Co-funded by the
Erasmus+ Programme
of the European Union

Módulo 5

Sistema operativo de tiempo real: FreeRTOS

Sistema Operativo en Tiempo Real (RTOS)

- Multitasking preemptivo
- Tareas priorizadas y ejecutadas por scheduler
- Prioridades
- Tareas se bloquean si esperan eventos o recursos
- Tiempo de respuesta: suma de los tiempos de ISR + tareas de mayor prioridad

¿Por qué usar un RTOS?

- Mantenimiento
- Modularidad
  - Testing más fácil
  - Reusar código
- Eficiencia
- Tiempo Idle
  - Power management

RTOS: Conceptos

RTOS Suave

Las tareas se ejecutan lo más rápido posible pero no tienen un deadline estricto.

RTOS Duro

Las tareas deben ejecutarse rápido, bien, y en tiempo.

RTOS: Conceptos

Recurso: Es una entidad utilizada por una tarea. Por ejemplo: cualquier dispositivo de I/O (teclado, display, etc) o una variable.

Recurso compartido: Es un recurso que puede ser utilizado por más de una tarea. Se debe utilizar algún mecanismo para garantizar que un recurso compartido no se utilice por más de una tarea al mismo tiempo. (Exclusión mutua)

RTOS: Conceptos

Sección crítica: Secciones de código que son indivisibles. Una sección crítica no puede ser interrumpida. ¿Cómo? Deshabilitando int antes y habilitando después (recurso compartido)

Tarea: Es un programa que opera bajo el supuesto de que es lo único ejecutándose en la CPU. Cada tarea tiene una prioridad, su juego de registros de CPU y su stack

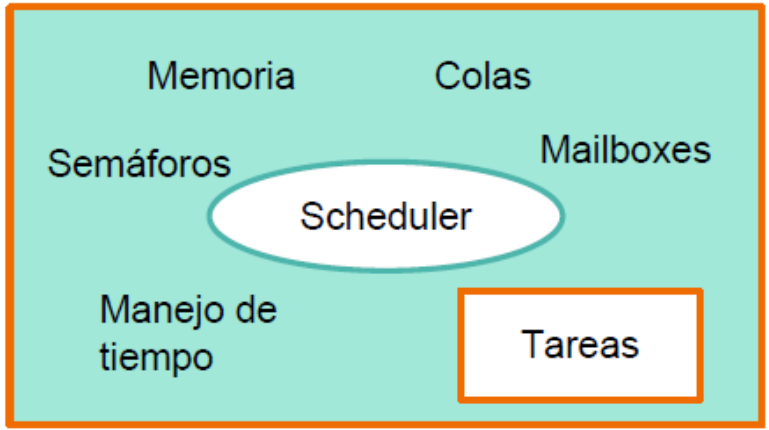
RTOS: Conceptos

Context switch (task switch): Acción que ocurre cuando el kernel decide correr otra tarea. Se guarda el estado actual de la tarea en ejecución (task's context, CPU registers) en el propio stack de la tarea. Cuantos más registros tenga el CPU, más demora.

Kernel: la parte del sistema operativo responsable de administrar las tareas y la comunicación entre ellas. Tener en cuenta que ocupa memoria RAM, y ROM, así como recursos de CPU. Permite el uso de semáforos, mailboxes, colas, memoria dinámica, tiempo, etc.

RTOS: Conceptos

Scheduler: Es la parte del kernel responsable de determinar cuál tarea debe ejecutarse. En un kernel basado en prioridades, el scheduler siempre le ejecuta la tarea con mayor prioridad cuyo estado sea READY.



RTOS: Conceptos. No preemptivo vs

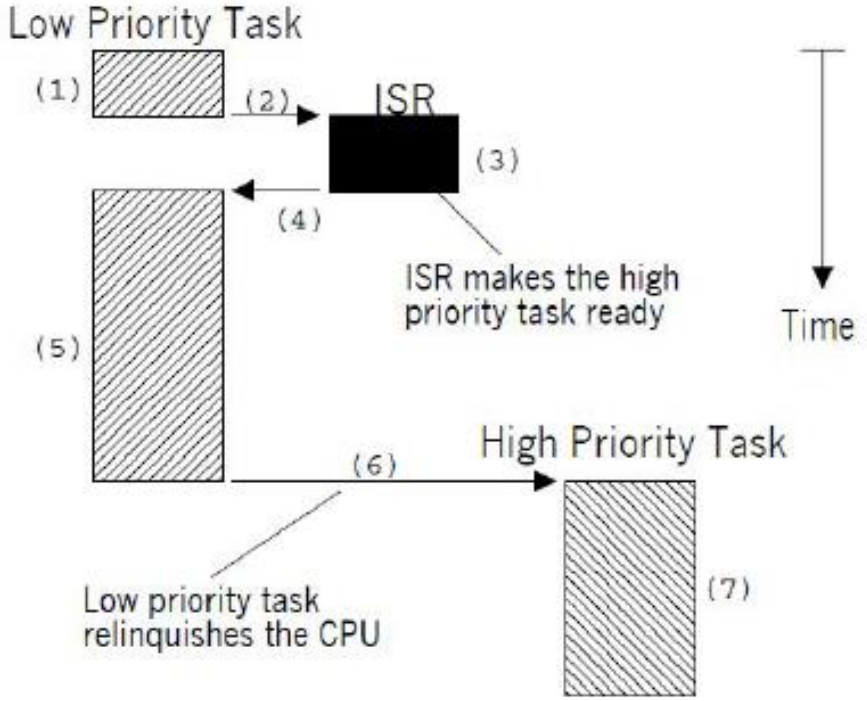


Figure 2-4, Non-preemptive kernel

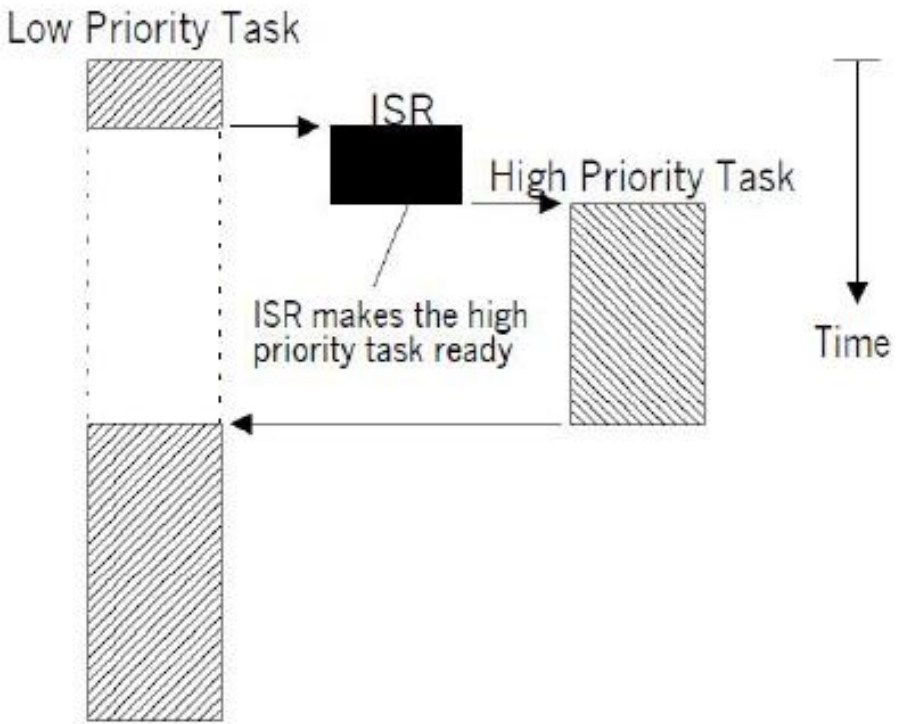


Figure 2-5, Preemptive kernel

RTOS: Conceptos

Exclusión mutua: Cuando dos o más tareas comparten variables globales, se debe utilizar algún método para acceder a los datos de forma exclusiva. Las formas de hacer esto son:

1. Deshabilitar interrupciones: taskENTER_CRITICAL() y taskEXIT_CRITICAL()
2. Deshabilitar el scheduler: vTaskSuspendAll() y xTaskResumeAll()
3. Utilizar semáforos

RTOS: Conceptos

Semáforos: son un mecanismo para permitir el acceso de forma segura a un recurso compartido.

Hay dos tipos de semáforos: binary y counting.
Un semáforo binario (mutex) es un semáforo contador inicializado en 1.

Las operaciones que se pueden con un semáforo son WAIT / PEND / **TAKE** y SIGNAL / POST / **GIVE**.

RTOS: Conceptos

Comunicación entre tareas: se pueden comunicar a través de variables globales o enviando mensajes. El envío de mensajes es un servicio del kernel.

- Queues (colas)
- Mailboxes
- Memoria

RTOS:

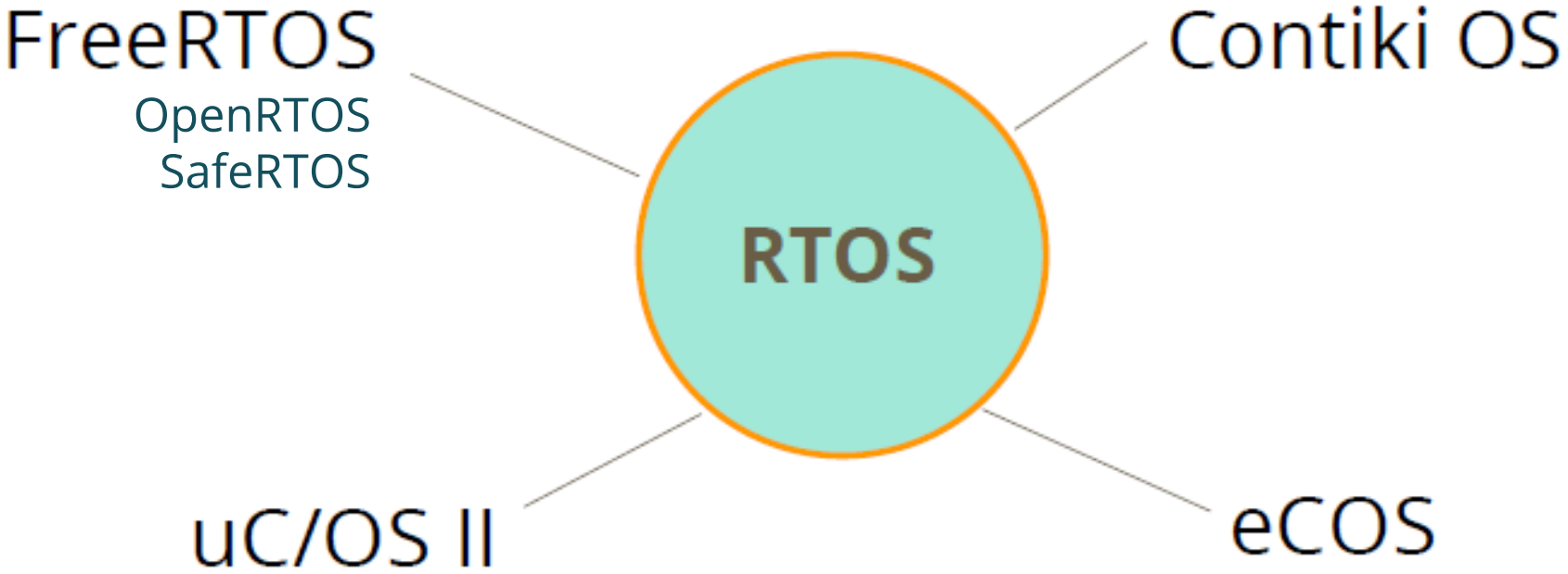
Ventajas:

- Estabilidad cuando el código cambia: agregar una tarea no afecta el tiempo de respuesta de las tareas de mayor prioridad.
- Muchas opciones disponibles: comerciales y GNU.

Desventajas:

- Complejidad: mucho dentro de RTOS, algo para usarlo.
- Requerimientos mínimos y overhead del RTOS.

Sistema Operativo en tiempo real



freeRTOS

Características:

- Operación preemptiva o cooperativa
- Asignación flexible de prioridades
- Colas (Queues)
- Semáforos
- Timers por software
- Chequeo de stack overflow
- Trace recording
- Recolección de estadísticas en run-time
- Bajo consumo
- GRATIROLA!

freeRTOS: Types

BaseType_t

Está siempre definido como el tipo de datos más eficiente para la arquitectura dada. Típicamente, es 32-bit en una arquitectura de 32-bit, 16-bit en una arquitectura de 16-bit, etc.

BaseType_t se usa generalmente para *returns* de funciones que pueden tomar solo un número muy limitado de valores.

freeRTOS: Types

TickType_t

FreeRTOS configura una interrupción periódica llamada “Interrupción de Tick”.

La cantidad de interrupciones de tick que ocurrieron desde que arrancó a correr FreeRTOS se llama *tick count*, y se usa como medida de tiempo.

El tiempo entre dos interrupciones de tick se llama “período de tick”. Los tiempos se especifican como múltiplos de este período.

TickType_t es el tipo de dato usado para la variable que almacena el *tick count*, y para especificar tiempos.

freeRTOS: Tasks

- Son funciones de C con un prototipo especial:

```
void ATaskFunction( void *pvParameters );
```

- Tienen un punto de entrada y generalmente van a estar ejecutándose en un loop infinito.
- No tienen `return`
- Una tarea puede crear a otras múltiples tareas.
- Cada tarea tiene reservado su stack y su copia de variables locales

freeRTOS: Tasks

```
void ATaskFunction( void *pvParameters ) {

   int32_t lVariableExample = 0;

   for(;;) {
      // El código que ejecuta la tarea
   }

   vTaskDelete(NULL);
}
```

freeRTOS Tasks

```
void ATaskFunction( void *pvParameters )
{
/* Variables can be declared just as per a normal function.  Each instance of a task
created using this example function will have its own copy of the lVariableExample
variable.  This would not be true if the variable was declared static – in which case
only one copy of the variable would exist, and this copy would be shared by each
created instance of the task. (The prefixes added to variable names are described in
section 1.5, Data Types and Coding Style Guide.) */
int32_t lVariableExample = 0;

    /* A task will normally be implemented as an infinite loop. */
    for( ;; )
    {
        /* The code to implement the task functionality will go here. */
    }

    /* Should the task implementation ever break out of the above loop, then the task
    must be deleted before reaching the end of its implementing function.  The NULL
    parameter passed to the vTaskDelete() API function indicates that the task to be
    deleted is the calling (this) task.  The convention used to name API functions is
    described in section 0, Projects that use a FreeRTOS version older than V9.0.0
must build one of the heap_n.c files.  From FreeRTOS V9.0.0 a heap_n.c file is only
required if configSUPPORT_DYNAMIC_ALLOCATION is set to 1 in FreeRTOSConfig.h or if
configSUPPORT_DYNAMIC_ALLOCATION is left undefined.  Refer to Chapter 2, Heap Memory
Management, for more information.
Data Types and Coding Style Guide. */
    vTaskDelete( NULL );
}
```

Listing 12. The structure of a typical task function

freeRTOS: Estados de una tarea

- SUSPENDED: la tarea está en memoria, pero no está disponible para el kernel.
- READY: cuando puede ser ejecutada pero su prioridad es menor que la de la tarea actualmente en ejecución.
- RUNNING: cuando tiene el control de la CPU.
- BLOCKED (WAITING FOR EVENT): cuando requiere que suceda un evento para continuar (ej: que se complete una operación de I/O, pase un cierto tiempo, se libere un recurso, etc.).
- INTERRUPTED: cuando ocurre una interrupción y la CPU está procesando la rutina de interrupción.

freeRTOS: Estados de una tarea

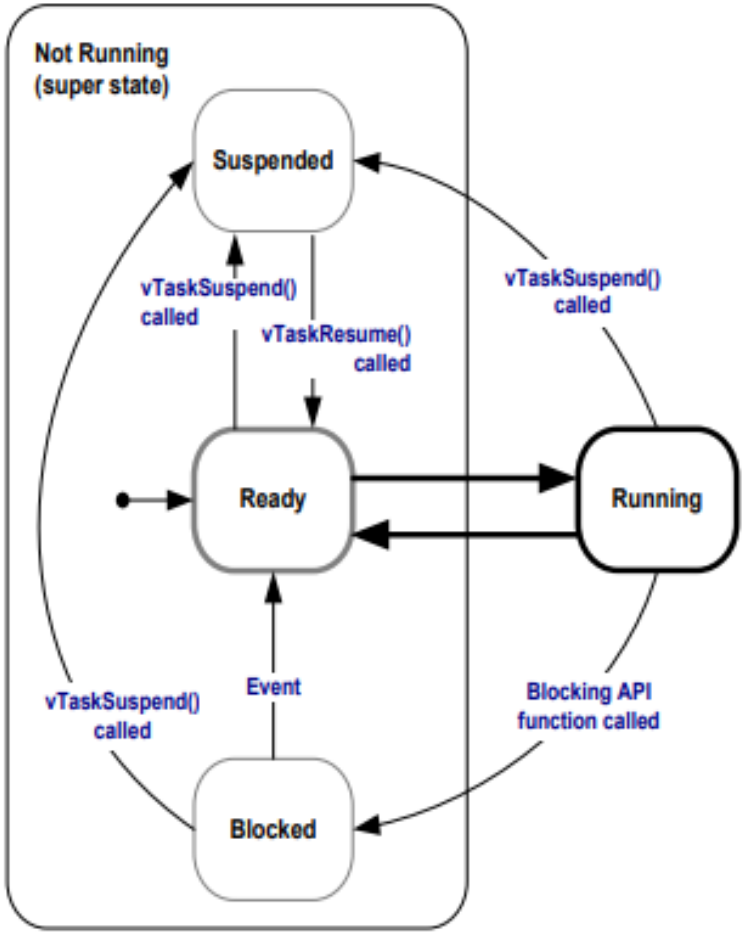


Figure 15. Full task state machine

freeRTOS: Idle Task

- Está corriendo cuando ninguna otra tarea está READY.
- Se le puede asociar una tarea a realizar con la Idle Task Hook.
- El tamaño de stack reservado para esta tarea está definido en el archivo *FreeRTOSConfig.h* por **configMINIMAL_STACK_SIZE**
- El tamaño de stack se define como la constante * 4 bytes (tamaño de palabra)
- Cumple el rol de “garbage collector”

freeRTOS: Prioridades de las tareas

- Van de 0 a (configMAX_PRIORITIES – 1). Siendo 0 la menor prioridad.
- configMAX_PRIORITIES es configurable y no puede ser mayor a 32.
- La Idle Task tiene prioridad definida por tskIDLE_PRIORITY y vale 0.
- Varias tareas pueden tener misma prioridad.
  - Si configUSE_TIME_SLICING no está definida, se define en 1, y las tareas de igual prioridad comparten procesamiento usando time slice.

freeRTOS: Prioridades de las tareas

- Se puede cambiar la prioridad con vTaskPrioritySet()
- Se puede consultar la prioridad de una tarea con uxTaskPriorityGet().

freeRTOS: Ejemplo - Creación de tareas

```
void vTask1( void *pvParameters ) {

    const char *pcTaskName = “Task 1 is running\n”;
        volatile uint32_t ul;

    for(;;) {
        USB_send( pcTaskName );
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );
    }

    vTaskDelete(NULL);
}
```

freeRTOS: Ejemplo - Creación de tareas

```
void vTask2( void *pvParameters ) {

    const char *pcTaskName = “Task 2 is running\n”;
        volatile uint32_t ul;

    for(;;) {
        USB_send( pcTaskName );
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );
    }

    vTaskDelete(NULL);
}
```

freeRTOS: Ejemplo - Creación de tareas

```
void main( void ) {

    xTaskCreate( vTask1, “Task 1”, 100, NULL, 1, NULL);
    xTaskCreate( vTask2, “Task 2”, 100, NULL, 1, NULL);

    vTaskSartScheduler();

    for(;;);
}
```

freeRTOS: Ejemplo - Creación de ta

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

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

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 14. Implementation of the first task used in Example 1

freeRTOS: Ejemplo - Creación de ta

```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

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

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 15. Implementation of the second task used in Example 1

freeRTOS: Ejemplo - Creación de tare

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* This example does not use the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```

Listing 16. Starting the Example 1 tasks

freeRTOS: Ejemplo - Creación de tarea

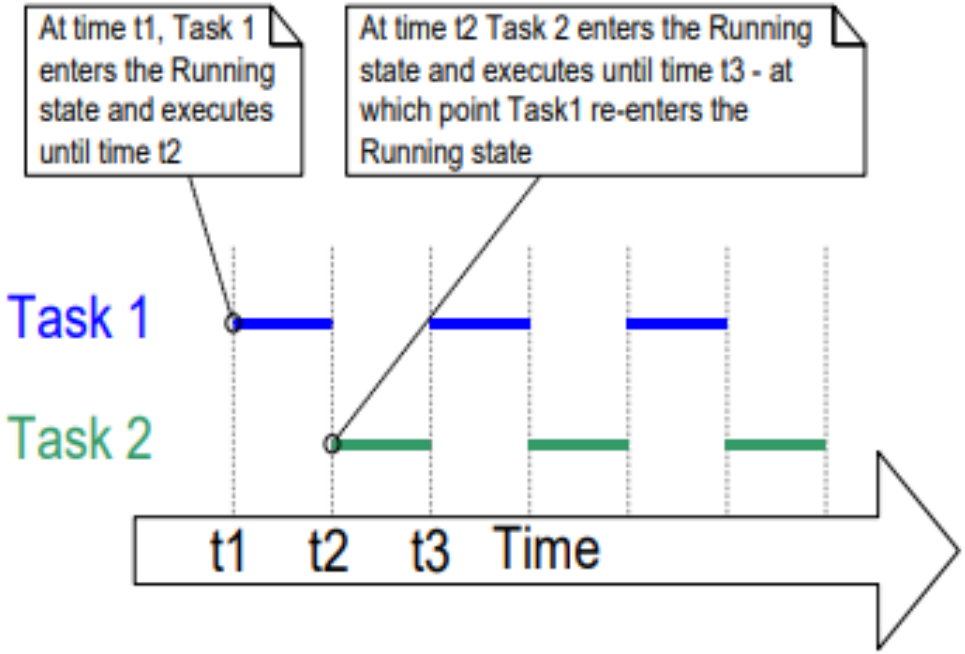


Figure 11. The actual execution pattern of the two Example 1 tasks

freeRTOS: Tareas que crean tareas

```
void vTask1( void *pvParameters ) {

    const char *pcTaskName = “Task 1 is running\n”;
        volatile uint32_t ul;

    xTaskCreate( vTask2, “Task 2”, 100, NULL, 1, NULL);

    for(;;) {
        USB_send( pcTaskName );
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );
    }

    vTaskDelete(NULL);
}
```

freeRTOS: Tareas que crean tareas

```
void vTask1( void *pvParameters )
{
const char *pcTaskName = "Task 1 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* If this task code is executing then the scheduler must already have
    been started.  Create the other task before entering the infinite loop. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( pcTaskName );

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later examples will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 17. Creating a task from within another task after the scheduler has started

freeRTOS: Ejemplo - Usando parámetros

```
void vTaskFunction( void *pvParameters ) {
    char *pcTaskName;
    volatile uint32_t ul;

    pcTaskName = (char*)pvParameters;

    for(;;) {
        USB_send( pcTaskName );
        for( ul = 0 ; ul < mainDELAY_LOOP_COUNT ; ul++ );
    }

    vTaskDelete(NULL);
}
```

freeRTOS: Ejemplo - Usando parámetros

```
static const char *pcTextForTask1 = “Task 1 is running\n”;
static const char *pcTextForTask2 = “Task 2 is running\n”;

void main( void ) {

    xTaskCreate( vTask1, “Task 1”, 100, (void*)pcTextForTask1, 1, NULL);
    xTaskCreate( vTask2, “Task 2”, 100, (void*)pcTextForTask2, 1, NULL);

    vTaskSartScheduler();

    for(;;);
}
```

freeRTOS: Ejemplo - Usando para

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

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

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

        /* Delay for a period. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )
        {
            /* This loop is just a very crude delay implementation.  There is
            nothing to do in here.  Later exercises will replace this crude
            loop with a proper delay/sleep function. */
        }
    }
}
```

Listing 18. The single task function used to create two tasks in Example 2

freeRTOS: Ejemplo - Usando

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

int main( void )
{
    /* Create one of the two tasks. */
    xTaskCreate(    vTaskFunction,          /* Pointer to the function that
                                               implements the task. */
                    "Task 1",               /* Text name for the task.  This is to
                                               facilitate debugging only. */
                    1000,                   /* Stack depth - small microcontrollers
                                               will use much less stack than this. */
                    (void*)pcTextForTask1,  /* Pass the text to be printed into the
                                               task using the task parameter. */
                    1,                      /* This task will run at priority 1. */
                    NULL );                 /* The task handle is not used in this
                                               example. */
```

freeRTOS: Ejemplo - Usando

```
    /* Create the other task in exactly the same way.  Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction).  Only
    the value passed in the parameter is different.  Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

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

    /* If all is well then main() will never reach here as the scheduler will
    now be running the tasks.  If main() does reach here then it is likely that
    there was insufficient heap memory available for the idle task to be created.
    Chapter 2 provides more information on heap memory management. */
    for( ;; );
}
```

Listing 19. The main() function for Example 2.

freeRTOS: Eliminación de tareas

- Debe estar INCLUDE_vTaskDelete definido en 1 en FreeRTOSConfig.h.
- `vTaskDelete()`
- Las tareas eliminadas ya no existen y no pueden entrar a un estado “running”
- La Idle Task es la que se encarga de liberar la memoria de cualquier tarea que haya sido eliminada

```
void vTaskDelete( TaskHandle_t pxTaskToDelete );
```

freeRTOS: Ejemplo - Eliminación tareas

```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters ) {
        const TickType_t xDelay100ms = pdMS_TO_TICKS(100UL);

    for(;;) {
        USB_send( “Task 1 is running\n” );

        xTaskCreate( vTask2, “Task 2”, 100, NULL, 2, &xTask2Handle);

        vTaskDelay(xDelay100ms);
    }

    vTaskDelete(NULL);
}
```

freeRTOS: Ejemplo - Eliminación tareas

```
void vTask2( void *pvParameters ) {
    USB_send( “Task 2 is running\n” );

    vTaskDelete(NULL);
}
```

```
void main( void ) {

    xTaskCreate( vTask1, “Task 1”, 100, NULL, 1, NULL);

    vTaskSartScheduler();

    for(;;);
}
```

freeRTOS: Ejemplo - Eliminación tareas

```
int main( void )
{
    /* Create the first task at priority 1.  The task parameter is not used
    so is set to NULL.  The task handle is also not used so likewise is set
    to NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );
    /* The task is created at priority 1 ______^. */

    /* Start the scheduler so the task starts executing. */
    vTaskStartScheduler();

    /* main() should never reach here as the scheduler has been started. */
    for( ;; );
}
```

Listing 37. The implementation of main() for Example 9

freeRTOS: Ejemplo - Eliminación tare

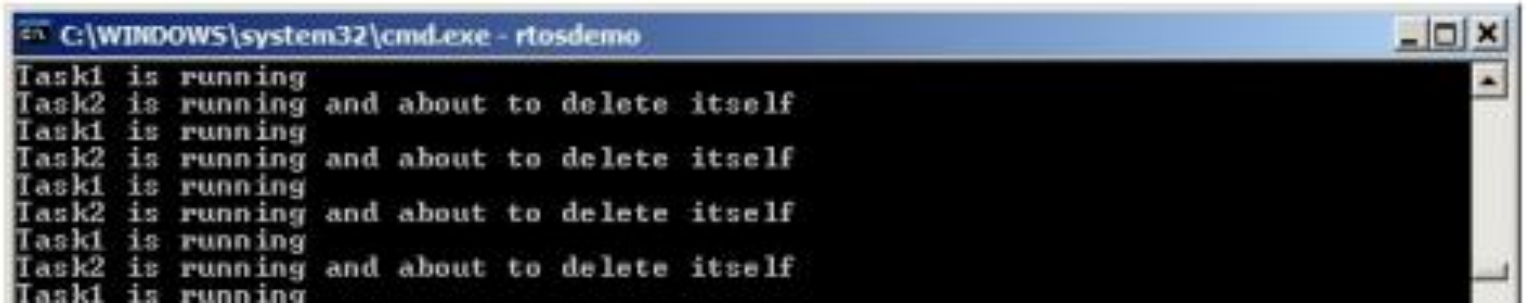
```
TaskHandle_t xTask2Handle = NULL;

void vTask1( void *pvParameters )
{
const TickType_t xDelay100ms = pdMS_TO_TICKS( 100UL );

    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority.  Again the task parameter is not
        used so is set to NULL - BUT this time the task handle is required so
        the address of xTask2Handle is passed as the last parameter. */
        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );
        /* The task handle is the last parameter _____^^^^^^^^^^^^^ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here Task 2
        must have already executed and deleted itself.  Delay for 100
        milliseconds. */
        vTaskDelay( xDelay100ms );
    }
}
```

Listing 38. The implementation of Task 1 for Example 9

freeRTOS: Ejemplo - Eliminación ta

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself.  To do this it could call vTaskDelete()
    using NULL as the parameter, but instead, and purely for demonstration purposes,
    it calls vTaskDelete() passing its own task handle. */
    vPrintString( "Task 2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}
```

Listing 39. The implementation of Task 2 for Example 9



freeRTOS: Time slices - Tick

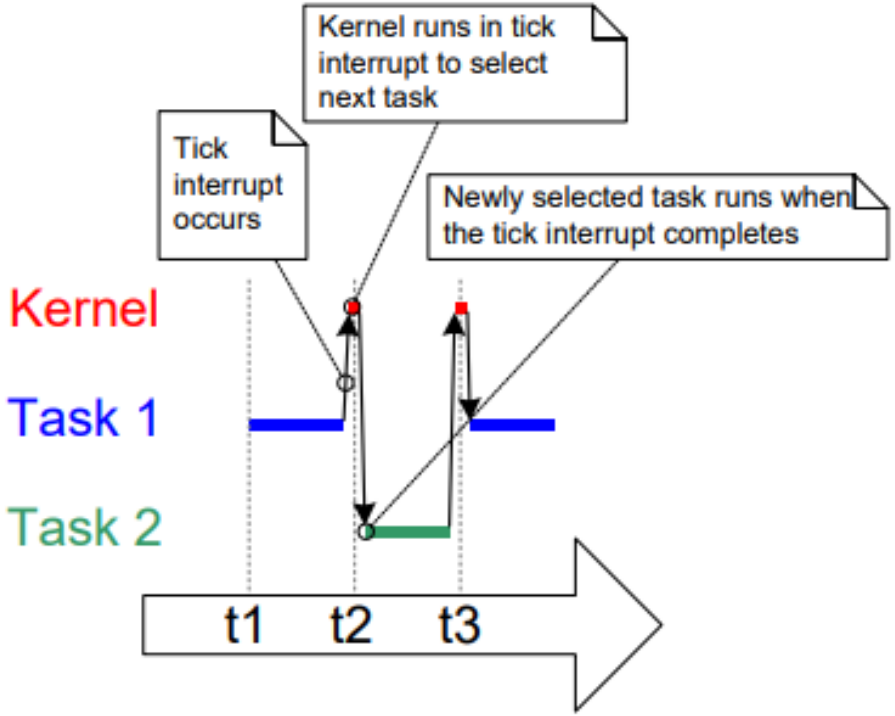


Figure 12. The execution sequence expanded to show the tick interrupt executing

freeRTOS: Delays

- Forma que tiene el scheduler de “darse cuenta” que una tarea espera por un tiempo y puede “bloquearse”
  Un delay en un loop, como en el ejemplo, no lo hace.

```
void vTaskDelay( TaskType_t xTicksToDelay );
```

- `vTaskDelay( pdMS_TO_TICKS( 100 ) )`
  Para hacer un delay de 100 ms

freeRTOS:

Eventos

Conceptos previos: Reentrante

Función reentrante: Se puede acceder desde más de una tarea a la vez, sin que los datos se corrompan.

```
void addOneHundred( long var ) {

    return var + 100;
}
```

Ejemplo de NO reentrante

```
long total = 0;

void addOneHundredToTotal( void ) {

    return total + 100;
}
```

Conceptos previos: Func. Reentrante

Función reentrante: Se puede acceder desde más de una tarea a la vez, sin que los datos se corrompan.

```
/* A parameter is passed into the function.  This will either be passed on the stack,
or in a processor register.  Either way is safe as each task or interrupt that calls
the function maintains its own stack and its own set of register values, so each task
or interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( long lVar1 )
{
/* This function scope variable will also be allocated to the stack or a register,
depending on the compiler and optimization level.  Each task or interrupt that calls
this function will have its own copy of lVar2. */
long lVar2;

    lVar2 = lVar1 + 100;
    return lVar2;
}
```

Listing 112. An example of a reentrant function

Conceptos previos: Func. Reentrante

EJEMPLO DE FUNCIÓN NO REENTRANTE

```
/* In this case lVar1 is a global variable, so every task that calls
lNonsenseFunction will access the same single copy of the variable. */
long lVar1;

long lNonsenseFunction( void )
{
/* lState is static, so is not allocated on the stack.  Each task that calls this
function will access the same single copy of the variable. */
static long lState = 0;
long lReturn;

    switch( lState )
    {
        case 0 : lReturn = lVar1 + 10;
                 lState = 1;
                 break;

        case 1 : lReturn = lVar1 + 20;
                 lState = 0;
                 break;
    }
}
```

Listing 113. An example of a function that is not reentrant

Conceptos previos: Func. Reentrante

Baja prioridad

```
...
while(1) {
        x = 1;
    y = 1;

    swap(&x, &y);

    vTaskDelay(10);
}
```

Alta prioridad

```
...
while(1) {
        z = 3;
    t = 4;

    swap(&z, &t);

    vTaskDelay(10);
}
```

```
void swap(int *a, int* b) {
    temp = *a;
    *a = *b;
    *b = temp;
}
```

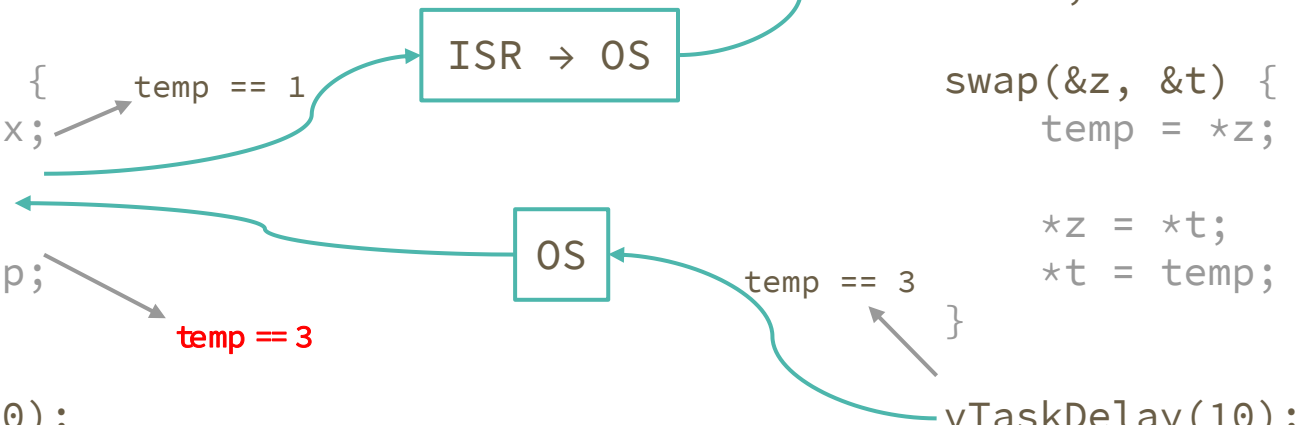
Conceptos previos: Func. Reentrante

Baja prioridad

```
...
while(1) {
        x = 1;
    y = 1;

    swap(&x, &y) {
        temp = *x;

        *x = *y;
        *y = temp;
    }

    vTaskDelay(10);
}
```

Alta prioridad

```
...
while(1) {
        z = 3;
    t = 4;

    swap(&z, &t) {
        temp = *z;

        *z = *t;
        *t = temp;
    }

    vTaskDelay(10);
}
```



freeRTOS: Sincronización

Para la sincronización entre tareas y uso de recursos compartidos, se utilizan:

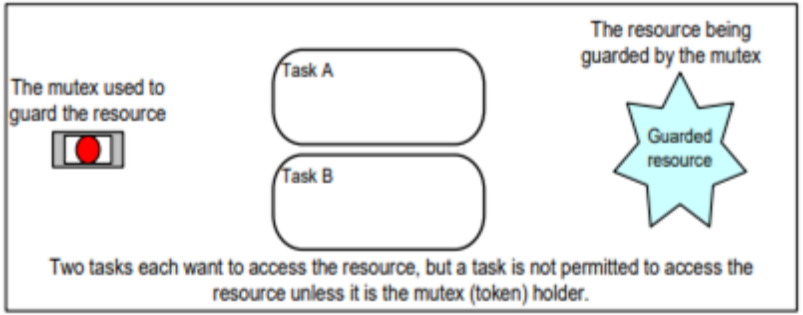
- Mutex (y recursive mutex)
- Binary semaphore (sincronización)
- Counting semaphore

freeRTOS: Mutex

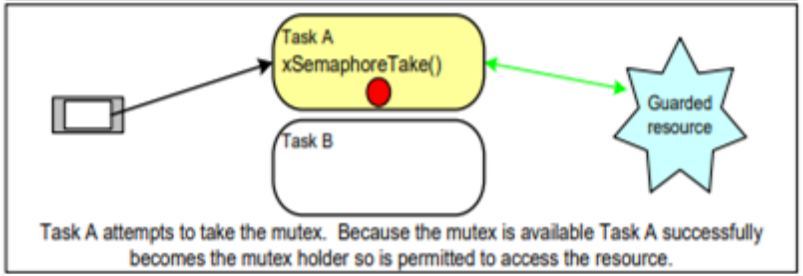
- Controla el acceso a un recurso compartido por dos o más tareas.
- Funciones:

```
SemaphoreHandle_t xSemaphoreCreateMutex(void);
SemaphoreHandle_t xSemaphoreCreateMutexStatic(void);

bool xSemaphoreTake(SemaphoreHandle_t xSemaphore,
                    TickType_t xTicksToWait);

bool xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

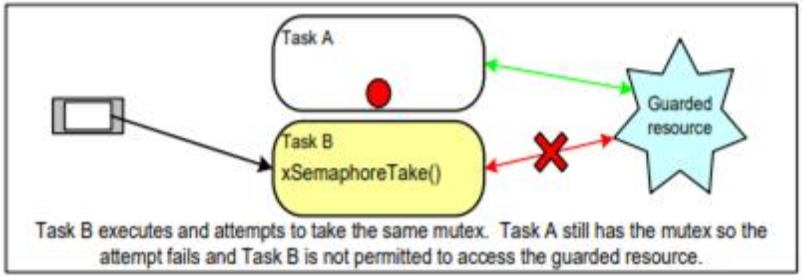
freeRTOS: Mutex



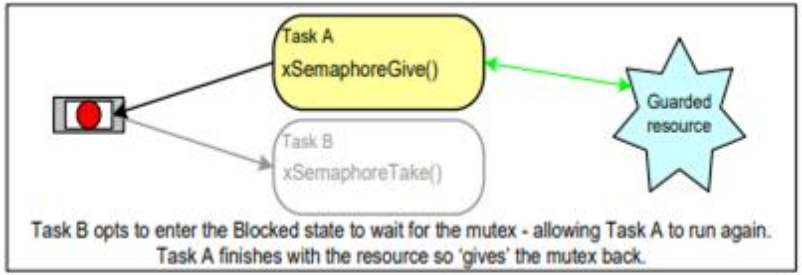
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.



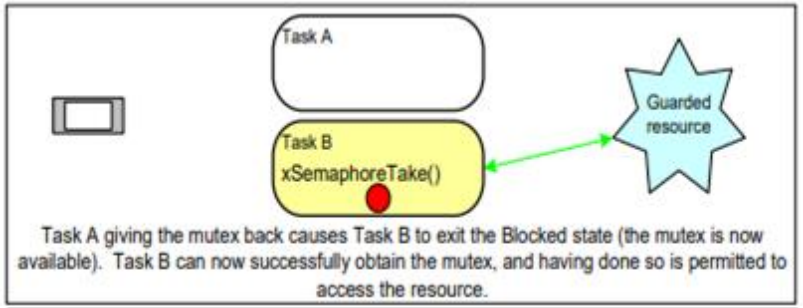
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.



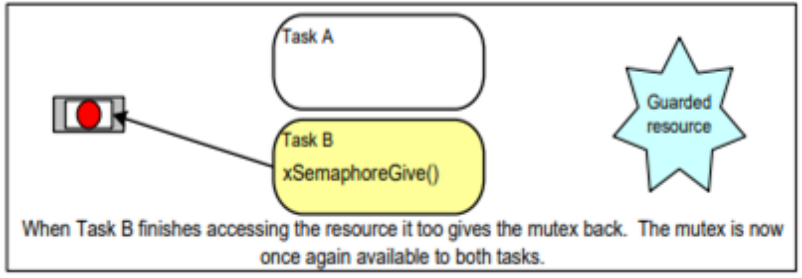
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.



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



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

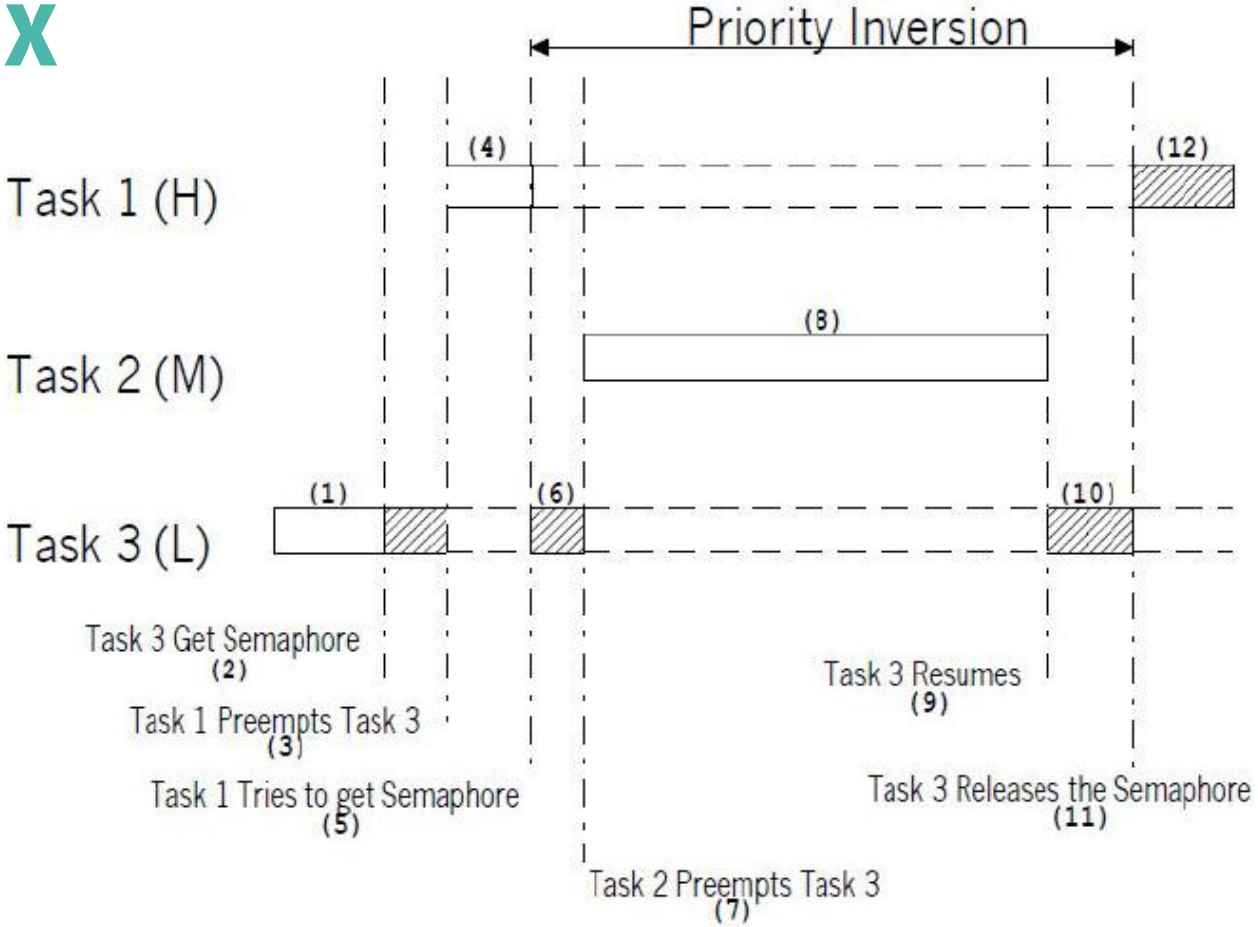


When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

RTOS: Mutex

Inversión de prioridades

Mutex implementan “priority inheritance” para resolverlo



freeRTOS: Mutex recursivo

- Similar al mutex, pero permite que sea tomada varias veces por una misma task. Debe ser devuelto tantas veces como fue tomado antes de estar disponible para otra task.
- Funciones:

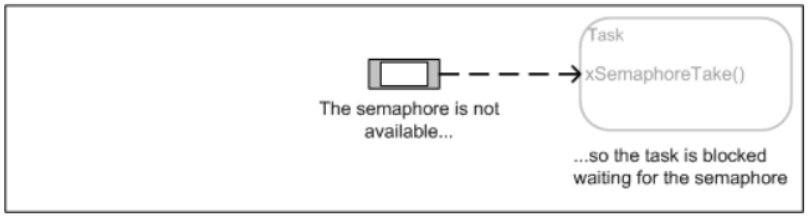
```
SemaphoreHandle_t xSemaphoreCreateRecursiveMutex(void);
SemaphoreHandle_t xSemaphoreCreateRecursiveMutexStatic(void);

bool xSemaphoreTake(SemaphoreHandle_t xSemaphore,
                    TickType_t xTicksToWait);

bool xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

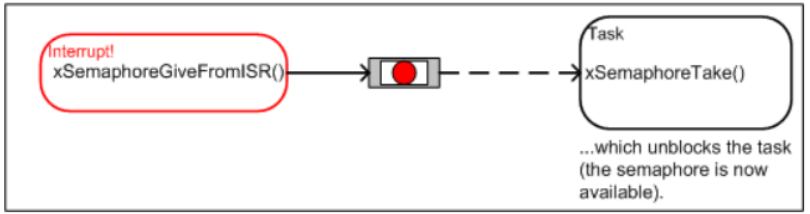
freeRTOS: Binary Semaphore

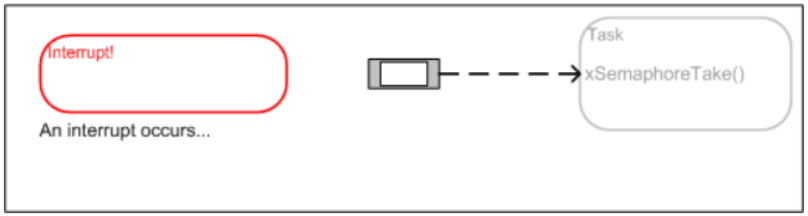
- Más apropiado para sincronización y señalización entre tareas
- Funciones:

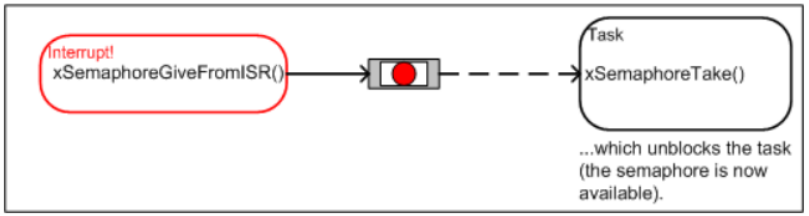
```
SemaphoreHandle_t xSemaphoreCreateBinary(void);
SemaphoreHandle_t xSemaphoreCreateBinaryStatic(void);

bool xSemaphoreTake(SemaphoreHandle_t xSemaphore,
                    TickType_t xTicksToWait);

bool xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

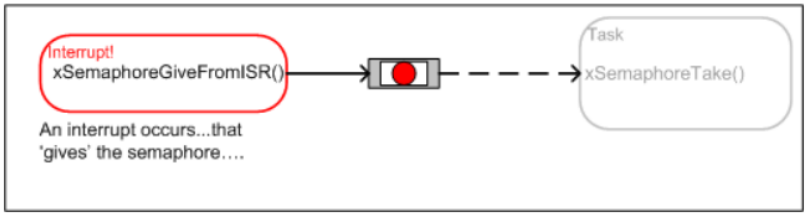
freeRTOS: Binary Semaphore

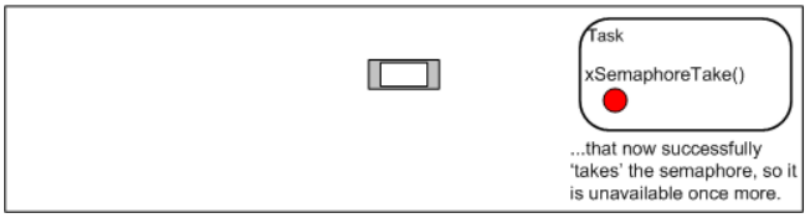












freeRTOS: Queues

- Cola de datos de algún tipo con un largo determinado. El largo y el tipo se fijan al crearse.
- FIFO
- Pueden guardar datos (queue by copy) o punteros (queue by reference)

freeRTOS: Queues

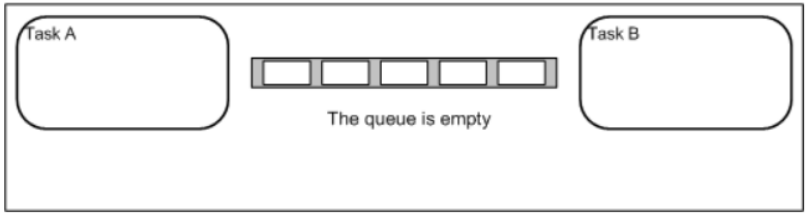
```
QueueHandle_t xQueueCreate (UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

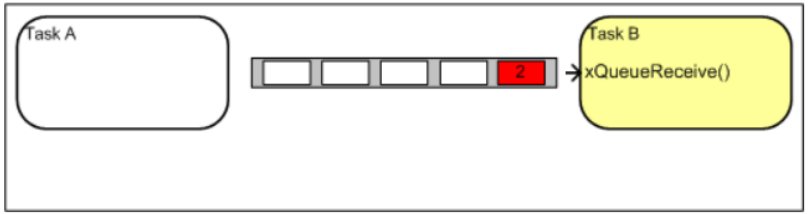
```
BaseType_t xQueueSendToBack (QueueHandle_t xQueue,
                             const void *pvItemToQueue,
                             TickType_t xTicksToWait);
```

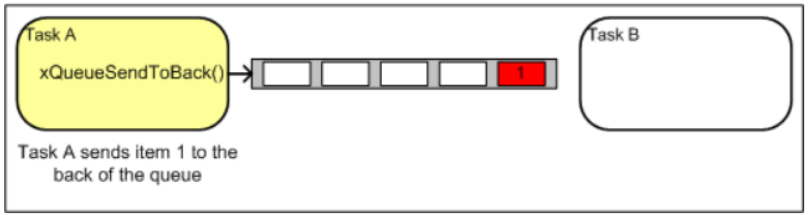
```
BaseType_t xQueueReceive (QueueHandle_t xQueue,
                          void * const pvBuffer,
                          TickType_t xTicksToWait);
```

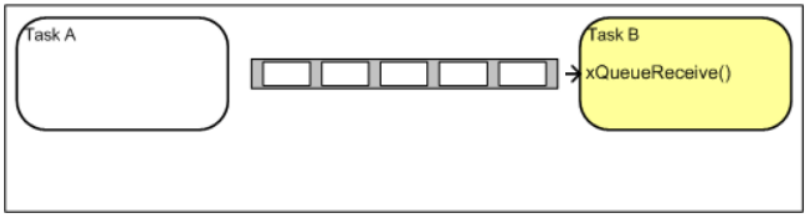
```
UBaseType_t uxQueueMessagesWaiting (QueueHandle_t xQueue);
```

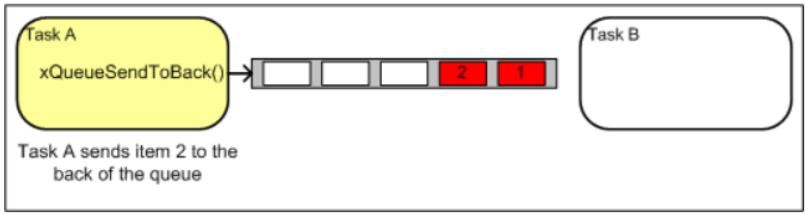
freeRTOS: Queues

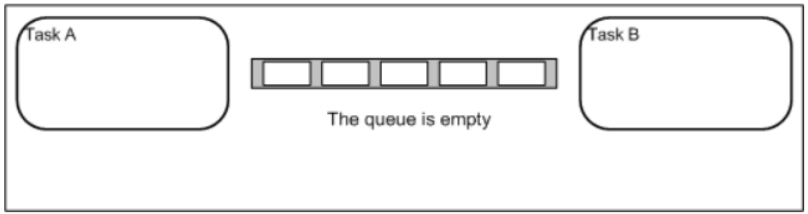












freeRTOS: Timers

- Permite que una función (callback) se ejecute en el futuro (por única vez, o de forma periódica)

- Funciones:

```
TimerHandle_t xTimerCreate (const char * const pcTimerName,
                            const TickType_t xTimerPeriod,
                            const UBaseType_t uxAutoReload,
                            void * const pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction);

void vTimerCallback (TimerHandle_t xTimer);    //Callback
```

freeRTOS: Memoria dinámica

- FreeRTOS gestiona la memoria heap, y existen funciones para solicitar y liberar dinámicamente bloques de memoria
- Funciones:

```
void* malloc (size_t size);
void* calloc (size_t nitems, size_t size);
void* realloc (void *ptr, size_t size);

void free (void* ptr);
```

freeRTOS: Memoria dinámica vs estática

- El kernel del RTOS necesita RAM cada vez que se crea una tarea, una cola, un semáforo u otro evento.
- La memoria se puede asignar de forma dinámica por el RTOS o estática por el programador. Cuál utilizar depende del programador y cada una tiene ventajas y desventajas.

freeRTOS: Ventajas de memoria dinámica

- Mayor simplicidad
- Potencial disminución del máximo de RAM utilizado:
  - Menos parámetros necesarios
  - La memoria la aloja la API del RTOS
  - El programador no debe preocuparse del alojamiento de memoria.
  - Reutilización de RAM entre objetos

freeRTOS: Ventajas de memoria estática

- Mayor control por parte del programador:
  - Objetos colocados en direcciones de memoria específicas.
  - Fijar el máximo de RAM a utilizar a nivel de linker y no de ejecución.
  - Permite utilizar el RTOS en sistemas que no permiten manejo de memoria dinámico

Referencias

- Richard Barry: ***Mastering the FreeRTOS™ Real Time Kernel.*** *A Hands-On Tutorial Guide.* Real Time Engineers Ltd. 2016.
- ***Dynamic C.*** *User's Manual.* Digi International® Inc. 2011