UNIVERSIDAD NACIONAL DE ROSARIO
FACULTAD DE CIENCIAS EXACTAS, INGENIERÍA Y AGRIMENSURA
DEPARTAMENTO DE SISTEMAS E INFORMÁTICA

Introducción a FreeRTOS

Ing. Gustavo Muro

CC BY NC ND

Introducción a RTOS

- Modelo de programación Bare Metal
  - Se denomina programación “bare-metal” cuando el MCU no utiliza recursos de un sistema operativo.
  - Ejemplos:
    - Super loop
    - Foreground-background
  - Generalmente el comportamiento es cooperativo.

Super Loop

- Un único bucle infinito chequea en orden cíclico el estado de los dispositivos de E/S y los atiende cuando es necesario
  - Sin interrupciones, toda la E/S por pooling.
  - Una única ejecución: no existen datos globales compartidos
  - Es simple.
- Problemas
  - Las latencias/tiempos de respuesta son poco predecibles.
    - En el peor caso, es el tiempo que tarda el bucle en realizar una iteración completa y en ejecutar todas la funciones asociadas a cada uno de los dispositivos
  - La arquitectura es frágil.
    - Cualquier cambio en el código altera los tiempos de respuesta.
    - El comportamiento del sistema es muy dependiente de la estructura del código y de la secuencia de eventos.

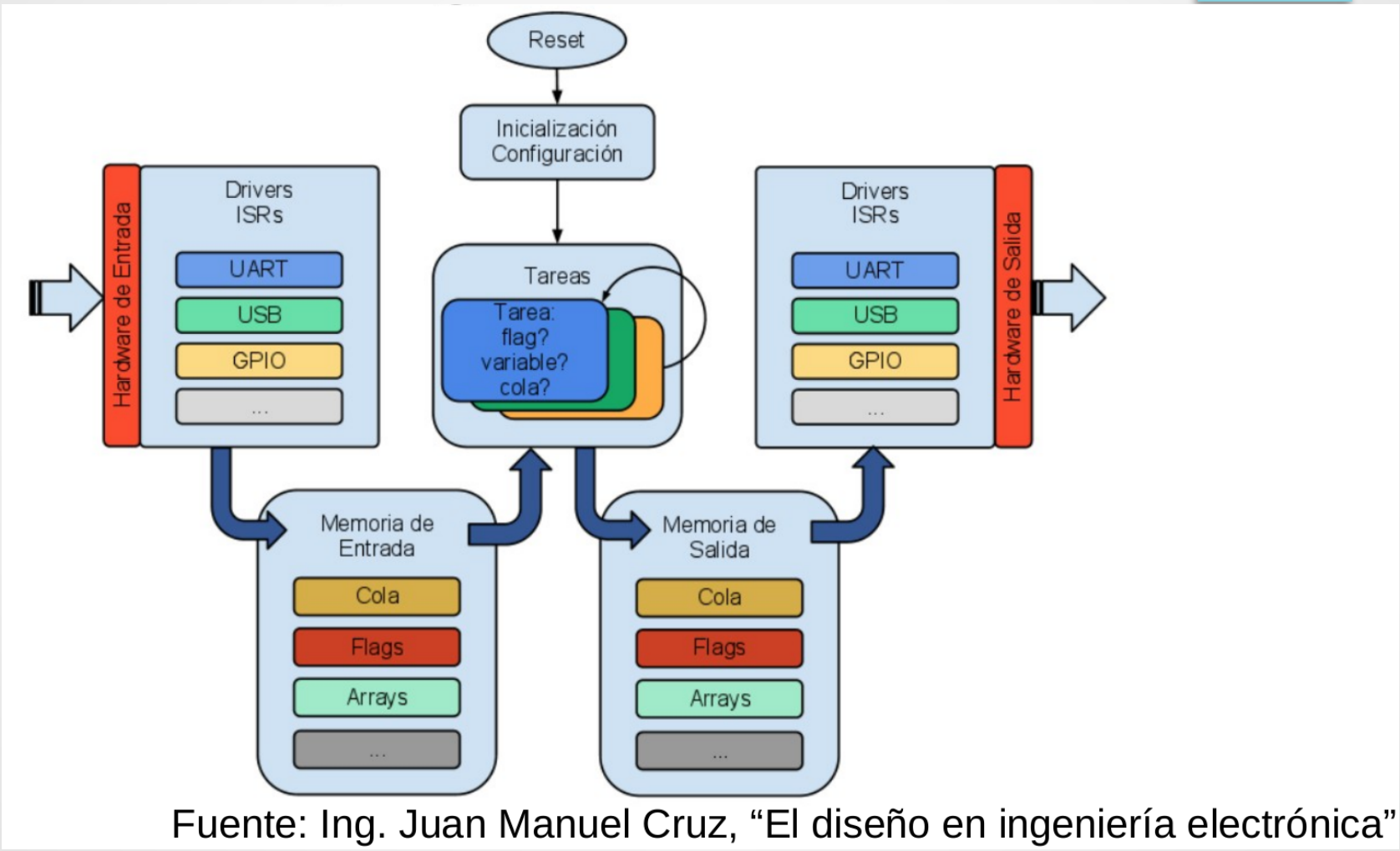
Fuente: http://www.fdi.ucm.es/profesor/mendias/PSyD/docs/PSyDtema6.pdf

Foreground / Background

- Super-loop con interrupciones
  - Distintas rutinas de servicio de interrupción atienden a los dispositivos de E/S según su prioridad cuando estos lo solicitan por interrupción
    - Estas rutinas, típicamente, solo señalizan eventos y/o escriben/leen datos en FIFOs
  - La ejecución en background (main) realiza un bucle infinito de mínima prioridad
    - Típicamente realiza todas las funciones no críticas en tiempo y todo el procesamiento de datos encolados
  - Las diferentes rutinas utilizan variables globales para comunicarse
    - Es necesario detectar y resolver el acceso a secciones críticas.
- Problemas
  - Si el procesamiento se realiza en background
    - Las funciones asociadas a cualquier dispositivo tienen la misma prioridad
    - En el peor caso, los procesamientos se realizan con retrasos similares al modelo super-loop
  - Si el procesamiento se realiza en foreground
    - Las ISR se alargan, y aunque las funciones se realizan según su prioridad, penalizan el tiempo de respuesta de interrupciones menos prioritarias.

Fuente: http://www.fdi.ucm.es/profesor/mendias/PSyD/docs/PSyDtema6.pdf

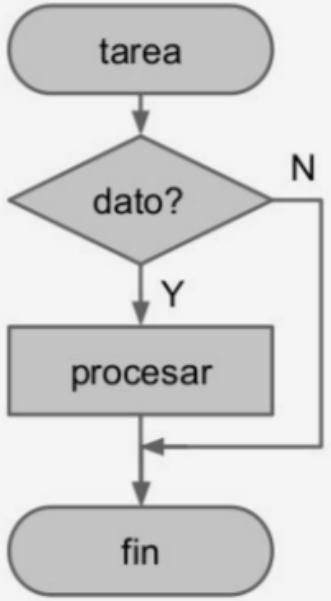
Introducción a RTOS
Modelo de programación

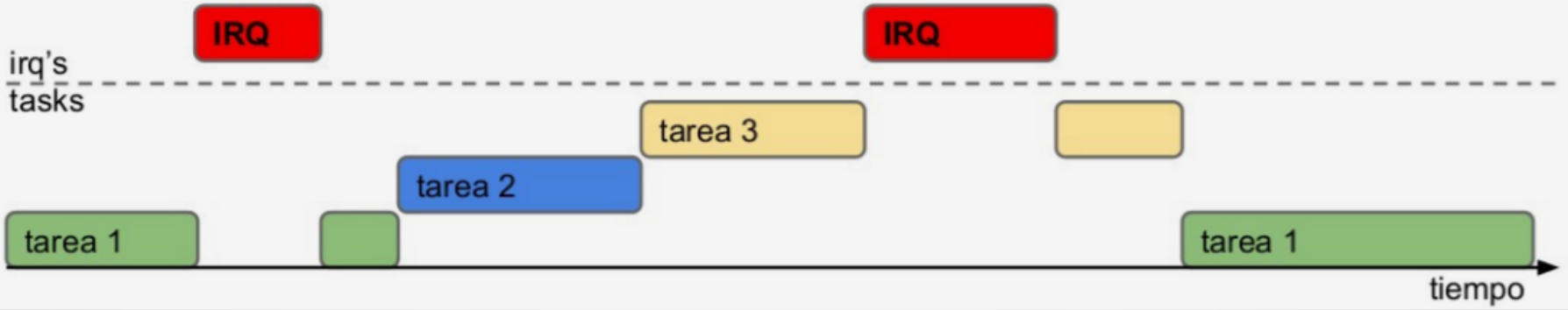


Fuente: Ing. Juan Manuel Cruz, “El diseño en ingeniería electrónica”

Introducción a RTOS
Modelo de programación bare metal

```
while(1)
{
   tarea1();
   tarea2();
   tarea3();
   ...
}
```



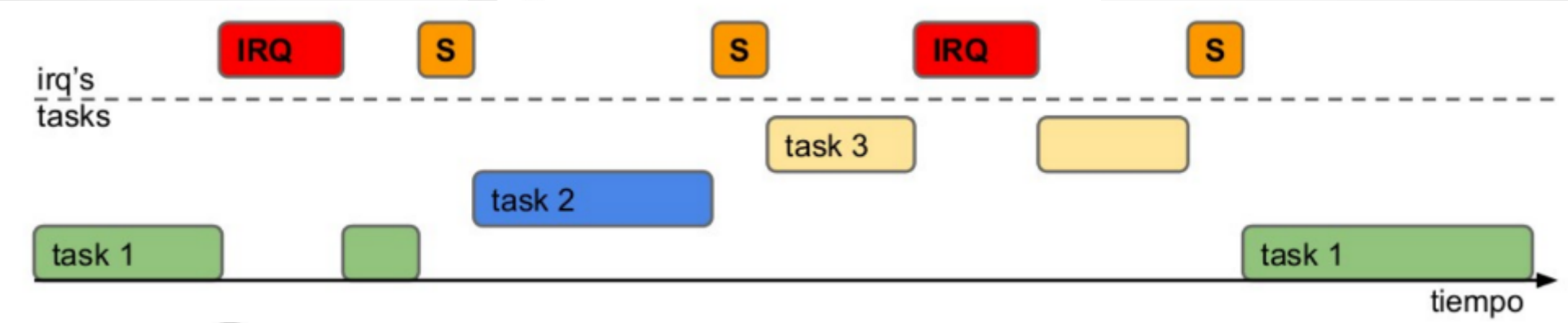


Introducción a RTOS
¿Por qué un RTOS?

- Diseñar un programa teniendo presente el comportamiento cooperativo implica una mayor carga para el programador.
- Una de las principales ventajas de usar un OS es la multitarea. Es decir que un hilo puede suspender su ejecución para que otro hilo se ejecute sin que el hilo original pierda su contexto.
- Por eso decimos que usar un OS aumenta la confiabilidad del sistema.

Modelo de programación Multitarea

```
int main(void)
{
     addTask(tarea1);
     addTask(tarea2);
     addTask(tarea3);
     startScheduler();
}

void tarea1(void)
{
     ...
     while(1)
     {
          ...
     }
}
```



Introducción a RTOS
¿Por qué un RTOS?

- Existen aplicaciones donde el tiempo de respuesta a un estímulo del sistema es un parámetro crítico.
- Es requisito que la respuesta del sistema se ubique dentro de una ventana de tiempo. Respuestas demasiado tempranas como demasiado tardías pueden ser indeseables.
- Por eso decimos que además de la confiabilidad, el uso de un RTOS contribuye a aumentar el determinismo del sistema.

Introducción a RTOS
¿Por qué un RTOS?

- El RTOS pone a disposición del programador recursos que permiten asegurar la respuesta del sistema, con cierta tolerancia.
  - Tareas con prioridad definida.
  - Semáforos, Mutex, Colas de Mensajes
- En función de los requerimientos para dicha tolerancia, podemos clasificar a los sistemas en dos grandes grupos.
  - Soft Real-Time.
  - Hard Real-Time.

Introducción a RTOS
¿Por qué un RTOS?

- Un sistema soft Real-Time intenta asegurar el tiempo de respuesta, aunque no se considera crítico que dicho tiempo no se cumpla ya que el sistema sigue siendo utilizable (ejemplo: interfaz de usuario).
- Un sistema hard Real-Time debe responder dentro de un período de tiempo definido por los requerimientos del sistema. Una respuesta por fuera de ese período puede significar una falla grave (ejemplo: airbag).

Introducción a RTOS
¿Por qué un RTOS?

- Las prioridades de las tareas ayudan a cumplir con los tiempos de respuesta de la aplicación
- Mejora el mantenimiento y la escalabilidad del sistema
- Ayuda a la modularidad de las tareas
- Ayuda a la definición de interfaces de los distintos módulos
- Facilita la reutilización de código
- Permite realizar tareas (idle task) cuando el sistema está desocupado o ir a un estado de bajo consumo
- Es flexible con el manejo de interrupciones

FreeRTOS

- FreeRTOS es un kernel de tiempo real (o un Scheduler de tiempo real)
  - Simple
  - Portable
  - Libre

- Mayormente escrito en C
- Algunas pocas funciones escritas en assembler

FreeRTOS
Algunas características

- Scheduler configurable: Pre-emptive o co-operative
- Asignación de prioridades y modificación en tiempo de ejecución
- Queues (colas de mensajes)
- Binary Semaphores
- Counting Semaphores
- Recursive Semaphores
- Mutex
- Tick hook function
- Idle hook function
- Stack overflow checking
- Trace hook macros

FreeRTOS - Distribución

- FreeRTOS puede ser compilado por más de veinte compiladores diferentes y puede ejecutarse en más de 30 arquitecturas de procesadores diferentes. Cada combinación define un “port del sistema operativo”
- FreeRTOS es configurado a través del archivo FreeRTOSConfig.h

```
FreeRTOS
    └─Source
        ├─tasks.c          FreeRTOS source file - always required
        ├─list.c           FreeRTOS source file - always required
        ├─queue.c          FreeRTOS source file - nearly always required
        ├─timers.c         FreeRTOS source file - optional
        ├─event_groups.c   FreeRTOS source file - optional
        └─croutine.c       FreeRTOS source file - optional
```

Cap 1.2 Mastering_the_FreeRTOS_Real_Time_Kernel

FreeRTOS
Algunas convenciones

- Data Types

TickType_t	FreeRTOS configures a periodic interrupt called the tick interrupt.
BaseType_t	This is always defined as the most efficient data type for the architecture. Typically, this is a 32-bit type on a 32-bit architecture, a 16-bit type on a 16-bit architecture, and an 8-bit type on an 8-bit architecture.

- Function Names

  - v**Task**PrioritySet() returns a void and is defined within **task**.c.
  - x**Queue**Receive() returns a variable of type BaseType_t and is defined within **queue**.c.
  - pv**Timer**GetTimerID() returns a pointer to void and is defined within **timers**.c.

Cap 1.5 Mastering_the_FreeRTOS_Real_Time_Kernel

FreeRTOS
Recursos utilizados

- FreeRTOS usa:
  - SysTick, PendSV y SVC. Estas interrupción no deben ser utilizadas por la aplicación
- Memoria
  - El kernel compilado ocupa aproximadamente 6KB de memoria Flash y pocos cientos de bytes de RAM
  - Cada tarea usa RAM para alojar el stack

Cap 1.5 Mastering_the_FreeRTOS_Real_Time_Kernel

FreeRTOS - Task

- Task Manegment
  - Cómo implementar una tarea (task)
  - Cómo crear una o más instancias de una tarea
  - Cómo usar los parámetros de una tarea
  - Cómo cambiar la prioridad de una tarea ya creada
  - Cómo borrar una tarea
  - Cómo implementar un procesamiento periódico
  - Cómo usar la tarea Idle

Cap 3 Mastering_the_FreeRTOS_Real_Time_Kernel

FreeRTOS - Task

- En *FreeRTOS* la aplicación se estructura en un conjunto de n tareas autónomas entre si
- Cada tarea se ejecuta en su propio contexto (*stack*)
- El *Scheduler* es responsable de iniciar su ejecución, detener para pasar a ejecutar otra o terminar la ejecución

FreeRTOS – Task
Declaración de función

- Las tareas son funciones escritas en C
- El prototipo de la función debe devolver void y recibir como parámetros un puntero a void
- Generalmente las tareas tienen un loop infinito donde se ejecuta su código y nunca se sale de ese loop por lo que nunca debe retornar
- Si se necesita, la tarea puede ser borrada si terminó su procesamiento
- Una tarea puede ser definida para luego crear múltiples instancias de ella:
  - Cada instancia tiene su propia ejecución
  - Cada instancia tiene su propio stack
  - Cada instancia tiene su propia copia de variables automáticas

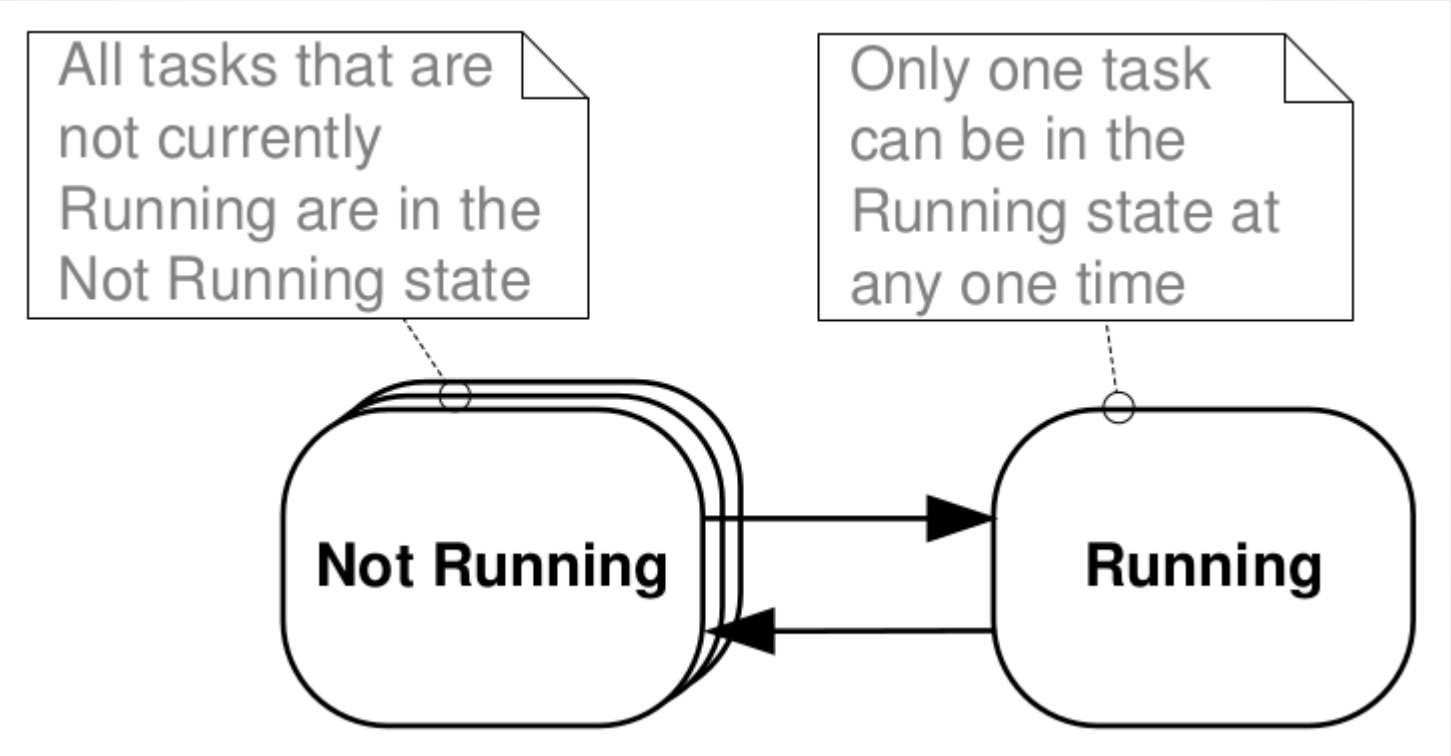
FreeRTOS – Task
Declaración de función

```
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 );
}
```

FreeRTOS – Task
Estados de una tarea

- La aplicación esta compuesta por varias tareas
- Solamente 1 tarea se ejecuta a la vez en el sistema (en un sistema simple core)
- Esto quiere decir que cada tarea puede estar en 2 posibles estados
  - Running
  - Not Running
- “switched in” o “swapped in”: cuando pasa de Not Running a Running
- “switched out” o “swapped out”: cuando pasa de Running a Not Running
- El Scheduler es el responsable de manejar el contexto:
  - Registros
  - Stack

FreeRTOS – Task
Estados de una tarea



FreeRTOS – Task
Crear una tarea

- Para crear una tarea se debe llamar a la función:

```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        uint16_t usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

- pPvTaskCode: Puntero a la función que implementa la tarea
- pcName: Nombre de la tarea
- usStackDepth: Tamaño de stack de la tarea especificado en words (ancho del stack del sistema)
- pvParameters: Parámetro que se le pasa a la tarea, por lo general es un puntero a una estructura

FreeRTOS – Task
Crear una tarea

- uxPriority: Es la prioridad de la tarea. 0 es la mínima prioridad, y la máxima es configMAX_PRIORITIES – 1
- pxCreatedTask: es utilizado para obtener un handler de la tarea, si no se desea utilizar se pasa NULL
- Retorna:
  - pdPASS: Si la tarea fue creada exitosamente
  - pdFAIL: Indica que la tarea no fue creada, por insuficiente memoria

FreeRTOS – Task
Crear una tarea

```
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. */
        }
    }
}
```

FreeRTOS – Task
Crear una tarea

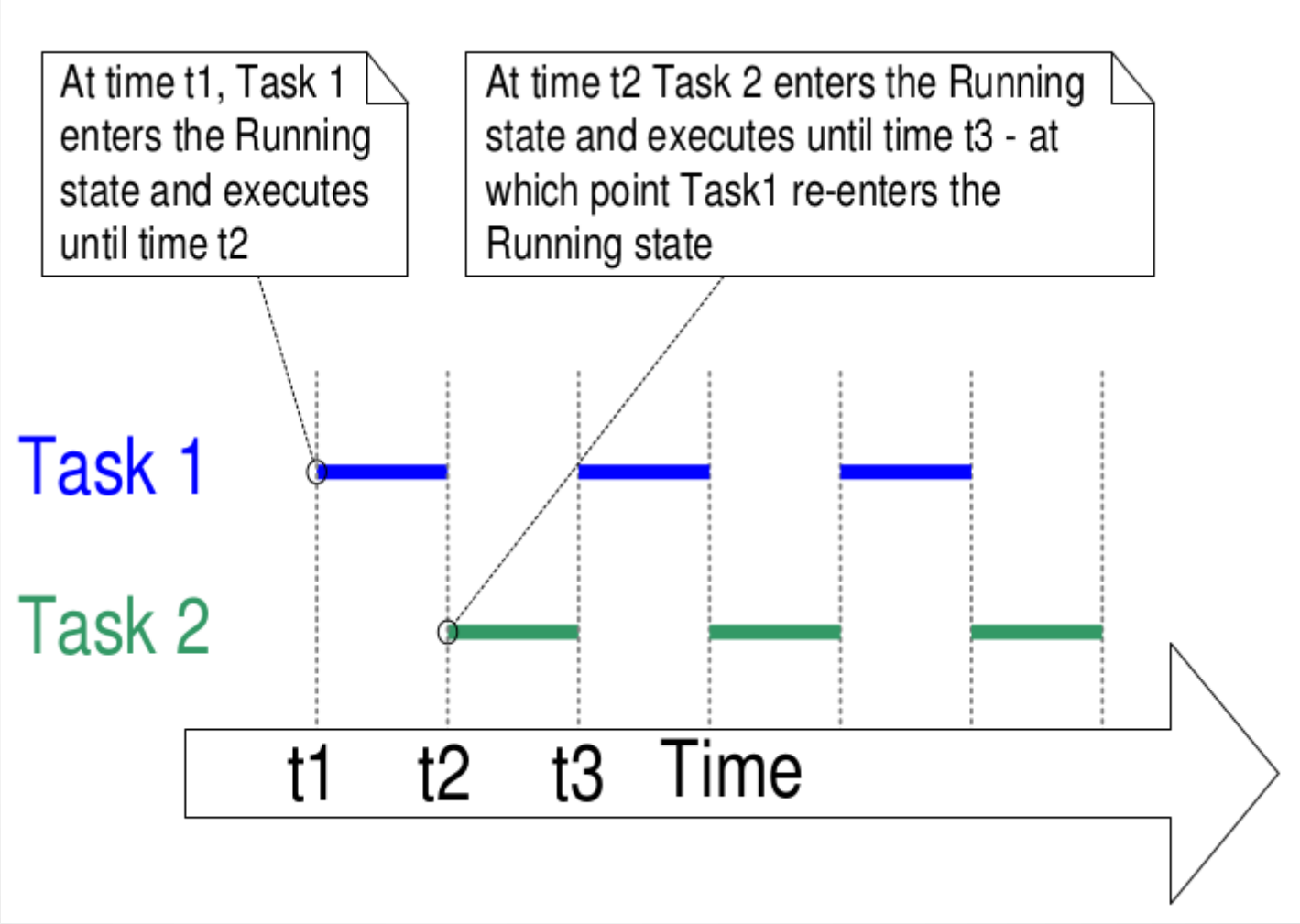
```
void vTask2( void *pvParameters )
{
const char *pcTaskName = "Task 2 is running\r\n";
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

    /* 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. */
        }
    }
}
```

FreeRTOS – Task
Crear una tarea

```
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( ;; );
}
```

FreeRTOS – Task
Crear una tarea

```
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
```



FreeRTOS – Task
Creando una tarea que recibe parámetros

```
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. */
        }
    }
}
```

FreeRTOS – Task
Creando una tarea que recibe parámetros

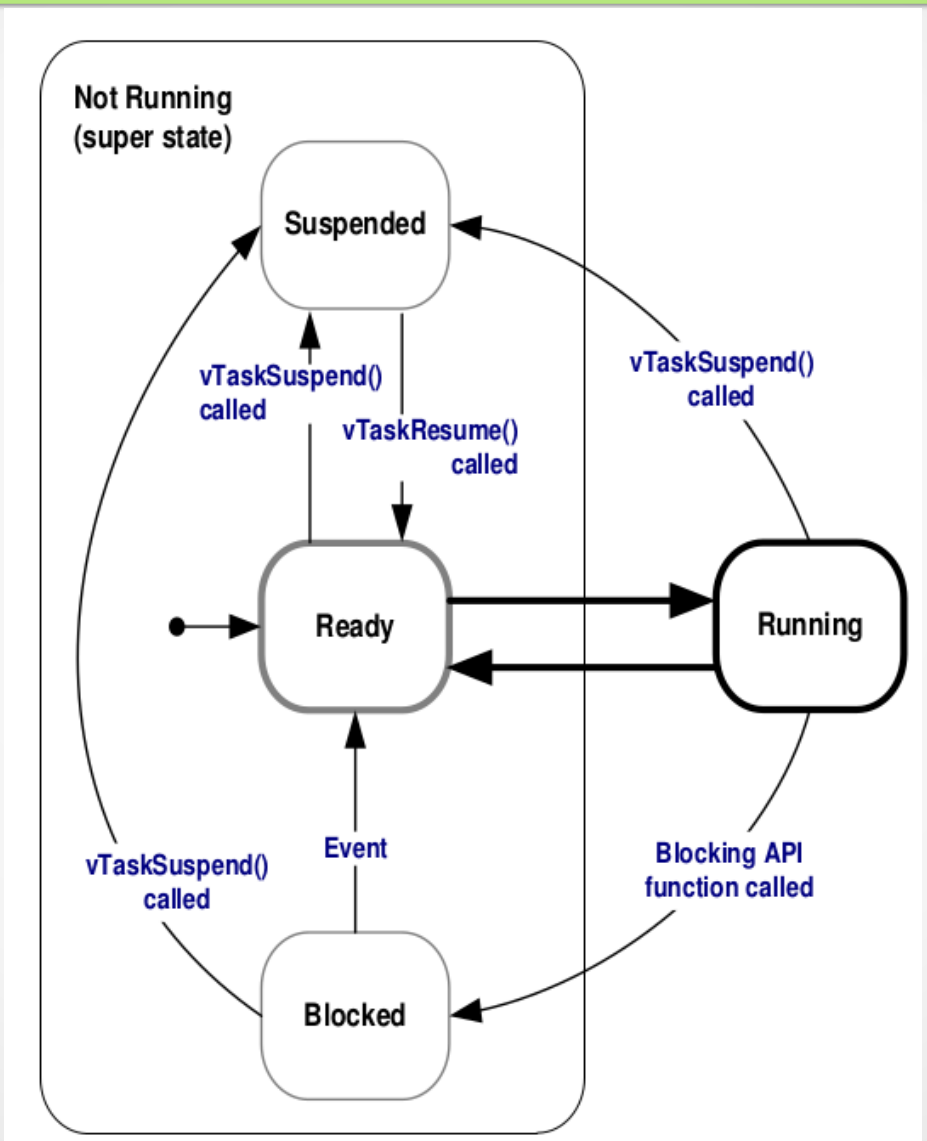
```
/* Define the strings that will be passed in as the task parameters.  These are
defined const and not on the stack to ensure they remain valid when the tasks are
executing. */
static const char *pcTextForTask1 = "Task 1 is running\r\n";
static const char *pcTextForTask2 = "Task 2 is running\r\n";

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. */

    /* 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( ;; );
}
```

“Expandiendo” el estado

Not Running

Cap 3 Mastering_the_FreeRTOS_Real_Time_Kernel

Estados de una tarea



Ejemplo: tarea en estado blocked por un delay

- El ejemplo anterior genera un delay mediante un loop
- Una manera eficiente de hacer que una tarea espere por una cantidad de tiempo es usar una de las funciones de la API de FreeRTOS:
  - vTaskDelay(): pone a la tarea en estado blocked por un número de Ticks
  - vTaskDelayUntil(): pone a la tarea en estado blocked por un número de Ticks, pero teniendo en cuenta el valor del Tick que se le pase como parámetros

Ejemplo: tarea en estado blocked por un delay

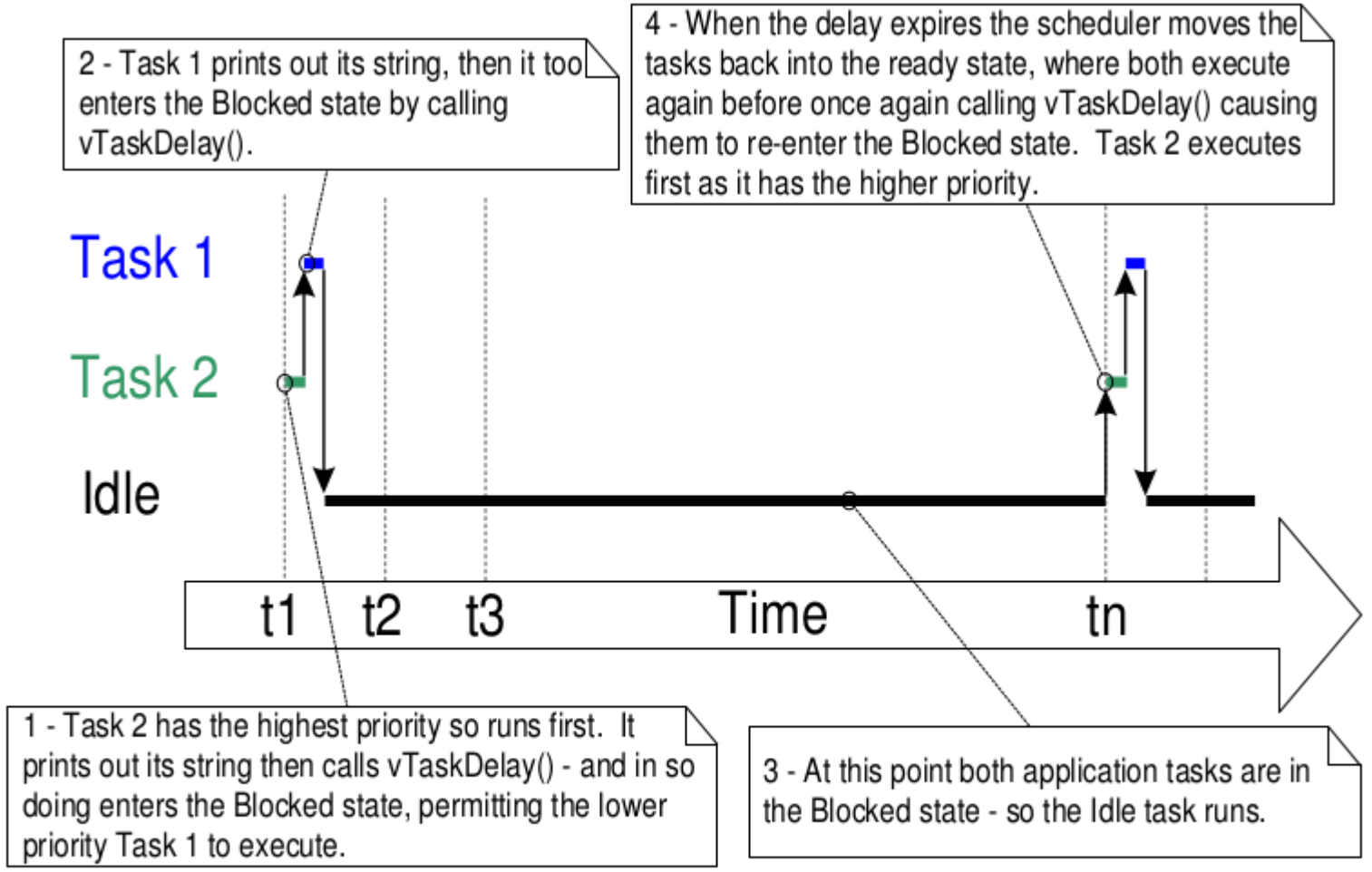
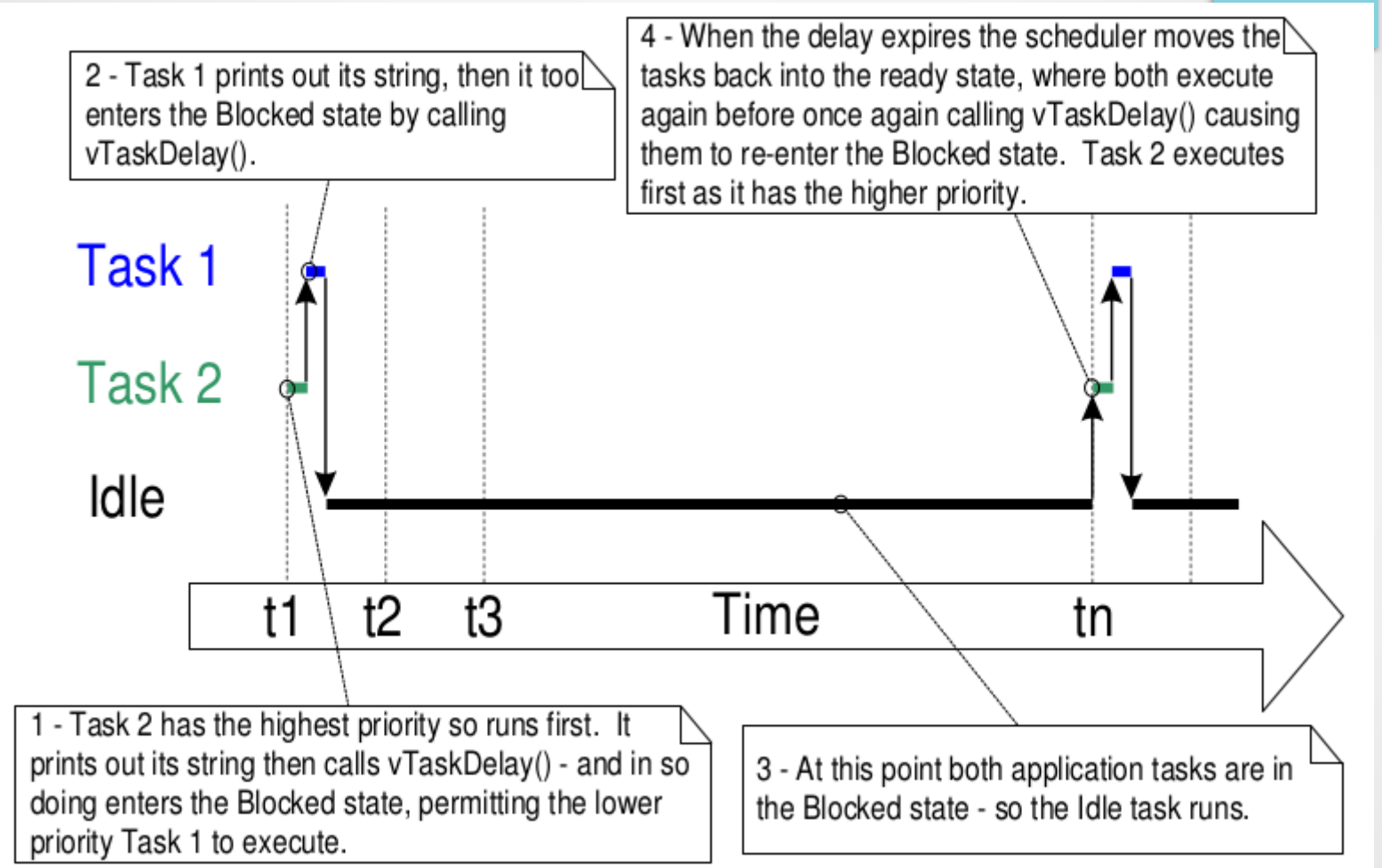


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Ejemplo: tarea en estado blocked por un delay

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

    /* 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.  This time a call to vTaskDelay() is used which places
        the task into the Blocked state until the delay period has expired.  The
        parameter takes a time specified in 'ticks', and the pdMS_TO_TICKS() macro
        is used (where the xDelay250ms constant is declared) to convert 250
        milliseconds into an equivalent time in ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

Ejemplo: tarea en estado blocked por un delay



Ejemplo: Tarea bloqueante
Tarea no bloqueante

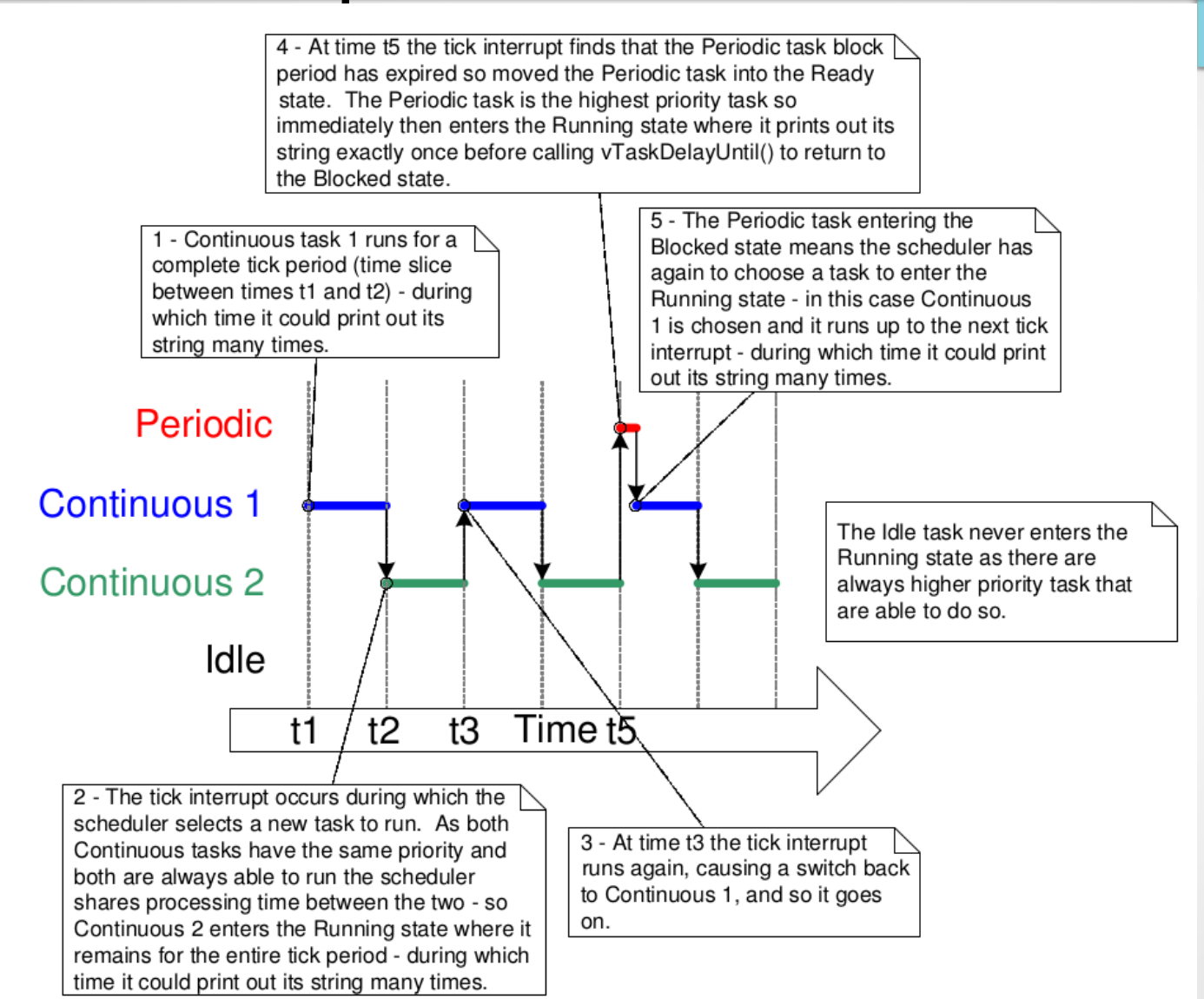
- 2 tareas bloqueantes
- 1 tarea de mayor prioridad que las dos anteriores

Ejemplo: Tarea bloqueante
Tarea no bloqueante

```
void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;

    /* 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.  This task just does this repeatedly
        without ever blocking or delaying. */
        vPrintString( pcTaskName );
    }
}
```

```
void vPeriodicTask( void *pvParameters )
{
TickType_t xLastWakeTime;
const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that this is the only time the variable is explicitly written to.
    After this xLastWakeTime is managed automatically by the vTaskDelayUntil()
    API function. */
    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
    for( ;; )
    {
        /* Print out the name of this task. */
        vPrintString( "Periodic task is running\r\n" );

        /* The task should execute every 3 milliseconds exactly - see the
        declaration of xDelay3ms in this function. */
        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
    }
}
```

Ejemplo: Tarea bloqueante
Tarea no bloqueante



Tarea IDLE

- La tarea IDLE se ejecuta cuando ninguna de las tareas del sistema esta en Ready
- Esta tarea se crea automáticamente por el scheduler
- La tarea IDLE tiene la menor prioridad (0)
- El código de la aplicación se coloca en la función void vApplicationIdleHook(void)
- Nunca se puede bloquear o suspender la tarea IDLE llamando a la API de FreeRTOS

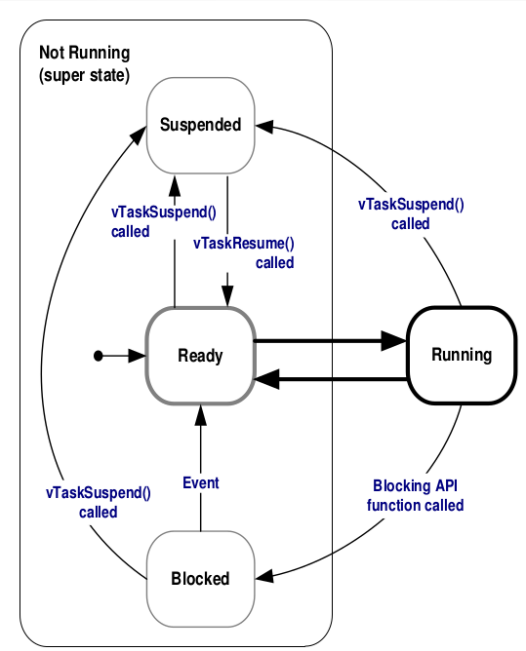
Cambiando la prioridad de una Tarea

- Al iniciar el sistema, las tareas tienen la prioridad que se les dio al momento de crearlas
- La prioridad de una tarea puede ser consultada mediante la función: uxTaskPriorityGet()
- La prioridad de una tarea puede ser cambiada mediante la función: uxTaskPrioritySet()
  - Ver Ejemplo 8

Algoritmos de Scheduling

Recapitulando estados de una Tarea

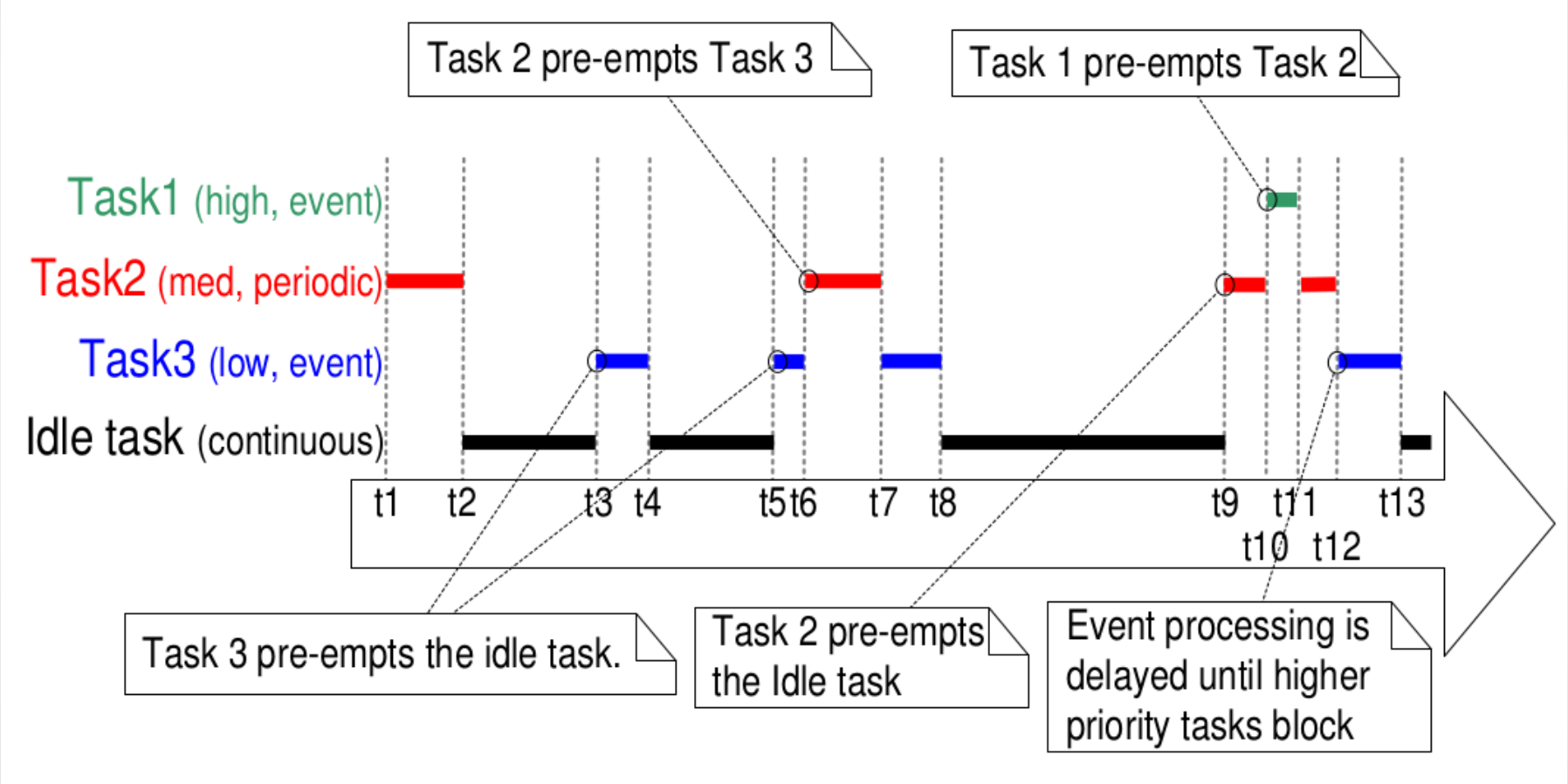
- En un sistema de un solo core, un única tarea puede estar en estado Running, el resto de las tareas se encuentra en estado Blocked, Suspended o Ready. Las que se encuentran en Ready, pueden ser seleccionadas por el Scheduler para pasar a estado Running.



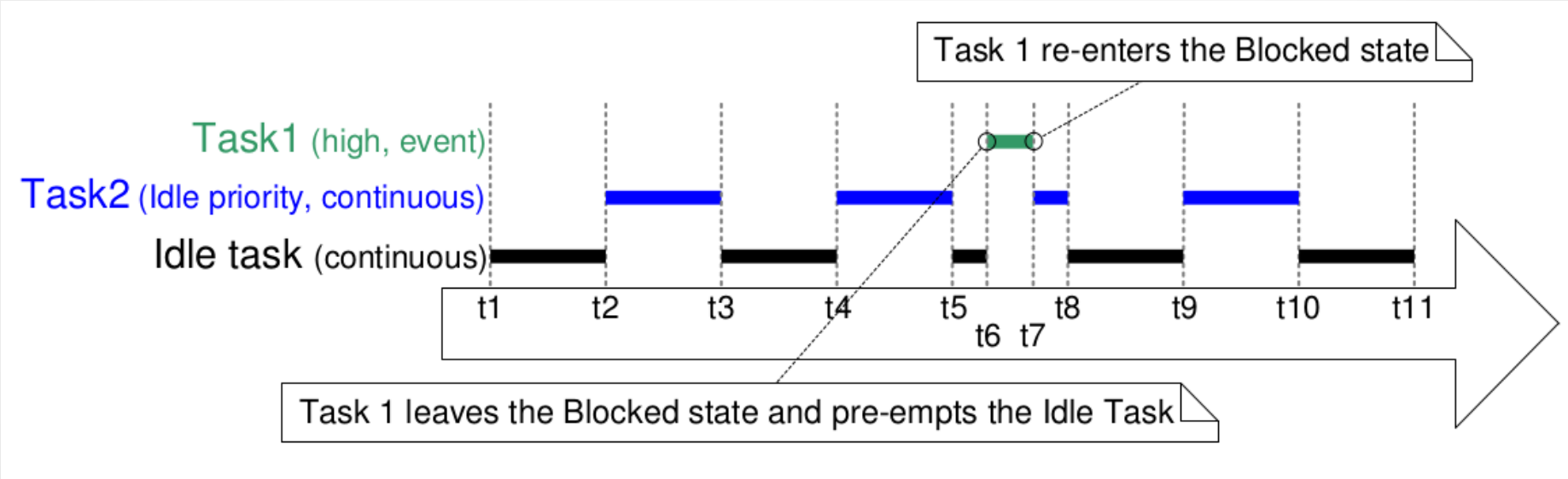
Algoritmo de Scheduling

- FreeRTOS implementa el algoritmo que comunmente se lo conoce como: Round Robin Scheduling
- Configuración:
  - configUSE_PREEMPTION: Si se configura esta constante en 1, si una tarea de mayor prioridad a la que se está ejecutando, esta pasará a estado Running y la otra a Ready
  - configUSE_TIME_SLICING: Si se configura esta constante en 1, tareas de igual el procesador se “repartirá” entre tareas de igual prioridad que se encuentren en estado Ready

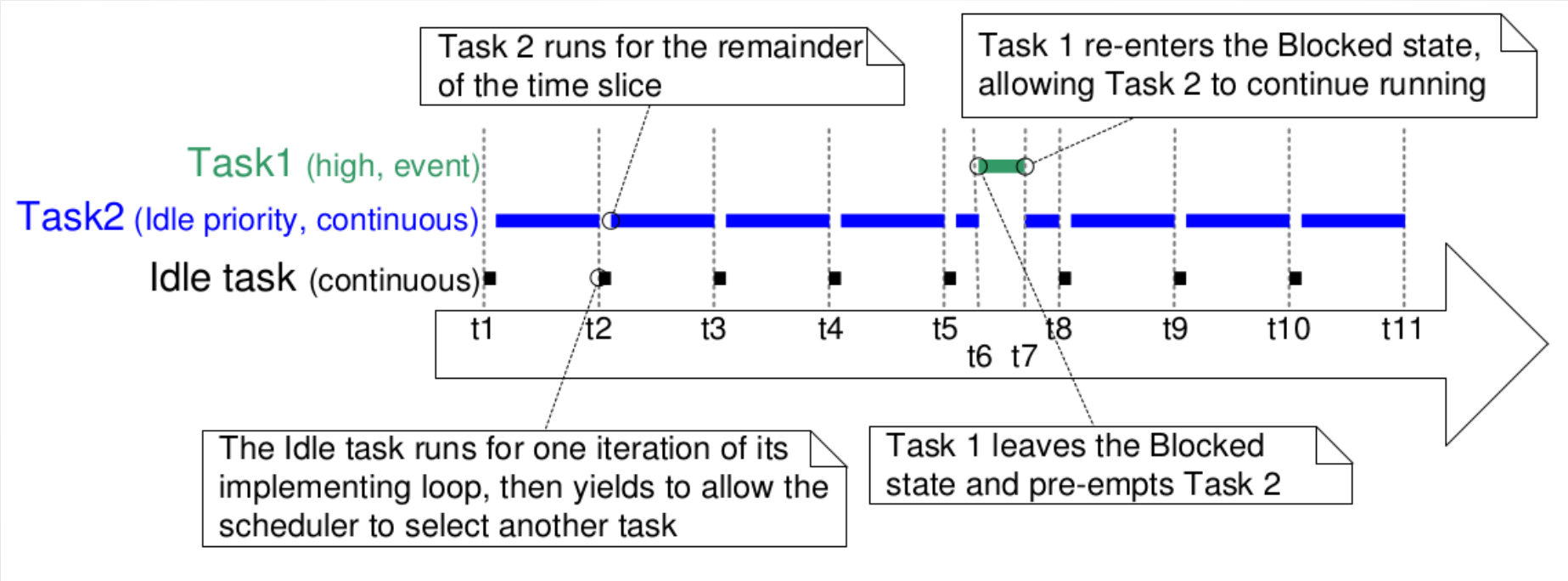
Ejemplo configUSE_PREEMPTION = 1



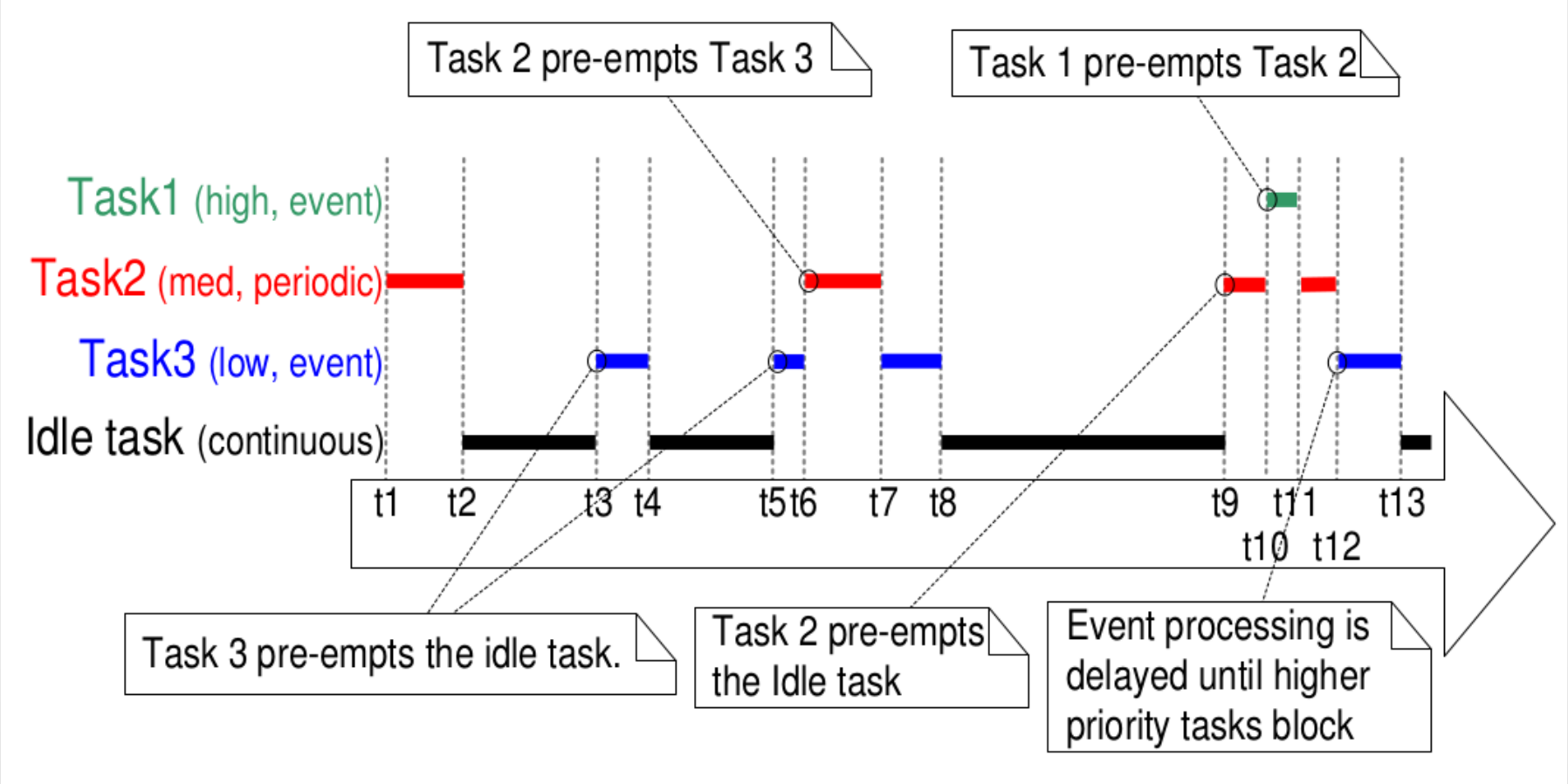
Ejemplo configUSE_TIME_SLICING = 1



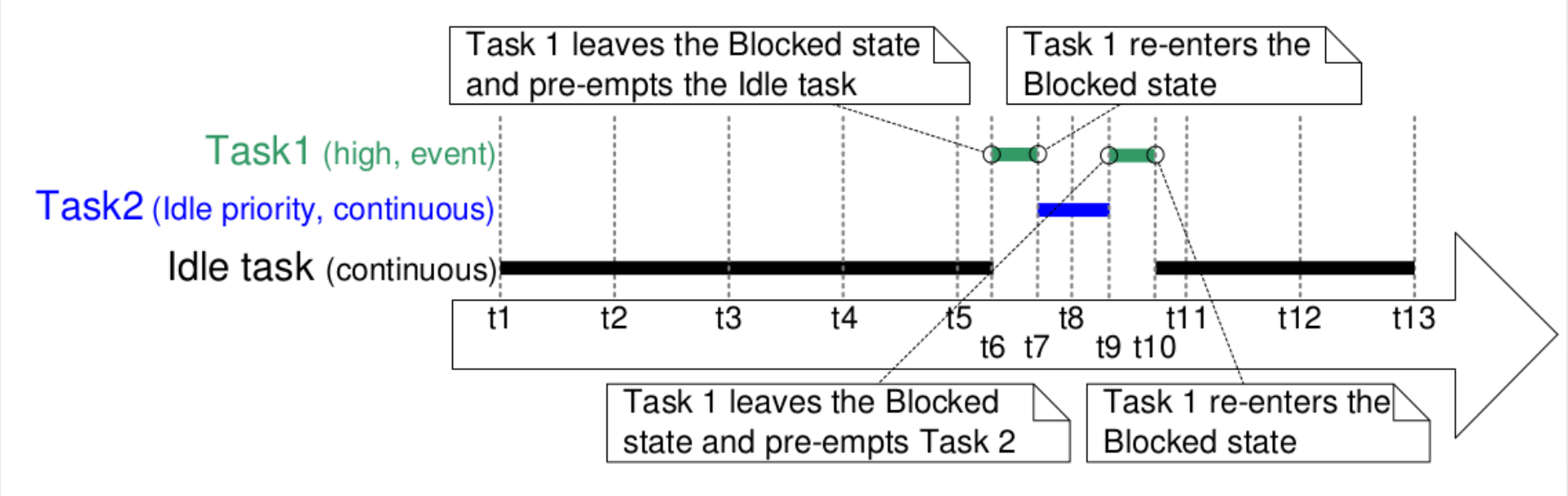
Ejemplo configIDLE_SHOULD_YIELD = 1



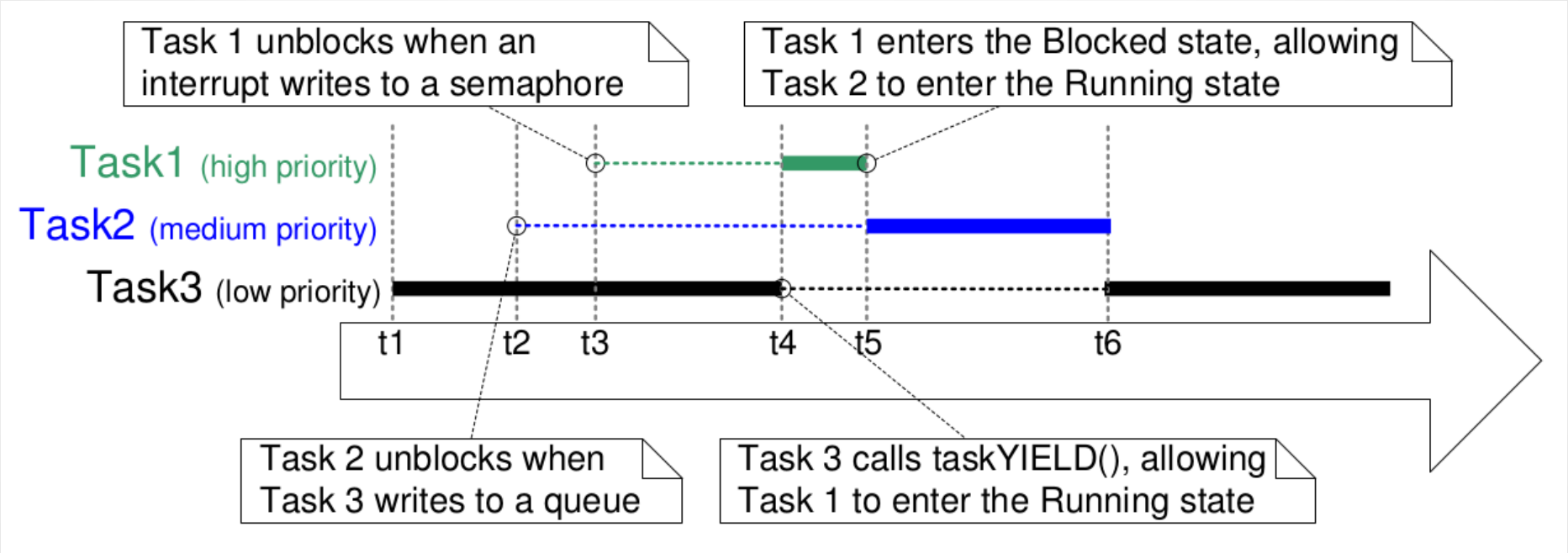
Ejemplo configUSE_PREEMPTION = 1



configUSE_PREEMPTION = 1
configUSE_TIME_SLICING = 0



configUSE_PREEMPTION = 0
configUSE_TIME_SLICING = X



Manejo de Memoria

Cap 2 Mastering_the_FreeRTOS_Real_Time_Kernel

Manejo de memoria

- Nota: A partir de la versión 9.0.0, FreeRTOS los objetos del kernel pueden ser ubicados estáticamente en tiempo de compilación o dinámicamente en tiempo de ejecución
- Para abordar un uso más simple de la API no usaremos ubicación estática.
- Cada vez que se crea un objeto nuevo (task, queue, mutex, etc) se le asigna memoria dinámicamente. Y cada vez que se elimina, se libera esa memoria.
- La asignación dinámica de memoria que por lo general proveen los compiladores no son adecuadas para sistemas de tiempo real.

Manejo de memoria

- Las funciones malloc y free provistas por el compilador por lo general no son apropiadas por las siguientes razones:
  - En sistemas embebidos pequeños, no está disponible
  - Algunas implementación son complejas y ocupan mucho código
  - Rara vez funcionan bien en un sistema con tareas
  - Su tiempo de ejecución no es determinístico
  - Pueden sufrir fragmentación
  - Puede que requiera una configuración extra en el Linker
  - No se provee el código fuente

Manejo de memoria

- IMPORTANTE: de ahora en más, la aplicación deberá llamar a las funciones:
  - pvPortMalloc()
  - vPortFree()

Manejo de memoria

- Heap_1: Es la implementación más simple (no permite hacer free)
- Heap_2: Se sigue distribuyendo en las versiones de FreeRTOS para mantener retrocompatibilidad, pero no se recomienda su uso
- Heap_3: usa las funciones malloc() y free() de la biblioteca standard
- Heap_4: Usa un algoritmo para asignar el bloque de memoria a la porción que mejor ajusta en tamaño. Permite combinar bloques de memoria libre adyacentes
- Heap_5: Es igual al 4, pero admite múltiples espacios de memoria separados.

Queue Management

Queue

Temas:

- Cómo crear una Queue
- Cómo maneja los datos la Queue
- Cómo enviar datos a una Queue
- Cómo recibir datos de una Queue
- Bloquear al escribir o leer una Queue

Características de una Queue

- Una Queue se utiliza para intercambiar datos entre distintas tareas
- La Queue almacena una cantidad finita de elementos de un determinado tamaño (todos iguales)
- Funcionamiento: FIFO
- Al escribir en una Queue, se copian los bytes en la Queue
- Al leer la Queue, se remueven esos datos
- Varias tareas pueden Leer o escribir en la misma Queue

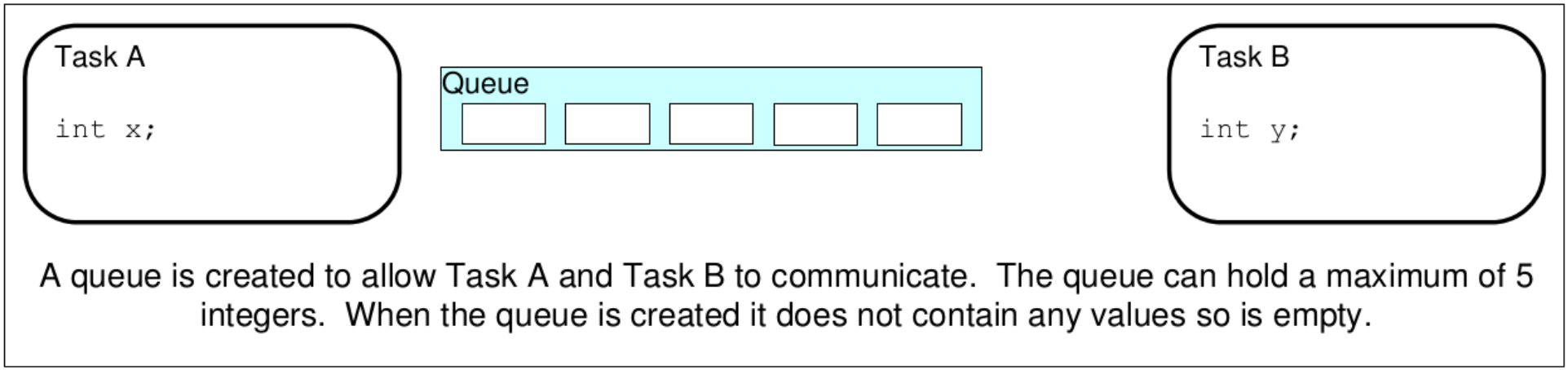
Bloquear al leer una Queue

- Si una Queue está vacía y una tarea intenta leerla, se bloquea hasta que otro coloque un dato en la Queue
  - Cuando una tarea coloque un dato en la Queue, la tarea que estaba bloqueada, pasa a estado Ready
  - Si más de una tarea estaba bloqueada, pasa a estado Ready la de mayor prioridad, si son de igual prioridad, pasa a Ready la que lleva más tiempo esperando
  - Se puede especificar un tiempo máximo de espera por datos en la Queue, si transcurre ese tiempo la tarea pasa a estado Ready

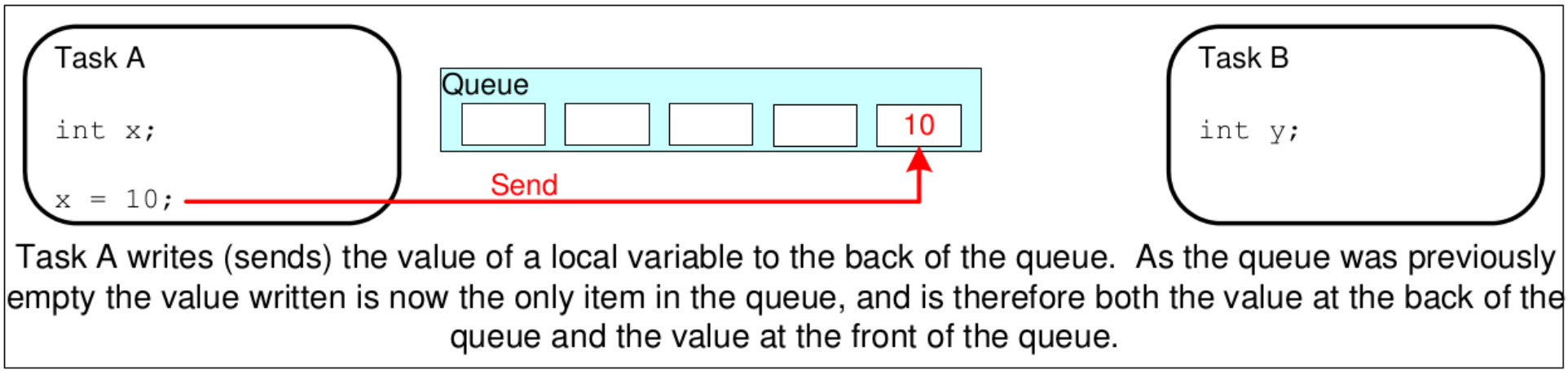
Bloquear al escribir una Queue

- Si una Queue llena y una tarea intenta escribir, se bloquea hasta que otro lea un dato de la Queue
  - Cuando una tarea lee un dato de la Queue, la tarea que estaba bloqueada, pasa a estado Ready
  - Si más de una tarea estaba bloqueada, pasa a estado Ready la de mayor prioridad, si son de igual prioridad, pasa a Ready la que lleva más tiempo esperando
  - Se puede especificar un tiempo máximo de espera para poner datos en la Queue, si transcurre ese tiempo la tarea pasa a estado Ready

Queue - Funcionamiento

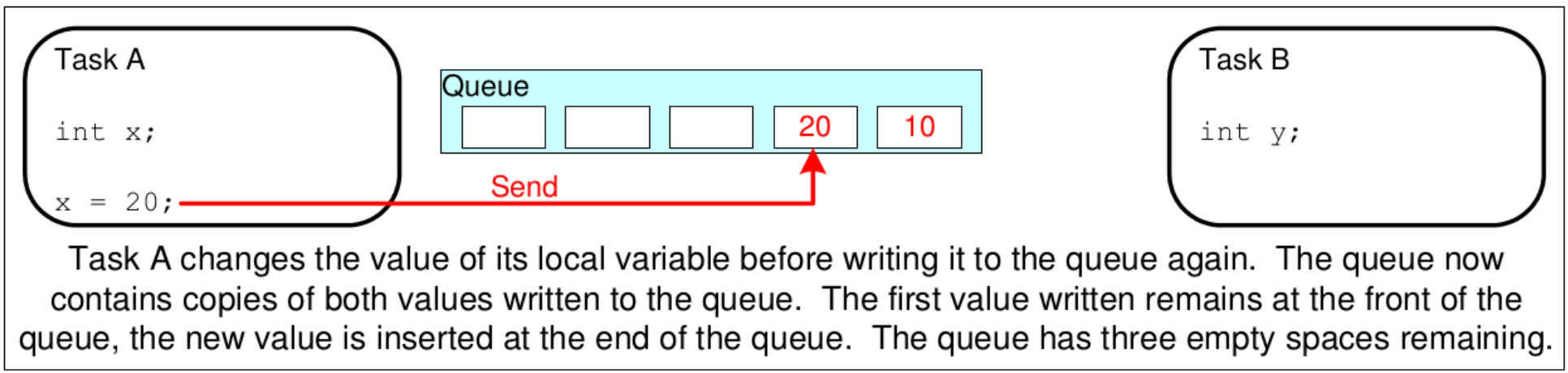


A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.

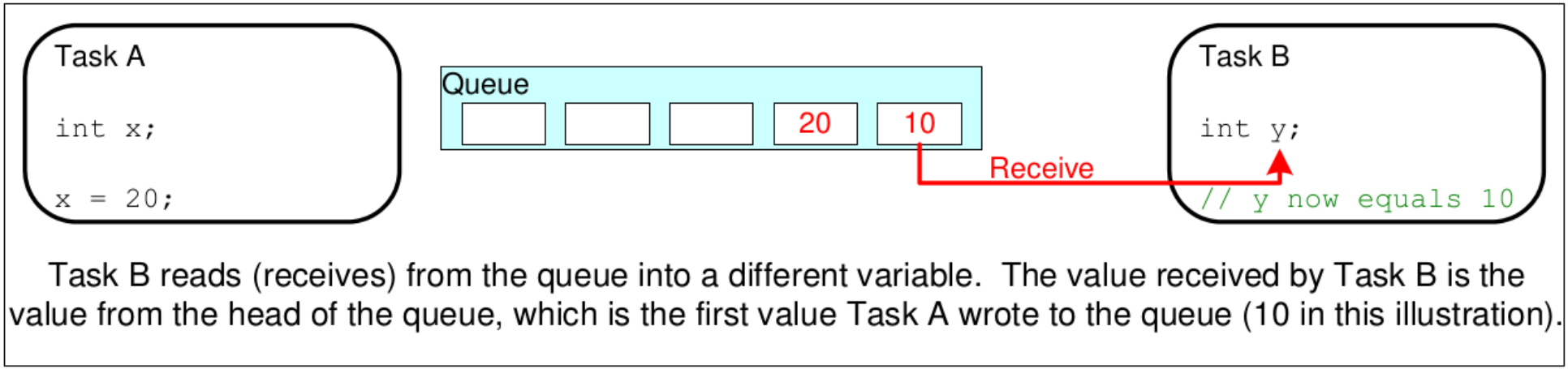


Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.

Queue - Funcionamiento

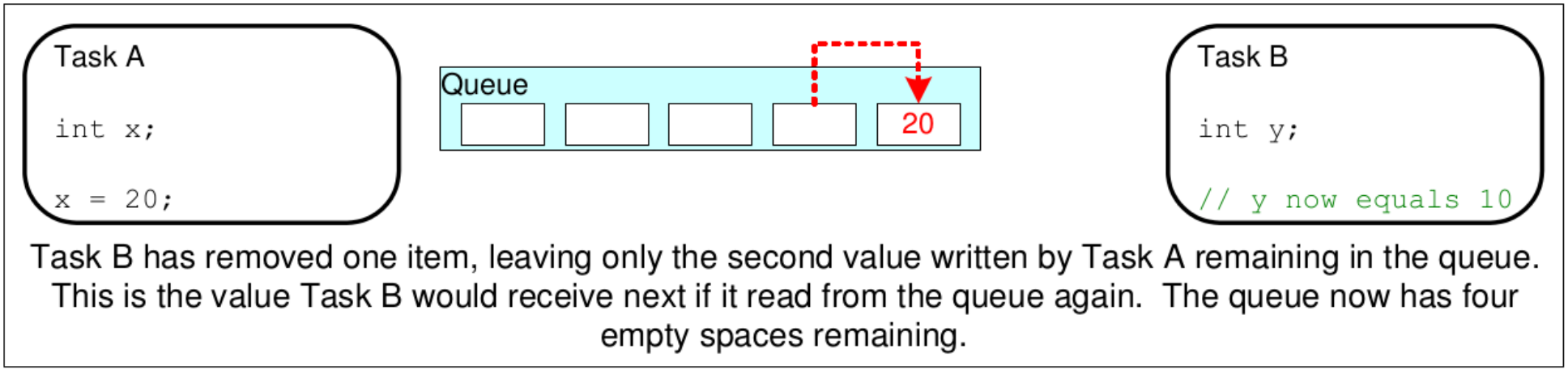


Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).

Queue - Funcionamiento



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Ver ejemplo 10

Manejo de Interrupciones

Cap 6 Mastering_the_FreeRTOS_Real_Time_Kernel

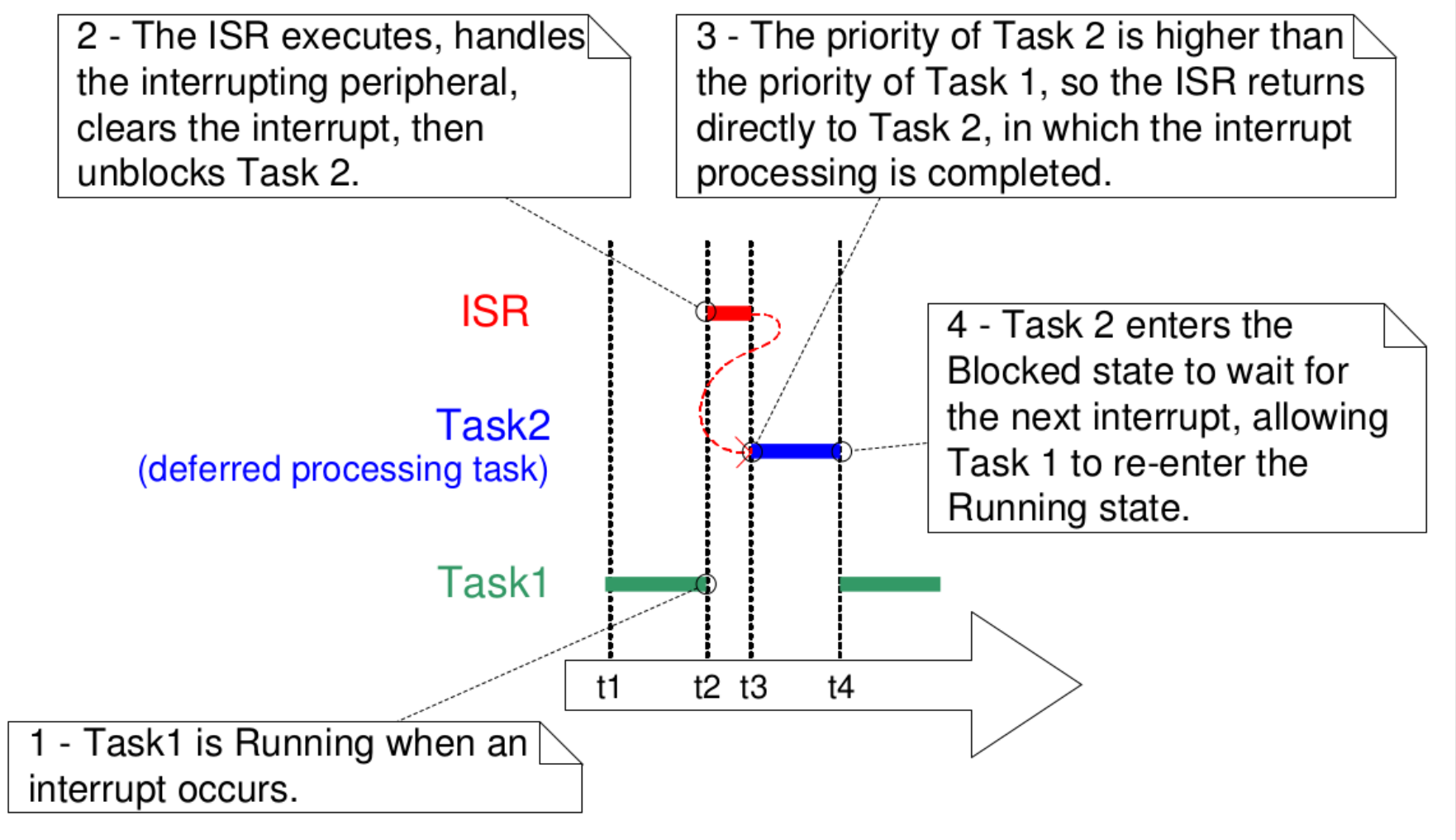
Interrupciones

- Temas
  - Qué funciones de la API pueden ser usadas desde una ISR
  - Procesamiento “posterior” de datos de una ISR
  - Cómo usar semáforos en una ISR
  - Diferencias entre semáforos binarios y contadores
  - Cómo pasar datos usando una Queue

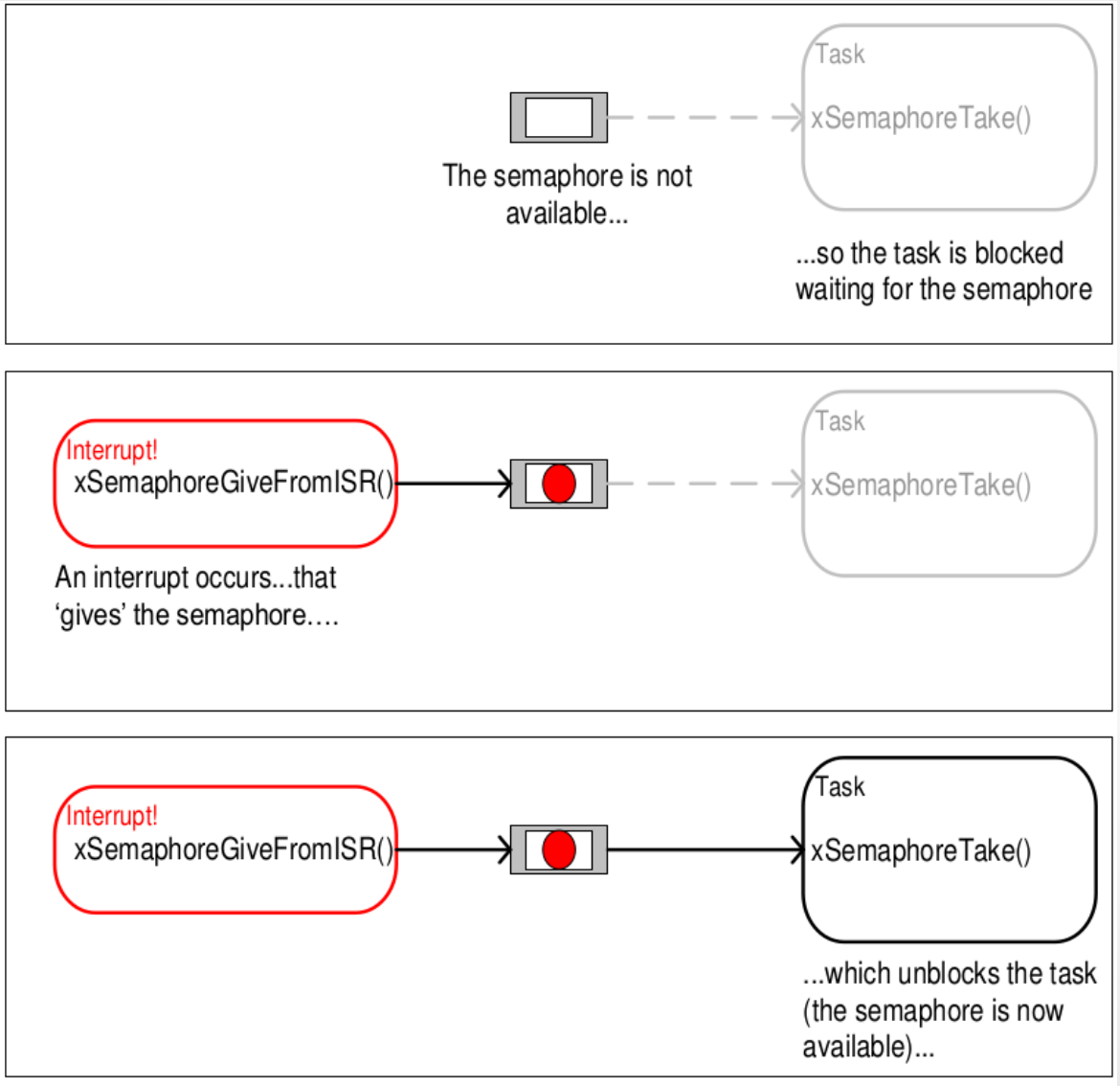
Interrupciones
Procesamiento de datos

- Sincronización mediante semáforo Binario
  - Se puede utilizar un semáforo binario para “desbloquear” una tarea que debe procesar datos luego de que se da un evento
  - La mayor parte del procesamiento que se hacía antes en una ISR, ahora se realiza en una tarea
  - Solamente una porción de código muy rápido se coloca en las ISR
  - Por eso se dice que el procesamiento es “deferred” (diferido) en una tarea “handler”
  - La tarea “handler” entra en estado Blocking al hacer un “take” de un semáforo quedando a la espera de que ocurra el evento
  - Cuando ocurre el evento, la ISR hace un give sobre el semáforo para que la tarea pase a estado Ready

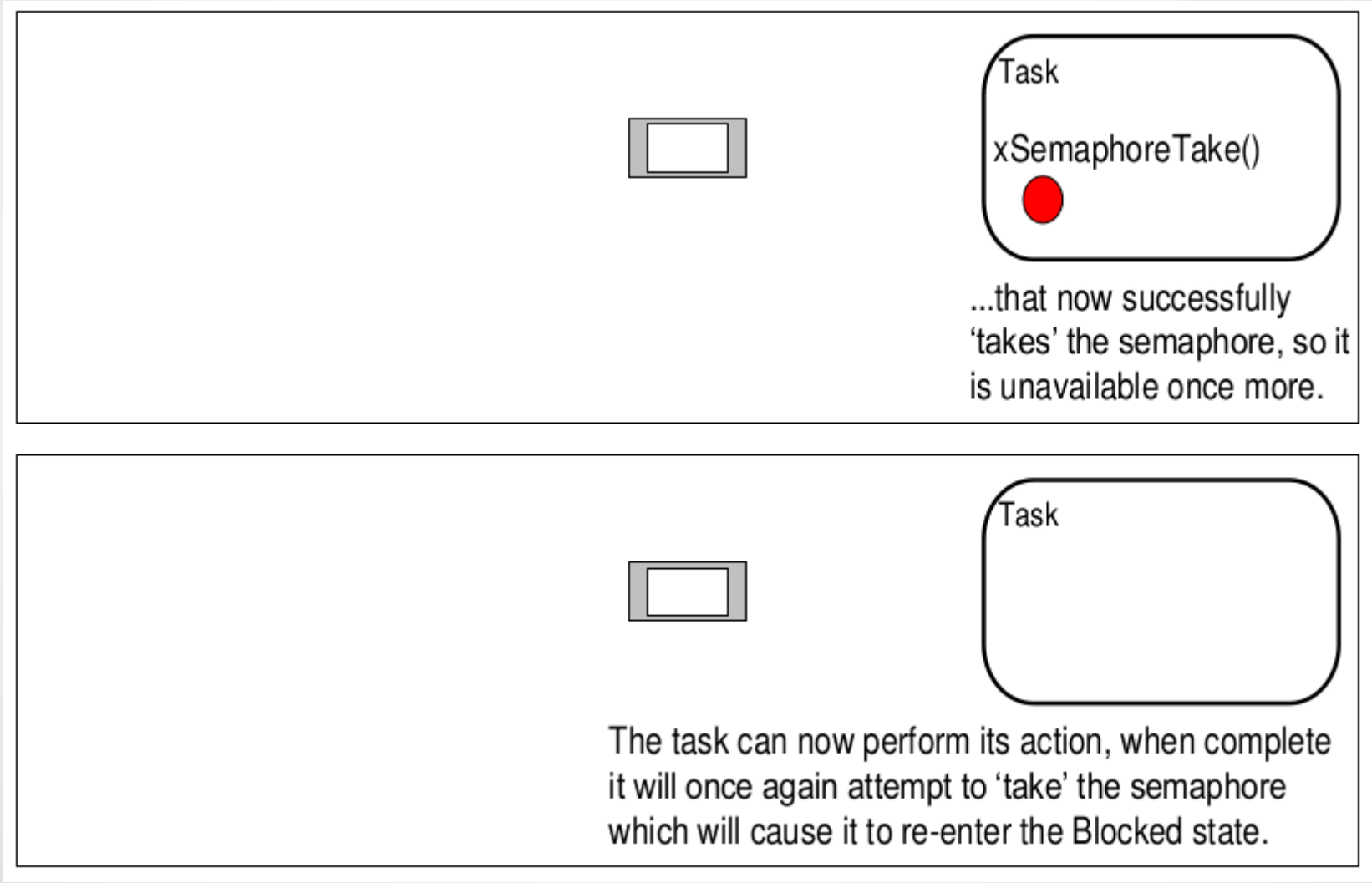
Interrupciones
Procesamiento de datos



Interrupciones
Procesamiento de datos



Interrupciones
Procesamiento de datos



Manejo de Recursos

Cap 7 Mastering_the_FreeRTOS_Real_Time_Kernel

Manejo de Recursos

- Un recurso puede ser compartido entre distintas tareas o entre tareas e interrupción. Por lo que es necesario generar una “exclusión mutua” para mantener la consistencia de los datos (o el funcionamiento correcto de un dispositivo)
- Temas:
  - Cuándo y por qué es necesario manejar y controlar un recurso
  - Qué es una sección crítica
  - Qué significa exclusión mutua
  - Qué significa suspender el Scheduler
  - Cómo usar un Mutex
  - Cómo crear y usar una tarea gatekeeper
  - Qué es la inversión de prioridades y cómo la herencia de prioridad puede mejorar (no eliminar) su impacto

Secciones críticas básicas

- Una sección crítica es una región de código rodeada por dos macros taskENTER_CRITICAL() y taskEXIT_CRITICAL()

```
/* The C code being compiled. */
PORTA |= 0x01;

/* The assembly code produced when the C code is compiled. */
LOAD    R1,[#PORTA] ; Read a value from PORTA into R1
MOVE    R2,#0x01    ; Move the absolute constant 1 into R2
OR      R1,R2       ; Bitwise OR R1 (PORTA) with R2 (constant 1)
STORE   R1,[#PORTA] ; Store the new value back to PORTA
```

```
/* Ensure access to the PORTA register cannot be interrupted by placing it within a
critical section.  Enter the critical section. */
taskENTER_CRITICAL();

/* A switch to another task cannot occur between the call to taskENTER_CRITICAL() and
the call to taskEXIT_CRITICAL().  Interrupts may still execute on FreeRTOS ports that
allow interrupt nesting, but only interrupts whose logical priority is above the
value assigned to the configMAX_SYSCALL_INTERRUPT_PRIORITY constant – and those
interrupts are not permitted to call FreeRTOS API functions. */
PORTA |= 0x01;

/* Access to PORTA has finished, so it is safe to exit the critical section. */
taskEXIT_CRITICAL();
```

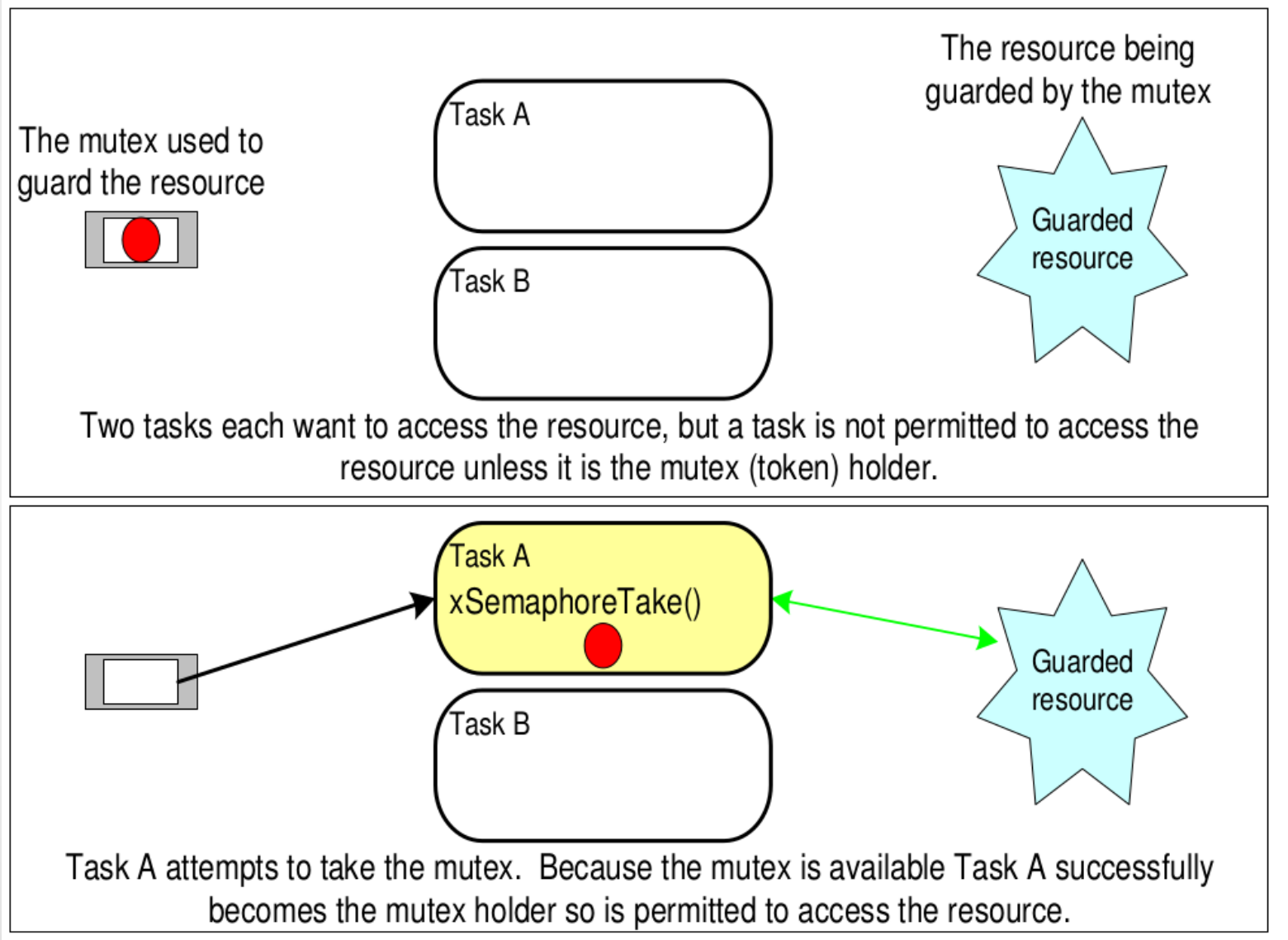
Suspendiendo el Scheduler

- También se puede crear una sección crítica suspendiendo el Scheduler
- Dependiendo de lo que interese suspender y de la duración se puede suspender solo el scheduler o suspender las interrupciones
- Las funciones son:
  - vtaskSuspendAll()
  - xTaskResumeAll()

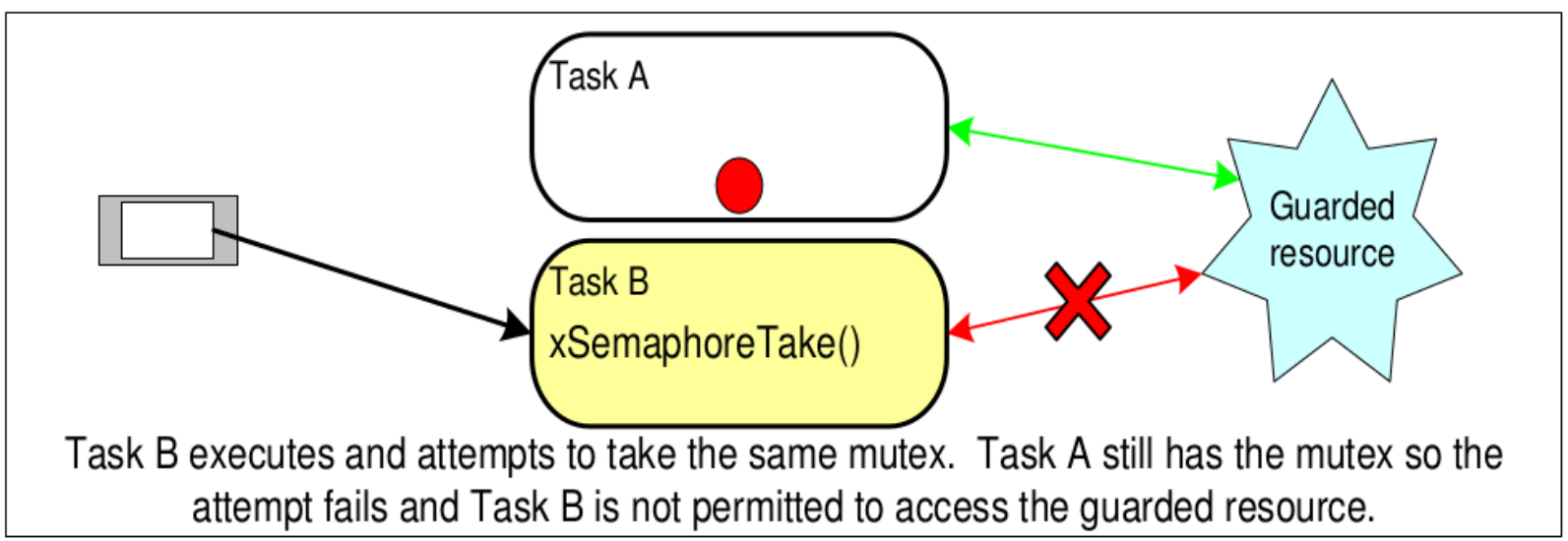
Mutex

- El Mutex es un tipo de semáforo binario especial que se usa para controlar el acceso a un recurso por dos o más tareas
- El mutex puede pensarse como un token asociado al recurso compartido
- Para poder acceder al recurso, primero se debe tomar “take” el token (mutex)
- Una vez que se realizar el take, luego se debe hacer un give para que otro pueda tomarlo

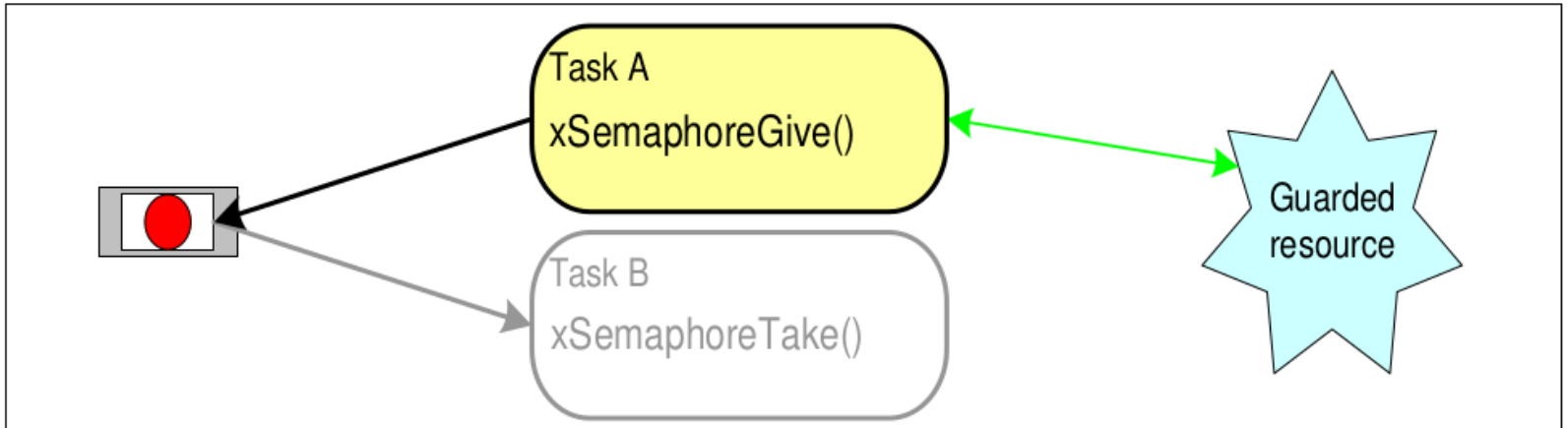
Exclusión mutua - Mutex



Exclusión mutua - Mutex

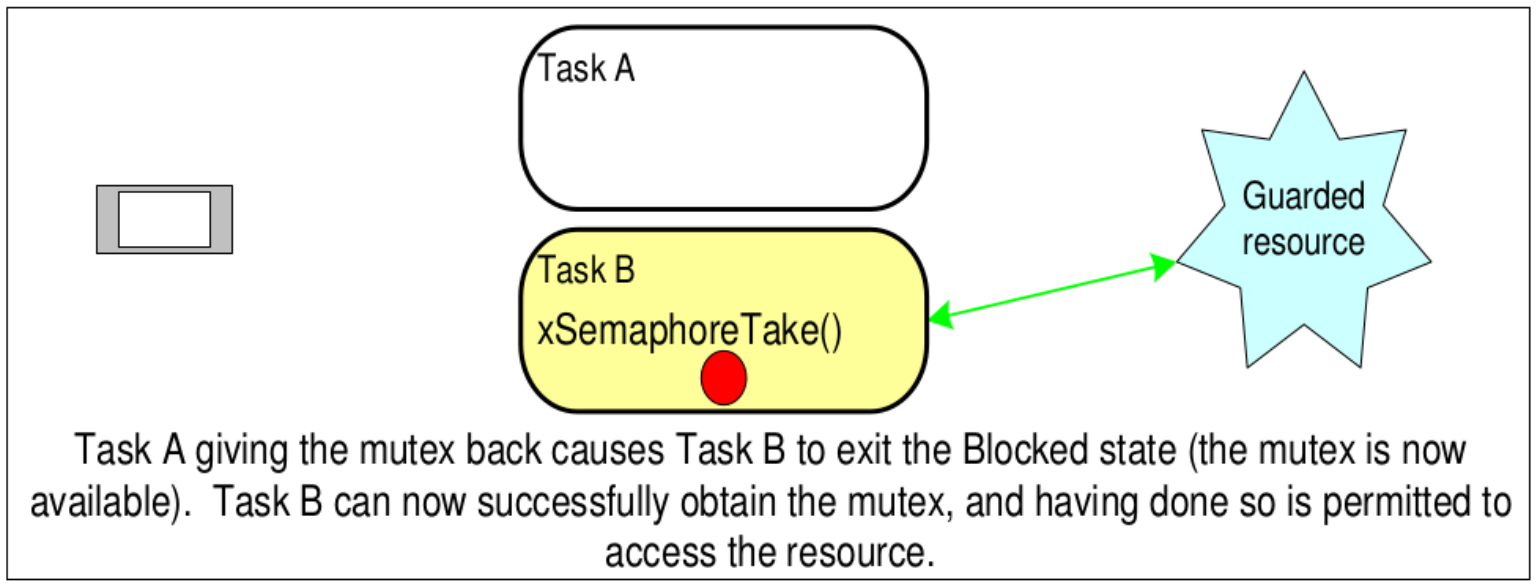


Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.

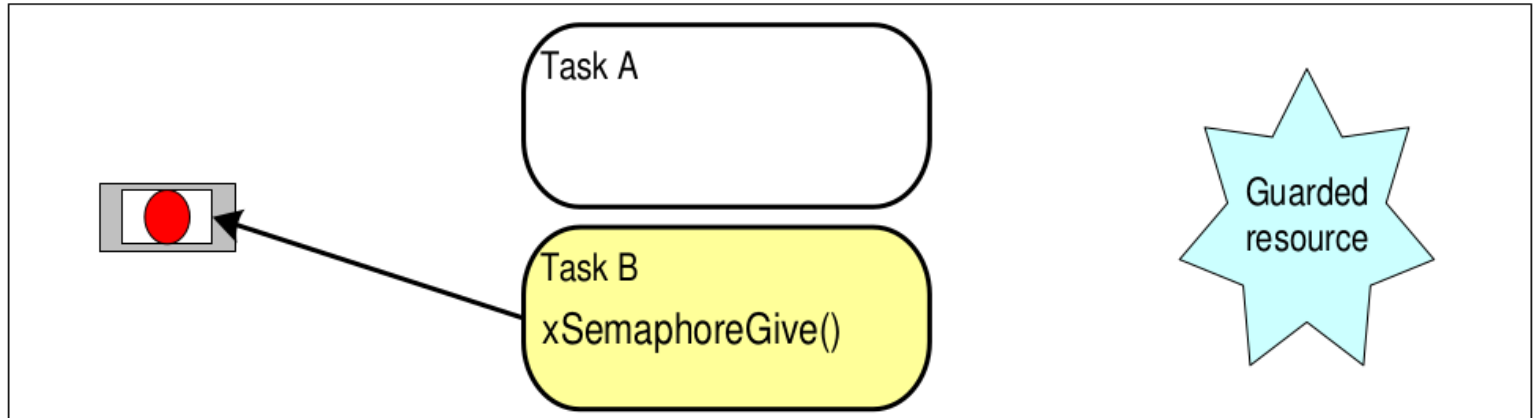


Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.

Exclusión mutua - Mutex

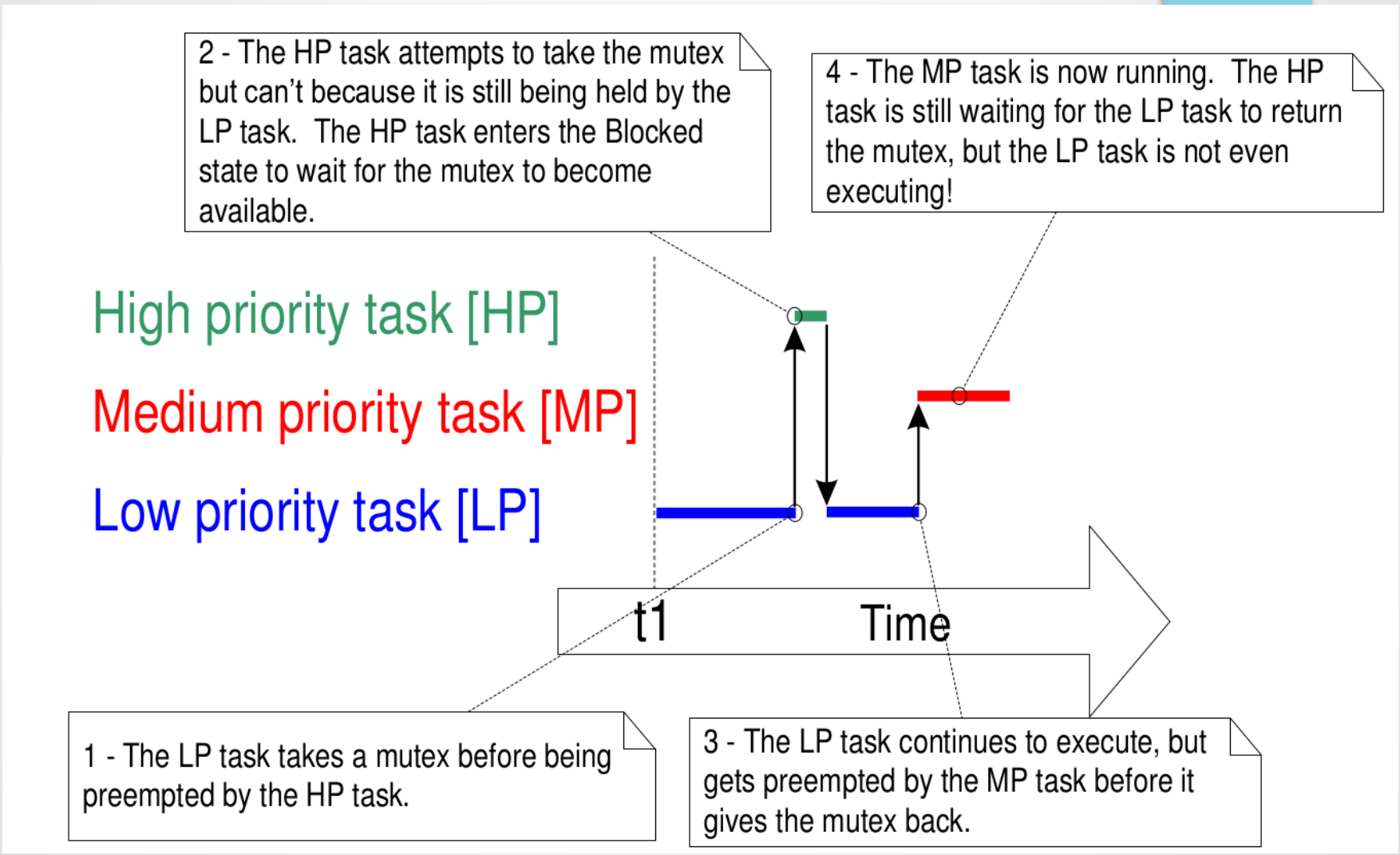


Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.

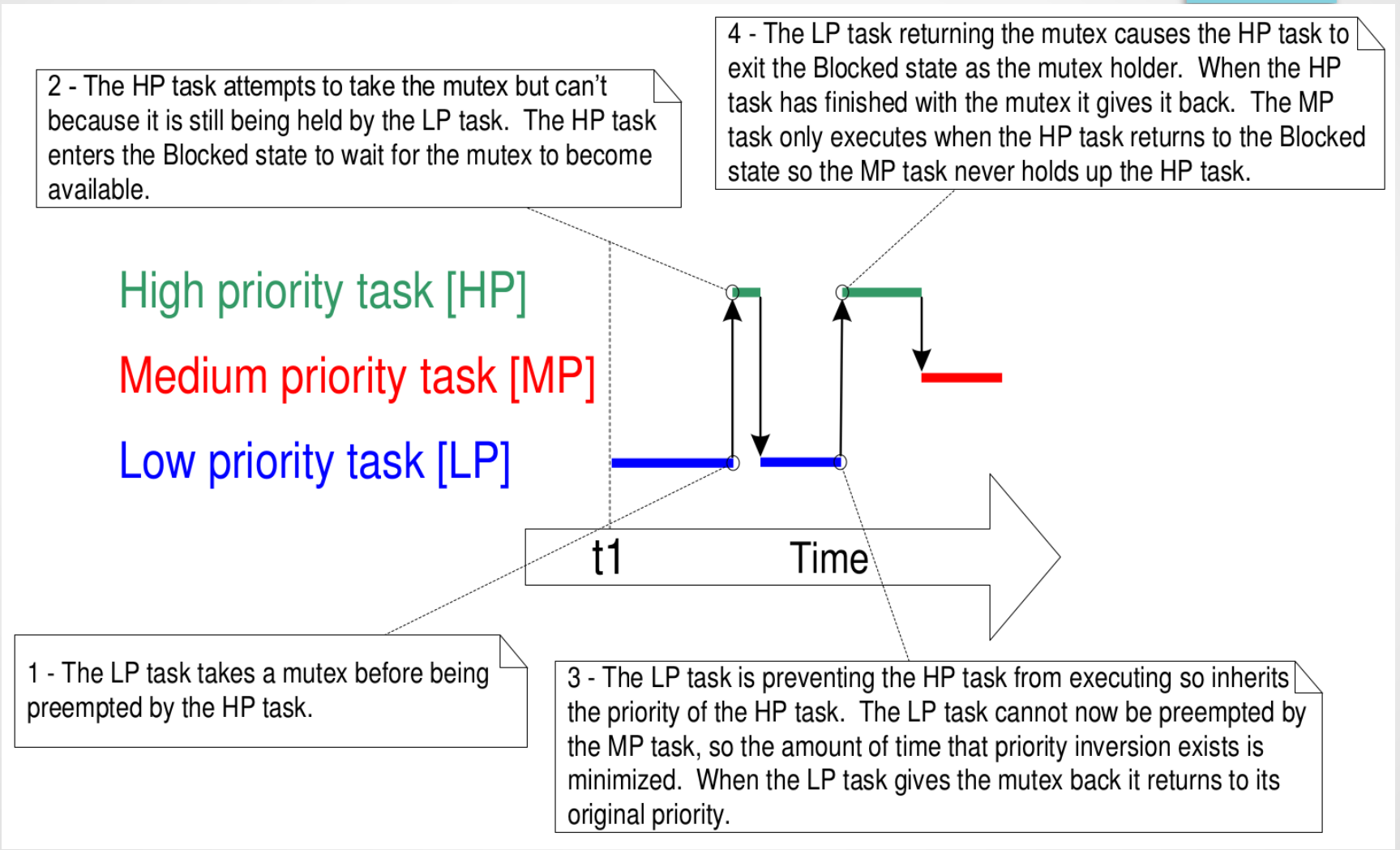


When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

Inversión de prioridad
Herencia de prioridad



Inversión de prioridad
Herencia de prioridad



¿Preguntas?