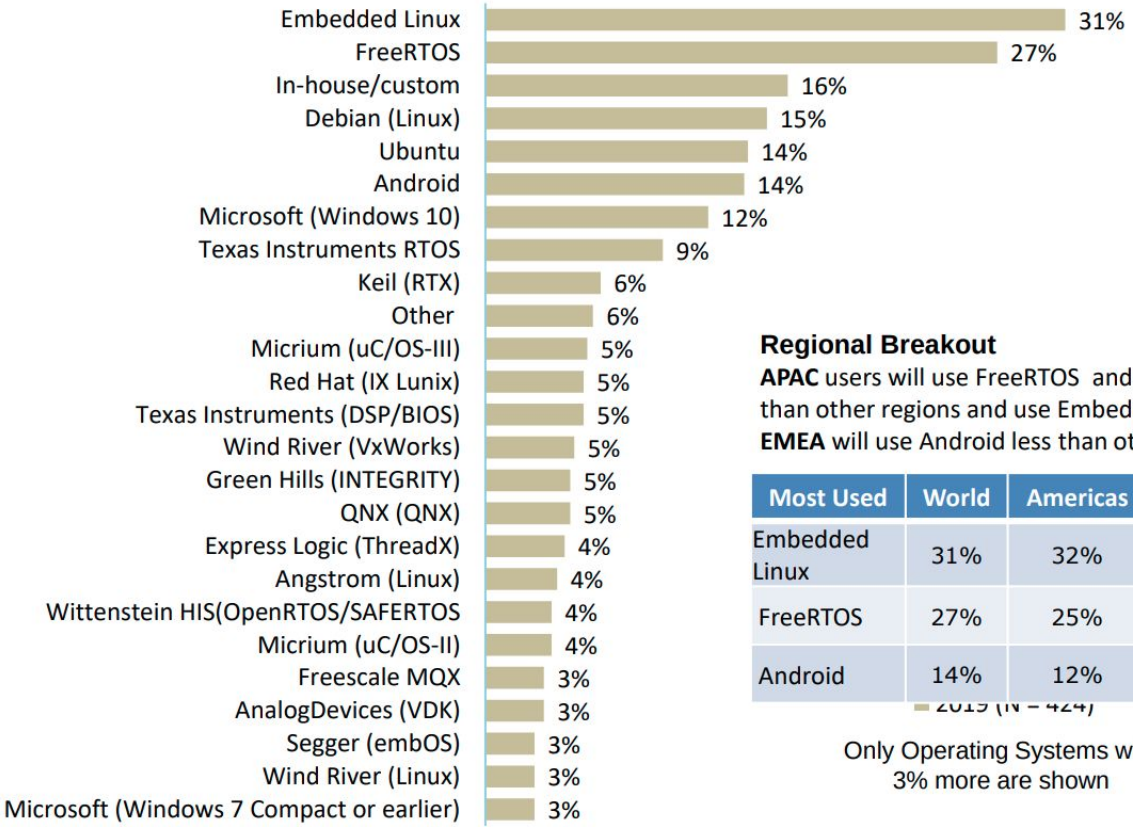
Sistemas Embebidos

Unidad 4

RTOS (Real Time Operating System)

- Hay **deadlines** (tiempos límite) para todas o algunas tareas.
- **Soft** RTOS: Si no se cumplen los deadlines, la **performance se degrada**, pero el sistema no se vuelve inutil o provoca una falla completa del sistema.
  - Ejemplos:
    - Sistema que debe medir variables ambientales a intervalos regulares de tiempo con timestamps.
    - Sistema que debe monitorear eventos, agregando timestamps.
- **Hard** RTOS: Si no se cumplen los deadlines, el sistema falla por completo.
  - Ejemplos:
    - Sistemas de seguridad: airbag, sensores de gases peligrosos, etc.

Sistemas operativos para sistemas embebidos



Regional Breakout

APAC users will use FreeRTOS and Android much more than other regions and use Embedded Linux much less.
EMEA will use Android less than other regions.

Most Used	World	Americas	EMEA	APAC
Embedded Linux	31%	32%	31%	**26%**
FreeRTOS	27%	25%	24%	**37%**
Android	14%	12%	**10%**	**26%**

2019 (N = 424)

Only Operating Systems with 3% more are shown

Figura obtenida de EETimes, “2019 Embedded Markets Study”.
https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf

FreeRTOS

- **Hard RTOS** para microcontroladores y procesadores pequeños.
  - SO más utilizado para sistemas embebidos de bajo poder de procesamiento.
- Desarrollado por Richard Barry, luego mantenido por Real Time Engineers ltd., luego Amazon Web Services.
- Licencia open source MIT license.
  - Licencia libre permisiva.
  - Permite reutilizar software dentro de software propietario.
- Objetivos:
  - Velocidad, pequeño tamaño, simplicidad, compacto, confiabilidad.
- Plataformas: ARM, x86, Atmel AVR, TI MSP432, Xtensa, Microchip PIC etc.
- Foro de soporte muy activo y varios libros accesibles sin costo.

Soportado por: https://www.freertos.org/RTOS_ports.html

FreeRTOS

- Implementa:
  - Scheduler (tareas).
  - Exclusión mutua, semáforos, colas.
  - Temporizadores.
  - Modos bajo consumo de energía.
  - Manejo de interrupciones.
  - Notificaciones entre tareas.
  - Integración con servicios en la nube de AWS (AWS IoT services).
- No implementa:
  - Controladores de dispositivos (drivers).
  - Administración de memoria.
  - Cuentas de usuario.

FreeRTOS

- Características:
  - El kernel requiere entre 6KB y 12KB de memoria (El ATmega 328p posee 32 KB).
  - Muy escalable.
  - SafeRTOS: Versión certificada para aplicaciones críticas como medicina, automóviles e industriales.
  - Código fuente lenguaje C.
    - Puede ser compilado en más de 20 compiladores.
    - Puede ser compilado para más de 30 arquitecturas.

FreeRTOS

- Tareas:
  - Se crean en tiempo de ejecución.
  - freeRTOS les asigna memoria cada vez que se crea una tarea, y libera memoria cada vez que la tarea finaliza.
  - No se utiliza malloc() y free() para asignar y liberar memoria. Se utiliza pvPortMalloc() y vPortFree().

FreeRTOS - Memorias en tareas

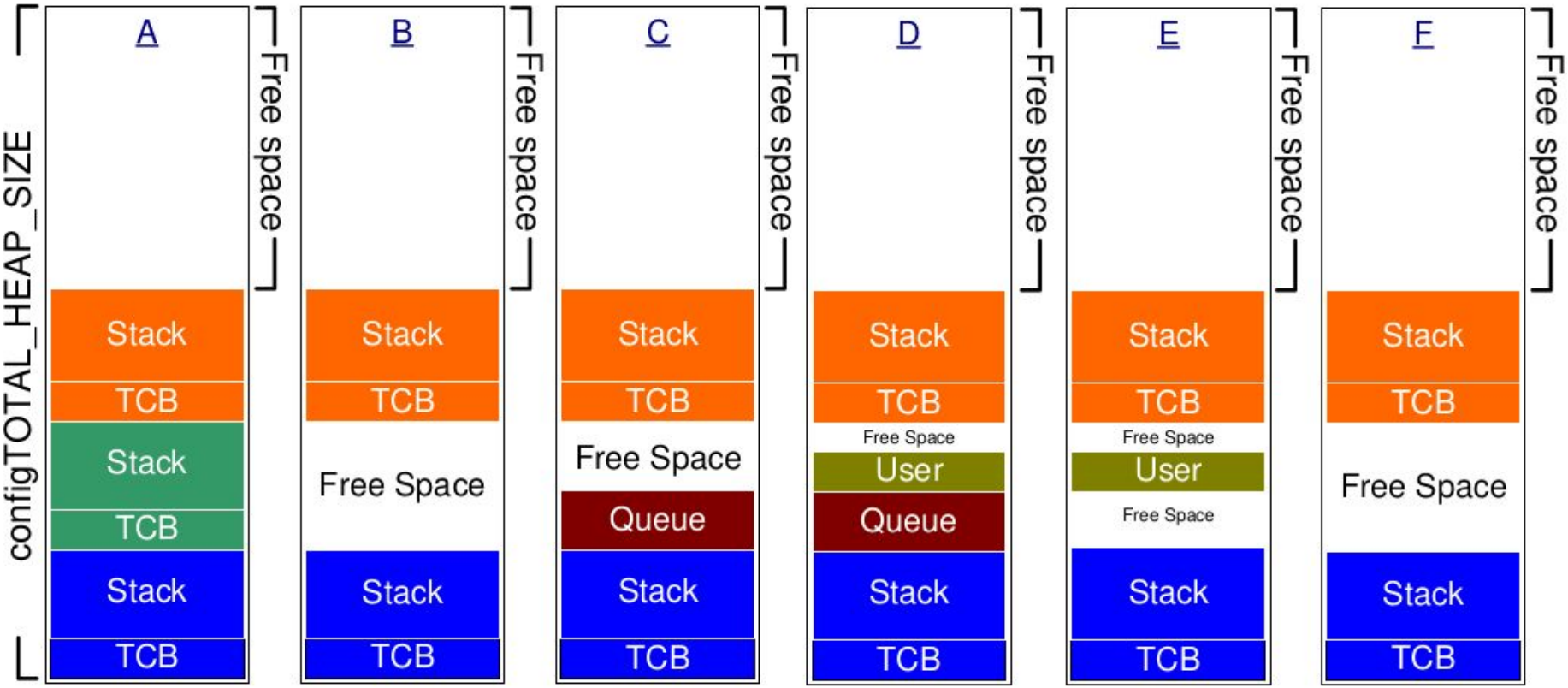


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 33.

Tareas - Estados

- **Running**: una tarea por núcleo.
- **Ready**: Esperando que el scheduler la pase a estado running.
- **Blocked**: No puede pasar a estado running. Con timeout. Pasa a estado Ready por:
  - Evento externo o temporal.
  - Timeout finaliza.
- **Suspended**: No puede pasar a estado running. Sin timeout. Solo entra o sale con vTaskSuspend() and xTaskResume()

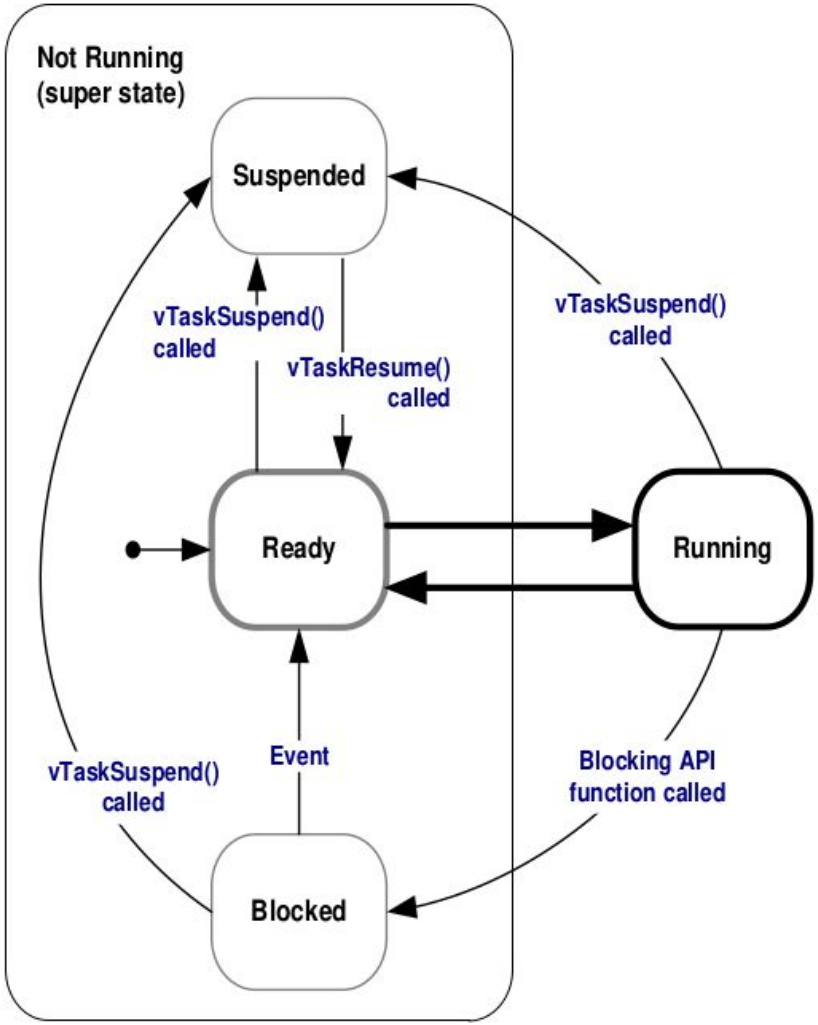


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 67

FreeRTOS

Scheduler

- Preemptive o con prioridades.
- Round-robin, time sliced (intervalos de tiempo fijo) con prioridades.
- Prioridades: Valor desde 0 a un máximo. Mayor valor, mayor prioridad. Fijas.
  - Se ejecutan siempre tareas de mayor prioridad.
    - Tareas de mayor prioridad deben entrar en modo blocked, de lo contrario, se ejecutarán todo el tiempo.
- Idle task: creada cuando el scheduler arranca. Tiene la más baja prioridad. La única función que ejecuta es liberar memoria asignada a tareas eliminadas.

FreeRTOS

- Se ejecuta sobre procesadores:
  - Simple core, simetric multicore: una sola instancia de FreeRTOS para todos los núcleos.
  - Asimetric multicore: Cada core ejecuta su instancia de FreeRTOS.

Implementación de una tarea

```
void vATaskFunction( void *pvParameters )
    {
        for( ;; )
        {
            -- Task application code here. --
        }
        /*No se debe llamar a return*/
        vTaskDelete( NULL ); /*Finaliza la tarea*/
    }
```

FreeRTOS

Creación de una tarea

```
xTaskCreate(
   TaskBlink13,  //Nombre de la función que implementa la tarea.
                 //Variable tipo "TaskFunction_t" (vector).
   "Blink",      //Nombre entendible por humanos. No usado por freeRTOS
   128,          //tamaño del stack asignado a la tarea. Tipo: "uint16_t"
   pvParameters, //Parámetros. Tipo "void *" (se pasan por referencia).
                 //Si no hay parámetros, se escribe NULL.
   2,            // Prioridad
   pxCreatedTask,//Manejador de tarea. Puede invocarse desde otra tarea
                 //para realizar operaciones como cambiar su prioridad
                 //o eliminar la tarea.
                 //Variable tipo "TaskHandle_t *" (se pasa por referencia).
                 //Puede escribirse NULL si la tarea no será modificada.
   );
```

FreeRTOS

Creación de una tarea

- La función xTaskCreate() retorna:
  - pdPASS si la tarea fue creada exitosamente.
  - pdFAIL si hubo errores al crear la tarea (por ejemplo, si no hay suficiente RAM).
- Pueden crearse varias tareas desde una misma función.
- Tick Interrupt: Interrupción que se ejecuta para que el scheduler ejecute la siguiente tarea.

FreeRTOS

Ejecución de dos tareas periódicas con igual prioridad.

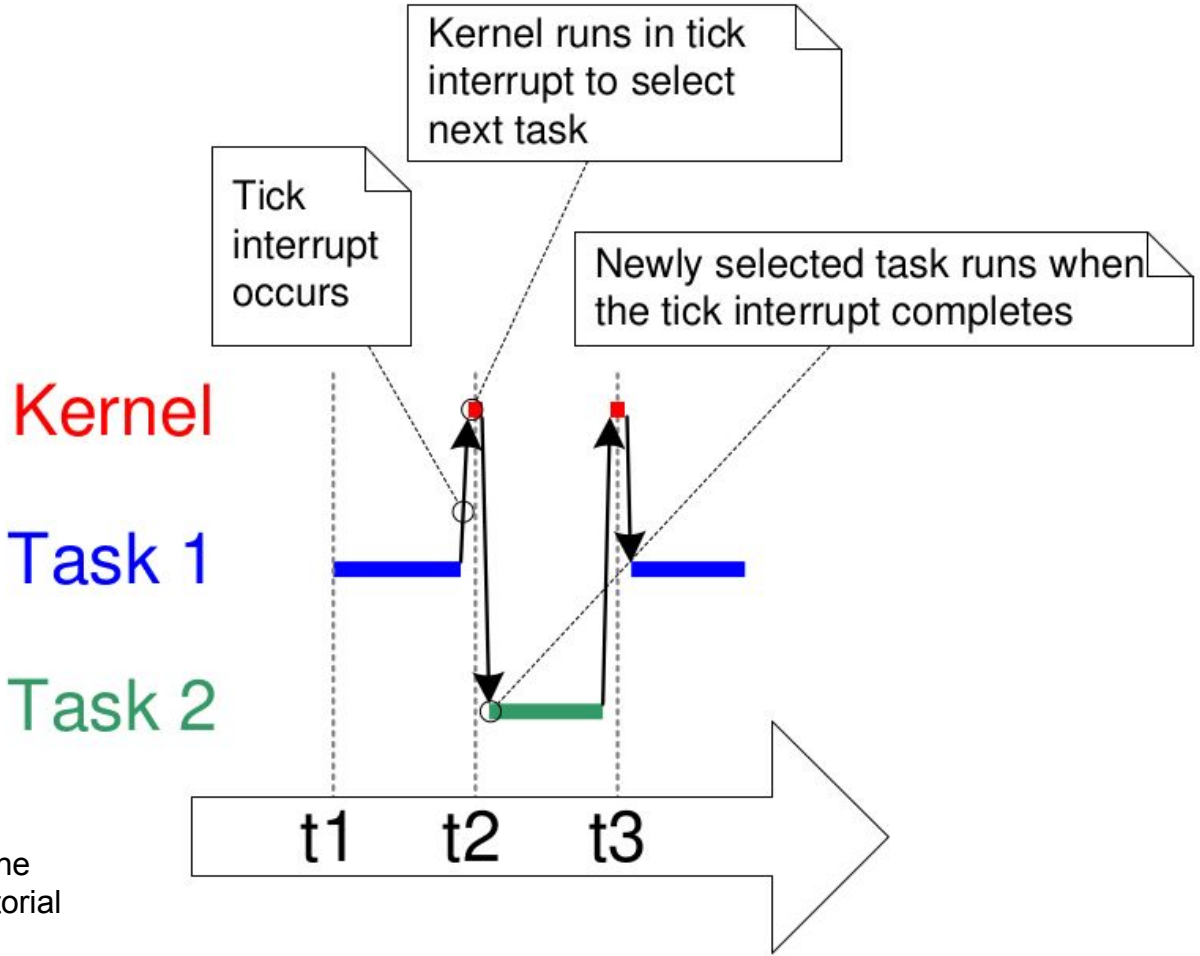


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 62

FreeRTOS

Ejecución de dos tareas periódicas con distinta prioridad. La tarea 2 posee mayor prioridad.

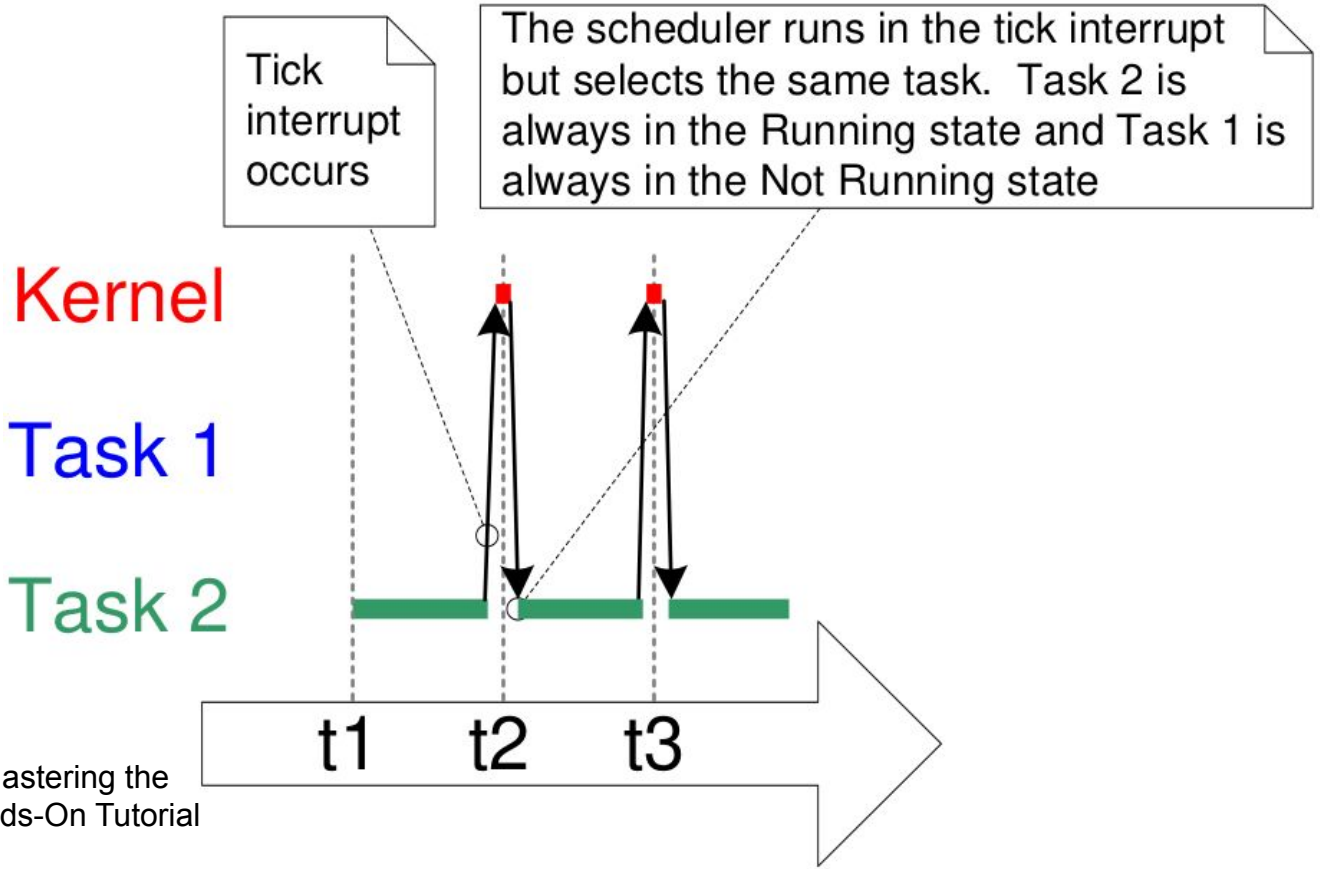


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 64

FreeRTOS

- Primitivas interesantes:
  - vTaskDelay(TickType_t xTicksToDelay): Coloca la tarea en modo Blocked hasta que transcurra el tiempo indicado.
  - vTaskDelayUntil(TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement): Coloca la tarea en modo Blocked hasta un momento de tiempo indicado.
    Ejemplo:

```
TickType_t xLastWakeTime;
xLastWakeTime = xTaskGetTickCount();
vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
```

  - void vTaskDelete(TaskHandle_t pxTaskToDelete): Elimina una tarea. Si se ejecuta dentro de la tarea a eliminar, pxTaskToDelete puede valer NULL.

FreeRTOS

Ejecución de dos tareas manejadas por eventos.

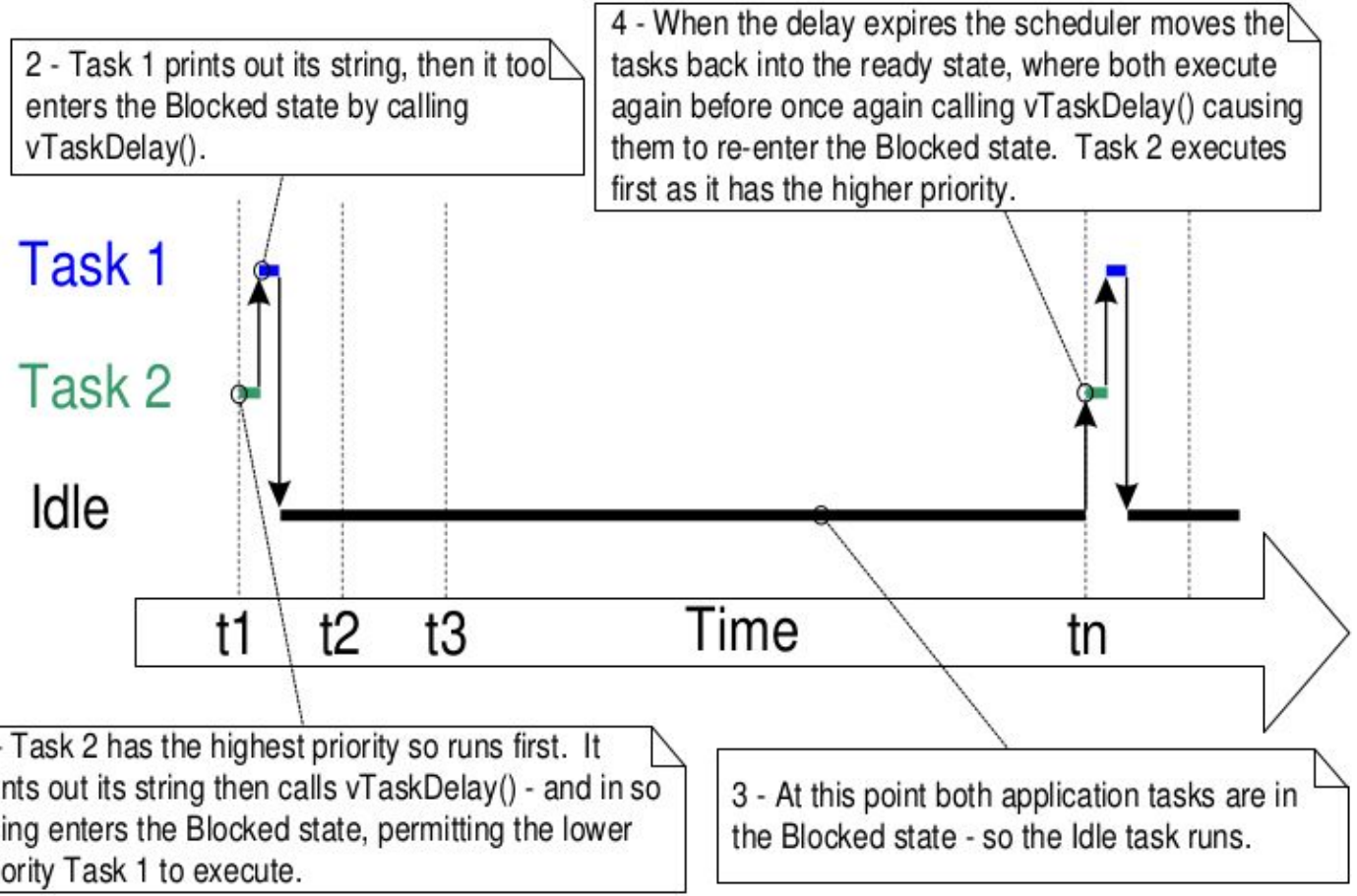


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 70

UNCUYO
UNIVERSIDAD
NACIONAL DE CUYO

FreeRTOS

Ejecución de dos tareas periódicas y una tarea manejada por eventos de mayor prioridad.

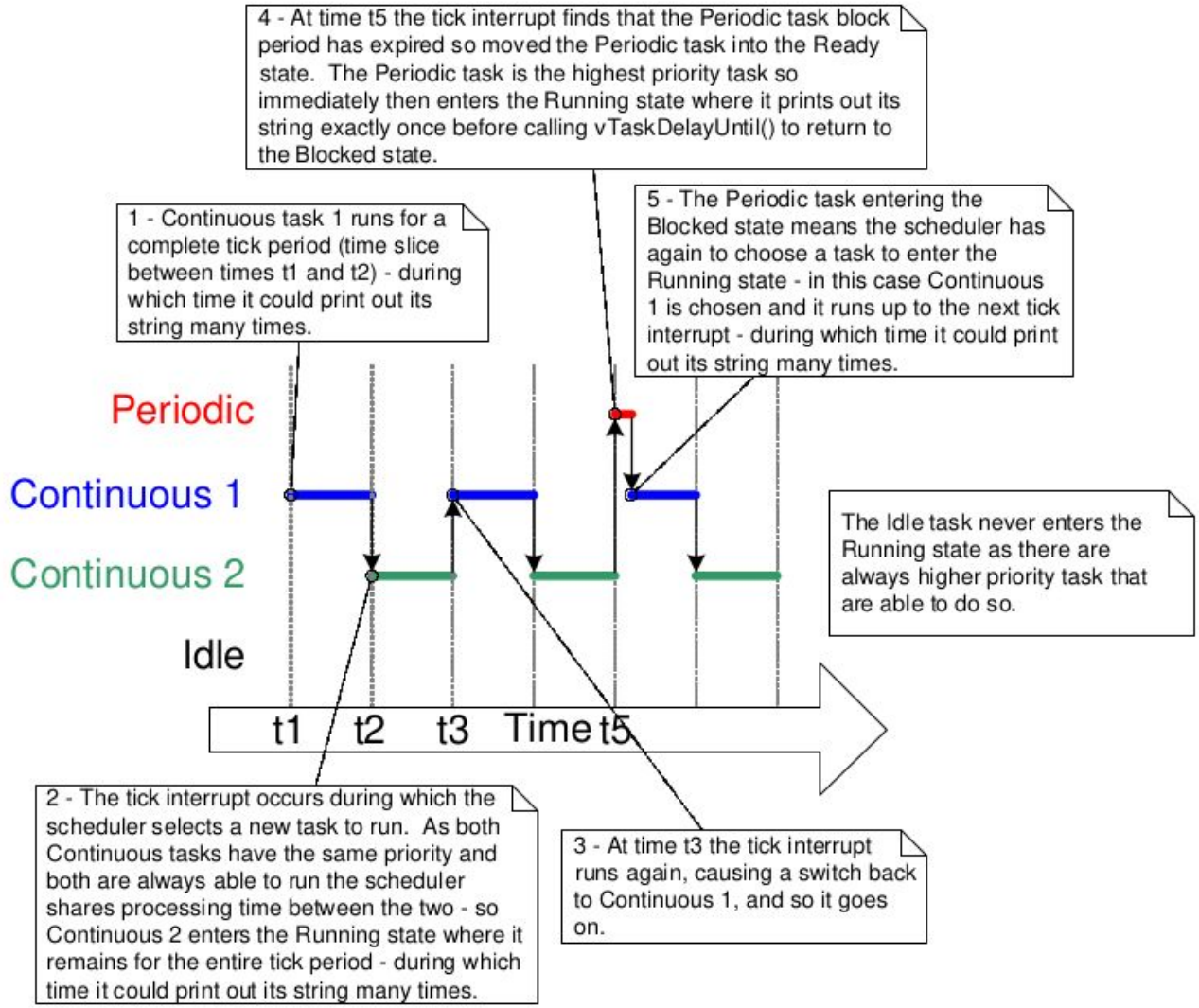


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 75

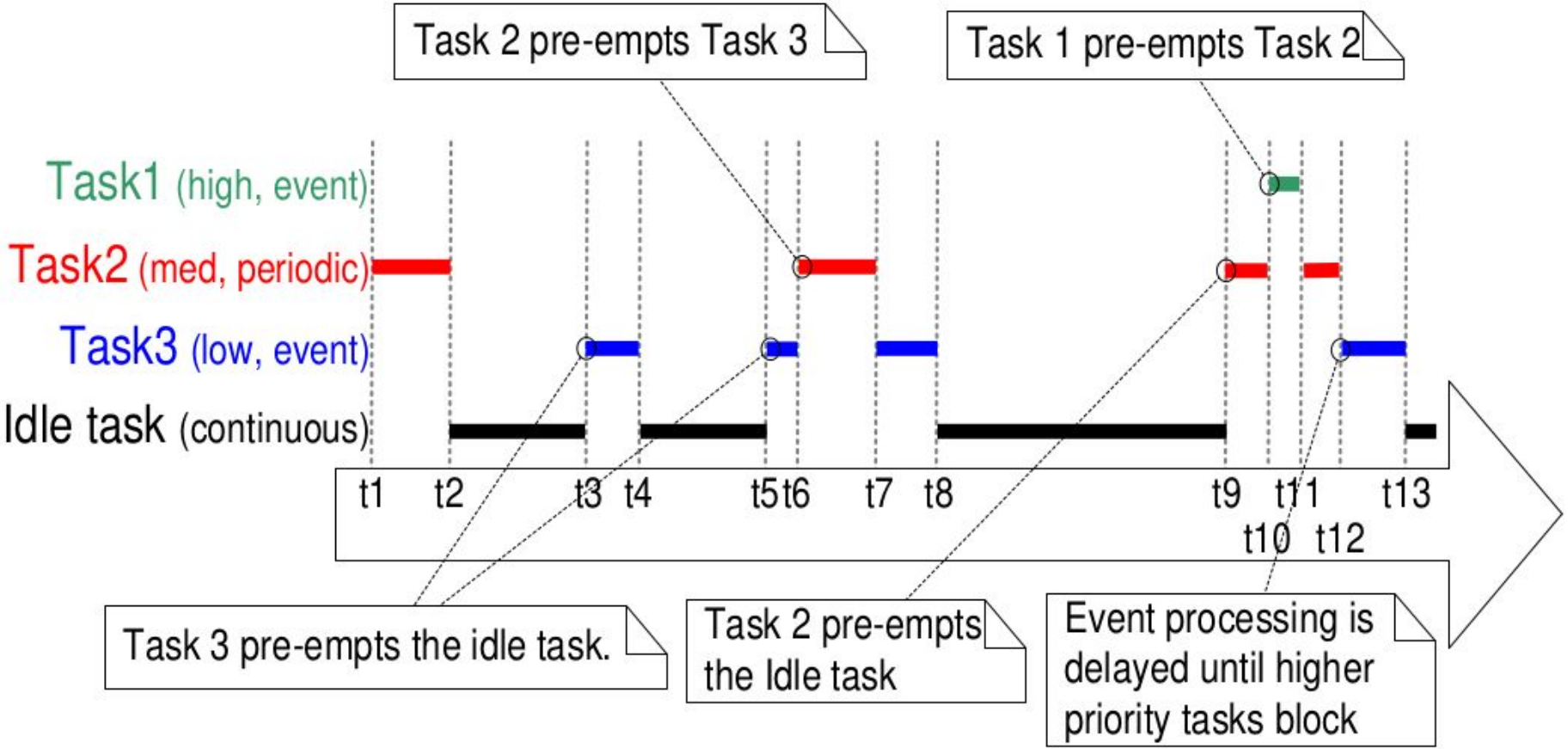
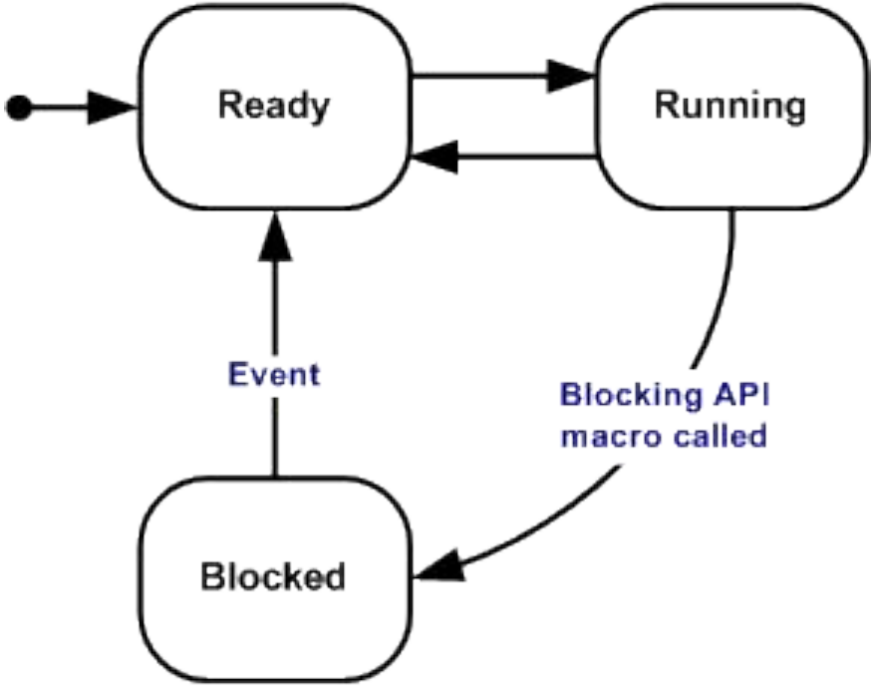


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 93

Co-rutinas

- Destinadas a procesadores de altas restricciones de RAM.



Co-rutinas

- Implementación
- Se crean con xCoRoutineCreate().
- Las tareas tienen prioridad sobre las corrutinas.

```
void vACoRoutineFunction( CoRoutineHandle_t xHandle,
                          UBaseType_t uxIndex )
{
    crSTART( xHandle );

    for( ;; )
    {
        -- Co-routine application code here. --
    }

    crEND();
}
```

FreeRTOS
Sintaxis

- Variables:
  - TickType_t: tipo de variable de la interrupción tick interrupt. Sirve para medir tiempos.
  - BaseType_t: Tipo de datos más eficiente de la arquitectura (depende del número de bits de la arquitectura).
  - Prefijos:
    - c: para char.
    - s: para int16_t (short).
    - l: para int32_t (long).
    - x: para BaseType_t
    - u: unsigned
    - p: punteros

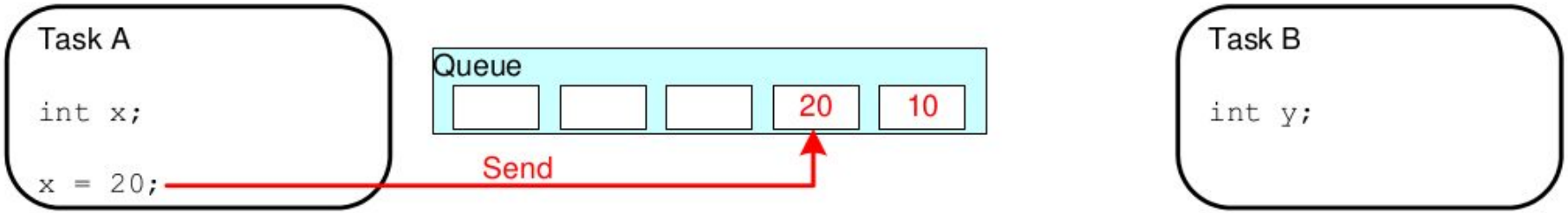
FreeRTOS
Sintaxis

- Prefijos en las funciones:
  - **vTask**PrioritySet(): retorna **void** y está definida en **task.c**.
  - **xQueue**Receive(): retorna variable del tipo **BaseType_t** y está definida en **queue.c**.
  - **pvTimer**GetTimerID(): retorna un **puntero** y está definida en **timers.c**.
  - Funciones **privadas** en el alcance de un archivo añaden el prefijo **prv**.

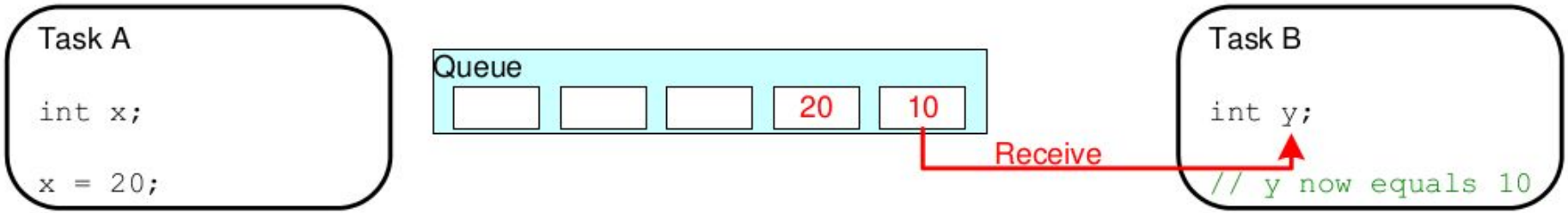
FreeRTOS
Comunicación entre tareas - Colas

- Permiten comunicación entre:
  - Tareas hacia tareas.
  - Tareas hacia interrupciones.
  - Interrupciones hacia tareas.
- Dos tipos:
  - Cola por copia: el dato es copiado en la cola, byte por byte.
  - Cola por referencia: La cola contiene un puntero a los datos.
- Tipo FIFO.
- Datos de tamaño variable: estructuras formadas por un vector apuntando al comienzo de los datos y un dato que contiene el tamaño de los datos.
- Una tarea que intenta leer datos de una cola vacía pasa a estado blocked.
- Una tarea que intenta escribir datos en una cola llena se bloqueará.

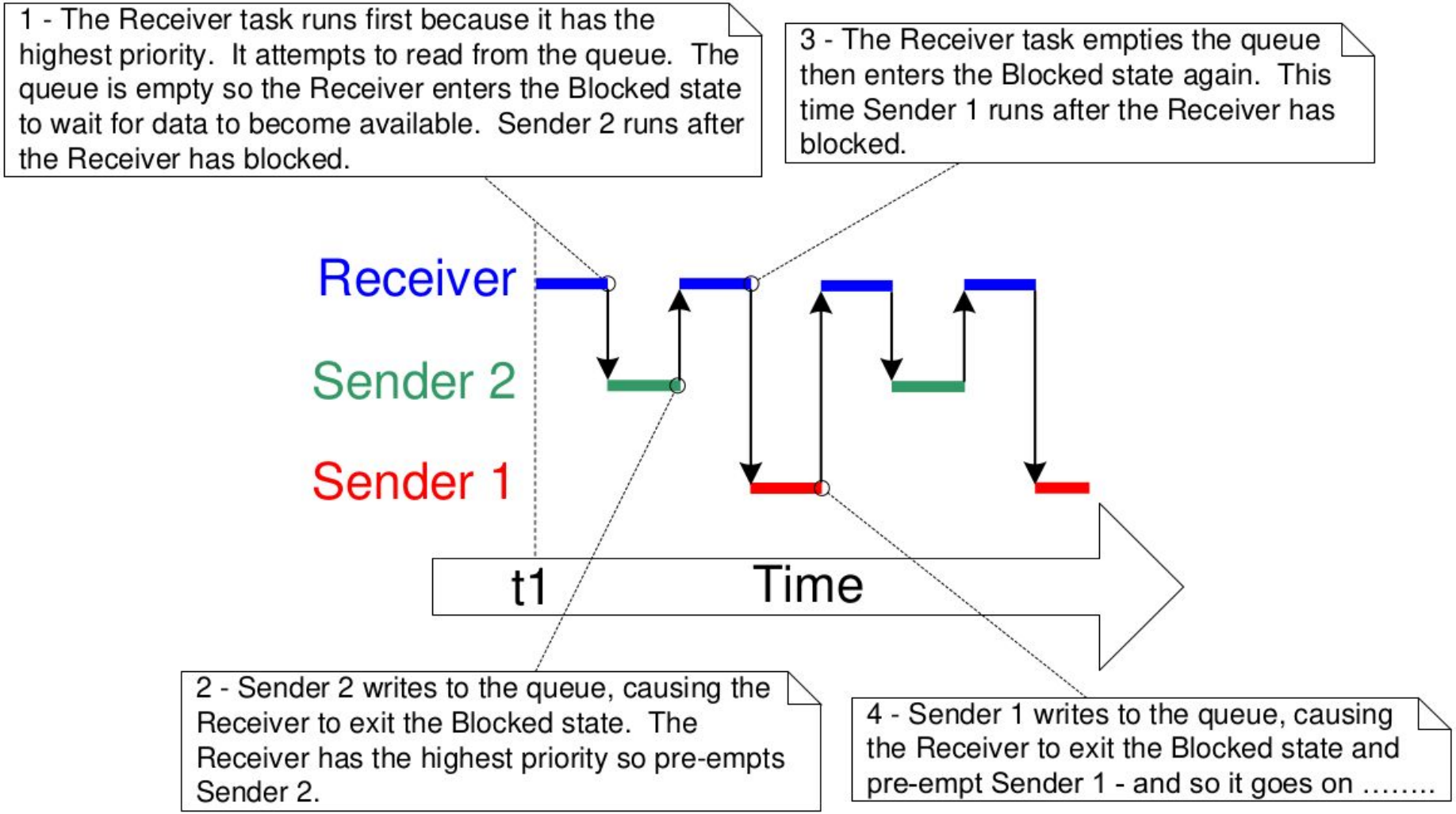
FreeRTOS - Comunicación entre tareas - Colas



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).


1 - The Receiver task runs first because it has the highest priority. It attempts to read from the queue. The queue is empty so the Receiver enters the Blocked state to wait for data to become available. Sender 2 runs after the Receiver has blocked.
3 - The Receiver task empties the queue then enters the Blocked state again. This time Sender 1 runs after the Receiver has blocked.
Receiver
Sender 2
Sender 1
t1
Time
2 - Sender 2 writes to the queue, causing the Receiver to exit the Blocked state. The Receiver has the highest priority so pre-empts Sender 2.
4 - Sender 1 writes to the queue, causing the Receiver to exit the Blocked state and pre-empt Sender 1 - and so it goes on ……..

Primitivas para las colas

QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);

- uxQueueLength: número máximo de ítems que la cola puede tener.
- uxItemSize: Tamaño de cada elemento (en bytes).
- Return Value:
  - NULL: la cola no pudo crearse.
  - non-NULL: la cola se creó con éxito. Se retorna el manejador.

Primitivas para las colas

BaseType_t xQueueSendToFront(QueueHandle_t xQueue,
const void * pvItemToQueue, TickType_t xTicksToWait);
(copia datos al frente de la cola)

BaseType_t xQueueSendToBack(QueueHandle_t xQueue,
const void * pvItemToQueue, TickType_t xTicksToWait);
(copia datos al final de la cola. Equivalente a xQueueSend()).

- xQueue: Manejador de la cola.
- pvItemToQueue: Puntero a los datos a escribir en la cola.
- xTicksToWait: Máxima cantidad de tiempo a esperar si la cola está llena (La tarea se bloquea).
- Returned value:
  - pdPASS: El dato se escribió correctamente en la cola.
  - errQUEUE_FULL: El dato no se escribió por estar la cola llena.

Primitivas para las colas

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

- xQueue: Manejador de la cola
- pvBuffer: puntero a un buffer donde se escribirán los datos recibidos.
- xTicksToWait: tiempo máximo a esperar si no hay datos disponibles en la cola. La tarea se bloquea.
- Returned value:
  - pdPASS: Se leyeron datos de la cola satisfactoriamente.
  - errQUEUE_EMPTY: No se leyeron datos por estar la cola vacía.

Interrupciones en FreeRTOS

- Las rutinas de servicio tienen prioridad sobre las tareas
  - La ISR de menor prioridad interrumpirá a la tarea de mayor prioridad.
  - Nunca una tarea interrumpirá a una ISR.
- Dos versiones de las funciones definidas por la API:
  - Funciones de la API para ser llamadas desde las tareas (colocan la tarea en estado bloqueado).
  - Funciones de la API para ser llamadas desde las ISR (No se puede colocar una ISR en estado bloqueado).
    - Se agrega **FromISR** al nombre de la función.
      - xQueueSendToBack(...)
      - xQueueSendToBackFromISR(....)

Procesamiento de interrupción diferida

- Objetivo: que la duración de la rutina de servicio sea mínima.
  - La rutina de servicio solo:
    - Desbloquea una tarea que atenderá el evento que causó la interrupción.
    - Realiza alguna tarea mínima.
- Ventajas:
  - Evitar que las rutinas de servicio causen variaciones temporales en la ejecución de las tareas (afecten comienzo y finalización de las tareas).
  - Evitar el anidamiento de ISRs
    - Cada ISR requiere recursos.
    - Cada ISR requiere tiempo de ejecución (Se puede perder previsibilidad).

Procesamiento de interrupción diferida

- taskYIELD(): Pide un cambio de contexto. Es llamada desde una tarea:
  - portYIELD_FROM_ISR() y portEND_SWITCHING_ISR(): Versión para llamar desde una ISR (significan lo mismo. Distintas implementaciones implementan una u otra).
  - portEND_SWITCHING_ISR(xHigherPriorityTaskWoken);
    - xHigherPriorityTaskWoken variable seteada por la API si debe ocurrir un cambio de contexto.

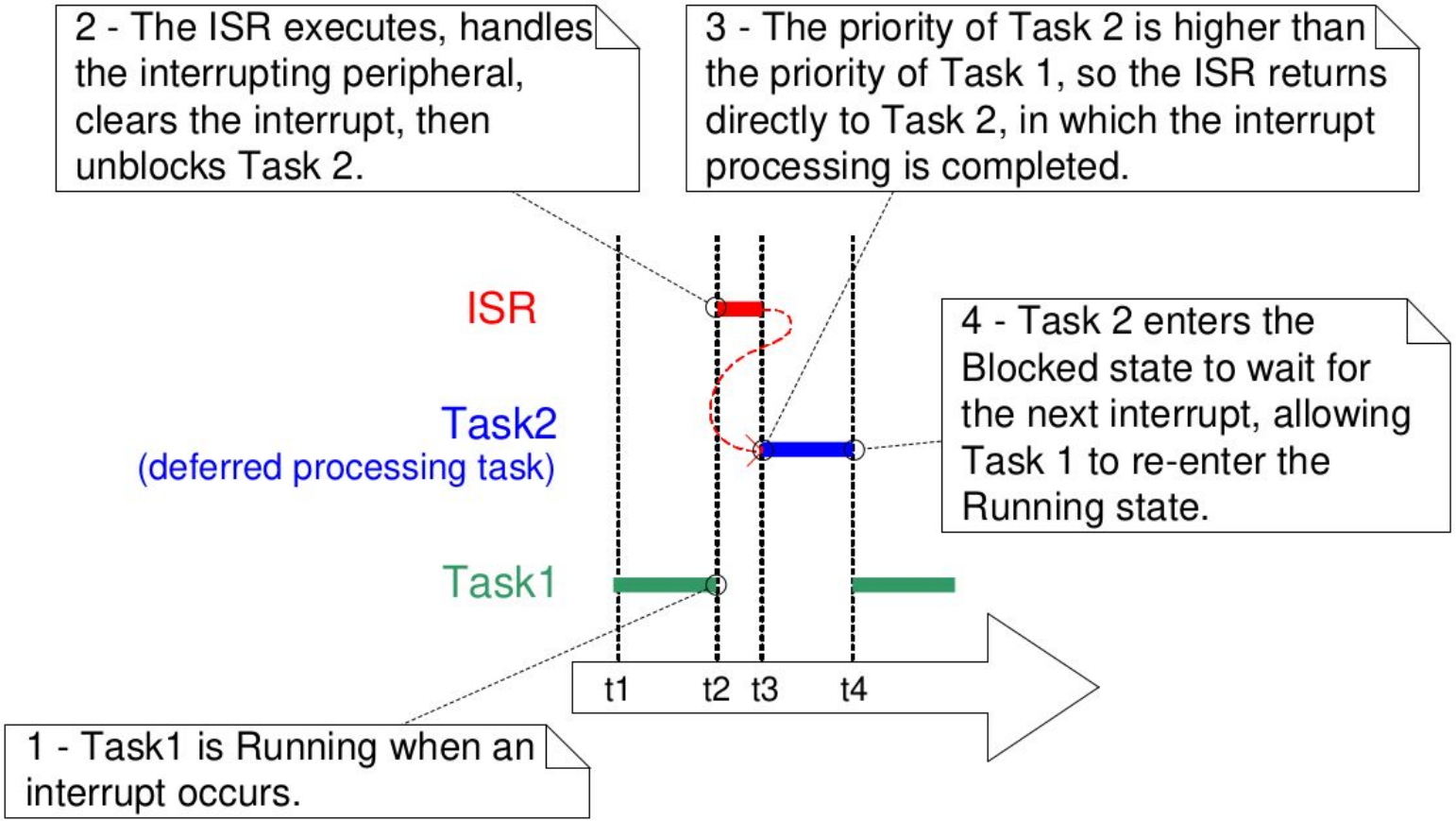
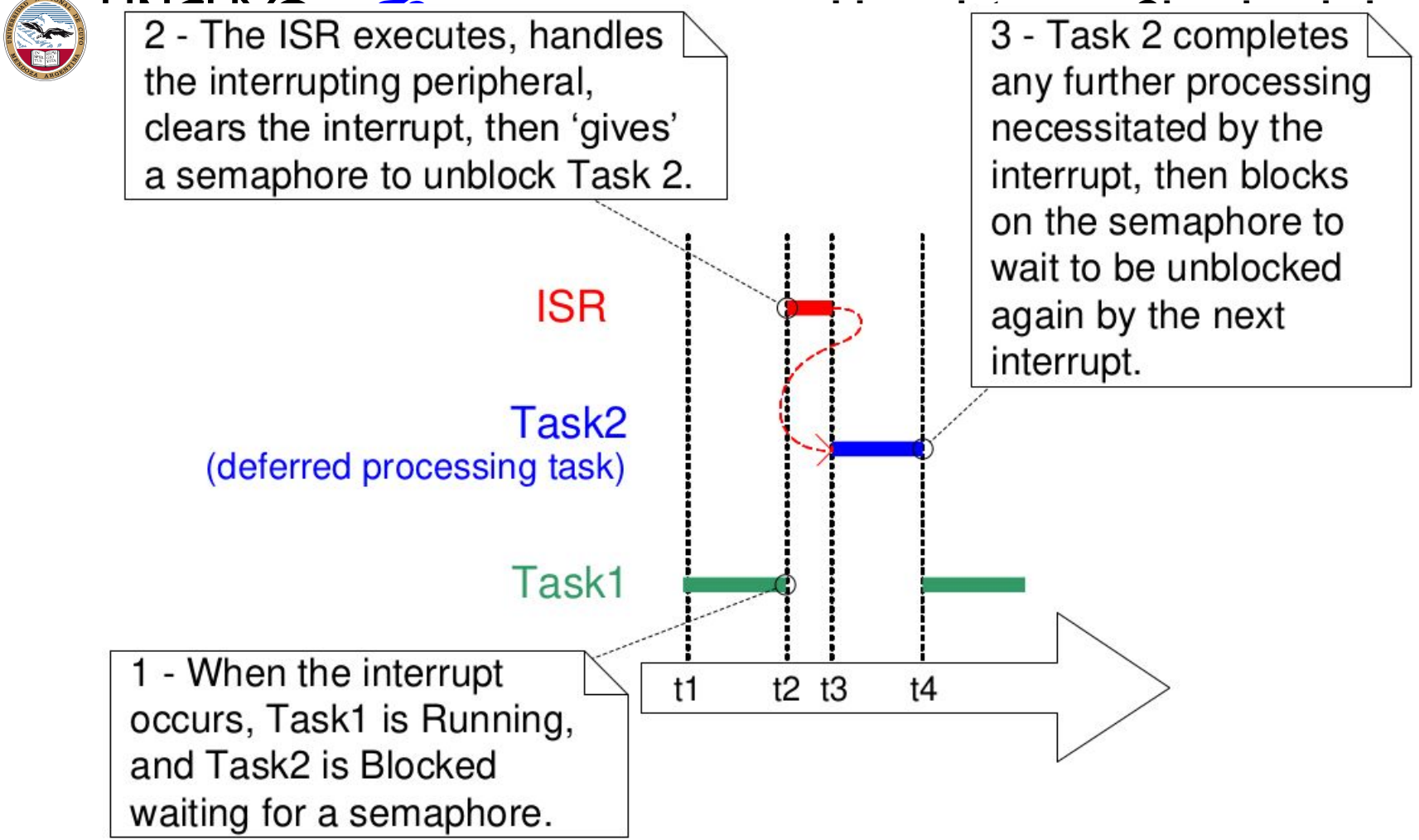


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 190.

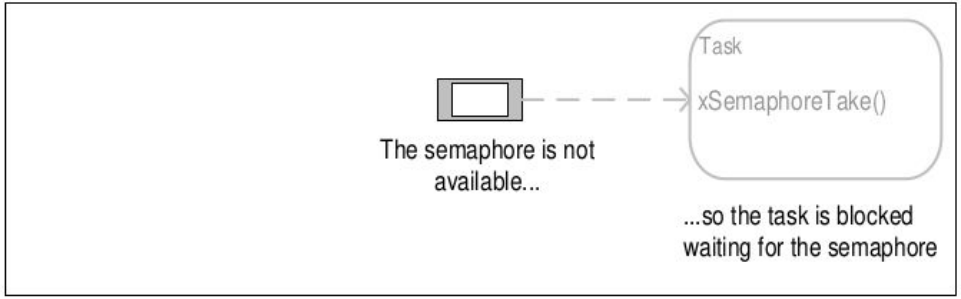

2 - The ISR executes, handles the interrupting peripheral, clears the interrupt, then 'gives' a semaphore to unblock Task 2.
3 - Task 2 completes any further processing necessitated by the interrupt, then blocks on the semaphore to wait to be unblocked again by the next interrupt.
ISR
Task2
(deferred processing task)
Task1
1 - When the interrupt occurs, Task1 is Running, and Task2 is Blocked waiting for a semaphore.
t1
t2
t3
t4

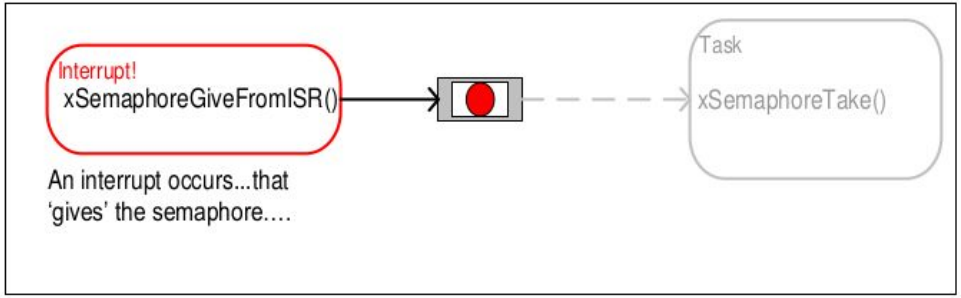
Semáforos binarios

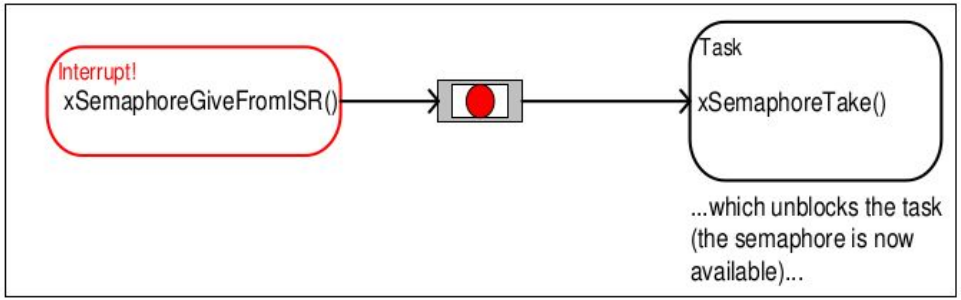
- Mecanismo de sincronización (Dijkstra, 1965).
- SemaphoreHandle_t xSemaphoreCreateBinary(void);
  - Crea un semáforo. Devuelve NULL si el semáforo no pudo crearse (falta de memoria).
- BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
  - Intenta tomar el semáforo (si está disponible).
  - xSemaphore: Identificador del semáforo.
  - xTicksToWait: Tiempo máximo a esperar (en ticks).
    - pdMS_TO_TICKS() convierte milisegundos a ticks.
    - Si es configurado como portMAX_DELAY, la tarea esperará indefinidamente.
  - Valor retornado:
    - pdPASS: El semáforo fue obtenido exitosamente.
    - pdFALSE: El tiempo de espera expiró.

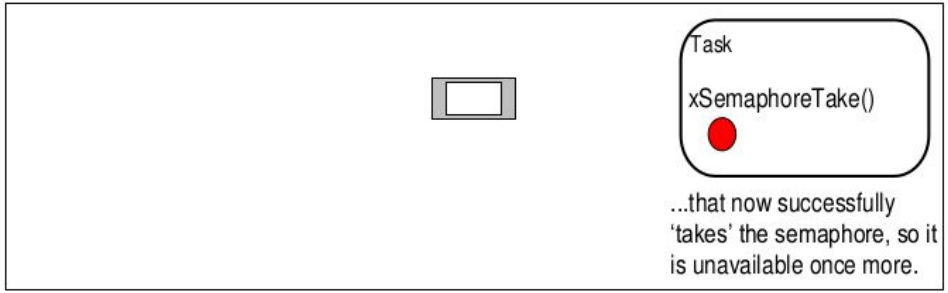
Semáforos binarios

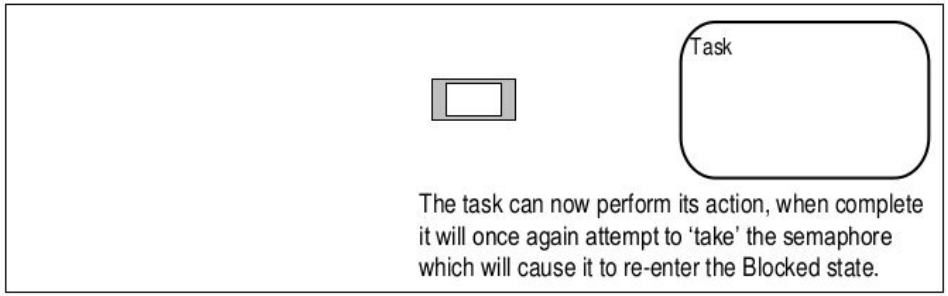
- xSemaphoreGive(SemaphoreHandle_t xSemaphore);
  - Otorga un semáforo. Para todos los tipos de semáforos (binarios, contador, etc).
- BaseType_t xSemaphoreGiveFromISR(SemaphoreHandle_t xSemaphore, BaseType_t *pxHigherPriorityTaskWoken);
  - Otorga el semáforo desde una ISR.
  - pxHigherPriorityTaskWoken es seteada a pdTRUE si se produce un cambio de contexto.


Task
xSemaphoreTake()
The semaphore is not available...
...so the task is blocked waiting for the semaphore


Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
An interrupt occurs...that 'gives' the semaphore....


Interrupt!
xSemaphoreGiveFromISR()
Task
xSemaphoreTake()
...which unblocks the task (the semaphore is now available)...

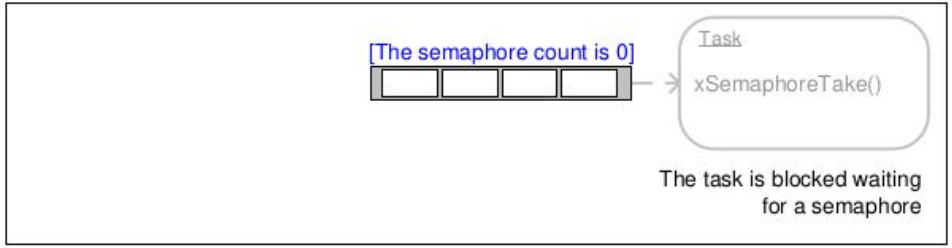

Task
xSemaphoreTake()
...that now successfully 'takes' the semaphore, so it is unavailable once more.

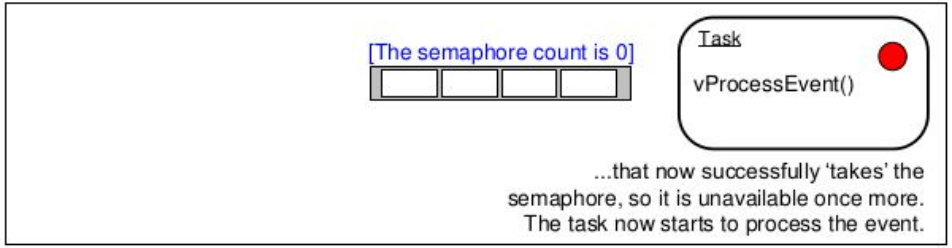

Task
The task can now perform its action, when complete it will once again attempt to 'take' the semaphore which will cause it to re-enter the Blocked state.

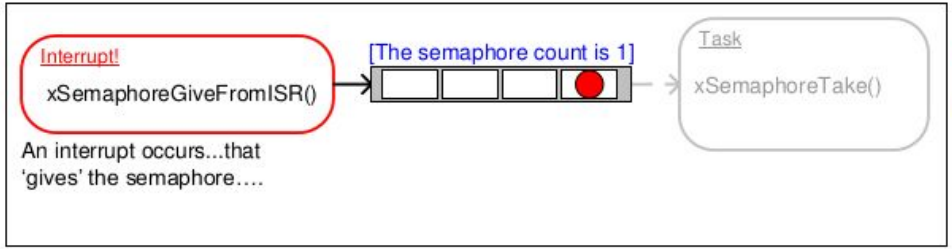
Semáforos contadores

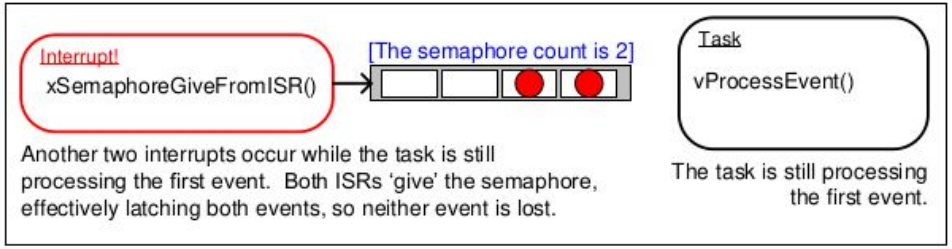
- Objetivos en los sistemas embebidos:
  - Contar eventos.
  - Manejar recursos.
- SemaphoreHandle_t xSemaphoreCreateCounting(UBaseType_t uxMaxCount, UBaseType_t uxInitialCount);
  - Valor de retorno:
    - NULL: El semáforo no pudo crearse.
    - non-NULL. El semáforo fue creado con éxito y devuelve un manejador de semáforo.
- Otorgar y tomar el semáforo (funciones vistas antes):
  - xSemaphoreGive(SemaphoreHandle_t xSemaphore);
  - BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);

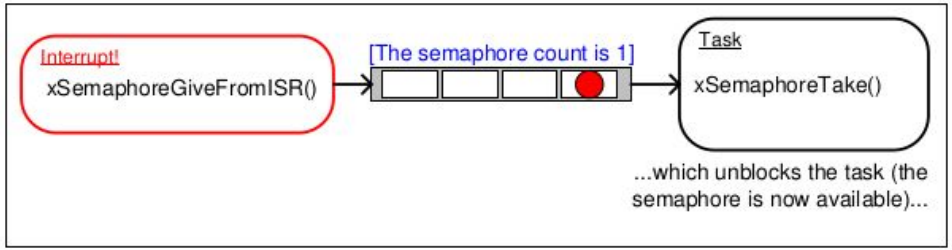
Semáforos múltiples

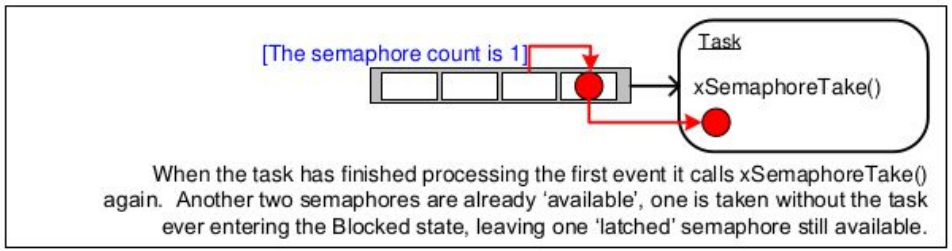












Exclusión mutua

Multitasking + recursos compartidos: Recursos estado inconsistentes.

- Una tarea1 comienza una operación, otra tarea2 interrumpe a tarea1 antes de que tarea1 finalice. El recurso puede quedar en estado inconsistente.
  - Ejemplos:
    - Tarea1 escribe “Hello world”, y tarea2 escribe “exit?”, el estado final puede ser: Hello worexit?ld.
    - Operaciones Read, Modify, Write.
    - Acceso a variables no atómicas.
- Exclusión mutua: Asegurar que una tarea tiene acceso exclusivo a un recurso.

- Regiones críticas: Un switch a otra tarea no puede ocurrir dentro de una región crítica.
  - Deshabilita interrupciones con prioridades menor a cierto umbral (el umbral depende de la implementación).
  - No se puede llamar a funciones desde una ISR que fue llamada desde una sección crítica.

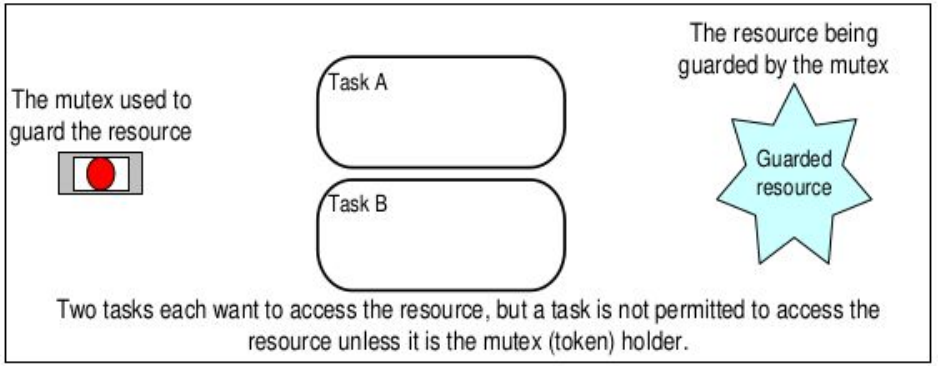
```
taskENTER_CRITICAL();
….Región crítica…..
taskEXIT_CRITICAL();
```

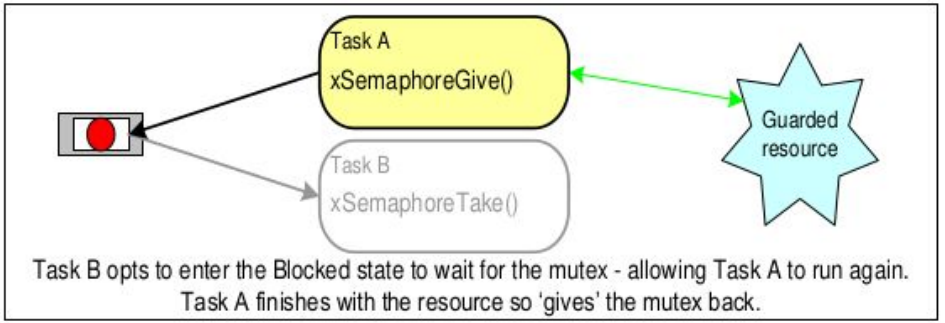
Versión para interrupciones:

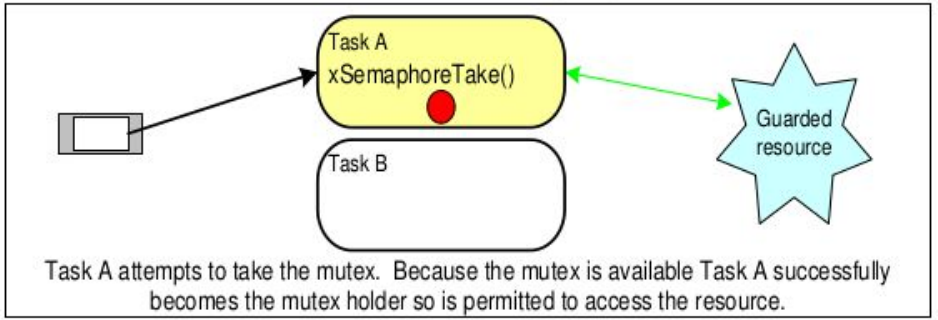
```
UBaseType_t  uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();
taskEXIT_CRITICAL_FROM_ISR(UBaseType_t  uxSavedInterruptStatus );
```

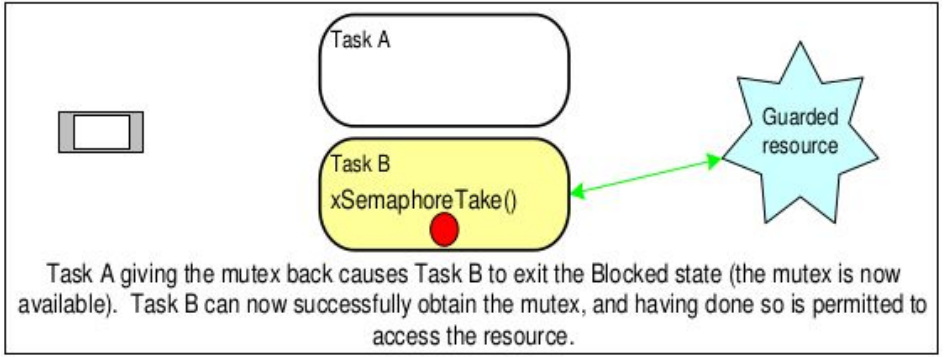
- Suspender el Scheduler.
  - Evita un switch de una tarea hacia otra.
  - No deshabilita interrupciones.
  - void vTaskSuspendAll(void);
  - BaseType_t xTaskResumeAll(void);
    - Valor de retorno:
      - pdTRUE si un cambio de contexto está pendiente.
      - pdFALSE si no hay cambio de contexto pendiente.

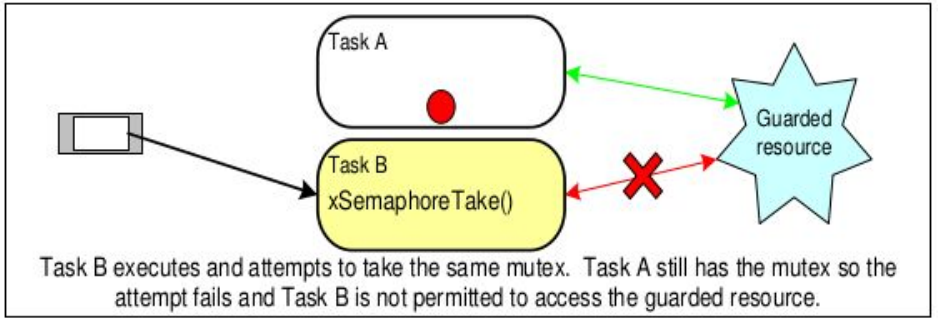
- MUTEX (MUTual EXclusion): Tipo de semáforo.
- Funcionamiento: similar a un token.
  - La tarea que necesita acceder a un recurso debe tomar el token.
  - Una vez que la tarea finaliza su trabajo con el recurso, debe liberar el token.
- El mutex debe siempre ser devuelto (los semáforos no es obligatorio).











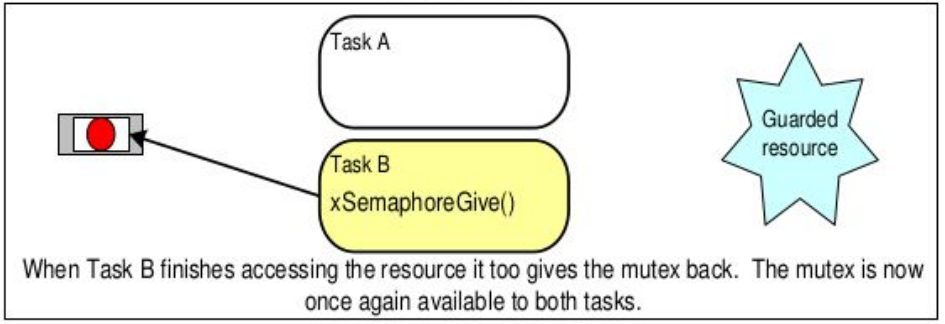


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 245.

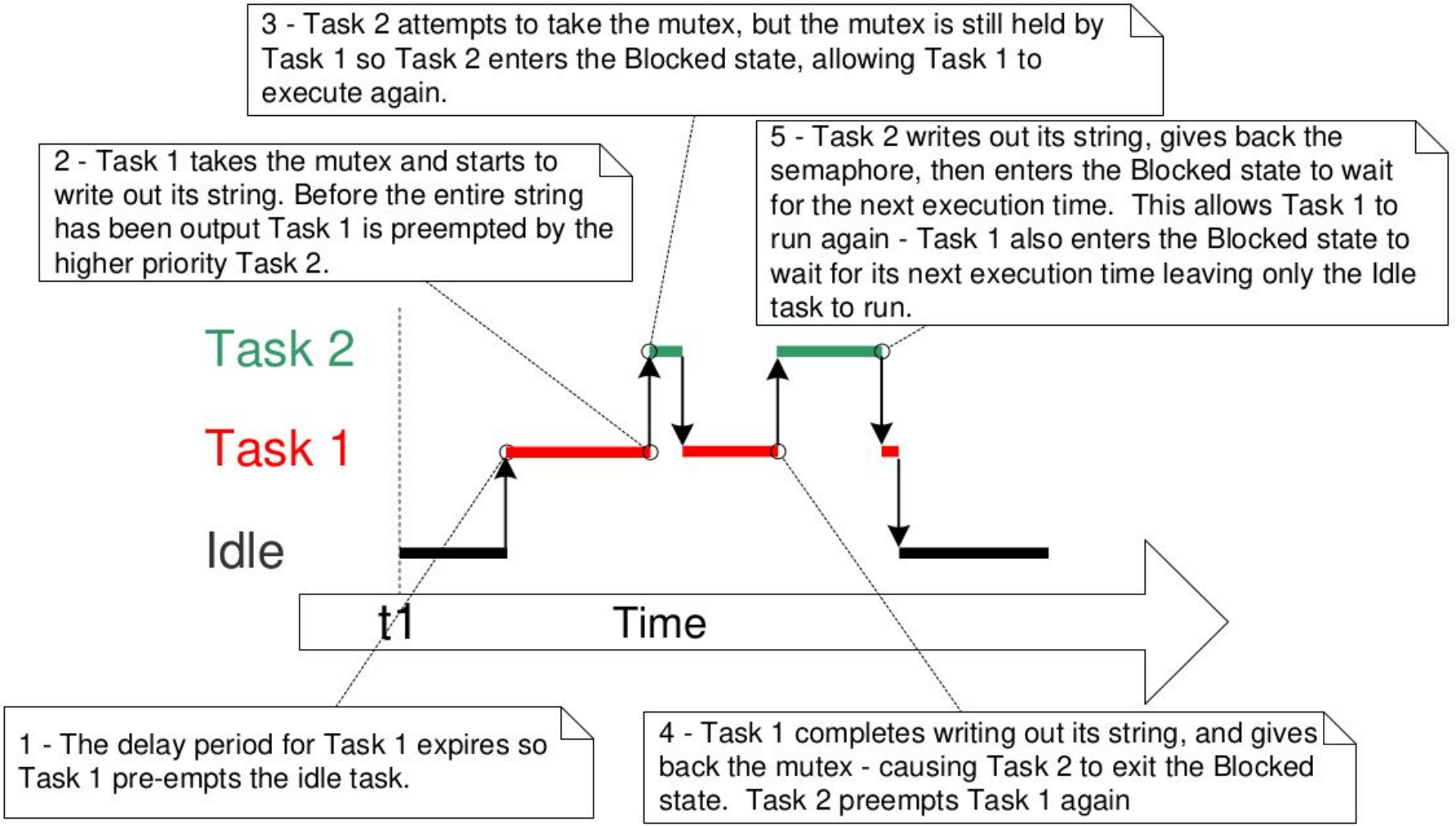


Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 250.

- SemaphoreHandle_t xSemaphoreCreateMutex(void);
  - Returned value:
    - NULL: El mutex no pudo ser creado.
    - non-NULL: El mutex pudo crearse. Retorna un manejador.
- BaseType_t xSemaphoreTake(SemaphoreHandle_t xMutex, TickType_t xTicksToWait);
- xSemaphoreGive(SemaphoreHandle_t xSemaphore);

Task Notifications

- Las tareas pueden comunicarse a través de: colas, semáforos, notificaciones.

- BaseType_t xTaskNotifyGive(TaskHandle_t xTaskToNotify);
  - xTaskToNotify: manejador de tarea.

- uint32_t ulTaskNotifyTake(BaseType_t xClearCountOnExit, TickType_t xTicksToWait);
  - xClearCountOnExit: si vale:
    - pdTRUE: se pone a cero el contador de notificaciones.
    - pdFALSE: se decrementa el contador de notificaciones.
  - xTicksToWait: Tiempo máximo que la tarea permanecerá bloqueada esperando la notificación
  - Retorna si el contador de notificaciones es >0.

Task Notifications

```
void vTask1( void *pvParam )
{
  for( ;; )
  {
    /* Write function code
    here. */
    ....

    /* At some point vTask1
    sends an event to
    vTask2 using a direct to
    task notification.*/
    ASendFunction();
  }
}
```

This time there is no communication object in the middle

```
void vTask2( void *pvParam )
{
  for( ;; )
  {
    /* Write function code
    here. */
    ....

    /* At some point vTask2
    receives a direct
    notification from vTask1
    */
    AReceiveFunction();
  }
}
```

Figura obtenida de Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”. Página 296.

Bibliografía:

- Richard Barry. “Mastering the FreeRTOS Real Time Kernel - A Hands-On Tutorial Guide”.
- https://www.freertos.org/