack)

Cambio de contexto



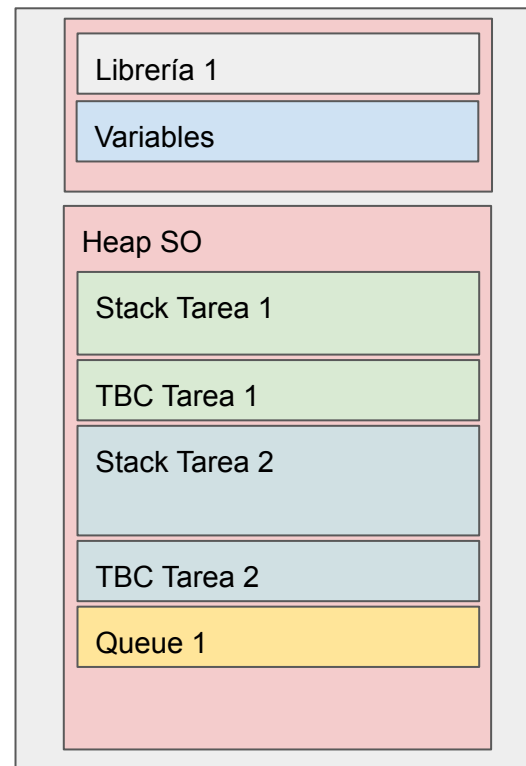
Administración de memoria

- El OS administra
 - Stacks para cada tarea (incluyendo la funcionalidad del sistema)
 - Las TCBs con distintas estrategias (puede ser junto con cada stack)
 - Elementos de comunicación de datos y señales y sincronización (queues, mutex, etc)
- Los elementos administrados por el OS conviven con los usados por otras librerías
- La asignación de recursos puede ser:
 - Estática
 - Se declaran los arreglos que constituirán la memoria para los recursos en tiempo de compilación
 - Dinámica
 - La creación de tareas involucra la reserva de memoria dinámica para su stack. En estos casos, el OS contará con un “Heap” que irá creciendo (o se irá llenando) a medida que reserve memoria
 - Puede haber esquemas que reserven y liberen memoria dinámicamente lo cual puede producir fragmentación y conllevar tiempos no deterministas.

Administración de memoria - Distribución de sectores

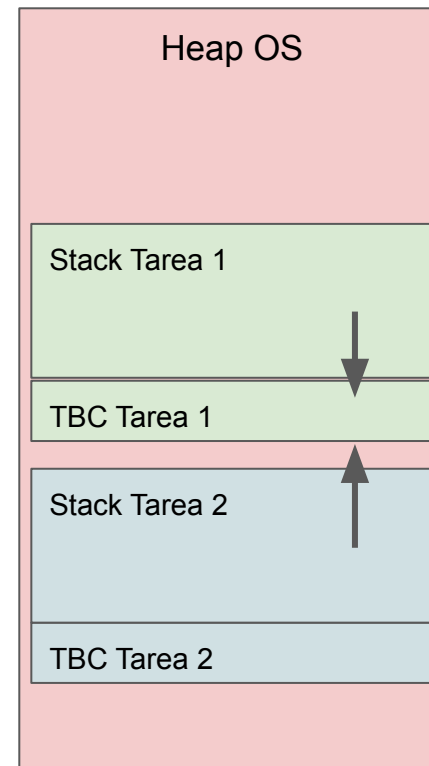
- Al configurar el OS debemos asignar tamaños a los sectores de la memoria RAM que se reservan para cada elemento
- Asignamos tamaño al Heap procurando que el OS tenga suficiente espacio para reservar sus elementos, y que quede suficiente espacio “por fuera”
- Asignamos tamaño a las tareas para que funcionen sin desbordar

Memoria
RAM



Administración de memoria - Stack overflow

- Al configurar el OS debemos asignar un tamaño al stack de cada tarea
- Ya sea porque dimensionamos mal el stack o porque cometemos un error en el código, puede pasar que nos extendamos más allá del área reservada para el stack
- Depende de la estrategia de administración de memoria elegida por el OS, al extendernos por fuera del stack podemos sobrescribir el TBC de la tarea, el TBC de otra tarea, o cualquier otra información importante
- Eso se conoce como “stack overflow”





FreeRTOS en el Cortex M3

- Características “de Folleto”:
 - Puede ser apropiativo o cooperativo
 - Tiene tareas con prioridades
 - Herramientas de concurrencia: Queues, Binary semaphores, Counting semaphores, Recursive semaphores, Mutexes
 - Permite implementar funcionalidad propia fácilmente para “enganchar” en eventos del SO:
 - **Tick hook, Idle hook, Task tag**
 - Y depurar (Chequeo de Stack overflow, Trace hooks)
 - Permite que interrupciones por encima de una prioridad determinada queden por encima del kernel asegurando la latencia y jitter para esos eventos.
- Soporte
 - Es de código abierto (puede leerse el código para depurar y aprender)
 - Tiene muy buena documentación: https://www.freertos.org/Documentation/RTOS_book.html
 - Un libro didáctico escrito por su creador Richard Barry “[Mastering the FreeRTOS Real Time Kernel - a Hands On Tutorial Guide](#)” . Existen versiones adaptadas al Cortex M3 pero son de una versión más vieja del OS
 - Un manual de referencia muy bien explicado y con ejemplos: [FreeRTOS V10.0.0 Reference Manual](#)
 - En la propia descarga del OS hay demos
 - Muchos libros de SE explican el FreeRTOS

FreeRTOS - Recursos utilizados en el Cortex M3 sin MPU

- Interrupciones

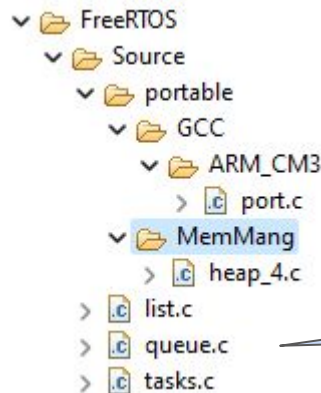
- SysTick: Utiliza el tick de sistema (periférico del Cortex) para su temporización
- PendSV: Utilizada para implementar los cambios de contexto
- SVC: Permite que el código de tareas realice llamadas al sistema

- Memoria

- “Approximately 6K bytes of Flash space and a few hundred bytes of RAM”.

Agregando FreeRTOS “bare” - Archivos de código

- Descargamos el FreeRTOS de la página
- Se descarga una carpeta con muchos archivos porque está portado a decenas de dispositivos distintos y tiene funcionalidades opcionales. Debemos seleccionar el subconjunto que nos sirva.
- El código “núcleo” es sorprendentemente compacto:



Código para adaptar el OS al hardware específico

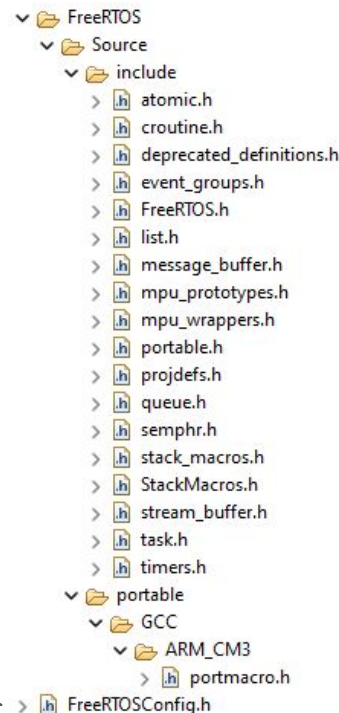
Algoritmo para asignar memoria

Kernel

Agregando FreeRTOS “bare” - Archivos de cabecera

- De especial importancia es el archivo **FreeRTOSConfig.h** que agregamos “por fuera” de la estructura de FreeRTOS
- Contiene definiciones que configuran las características y opciones que usaremos en el OS

Este es “nuestro” archivo de configuración. Aquí definimos todo lo que usamos del FreeRTOS y la configuración del sistema



Agregando FreeRTOS “bare” - Archivo de configuración

- Sistema operativo cooperativo o basado en preemption:
 - `#define configUSE_PREEMPTION 1`
- Los hooks permiten intercalar funciones propias cuando ocurren eventos particulares en el OS (pero si lo habilitamos y las funciones no están declaradas es un error)
 - `#define configUSE_IDLE_HOOK 0`
 - `#define configUSE_TICK_HOOK 0`
- Frecuencia del CPU
 - `#define configCPU_CLOCK_HZ ((unsigned long) 72000000)`
- Frecuencia del Tick del OS. Ejemplo 1 kHz (período 1 ms)
 - `#define configTICK_RATE_HZ ((TickType_t) 1000)`

```

/* FreeRTOSConfig.h */

/* Application specific definitions. */
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES ( 5 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE ( (size_t) 192 )
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configIDLE_SHOULD_YIELD 1
#define configUSE_MUTEXES 1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set to 1 to include the API function */
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1

/* Raw value as per the Cortex-M3 NVIC. */
#define configKERNEL_INTERRUPT_PRIORITY 255
/* Value 191 is equivalent to priority 11. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY 191
/* This is the value being used as per the ST library
which permits 16 priority values, 0 to 15. */
#define configLIBRARY_KERNEL_INTERRUPT_PRIORITY 15

```

Agregando FreeRTOS “bare” - Archivo de configuración

- Las tareas pueden tener prioridades comenzando en la 0 que es la más baja hasta (configMAX_PRIORITIES – 1) que es la más alta. Si luego asignamos una prioridad mayor se truncará a este valor.
 - `#define configMAX_PRIORITIES`
`(5)`
- El tamaño de stack mínimo para asignar a las tareas (por ejemplo será el que tenga la IDLE task). Se especifica en PALABRAS (no bytes)
 - `#define configMINIMAL_STACK_SIZE`
`((unsigned short) 128)`
- El tamaño del Heap, de donde se tomarán porciones para implementar los stacks
 - `#define configTOTAL_HEAP_SIZE`
`((size_t)8192)`

```

/* FreeRTOSConfig.h */

/* Application specific definitions. */
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES ( 5 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE ((size_t)8192)
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configIDLE_SHOULD_YIELD 1
#define configUSE_MUTEXES 1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set to 1 to include the API function */
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1

/* Raw value as per the Cortex-M3 NVIC. */
#define configKERNEL_INTERRUPT_PRIORITY 255
/* Value 191 is equivalent to priority 11. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY 191
/* This is the value being used as per the ST library
which permits 16 priority values, 0 to 15. */
#define configLIBRARY_KERNEL_INTERRUPT_PRIORITY 15

```

Agregando FreeRTOS “bare” - Archivo de configuración

- `configMAX_TASK_NAME_LEN`
Las tareas pueden tener nombres (cadenas de caracteres) para fines de debug y obtención del handler:
- `configUSE_TRACE_FACILITY`
Las herramientas de TRACE existen con fines de debug
- `configUSE_16_BIT_TICKS`
FreeRTOS permite usar una variable de contador de tick de 16 bits para hacer el código más eficiente en sistemas de 8 o 16 bits, pero no tiene utilidad en sistemas de 32 bits
- `configIDLE_SHOULD_YIELD`
Si se configura para “Yield” cederá la ejecución tras 1 iteración y el resto del time slice se adjudicará a la primera tarea que le siga en el RR
- `configUSE_MUTEXES`
Hay varias opciones distintas para lograr sincronizar el acceso a recursos compartidos; cada una requiere incluir el código correspondiente y en este caso los Mutex se pueden incluir o no opcionalmente.
- `CO_ROUTINES`
Las Co-rutinas son útiles en implementaciones cooperativas

```

/* FreeRTOSConfig.h */

/* Application specific definitions. */
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES ( 5 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE ((size_t)192)
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configIDLE_SHOULD_YIELD 1
#define configUSE_MUTEXES 1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

/* Set to 1 to include the API function */
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1

/* Raw value as per the Cortex-M3 NVIC. */
#define configKERNEL_INTERRUPT_PRIORITY 255
/* Value 191 is equivalent to priority 11. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY 191
/* This is the value being used as per the ST library
which permits 16 priority values, 0 to 15. */
#define configLIBRARY_KERNEL_INTERRUPT_PRIORITY 5

```

Agregando FreeRTOS “bare” - Archivo de configuración

- Pueden incluirse o no funcionalidades opcionales que se habilitan a través de los `#defines` de esta sección del archivo de configuración
- Finalmente, se configuran los valores de interrupciones en compatibilidad con el cortex M3
- Además, deberá incluirse un poco más de código para asegurar la compatibilidad con la HAL

```
/* FreeRTOSConfig.h */

/* Application specific definitions. */
#define configUSE_PREEMPTION 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configCPU_CLOCK_HZ ( ( unsigned long ) 72000000 )
#define configTICK_RATE_HZ ( ( TickType_t ) 1000 )
#define configMAX_PRIORITIES ( 5 )
#define configMINIMAL_STACK_SIZE ( ( unsigned short ) 128 )
#define configTOTAL_HEAP_SIZE ((size_t)192)
#define configMAX_TASK_NAME_LEN ( 16 )
#define configUSE_TRACE_FACILITY 0
#define configUSE_16_BIT_TICKS 0
#define configIDLE_SHOULD_YIELD 1
#define configUSE_MUTEXES 1

/* Co-routine definitions. */
#define configUSE_CO_ROUTINES 0
#define configMAX_CO_ROUTINE_PRIORITIES ( 2 )

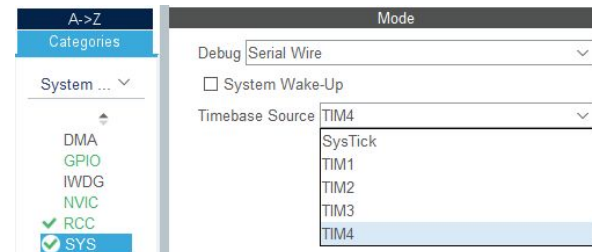
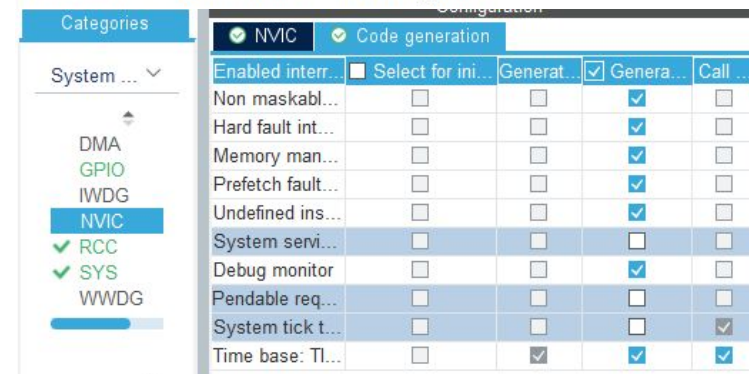
/* Set to 1 to include the API function */
#define INCLUDE_vTaskPrioritySet 1
#define INCLUDE_uxTaskPriorityGet 1
#define INCLUDE_vTaskDelete 1
#define INCLUDE_vTaskCleanUpResources 0
#define INCLUDE_vTaskSuspend 1
#define INCLUDE_vTaskDelayUntil 1
#define INCLUDE_vTaskDelay 1

/* Raw value as per the Cortex-M3 NVIC. */
#define configKERNEL_INTERRUPT_PRIORITY 255
/* Value 191 is equivalent to priority 11. */
#define configMAX_SYSCALL_INTERRUPT_PRIORITY 191
/* This is the value being used as per the ST library
which permits 16 priority values, 0 to 15. */
#define configLIBRARY_KERNEL_INTERRUPT_PRIORITY 5
```

Agregando FreeRTOS “bare” - Convivencia con la HAL

- FreeRTOS usa la interrupción de SysTick, PendSV y SVC y define los handlers
 - Por lo tanto debemos:
 - Decirle al FreeRTOS el nombre de los vectores de interrupción en FreeRTOSConfig.h
 - Deshabilitar la generación de código automática de los manejadores
- La HAL se basa en el SysTick para implementar funcionalidad como HAL_Delay y otras que dependen de una referencia de tiempo
 - Por lo tanto debemos asignarle una base de tiempo distinta para liberar el SysTick par el FreeRTOS

```
/* Definitions that map the FreeRTOS port interrupt handlers to their CMSIS
standard names. */
#define vPortSVCHandler SVC_Handler
#define xPortPendSVHandler PendSV_Handler
#define xPortSysTickHandler SysTick_Handler
```



Agregando FreeRTOS “bare” - Librerías

- En el código generado por el CubeMX podemos agregar la referencia a las librerías que utilizaremos
- Para “poner en marcha” el OS llamamos a la función `vTaskStartScheduler` que inicia la operación del sistema operativo.
- Si no hemos instalado ninguna tarea, pondrá a correr la tarea “Idle”

| Size | Free | Used | Usage (%) |
|-------|----------|---------|-----------|
| 20 KB | 18,25 KB | 1,75 KB | 8.75% |
| 64 KB | 57,19 KB | 6,81 KB | 10.64% |

| | | | |
|-------|----------|---------|--------|
| 20 KB | 10,09 KB | 9,91 KB | 49.53% |
| 64 KB | 54,23 KB | 9,77 KB | 15.26% |

```

/* USER CODE END Header */
/* Includes -----*/
#include "main.h"
/* Private includes -----*/
/* USER CODE BEGIN Includes */
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
/* USER CODE END Includes */

```

...

```

/* USER CODE BEGIN 2 */
vTaskStartScheduler();
/* USER CODE END 2 */

```

Corriendo la tarea IDLE



| FreeRTOS Task List | | | | | | | |
|--------------------|--------------------|----------------|------------------|---------|--------------|----------------|--|
| Name | Priority (Base/... | Start of Stack | Top of Stack | State | Event Obj... | Min Free St... | |
| → IDLE | 0/0 | 0x20000140 | 0x200002f8 <u... | RUNNING | | N/A | |

```

/* USER CODE END Header */
/* Includes
-----*/
#include "main.h"
/* Private includes
-----*/
/* USER CODE BEGIN Includes */
#include "FreeRTOSConfig.h"
#include "FreeRTOS.h"
#include "task.h"
/* USER CODE END Includes */

...

/* USER CODE BEGIN 2 */
vTaskStartScheduler();
/* USER CODE END 2 */

```

Agregando una tarea - Definición

Se crea una función que será la tarea (para eso se declara y define, como toda función)

```
/* USER CODE BEGIN PFP */
static void tareaParpadeo(void *pvParameters);
/* USER CODE END PFP */

...

/* USER CODE BEGIN 4 */
static void tareaParpadeo(void *pvParameters){
    for (;;) {
        vTaskDelay( pdMS_TO_TICKS(1000) );
        HAL_GPIO_TogglePin(GPIOC, GPIO_PIN_13);
    }
}
/* USER CODE END 4 */
```

```
void vAnotherTask( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some processing here. */

        ...

        /* Enter the Blocked state for 20 tick interrupts - the
        in the Blocked state is dependent on the tick frequency.
        vTaskDelay( 20 );

        /* 20 ticks will have passed since the first call to vTaskDelay()
        executed. */

        /* Enter the Blocked state for 20 milliseconds. Using the
        pdMS_TO_TICKS() macro means the tick frequency can change without
        affecting the time spent in the blocked state (other than the
        resolution of the tick frequency). */
        vTaskDelay( pdMS_TO_TICKS( 20 ) );
    }
}
```

Listing 21 Example use of vTaskDelay()

Agregando una tarea - xTaskCreate

2.6 xTaskCreate()

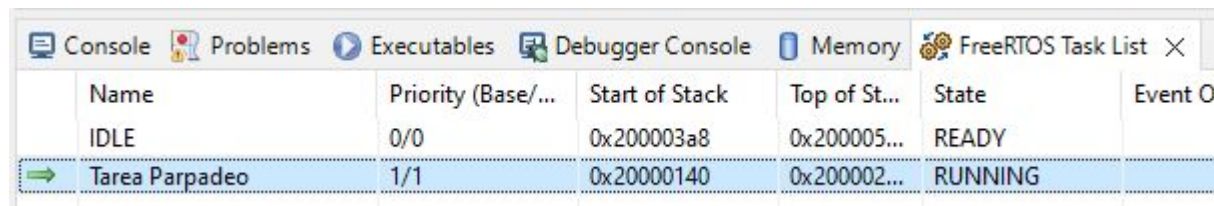
```
/* USER CODE BEGIN 2 */  
/* Instalar la tarea */  
xTaskCreate(    tareaParpadeo,  
               "Tarea Parpadeo",  
               128,  
               NULL,  
               tskIDLE_PRIORITY + 1,  
               NULL);
```

```
/* Start the scheduler. */  
vTaskStartScheduler();
```

```
/* USER CODE END 2 */
```

```
#include "FreeRTOS.h"  
#include "task.h"  
  
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,  
                        const char * const pcName,  
                        unsigned short usStackDepth,  
                        void *pvParameters,  
                        UBaseType_t uxPriority,  
                        TaskHandle_t *pxCreatedTask );
```

Listing 12 xTaskCreate() function prototype



| | Name | Priority (Base/... | Start of Stack | Top of St... | State | Event O |
|---|----------------|--------------------|----------------|--------------|---------|---------|
| | IDLE | 0/0 | 0x200003a8 | 0x200005... | READY | |
| ➡ | Tarea Parpadeo | 1/1 | 0x20000140 | 0x200002... | RUNNING | |

Tick

```

/* FreeRTOSConfig.h */
#define configUSE_TICK_HOOK          1

/* USER CODE BEGIN 4 */
...
void vApplicationTickHook( void ){

    HAL_GPIO_TogglePin (GPIOA,GPIO_PIN_5);
}
/* USER CODE END 4 */

```

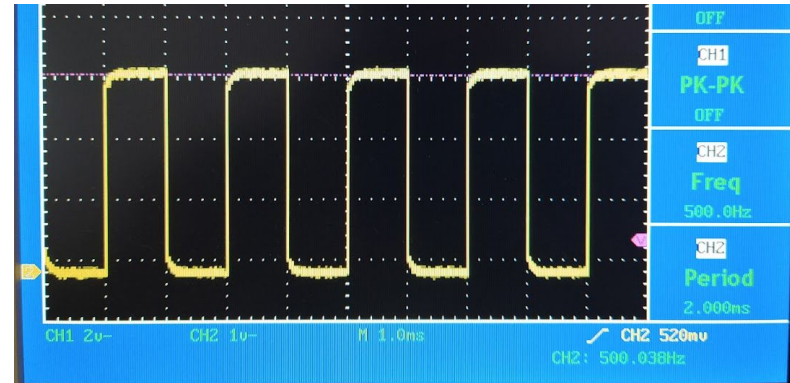
configUSE_TICK_HOOK

The tick hook function is a hook (or callback) function that, if defined and configured, will be called during each tick interrupt.

If configUSE_TICK_HOOK is set to 1 then the application must define a tick hook function. If configUSE_TICK_HOOK is set to 0 then the tick hook function will not be called, even if one is defined.

Tick hook functions must have the name and prototype shown in Listing 244.

```
void vApplicationTickHook( void );
```



Prioridad

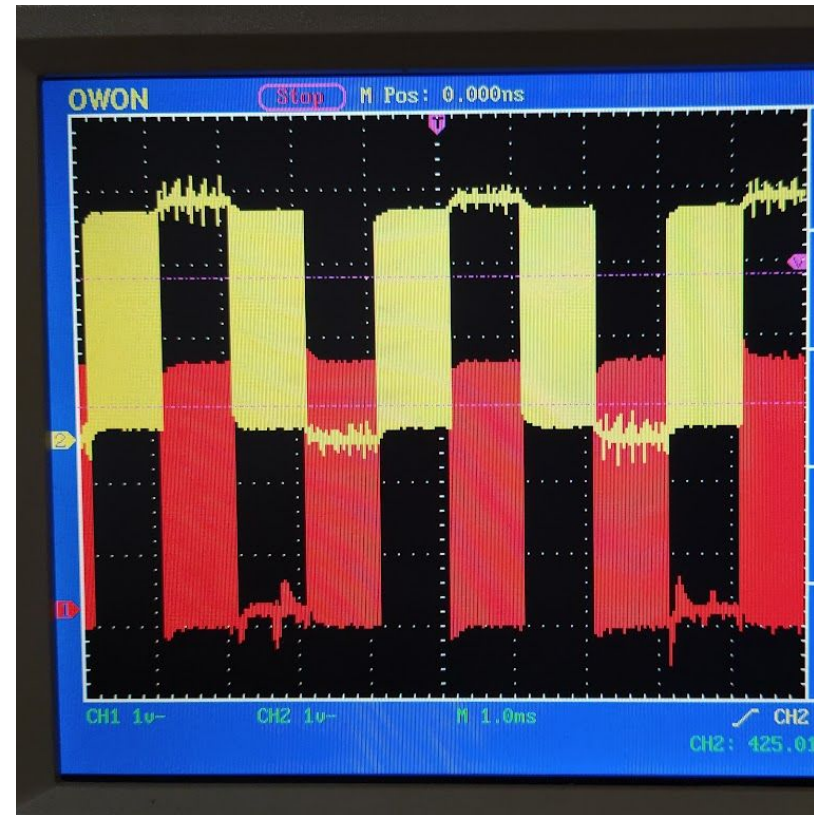
```
/* Instalar la tarea A */
xTaskCreate(tareaA, "Tarea A", 128, NULL,
            tskIDLE_PRIORITY+1, NULL);

/* Instalar la tarea B */
xTaskCreate(tareaB, "Tarea B", 128, NULL,
            tskIDLE_PRIORITY+1, NULL);

/* Start the scheduler. */
vTaskStartScheduler();

static void tareaA(void *pvParameters) {
    for (;;)
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_5);
}

static void tareaB(void *pvParameters) {
    for (;;)
        HAL_GPIO_TogglePin(GPIOA, GPIO_PIN_6);
}
```



vTaskDelay/Until

```
/* Instalar la tarea A */  
xTaskCreate(tareaA, "Tarea A", 128, NULL,  
tskIDLE_PRIORITY+2, NULL);
```

```
/* Instalar la tarea B */  
xTaskCreate(tareaB, "Tarea B", 128, NULL,  
tskIDLE_PRIORITY+1, NULL);
```

```
void mi_busy_delay(int ms){  
    int i,j;  
    int cyc_1ms = 5494;  
    for(i=0; i< ms; i++){  
        for(j=0; j<cyc_1ms; j++){  
            ;  
        }  
    }  
}
```

```
static void tareaA(void *pvParameters) {  
    int e = 40;  
    int p = 100;  
    TickType_t xLastWakeTime;  
    xLastWakeTime = xTaskGetTickCount();  
    for (;;) {  
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_SET);  
        mi_busy_delay(e);  
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_5, GPIO_PIN_RESET);  
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( p ) );  
    }  
}  
  
static void tareaB(void *pvParameters) {  
    int e = 20;  
    int p = 150;  
    ...  
    for (;;) {  
        HAL_GPIO_WritePin(GPIOA, GPIO_PIN_6, GPIO_PIN_SET);  
        ...  
    }  
}
```

vTaskDelay/Until

```
/* Instalar la tarea A */  
xTaskCreate(tareaA, "Tarea A", 128, NULL,  
tskIDLE_PRIORITY+2, NULL);
```

```
/* Instalar la tarea B */  
xTaskCreate(tareaB, "Tarea B", 128, NULL,  
tskIDLE_PRIORITY+1, NULL);
```

```
static void tareaA(void *pvParameters) {  
    int e = 40;  
    int p = 100;  
    ...  
}  
  
static void tareaB(void *pvParameters) {  
    int e = 20;  
    int p = 150;  
    ...  
}
```



Trace hooks

Trace hook macros (Callbacks)

- FreeRTOS provee de numerosos callbacks predefinidos como macros vacíos, los **trace hook macros**, que se ejecutan ante diversos eventos relacionados con el funcionamiento interno del RTOS y de la ejecución de las tareas.
- Si se redefinen, pueden invocar código de usuario y permitir un seguimiento minucioso de la actividad interna.
- El lugar recomendado para redefinir estos macros es al final de FreeRTOSConfig.h

Trace hooks

- Los trace hooks más útiles para realizar el seguimiento de la ejecución de las tareas son:
 - `traceTASK_SWITCHED_IN()`
 - `traceTASK_SWITCHED_OUT()`
- Hay decenas de trace hook macros adicionales, con los que se puede ejecutar código de usuario cada vez que se ejecutan la mayor parte de las funciones de FreeRTOS.
- Todos deben habilitarse, y luego redefinir el macro
- Otros Hooks útiles:
 - `void vApplicationIdleHook(void);`
 - `void vApplicationTickHook(void);`
 - `void vApplicationMallocFailedHook(void);`

Trace hooks para Tareas

- Los macros

`traceTASK_SWITCHED_IN()`

`traceTASK_SWITCHED_OUT()`

Se ejecutan cuando se pone o saca de ejecución una tarea si se habilita la opción `USE_APPLICATION_TASK_TAG`

- Es interesante usarlos asignando un GPIO a cada tarea y subirlo cuando está en ejecución y bajarlo cuando no. Pero el macro es uno sólo, cómo distinguimos cuál pin subir o bajar?
- Puede haber varias estrategias, pero la recomendada en el manual es utilizar un campo que se guarda en el TCB, el `pxTaskTag`. Es de tipo `*void` porque está pensado para pasar un puntero a “cualquier cosa” (una estructura, un puntero a función, etc)
- En este caso lo usamos para guardar un entero arbitrario que identifique a cada tarea.

Task tag

- Por default el tag de cada tarea es nulo, hay que invocar a la función **vTaskSetApplicationTaskTag** y asignar un valor distinto a cada una.

```
/* In this example an integer is set as the task tag value.
See the RTOS trace hook macros documentation page for an
example how such an assignment can be used. */
void vATask( void *pvParameters )
{
    /* Assign a tag value of 1 to myself. */
    vTaskSetApplicationTaskTag( NULL, ( void * ) 1 );

    for( ;; )
    {
        /* Rest of task code goes here. */
    }
}
```

Implementación de callbacks

- En FreeRTOSConfig.h se redefinen los macros para que invoquen funciones, pasando como parámetro el tag de la tarea actual: **xCurrentTCB->pxTaskTag**

```
/* USER CODE BEGIN Defines */
/* Section where parameter definitions can be added (for instance, to override
default ones in FreeRTOS.h) */

void callback_in(int);
void callback_out(int);
#define traceTASK_SWITCHED_IN() callback_in((int)pxCurrentTCB->pxTaskTag)
#define traceTASK_SWITCHED_OUT() callback_out((int)pxCurrentTCB->pxTaskTag)
/* USER CODE END Defines */
```

Debug con TaskHooks

```
/* FreeRTOSConfig.h*/  
void callback_in(int);  
void callback_out(int);  
#define traceTASK_SWITCHED_IN() callback_in((int)pxCurrentTCB->pxTaskTag)  
#define traceTASK_SWITCHED_OUT() callback_out((int)pxCurrentTCB->pxTaskTag)  
#define TAG_TASK_IDLE 0  
#define TAG_TASK_NORMAL 1  
#define TAG_TASK_ALTA 2
```

```
void callback_in(int tag) { /* Definición en main.c*/  
    switch (tag) {  
        case TAG_TASK_IDLE: HAL_GPIO_WritePin(GPIOB, GPIO_PIN_4, GPIO_PIN_SET);break;  
        case TAG_TASK_NORMAL: HAL_GPIO_WritePin(GPIOB, GPIO_PIN_5, GPIO_PIN_SET);break;  
        case TAG_TASK_ALTA: HAL_GPIO_WritePin(GPIOB, GPIO_PIN_6, GPIO_PIN_SET);break;  
    }  
}
```

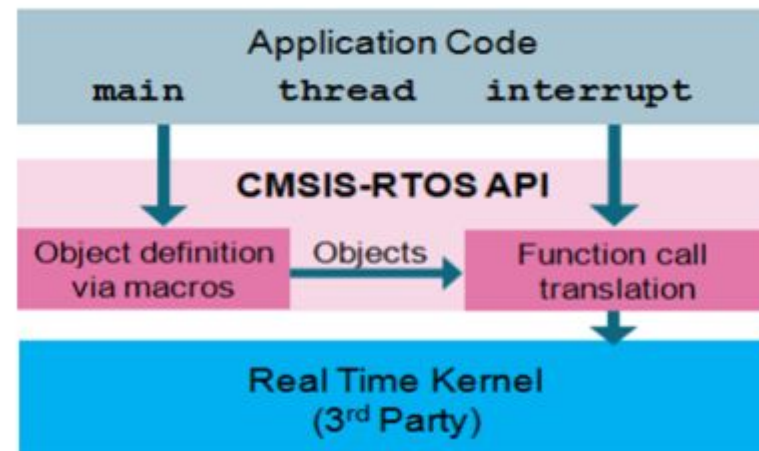
```
/* En cada tarea (incluir task.h)*/  
vTaskSetApplicationTaskTag(NULL, (void*) TAG_TASK_NORMAL);
```

FreeRTOS y CMSIS

¿FreeRTOS o CMSIS?

- ARM implementó una capa de abstracción que se llam **CMSIS** para portar código entre microcontroladores con procesadores Cortex
- ST implementó el uso de sistemas operativos en el IDE a partir de esta capa de abstracción, en particular la versión **CMSIS OS v2**
- Por debajo de esta capa puede correr el sistema operativo deseado, como FreeRTOS, ThreadX, etc
- La API de CMSIS no cubre toda la funcionalidad del OS de base
- Además, para entender en forma completa la funcionalidad dependemos del FreeRTOS (que es el que vamos a usar debajo de CMSIS)
- Por lo tanto se debe recurrir a las dos APIs

Figure 5. CMSIS-RTOS architecture



¿FreeRTOS o CMSIS?

```

cmsis_os2.c × *main.c
790
791 osStatus_t osDelay (uint32_t ticks) {
792     osStatus_t stat;
793
794     if (IS_IRQ()) {
795         stat = osErrorISR;
796     }
797     else {
798         stat = osOK;
799
800         if (ticks != 0U) {
801             vTaskDelay(ticks);
802         }
803     }
804
805     return (stat);
806 }

```

```

void vAnotherTask( void * pvParameters )
{
    for( ;; )
    {
        /* Perform some processing here. */

        ...

        /* Enter the Blocked state for 20 tick interrupts - the
        in the Blocked state is dependent on the tick frequency.
        vTaskDelay( 20 );

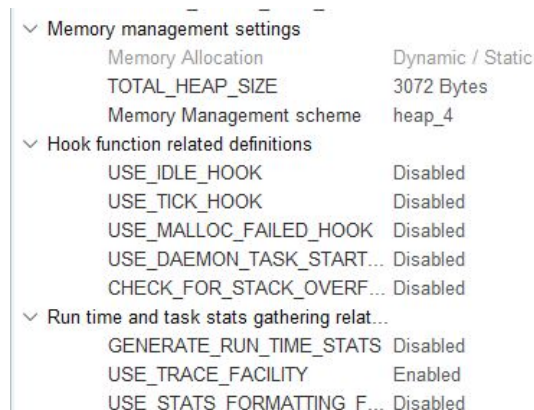
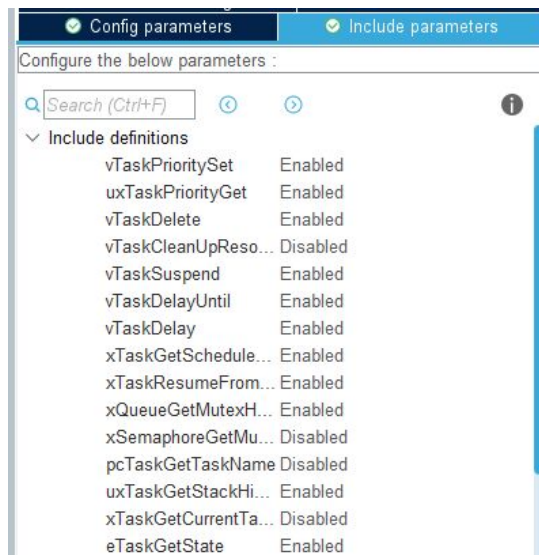
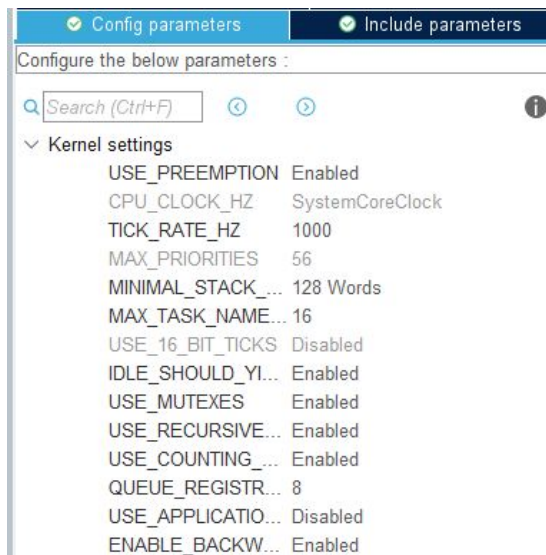
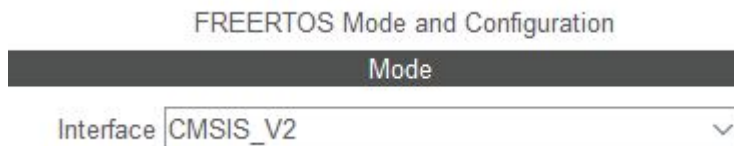
        /* 20 ticks will have passed since the first call to vTaskDelay()
        executed. */

        /* Enter the Blocked state for 20 milliseconds. Using the
        pdMS_TO_TICKS() macro means the tick frequency can change without
        affecting the time spent in the blocked state (other than the
        resolution of the tick frequency). */
        vTaskDelay( pdMS_TO_TICKS( 20 ) );
    }
}

```

Listing 21 Example use of vTaskDelay()

Instalando FreeRtos con capa CMSIS OS V2 desde Cube IDE



Detalles sobre la sintaxis de FreeRTOS

- Tipos de dato

- Se define **BaseType_t** como el tipo de dato más eficiente que en nuestra arquitectura es el de 32 bits
- Otro muy usado es **TickType_t** que especifica períodos de tiempo (e.g. las funciones TaskDelay lo admiten)

- Nombres de variables

- Se les agrega **c**, **s**, **l** si son de tipo char, short o long. **Además**, Si son unsigned se agrega **u** y si son punteros **p**
- En cambio, se les agrega **x** si son de un tipo específico de FreeRTOS como BaseType_t

- Nombres de las funciones

- La primera letra corresponde al tipo de dato que devuelven, agregando **v** para void
- La palabra siguiente corresponde al archivo donde se definen (task.c , queue.c, etc)
- Y luego el nombre específico

vTaskDelayUntil()

- Nombres de Macros

- Se definen en mayúscula y agrega un prefijo según su archivo .h de definición