

Usando el Kernel de Tiempo Real FreeRTOS:

(Traducción de la Guía práctica por Richard Barry)

Capítulo 1: Manejo de tareas

1.1 Introducción y alcance de este capítulo:

Introducción a los conceptos de multitareas y pequeños sistemas embebidos:

Los diferentes sistemas multitareas tienen diferentes objetivos. Tomando las estaciones de trabajo y los escritorios como un ejemplo:

- En los viejos tiempos, los procesadores eran caros, entonces la multitarea era usada para permitir a varios usuarios el acceso a un procesador simple. Los algoritmos de programación usados en estos tipos de sistemas fueron diseñados con el objetivo de permitir a cada usuario una “porción justa” del tiempo de procesamiento, acorde a la tarea a realizar.
- En la actualidad, el poder de procesamiento se ha disminuido su costo, entonces, cada usuario puede acceder a uno o más procesadores. Los algoritmos de programación en estos tipos de sistemas son diseñados para permitir que los usuarios puedan correr múltiples aplicaciones en forma simultánea, sin que la computadora deje de responder. Por ejemplo, un usuario puede correr un procesador de texto, una hoja de cálculos, responder emails y navegar por internet, todo esto al mismo tiempo, esperando que cada programa responda adecuadamente a las entradas solicitadas en cada momento.

El procesamiento en una computadora de escritorio se puede clasificar como de “tiempo real suave” (Soft Real Time). Para asegurar la mejor experiencia al usuario la computadora debería responder a cada entrada en un tiempo límite dado, pero una respuesta que se produce después de este tiempo límite es aceptable, no convierte a la computadora en inutilizable.

La multitarea en sistemas embebidos de tiempo real es conceptualmente similar a la multitarea en una computadora de escritorio, en el punto que describe múltiples hilos de ejecución usando un solo procesador. Sin embargo, los objetivos de los sistemas embebidos de tiempo real son un poco distintos a los objetivos de las computadoras de escritorio, especialmente cuando se espera que éstos provean un comportamiento de “Tiempo Real Duro” (Hard Real Time).

El comportamiento de Tiempo Real Duro **debe** responder dentro de un tiempo límite establecido. Si falla y demora más tiempo, esto significará una falla absoluta del sistema. El mecanismo de disparo de un Airbag es un ejemplo claro de esto. Éste debe desplegarse luego de un cierto tiempo después del impacto, si la respuesta se demora, el conductor podría sufrir heridas serias, que se hubieran evitado si el sistema respondía correctamente.

La mayoría de los sistemas embebidos implementan una mezcla de ambos comportamientos, duros y blandos.

Una nota acerca de la terminología:

En FreeRTOS cada hilo de ejecución se denomina “tarea” (Task). No hay un consenso absoluto en cuanto a terminologías en la comunidad de sistemas embebidos, pero yo prefiero la palabra “tareas” a “hilos” dado que un hilo puede tener un significado más específico dependiendo en su previa experiencia.

Alcance:

Este capítulo tiene como objetivo darle a los lectores una buena comprensión a cerca de:

- Cómo FreeRTOS asigna tiempos de procesamiento a cada tarea en una aplicación.
- Cómo FreeRTOS elige cuál tarea debería ejecutarse en cualquier tiempo dado.
- Cómo la prioridad relativa de cada tarea afecta en el comportamiento del sistema.
- Los estados en los que una tarea se puede encontrar.

Además espero que los lectores puedan obtener una buena comprensión sobre:

- Cómo implementar tareas.
- Cómo crear una o más instancias de una tarea.
- Cómo utilizar el parámetro de tarea.
- Cómo cambiar la prioridad de una tarea que ya ha sido creada.
- Cómo borrar una tarea.
- Cómo implementar procesamiento periódico.
- Cuándo se ejecutara la “tarea ociosa” (Idle Task) y cuando puede ser usada.

Los conceptos presentados en este capítulo son fundamentales para comprender como usar FreeRTOS y cómo se comportan las aplicaciones de FreeRTOS, por eso es que este es el capítulo más detallado en el libro.

1.2 Funciones de Tareas:

Las tareas son implementadas como funciones en C. Lo único particular acerca de ellas es su prototipo, el cual debe retornar vacío y tomar parámetro de puntero vacío. El prototipo se muestra en el Listado 1.

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

Listado 1 – Prototipo de la función de tarea.

Cada tarea es un pequeño programa. Tiene un punto de entrada, normalmente correrá en forma permanente en un lazo infinito, y nunca saldrá. La estructura de una tarea típica se muestra en el Listado 2.

Las tareas de FreeRTOS **no deben** poder retornar de su función de implementación en ningún caso, no deben contener una declaración de retorno y no se debe permitir que se ejecute pasado el final de la función.

Si una tarea no se requiere más, debe ser explícitamente borrada. Esto también se demuestra en el Listado 2.

Una sola definición de una función de tarea puede usarse para crear cualquier número de tareas. Cada tarea creada es una instancia separada de ejecución, con su propia pila y su propia copia de variables definida en la tarea misma.

```
void ATaskFunction( void *pvParameters )
{
    /* Las variables pueden ser declaradas igual que para las funciones normales. Cada instancia de una tarea
    creada usando esta función tendrá su propia copia de la variable iVariableExample. Esto no sería verdadero
    si la variable fuese declarada como estática (static), en tal caso, solo una copia de la variable existirá, y esta
    copia sería compartida con cada instancia creada por la tarea */
    int iVariableExample = 0;

    /* Una tarea normalmente será implementada como un lazo infinito */
    for ( ;; )
    {
        /* El código a implementar por la función irá acá. */

    }

    /* La tarea nunca debería salir del lazo anterior, y debería ser borrada antes del final de esta función.
    El parámetro NULL pasado a la función vTaskDelete() indica que la tarea a ser borrada es esta.
    */
    vTaskDelete( NULL);
}
```

Listado 2 - Estructura de una función de tarea típica.

1.3 Aplicaciones de mayor nivel:

Una aplicación puede consistir en muchas tareas. Si el micro controlador corriendo la aplicación sólo contiene un núcleo, entonces solo una tarea puede estar ejecutándose en un tiempo dado. Esto implica que una tarea puede existir en dos estados, “Ejecutándose” o “No ejecutándose”. Consideraremos este modelo simplificado en primer lugar, pero teniendo en mente que más adelante veremos que el estado “No ejecutándose” en realidad tiene más sub estados.

Cuando una tarea se encuentra ejecutándose, el procesador está ejecutando su código. Cuando una tarea no está ejecutándose, la tarea está inactiva, su estado ha sido salvado y está listo para retomar su ejecución la próxima vez que el planificador (scheduler) decida que debe entrar en el estado de ejecución. Cuando una tarea resume su ejecución, lo hace desde la misma instrucción que estaba por ejecutar antes que pasase al estado de no ejecución.

Todas las tareas que no se encuentran ejecutándose están en el estado de no ejecución. Solo una tarea puede estar ejecutándose en cualquier momento.

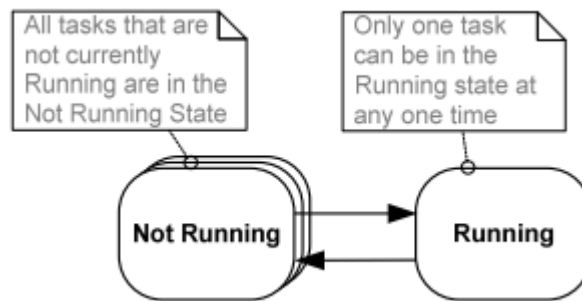


Figura 1 – Estados y Transiciones principales de las tareas.

Cuando una tarea que transiciona del estado de no ejecución a ejecutarse se dice que ha sido “switched in” o “swapped in”. Inversamente, una tarea que transiciona del estado de ejecución al estado de no ejecución se dice que ha sido “switched out” o “swapped out”. El planificador de FreeRTOS es la única entidad capaz de intercambiar los estados de las tareas, es decir, los pasajes de switch in y out.

1.4 Creando una tarea:

xTaskCreate() . Función API (Interface de programación de aplicación):

Las tareas son creadas usando la función de FreeRTOS xTaskCreate(). Probablemente ésta sea la más compleja de todas las funciones API, desafortunadamente es la primera que veremos, pero las tareas deben ser estudiadas primero, dado que son los componentes más importantes de un sistema multitarea. Todos los ejemplos de este libro hacen uso de la función xTaskCreate(), por lo que hay muchos ejemplos como referencia.

Apéndice 5: Describe los tipos de datos y los nombres convencionales usados.

```
portBASE_TYPE xTaskCreate( pdTASK_CODE pvTaskCode,
                           const signed portCHAR * const pcName,
                           unsigned portSHORT usStackDepth,
                           void *pvParameters,
                           unsigned portBASE_TYPE uxPriority,
                           xTaskHandle *pxCreatedTask
                           );
```

Listado 3 – Prototipo de la función API xTaskCreate().

Tabla 1 xTaskCreate() parámetros y valores de retorno

Nombre del Parámetro / Valor de Retorno	Descripción
pvTaskCode	Este parámetro es simplemente un puntero a la función (en realidad solo al nombre de la función) que implementa la tarea.
pcName	Un nombre descriptivo de la tarea. Este nombre no es usado por FreeRTOS, es incluido únicamente como una ayuda para el debug, para identificar la tarea más fácilmente.
usStackDepth	<p>Este parámetro le dice al Kernel que tan grande debe ser la pila para esta función. Especifica el número de palabras que la pila puede tener, no el número de bytes.</p> <p>Por ejemplo, si la pila tiene un ancho de 32 y el usStackDepth es 100, luego, 400 bytes serán asignados (100*4bytes). Este valor no debe exceder el valor máximo que puede ser contenido en una variable del tipo size_t. El tamaño de la pila usado por la tarea ociosa está definido por la constante "configMINIMAL_STACK_SIZE". Este es el mínimo espacio recomendable a asignar para cualquier aplicación de FreeRTOS. De ser necesario una pila de mayor tamaño, se deberá asignar un valor mayor. No hay un modo sencillo de calcular el tamaño de la pila requerido para una aplicación, si bien es</p>

	posible calcularlo, la mayoría de los usuarios solo asignan el valor que consideran razonable, y luego usando las características que brinda FreeRTOS, se aseguran que sea el valor adecuado, y no se está desperdiciando RAM. El capítulo 6 contiene información sobre como consultar el espacio de pila que usa una tarea.
pvParameters	Las funciones de las tareas aceptan un parámetro del tipo puntero void. El valor asignado a pvParameters será el valor que se le pasa a la tarea. Algunos ejemplos en este documento demuestran cómo se puede usar este parámetro.
uxPriority	Define la prioridad con la cual la tarea se ejecutará. Las prioridades se pueden asignar desde 0 (valor más bajo), hasta el valor "configMAX_PRIORITIES-1" (valor más alto). "configMAX_PRIORITIES" es una constante definida por el usuario. Con el fin de evitar desperdiciar memoria RAM, se debería usar el menor número de niveles de prioridad posible.
pxCreatedTask	Este parámetro sirve como handle de la tarea creada. El handle se utiliza para referenciar a esta tarea en llamados de otras funciones API, por ejemplo, para cambiar el nivel de prioridad o borrar esta tarea. Si no se utilizará el handle, se lo puede setear como NULL.
Valor Retornado	Hay dos posibles valores: 1. pdTRUE, que indica que la tarea se creo correctamente. 2. errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY, esto indica que la tarea no pudo ser creada porque no había suficiente memoria RAM para asignarle a la pila y las estructuras de la tarea. El capítulo 5 ofrece más información sobre el manejo de memoria

Ejemplo 1. Creación de Tareas:

Apéndice 1: contiene información sobre las herramientas requeridas para construir ejemplos de proyectos.

Este ejemplo demuestra los pasos necesarios para crear dos simples tareas, y luego comenzar la ejecución de las mismas. Las tareas periódicamente imprimen una cadena, usando un lazo nulo para crear un periodo de retardo. Ambas tareas son creadas con la misma prioridad y son idénticas salvo por la cadena que imprimen. Ver Listado 4 y Listado 5 por sus respectivas implementaciones.

```
void vTask1 ( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running\r\n";
    volatile unsigned long ul;

    /* Como en la mayoría de las tareas, se implementa en un lazo infinito. */
    for(;;)
    {
        /* Imprime el nombre de la tarea. */
        vPrintString ( pcTaskName );
        /* Retarda un periodo. */
        for ( ul=0; ul < mainDELAY_LOOP_COUNT; ul++)
        {
            /* Este lazo es solo una cruda implementación de un retardo. Los próximos ejemplos reemplazarán este retardo por una función apropiada de delay/sleep */
        }
    }
}
```

Listado 4 - Implementación de la primera tarea usada en el Ejemplo 1.

```
void vTask2 ( void *pvParameters )
{
    const char *pcTaskName = "Task 2 is running\r\n";
    volatile unsigned long ul;

    /* Como en la mayoría de las tareas, se implementa en un lazo infinito. */
    for(;;)
    {
        /* Imprime el nombre de la tarea. */
        vPrintString ( pcTaskName );
        /* Retarda un periodo. */
        for ( ul=0; ul < mainDELAY_LOOP_COUNT; ul++)
        {
```

```
/* Este lazo es solo una cruda implementación de un retardo. Los próximos ejemplos reemplazarán este retardo por una función apropiada de delay/sleep */
```

```
    }
}
}
```

Listado 5 - Implementación de la segunda tarea usada en el Ejemplo 1.

La función main() simplemente crea las tareas antes de arrancar el scheduler. Ver el Listado 6 para ver su implementación.

```
int main ( void )
{
    /* Crea una de las dos tarea. Notar que una aplicación real debería chequear el valor de
    retorno de xTaskCreate(), para verificar si la tarea se creó correctamente. */
    xTaskCreate (    vTask1, /*Puntero a la función que implementa la tarea. */
                  "Task 1", /*Nombre de la función, esto es solo para facilitar el debug.*/
                  1000,    /*Tamaño de la pila, la mayoría de las aplicaciones pequeñas de
                           micro controladores usaran un valor mucho menor a este. */
                  NULL,    /*No usaremos el parámetro de tarea. */
                  1,       /*La aplicación se ejecutará con prioridad 1. */
                  NULL ); /*No usaremos el handle de la tarea. */

    /*Creamos la otra tarea exactamente igual. */
    xTaskCreate ( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /*Arranca el Scheduler para que las tareas puedan ejecutarse. */
    vTaskStartScheduler();

    /*Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va
    a estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces es probable
    que la memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5
    provee más información sobre el manejo de memoria. */
    for ( ; ; );
}
```

Listado 6 - Creación de tareas del Ejemplo 1.

Ejecutar este ejemplo produce la siguiente salida, mostrada en la Figura 2.

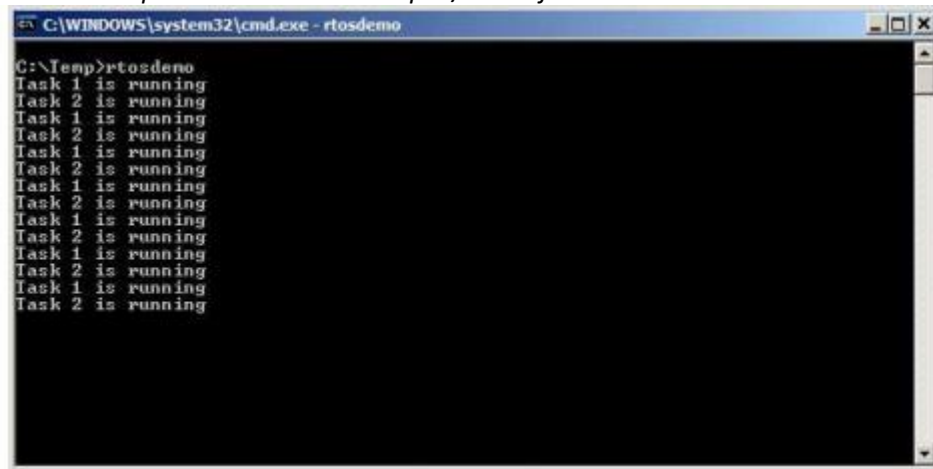


Figura 2 – Salida Producida cuando es ejecutado el Ejemplo 1.

La Figura 2 muestra las dos tareas aparentando estar ejecutándose simultáneamente, pero ambas tareas se están ejecutando en el mismo procesador, por lo tanto no pueden estar ejecutándose al mismo tiempo. En realidad, ambas tareas están entrando y saliendo rápidamente del estado de ejecución. Ambas tareas se ejecutan con la misma prioridad, entonces, comparten el tiempo de procesamiento del único procesador. Su verdadero patrón de ejecución se muestra en la Figura 3.

La flecha a lo largo de la parte inferior de la Figura 3 muestra el paso del tiempo desde t_1 en adelante. Las líneas coloreadas muestran cual tarea está ejecutándose en cada momento, por ejemplo, la tarea 1 se ejecuta entre los tiempos t_1 y t_2 .

Solo una tarea puede estar ejecutándose en cualquier momento, entonces, mientras una tarea entra al estado de ejecución (switched in), la otra sale del estado de ejecución para entrar al de no ejecución (switched out).

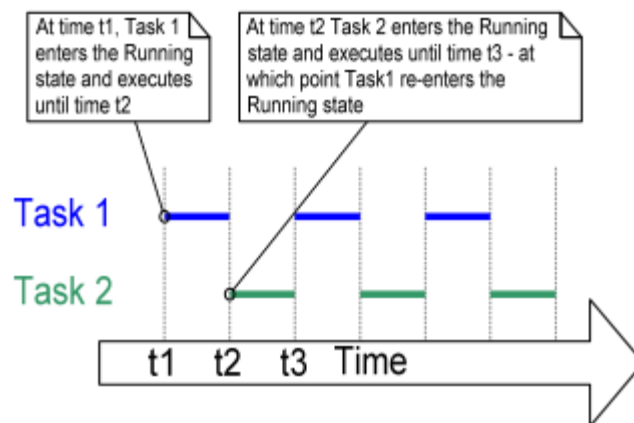


Figura 3 - Patrón Real de ejecución de las dos tareas del Ejemplo 1.

El ejemplo 1 creó dos tareas desde adentro del `main()`, antes de arrancar el scheduler. También es posible crear una tarea desde adentro de otra tarea. Podríamos haber creado la Tarea 1 desde el `main()`, y luego crear la Tarea 2 desde la Tarea1. Para hacer esto, nuestra Tarea 1 cambiará como se muestra en el Listado 7. La Tarea 2 no se creará hasta después que arranque el scheduler, pero la salida producida será exactamente la misma que la anterior.

```
void vTask1 ( void *pvParameters )
{
    const char *pcTaskName = "Task 1 is running \r\n";
    volatile unsigned long ul;

    /* Si este código se está ejecutando, entonces el scheduler ya se ha iniciado. Se crea ahora
    la otra tarea antes de entrar en el lazo infinito. */
    xTaskCreate ( vTask2, "Task2", 1000, NULL, 1, NULL );

    for ( ;; )
    {
        /* Imprime el nombre esta tarea. */
        vPrintString( pcTaskName );

        /* Retardo por un periodo. */
        for ( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ );
        {
            /* Este lazo es solo una cruda implementación de un retardo. Los próximos
            ejemplos reemplazarán este retardo por una función apropiada de
            delay/sleep */
        }
    }
}
```

Listado 7 - Creación de una tarea desde adentro de otra tarea, luego de haber iniciado el scheduler

Ejemplo 2. Uso del Parámetro de Tarea:

Las dos tareas creadas en el Ejemplo 1 eran prácticamente iguales, la única diferencia era el texto que imprimían. Esta duplicación puede eliminarse creando una misma función de implementación para las dos tareas. El parámetro de tarea puede usarse para pasarle a cada tarea la cadena que esa instancia debería imprimir.

El Listado 8 contiene el código de la única función de Tarea (vTaskFunction) usada en el Ejemplo 2. Esta única función reemplaza las dos funciones de tarea (vTask1 y vTask2) usadas en el Ejemplo 1. Notar como el parámetro de tarea es “casteado” como un puntero de caracter, para obtener la cadena que la tarea debe imprimir.

```
void vTaskFunction ( void *pcParameters )
{
    char *pcTaskName;
    volatile unsigned long ul;

    /* La cadena a imprimir es pasada por medio del parámetro. Se “castea” como puntero de
    caracter. */
    pcTaskName = ( char* ) pvParameters;

    /* Como en la mayoría de las tareas, se implementa dentro de un lazo infinito. */
    for(;;)
    {
        /* Imprime el nombre de esta tarea. */
        vPrintString( pcTaskName );

        /* Retardo por un periodo. */
        for ( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ );
        {
            /* Este lazo es solo una cruda implementación de un retardo. Los próximos
            ejemplos reemplazarán este retardo por una función apropiada de
            delay/sleep”
        }
    }
}
```

Listado 8 – La única función de implementación usada en el Ejemplo 2.

Usando el Kernel de Tiempo Real FreeRTOS – Cap 1 / Manejo de Tareas

Aunque ahora solo hay una función de implementación (vTaskFunction), más de una instancia de la tarea definida pueden ser creadas. Cada instancia creada se ejecutará independientemente bajo el control del scheduler de FreeRTOS.

El parámetro pvParameters a la función xTaskCreate(), es usado para pasarle la cadena de texto como se muestra en el Listado 9.

```
/* Define las cadenas que serán pasadas como parámetros de tarea. Estas constantes son constantes defini-
das estáticas, y no en la pila, para asegurar que permanezcan válidas cuando la tarea se esté ejecutando. */
```

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

```
int main( void )
{
```

```
    /* Crea una de las dos tareas. */
```

```
    xTaskCreate (    vTask1,                /*Puntero a la función que implementa la tarea. */
                    "Task 1",              /*Nombre de la función, esto es solo para facilitar el
                                           debug.*/
                    1000,                  /*Tamaño de la pila, la mayoría de las aplicaciones pe-
                                           queñas de micro controladores usaran un valor mucho
                                           menor a este. */
                    (void*)pcTextForTask1, /*No usaremos el parámetro de tarea. */
                    1,                     /*La aplicación se ejecutará con prioridad 1. */
                    NULL );                /*No usaremos el handle de la tarea. */
```

```
    /*Crea la otra tarea de igual manera. Notar que esta vez múltiples tareas están siendo creadas
    desde la misma implementación de tarea (vTaskFunction). Solo el valor pasado en el parámetro es
    distinto. Dos instancias de una misma tarea están siendo creadas.*/
```

```
    xTaskCreate( vTaskFunction, "Task2", 1000, (void*)pcTextForTask2,1,NULL );
```

```
    /* Inicio el scheduler para que nuestras tareas se ejecuten. */
```

```
    vTaskStartScheduler();
```

```
/*Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va a
estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces es probable que la
memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5 provee más informa-
ción sobre el manejo de memoria. */
for ( ; ; );
}
```

Listado 9 - Función main() para el ejemplo 2.

La salida impresa en pantalla por el Ejemplo 2 es exactamente igual a la del Ejemplo 1, en la Figura 2.

1.5 Prioridad de Tareas:

El parámetro `uxPriority` de la función API `xTaskCreate()` asigna una prioridad inicial a la tarea que está siendo creada. La prioridad puede ser cambiada luego que el scheduler ha iniciado, usando la función API `vTaskPrioritySet()`.

El máximo número de prioridades disponibles es seteado por la aplicación definida `configMAX_PRIORITIES` en la `FreeRTOSConfig.h`. FreeRTOS no limita el valor máximo de esta constante, pero cuanto más alto sea el valor de `configMAX_PRIORITIES` mayor será la RAM utilizada por el Kernel, por eso es aconsejable mantener este valor al mínimo necesario.

FreeRTOS no impone restricciones sobre como asignar las prioridades a las tareas. Cualquier número de tareas puede compartir la misma prioridad, asegurando una máxima flexibilidad de diseño. Puede asignarse una prioridad diferente a cada tarea, pero esta restricción no es impuesta de ninguna manera.

Valores numéricos bajos indican bajos niveles de prioridad, siendo la prioridad 0 la prioridad más baja posible. El rango disponible de prioridades es desde 0 hasta `configMAX_PRIORITIES-1`.

El scheduler asegurará que siempre la tarea de mayor prioridad que esté disponible para ser ejecutada será la que entre en el estado de ejecución. Cuando más de una tarea con el mismo nivel de prioridad esté disponible para ser ejecutada, el scheduler alternará cada tarea, ejecutando una a la vez. Este es el comportamiento observado en los ejemplos hasta ahora vistos, donde ambas tareas fueron creadas con los mismos niveles de prioridad, y siempre estaban disponibles para ser ejecutadas. Cada tarea se ejecutaba durante una porción de tiempo, y salía del estado de ejecución al concluir este tiempo. En la Figura 3 el tiempo entre `t1` y `t2` equivale a una porción de tiempo.

Para poder ejecutar la siguiente tarea el scheduler debe ejecutarse al finalizar cada porción de tiempo. Una interrupción periódica llamada Interrupción Tick es usada para éste propósito. La duración de la porción de tiempo es efectivamente seteada por la frecuencia de la Interrupción Tick, la cual es configurada por la constante `configTICK_RATE_HZ`, en `FreeRTOSConfig.h`. Por ejemplo, si

Usando el Kernel de Tiempo Real FreeRTOS – Cap 1 / Manejo de Tareas

configTICK_RATE_HZ es seteado en 100(HZ), la porción de tiempo será de 100ms. La Figura 3 puede ser expandida para mostrar la ejecución del scheduler en la secuencia. Esto se observa en la Figura 4.

Notar que FreeRTOS API siempre llama a la Interrupción Tick. La constante portTICK_RATE_MS es provista para permitir demoras de tiempo expresadas en milisegundos y no en Interrupciones de Tick. La resolución disponible depende de la frecuencia de los Ticks (Interrupciones de Tick).

La "Cuenta de Ticks" es el número total de Ticks que han ocurrido desde que el scheduler inició, asumiendo que esta variable no ha desbordado. Las aplicaciones de usuario no necesitan considerar desbordes cuando especifican demoras de tiempo, dado que la consistencia temporal es manejada por el Kernel.

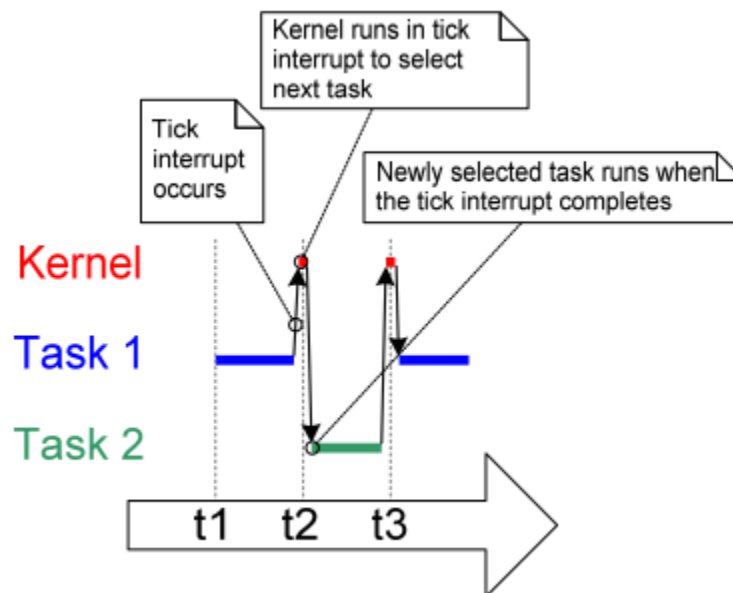


Figura 4 – Secuencia de ejecución expandida para mostrar la Interrupción Tick.

En la figura 4 la línea roja indica cuando el kernel mismo está ejecutándose. Las flechas negras muestran la secuencia de ejecución desde una tarea a la interrupción, luego desde la interrupción de vuelta hacia una nueva tarea.

Ejemplo 3. Experimentando con las prioridades:

El scheduler asegura que siempre la tarea de mayor prioridad que esté disponible para ser ejecutada será la elegida para entrar en el estado de ejecución. Hasta ahora en nuestros ejemplos se han creado dos tareas con las mismas prioridades, entonces, ambas entraban y salían del estado de ejecución en forma alternada. En este ejemplo observaremos que ocurre cuando cambiamos la prioridad de una de las dos tareas del Ejemplo 2. Ahora la primera tarea será creada con prioridad 1, y la segunda tarea con prioridad 2. El código para crear las tareas se muestra en el Listado 10. La única función que implementa ambas tareas no ha cambiado, y sigue imprimiendo periódicamente una cadena usando un lazo nulo para crear un retardo.

```
/* Se define la cadena que será pasada mediante el parámetro de tarea. Éstas son definidas como cons-
tantes y no en la pila, para asegurar que permanezcan válidas cuando la tarea se está ejecutando. */
```

```
static const char *pcTextForTask1 = "Task 1 is running \r\n";
```

```
static const char *pcTextForTask2 = "Task 2 is running \r\n";
```

```
int main (void)
```

```
{
```

```
    /* Creo la primera tarea con prioridad 1. La prioridad es el anteúltimo parámetro. */
```

```
    xTaskCreate ( vTaskFunction, "Task1", 1000, (void)*pcTextForTask1, 1, NULL );
```

```
    /* Creo la segunda tarea con prioridad 2. */
```

```
    xTaskCreate ( vTaskFunction, "Task2", 1000, (void)*pcTextForTask2, 2, NULL );
```

```
    /* Inicio el scheduler para que las tareas comiencen a ejecutarse. */
```

```
    vTaskStartScheduler();
```

```
    return 0;
```

```
}
```

Listado 10 – Creando dos tareas con diferentes prioridades.

La salida producida por el Ejemplo 3 se muestra en la Figura 5.

Usando el Kernel de Tiempo Real FreeRTOS – Cap 1 / Manejo de Tareas

El scheduler elegirá siempre la tarea de mayor prioridad que esté disponible para ser ejecutada. La tarea 2 tiene mayor prioridad que la tarea 1, y siempre está disponible para ser ejecutada, por lo tanto, la tarea 2 es la única tarea que se ejecuta. Como la tarea 1 nunca entra al estado de ejecución, nunca imprime su cadena.

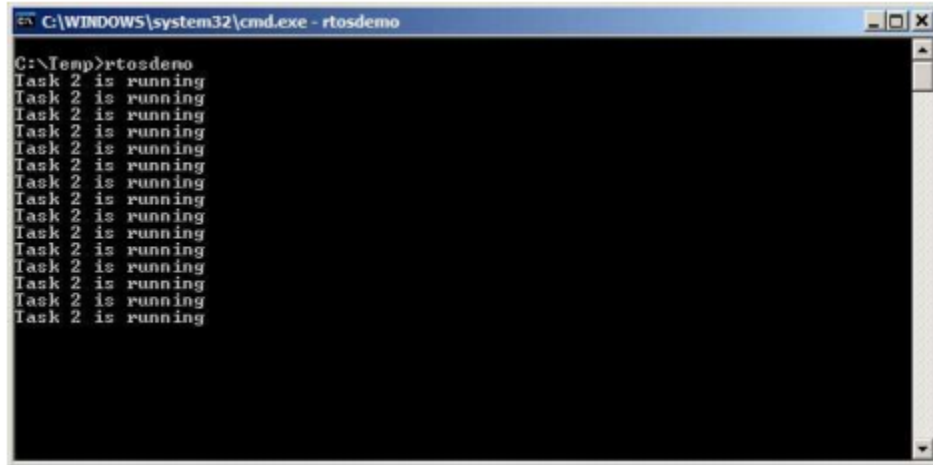


Figura 5 – Ejecutando dos tareas con diferentes prioridades.

La tarea 2 está siempre disponible para ser ejecutada porque nunca tiene que esperar que nada ocurra.

La tarea 2 estará siempre imprimiendo su cadena o en el lazo nulo que genera el retardo.

La Figura 6 muestra la secuencia de ejecución del Ejemplo 3.

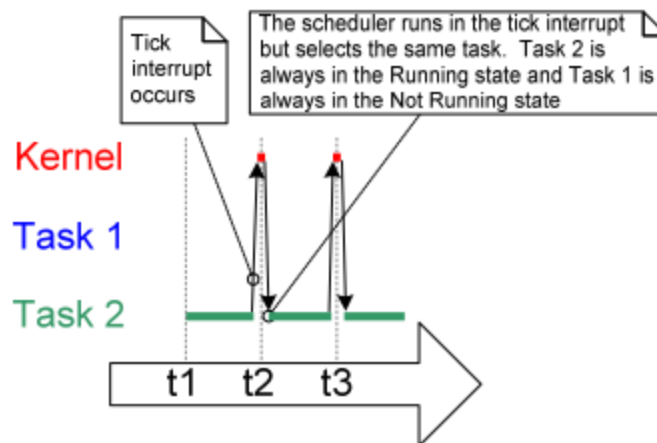


Figura 6 – Patrón de Ejecución cuando una tarea tiene prioridad mayor que la otra.

Quando la Interrupción Tick ocurre, el scheduler deja de ejecutar la tarea 2, pero luego vuelve a seleccionar a la misma tarea para ser ejecutada, dado que es la que tiene mayor prioridad. Por lo tanto la tarea 2 estará siempre en estado de ejecución mientras que la tarea 1 nunca se estará ejecutando.

1.6 Expandiendo el estado de “No Ejecución”:

Hasta ahora, en los ejemplos presentados, cada tarea creada ha podido ejecutar su código sin tener que esperar que ocurriese ningún evento. Por lo tanto, siempre se encontraban disponibles para ser ejecutadas. Este tipo de tareas de “procesamiento continuo” tiene una utilidad limitada, porque solo pueden ser creadas en la prioridad más baja. Si se ejecutan con otra prioridad, impedirán que tareas de menor prioridad entren en el estado de ejecución.

Para hacer nuestras tareas en verdad útiles, necesitamos una forma de permitirles ser manejadas por eventos. Una tarea manejada por un evento solo tiene trabajo (proceso) que realizar luego que haya ocurrido el evento que la dispara, y no puede entrar en el estado de ejecución antes que dicho evento ocurra. El scheduler siempre selecciona la tarea de mayor prioridad que se encuentre *disponible* para ser ejecutada. Si la tarea de mayor prioridad no se encuentra disponible para ser ejecutada, el scheduler seleccionará la siguiente tarea con mayor prioridad que si se encuentre disponible. Usar tareas manejadas por eventos significa entonces que las tareas pueden ser creadas en diferentes prioridades, sin que la tarea de mayor prioridad impida que las tareas de menor prioridad alguna vez se ejecuten.

El Estado Bloqueado:

Una tarea que está esperando que ocurra un evento se dice que se encuentra en estado “Bloqueado”, el cual es un sub estado del estado de No Ejecución.

Las tareas pueden entrar en el estado bloqueado para esperar por dos tipos distintos de eventos:

1. Eventos Temporales: el evento consta de un cierto retardo de tiempo que debe cumplirse o un tiempo absoluto que debe alcanzarse. Por ejemplo, una tarea puede bloquearse para esperar que pasen 10 milisegundos.
2. Eventos de Sincronización: donde el evento es originado desde otra tarea o interrupción. Por ejemplo, una tarea puede bloquearse para esperar que ingresen datos en una cola. Los eventos de sincronización cubren un amplio rango de tipos de eventos.

En FreeRTOS las colas, los semáforos binarios, los semáforos de conteo, los semáforos recursivos y lo-mutex pueden ser usados para crear eventos de sincronización. Los capítulos 2 y 3 cubren estos temas con más detalle.

Es posible que una tarea se bloquee con un evento de sincronización, pero con un determinado tiempo de espera. Por ejemplo, una tarea que debe esperar un máximo de 10 milisegundos a que llegue un dato en una cola. La tarea dejará de estar bloqueada si llega un dato antes de los 10 milisegundos, o luego de que pasen los 10 milisegundos y ningún dato haya llegado.

El Estado Suspendido:

El estado suspendido también es un sub estado del estado de No Ejecución. Las tareas en el estado suspendido no están disponibles para que el scheduler las ejecute. La única manera que una tarea entre en este estado es mediante un llamado a la función API `vTaskSuspend()`, y la única manera de salir de este estado es mediante un llamado a la función `vTaskResume()` o `vTaskResumeFromISR()`. La mayoría de las aplicaciones no usan este estado.

El Estado Disponible:

Las tareas que no se están ejecutando pero no están bloqueadas ni suspendidas están en estado disponible. Pueden ser ejecutadas, están “listas” para hacerlo, pero aún no fueron elegidas para ser ejecutadas.

Completando el Diagrama de Transiciones de estados:

La Figura 7 expande el diagrama de estados anterior, incluyendo todos los sub estados del estado de no ejecución. Las tareas creadas en los ejemplos hasta ahora no han usado los estados bloqueados o suspendidos, solo han alternado entre el estado disponible y el estado en ejecución.

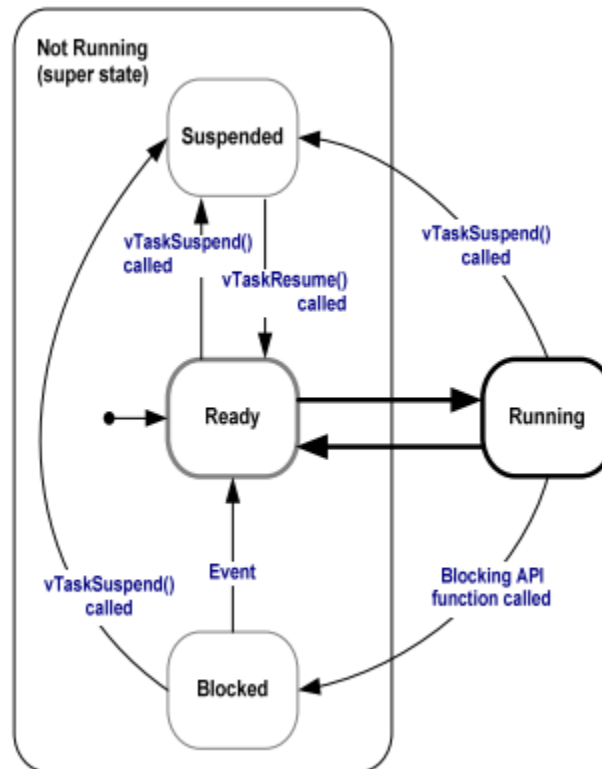


Figura 7 – Diagrama completo de estados y transiciones.

Nota: Suspend = Suspendido, Ready = Listo o Disponible, Blocked = Bloqueado, Running = En Ejecución.

Ejemplo 4. Usando el estado bloqueado para crear un retardo:

Todas las tareas creadas en los ejemplos presentados hasta ahora eran periódicas, tenían un retardo, imprimían una cadena, y volvían una vez más al estado de retardo, y así sucesivamente. El retardo ha sido generado muy crudamente, usando un lazo nulo (la tarea incrementaba una variable hasta alcanzar un valor prefijado). El ejemplo 3 mostraba claramente la desventaja de este método. Mientras se ejecutaba el lazo nulo, la tarea permanecía en estado disponible, impidiendo que la otra tarea, de menor prioridad, se ejecute.

Mientras la tarea se encuentra en el lazo nulo, no realiza ninguna acción, pero utiliza todo el tiempo de procesamiento, por lo que se están desperdiciando ciclos de procesamiento. El ejemplo 4 corrige este comportamiento reemplazando el lazo nulo por la función API `vTaskDelay()`, cuyo prototipo se muestra en el Listado 11. La nueva definición de la tarea se muestra en el Listado 12.

`vTaskDelay()` coloca a la tarea donde es ejecutado en estado bloqueado, por un número dado de interrupciones Tick. Mientras la tarea se encuentra en estado bloqueado, esta no consume tiempo de procesamiento, por lo que este tiempo es solo usado por la tarea que si tiene trabajo que realizar.

```
void vTaskDelay( portTickType xTicksToDelay );
```

Listado 11 – Prototipo de la Función API `vTaskDelay()`.

Tabla 2. Parámetros de la función vTaskDelay()

Nombre del parámetro	Descripción
xTicksToDelay	<p>Número de interrupciones de Tick que la tarea que la ejecuta debe permanecer en estado bloqueado antes de volver a entrar en ejecución. Por ejemplo, si una tarea llama a vTaskDelay(100), mientras la cuenta de ticks era 10000, entonces ingresa inmediatamente en estado bloqueado y permanece en ese estado hasta que la cuenta de ticks alcance el valor 10100.</p> <p>La constante portTICK_RATE_MS se emplea para convertir milisegundos en ticks.</p>

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

    /* La cadena a imprimir es pasada mediante el parámetro. Se castea a puntero de caracter */
    pcTaskName = ( char * ) pvParameters;

    /* Como en la mayoría de las tareas, se implementa dentro de un lazo infinito. */
    for( ;; )
    {
        /* Imprime el nombre de esta tarea. */
        vPrintString( pcTaskName );

        /*Retrasa por un periodo. El llamado a la función vTaskDelay() coloca a la tarea en estado
        bloqueado hasta que el periodo haya concluido. Este periodo se especifica en Ticks, pero la
        constante portTICK_RATE_MS puede usarse para especificar este valor en milisegundos.
        Para este ejemplo, el periodo durará 250 milisegundos. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Listado 12 – Código fuente para el ejemplo donde el lazo nulo fue reemplazado por un llamado a la función vTaskDelay().

Aunque las dos tareas son creadas con distintas prioridades, ahora ambas se ejecutarán. La salida del Ejemplo 4 es mostrada en la Figura 8, confirmando este comportamiento.

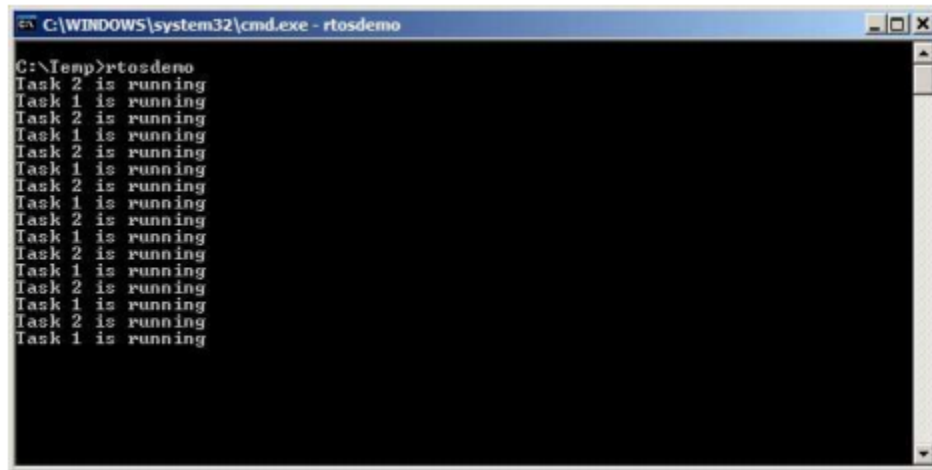


Figura 8 – Salida producida cuando el Ejemplo 4 es ejecutado.

La secuencia de ejecución mostrada en la Figura 9 explica por qué ambas tareas se ejecutan, aunque tengan distintas prioridades. La ejecución del propio kernel es omitida para simplificar el diagrama. La tarea ociosa es creada automáticamente cuando el scheduler es iniciado para asegurar que siempre haya una tarea disponible para ejecutar (por lo menos una tarea en estado disponible). La sección 1.7 describe la tarea ociosa con mayor detalle.

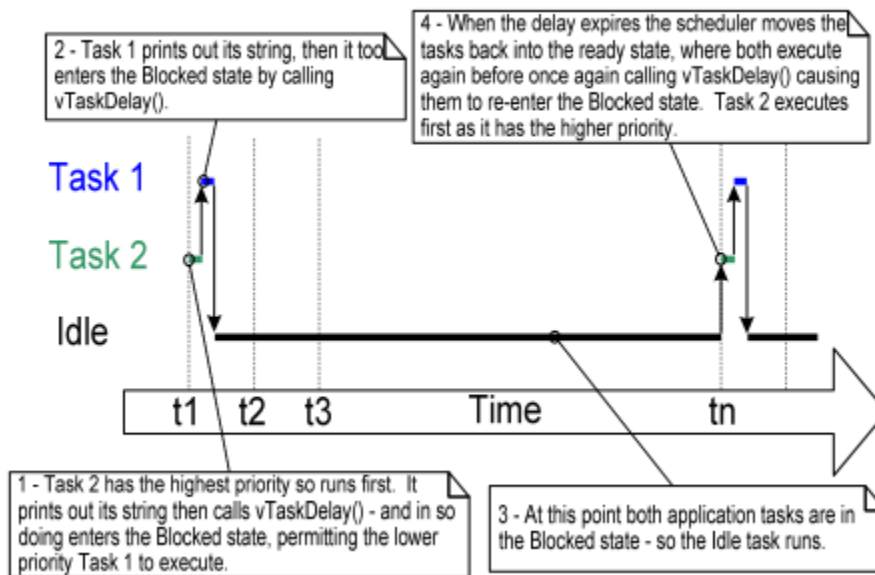


Figura 9 – Secuencia de Ejecución cuando las tareas usan la función vTaskDelay() en vez de un lazo nulo.

- 1- La tarea 2 tiene la prioridad más alta, por lo tanto, se ejecuta primero. Imprime su cadena y luego llama a la función `vTaskDelay()`, al hacer esto, se bloquea, permitiendo que la tarea 1, de menor prioridad, se ejecute.
- 2- La tarea 1 imprime su cadena y también se bloquea al ejecutar la función `vTaskDelay()`.
- 3- En este punto, ambas tareas están bloqueadas, por lo tanto se ejecuta la tarea ociosa.

Usando el Kernel de Tiempo Real FreeRTOS – Cap 1 / Manejo de Tareas

- 4- Cuando el periodo de retardo expira, el scheduler mueve la tarea al estado disponible, donde ambas se vuelven a ejecutar y nuevamente llaman a la función `vTaskDelay()`, causando que vuelvan a bloquearse. La tarea 2 siempre se ejecuta primero dado que tiene mayor prioridad.

Solo la implementación de nuestras dos tareas ha cambiado, no su funcionalidad. Comparando la Figura 9 con la Figura 4, se demuestra claramente que el mismo funcionamiento está siendo logrado de un modo mucho más eficiente.

La Figura 4 muestra el patrón de ejecución cuando las tareas están usando un lazo nulo para crear el retardo, entonces las tareas estaban siempre disponibles para ser ejecutadas, y como resultado se utilizaba mucho tiempo de procesamiento. La Figura 9 muestra el patrón de ejecución cuando las tareas se bloquean durante todo el periodo de retardo, entonces, solo se utiliza tiempo de procesamiento cuando realmente tiene trabajo que realizar, en este ejemplo, es simplemente la impresión por pantalla de una cadena.

En la Figura 9 también podemos observar que cada vez que una tarea deja de estar bloqueada, solo se ejecutan por una fracción de tiempo antes de volver a bloquearse. La mayor parte del tiempo ambas tareas están bloqueadas, por lo tanto no hay tareas disponibles, y entonces el scheduler ejecuta la tarea ociosa. La cantidad de tiempo de procesamiento que tiene la tarea ociosa es una medida de la capacidad de procesamiento extra que tiene el sistema.

Las líneas gruesas en la Figura 10 muestran las transiciones de estado realizadas por las tareas del Ejemplo 4, cada tarea pasando alternadamente entre el estado bloqueado y disponible.

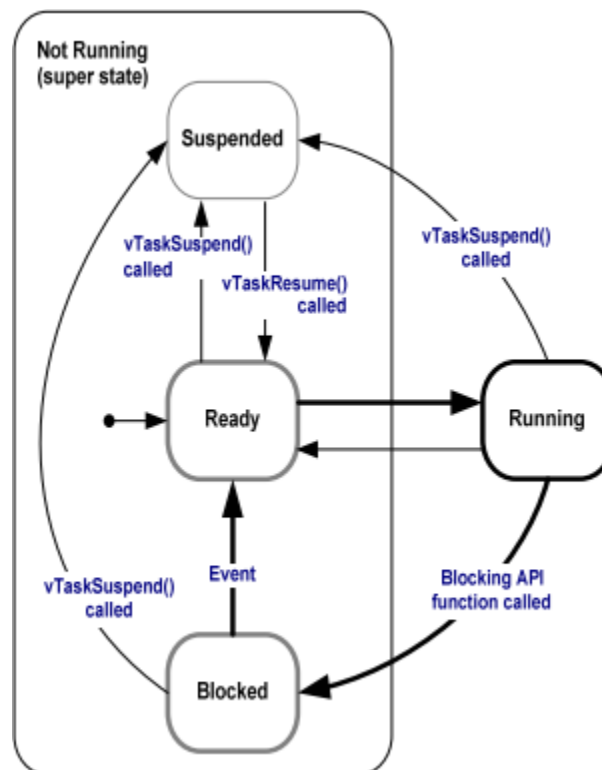


Figura 10 – Las líneas gruesas indican las transiciones de estado realizadas por las tareas del Ejemplo 4

Función API vTaskDelayUntil():

vTaskDelayUntil() es similar a vTaskDelay(). Como se acaba de demostrar, el parámetro la función vTaskDelay() especifica el número de Interrupciones Tick que deben ocurrir entre el llamado a la función vTaskDelay() y que la tarea salga del estado bloqueado. La cantidad de tiempo que la tarea permanece bloqueada es especificado por este parámetro, pero el tiempo real en el cual la tarea deja de estar bloqueada es relativo (depende) del momento en que se llamó a la función vTaskDelay(). En cambio, el parámetro de la función vTaskDelayUntil() especifica el valor exacto que debe tener la cuenta de interrupciones Tick, para que la función pase del estado bloqueado al estado disponible. vTaskDelayUntil() es la función API que debería usarse cuando se requiere un período fijo de ejecución, es decir, cuando uno quiere que su tarea se ejecute periódicamente con una frecuencia dada.

```
void vTaskDelayUntil ( portTickType *pxPreviousWakeTime, portTickType xTimeIncrement);
```

Listado 13 – Prototipo de la Función API vTaskDelayUntil().

Tabla 3. Parámetros de la función vTaskDelayUntil()

Nombre del parámetro	Descripción
pxPreviousWakeTime	<p>Este parámetro se denomina en el supuesto que la función vTaskDelayUntil () está siendo utilizada para implementar una tarea que se ejecuta periódicamente y con una frecuencia fija. En este caso pxPreviousWakeTime contiene el tiempo en que la tarea dejó por última vez el estado bloqueado. Este tiempo se utiliza como un punto de referencia para calcular el próximo momento en que la tarea debe dejar de estar bloqueada.</p> <p>La variable apuntada por pxPreviousWakeTime se actualiza automáticamente dentro de la función vTaskDelayUntil () y normalmente no sería modificada por el código de la aplicación, salvo la primera vez que es inicializada. El Listado 14 muestra cómo se realiza la inicialización</p>
xTimeIncrement	<p>Este parámetro también es denominado en el supuesto que la función vTaskDelayUntil() está siendo usada para implementar una tarea que se ejecuta periódicamente y con una frecuencia fija. La frecuencia se setea con el valor de xTimeIncrement, especificado en Ticks. Nuevamente se puede emplear la constante portTICK_RATE_MS para convertir los Ticks en milisegundos.</p>

Ejemplo 5. Convirtiendo las tareas anteriores para utilizar vTaskDelayUntil():

Las dos tareas creadas en el Ejemplo 4 son tareas periódicas, pero usando la función vTaskDelay() no se garantiza que la frecuencia a la que se ejecutan sea fija, dado que el tiempo en el cual las tareas se desbloquean es relativo al momento en el que llamaron a la función vTaskDelay(). Usando la función vTaskDelayUntil() en vez de vTaskDelay() resolvería este problema potencial.

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    portTickType xLastWakeTime;
    /* La cadena a imprimir es pasada a través del parámetro. Se castea como puntero de carácter. */
    pcTaskName = ( char * ) pvParameters;

    /* La variable xLastWakeTime necesita ser inicializada con el valor actual de la cuenta de interrupciones Tick. Notar que esta es la única vez que la variable es escrita explícitamente. Luego esta variable se actualiza automáticamente dentro de la función vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

    /* Como siempre, se implementa la tarea dentro de un lazo infinito. */
    for( ;; )
    {
        /* Imprime el nombre de la tarea. */
        vPrintString( pcTaskName );

        /* Esta tarea se debería ejecutar exactamente cada 250 milisegundos. Al igual que con la función vTaskDelay(), el tiempo se mide en Ticks, y la constante portTICK_RATE_MS se usa para convertir los milisegundos en Ticks. xLastTimeWake es automáticamente actualizada en la función vTaskDelayUntil() por lo que no hay que actualizarla explícitamente en el código de la tarea. */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

Listado 14 – Implementación de las tareas usando vTaskDelayUntil().

La salida producida por el Ejemplo 5 es exactamente la misma que la mostrada en la Figura 8, para el Ejemplo 4.

Ejemplo 6. Combinando estados bloqueados y no bloqueados:

Los ejemplos anteriores han examinado el comportamiento de tareas con lazos nulos, y con funciones bloqueantes por separado. Este ejemplo muestra una secuencia de ejecución donde estos dos esquemas se combinan del siguiente modo:

- Dos tareas se crean con prioridad 1. No hacen otra cosa más que imprimir constantemente una cadena.
Estas tareas no hacen ningún llamado a funciones API que pudieran hacer que las tareas se bloqueen, por lo que se encontrarán siempre ejecutándose o disponibles para ejecutarse. Las tareas de esta naturaleza se denominan “tareas de proceso continuo”, dado que siempre tienen trabajo que realizar, aunque sea un trabajo trivial en este caso. Las fuentes de estas tareas de proceso continuo se muestran en el Listado 15.
- Una tercera tarea se crea con prioridad 2 (mayor que la prioridad de las dos tareas anteriores). La tercera tarea también imprime una cadena solamente, pero utiliza periódicamente la función API `vTaskDelayUntil()` para bloquearse entre cada impresión. La fuente de esta tarea periódica se muestra en el Listado 16.

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

    /* La cadena a imprimir es pasada a través del parámetro. Se castea como puntero de carácter. */
    pcTaskName = ( char * ) pvParameters;

    /* Como siempre, se implementa la tarea dentro de un lazo infinito. */
    for( ;; )
    {
        /* Imprime el nombre de la tarea. Hace esto repetidamente sin bloquearse o entrar en retardos. */
        vPrintString( pcTaskName );
    }
}
```

Listado 15 - Tarea de Proceso continuo usada en el Ejemplo 6.

```
void vPeriodicTask( void *pvParameters )
{
    portTickType xLastWakeTime;

    /* La variable xLastWakeTime necesita ser inicializada con el valor actual de la cuenta de interrupciones Tick. Notar que esta es la única vez que la variable es escrita explícitamente. Luego esta variable se actualiza automáticamente dentro de la función vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();

    /* Como siempre, se implementa la tarea dentro de un lazo infinito. */
    for( ;; )
    {
        /* Imprime el nombre de la tarea. */
        vPrintString( "Periodic task is running\r\n" );
    }
}
```

```

/* Esta tarea debería ejecutarse cada 10 milisegundos exactamente. */
vTaskDelayUntil( &xLastWakeTime, ( 10 / portTICK_RATE_MS ) );
}
}

```

Listado 16 – Tarea Periódica usada en el Ejemplo 6.

La figura 11 muestra la salida producida por el Ejemplo 6, con una explicación de la conducta observada dada por la secuencia de ejecución que se muestra en la Figura 12.

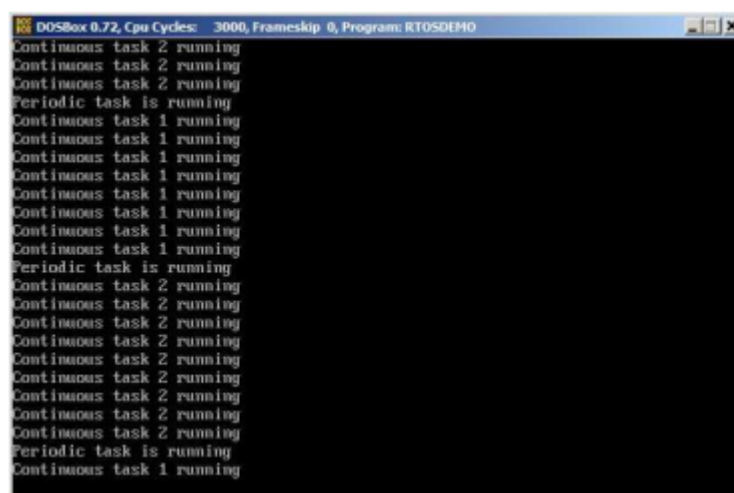


Figura 11 – Salida producida cuando se ejecuta el Ejemplo 6.

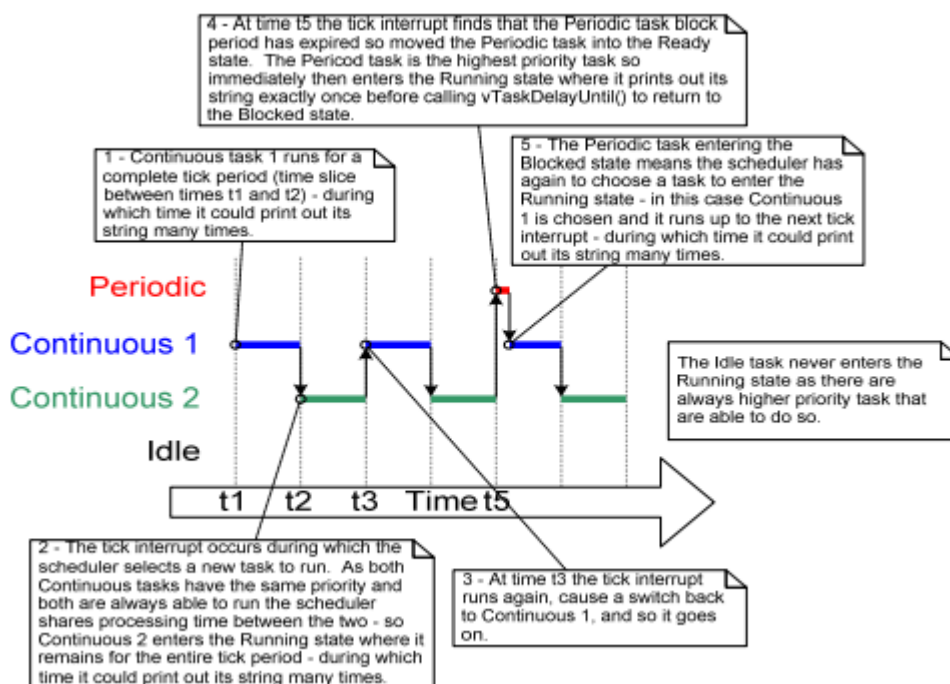


Figura 12 – Patrón de Ejecución del Ejemplo 6.

Esta salida se produjo usando el simulador DOSBox para retardar la ejecución a un nivel donde el comportamiento pudo observarse en una sola captura de pantalla.

1.7 La Tarea Ociosa y el Enlace de Tarea Ociosa:

Las tareas creadas en el Ejemplo 4 pasan la mayor parte de su tiempo en el estado bloqueado. Mientras que en se encuentran en este estado, no son capaces de ejecutarse y no puede ser seleccionado por el scheduler.

El procesador siempre necesita algo para ejecutar, es decir, debe haber siempre al menos una tarea que puede entrar en el estado de ejecución. Para asegurarse de esto, una tarea de ociosa (Idle Task) se crea automáticamente por el scheduler cuando se llama a la función `vTaskStartScheduler()`. La tarea ociosa no hace más que entrar en un bucle, así como las tareas de los ejemplos originales, y siempre es capaz de ejecutarse.

La tarea ociosa tiene la prioridad más baja posible (prioridad 0) para asegurarse que no impida que una tarea de mayor prioridad entre en el estado de ejecución. De todos modos, no hay nada que impida a los programadores crear una tarea con igual prioridad que de la tarea ociosa, si así lo desea.

Tan pronto como una tarea de mayor prioridad pasa al estado listo, la tarea ociosa dejará de ejecutarse para cederle su lugar a la nueva tarea. Esto se puede ver en el tiempo tn en la Figura 9, donde se intercambia inmediatamente la tarea ociosa para permitir que se ejecute la Tarea 2, en el instante que esta dejó de estar bloqueada.

Funciones de Enlace de Tarea Ociosa:

Es posible añadir funcionalidades específicas de la aplicación directamente en la tarea ociosa a través del uso del enlace de tarea ociosa. Esto es una función que se llama de forma automática por la tarea ociosa una vez por iteración del bucle tarea ociosa.

Los usos más comunes son:

- Ejecución de baja prioridad, de fondo, o de procesamiento continuo.
- Medir la cantidad de capacidad de procesamiento extra (la tarea ociosa sólo se ejecutará cuando todas las otras tareas no tienen trabajo para llevar a cabo, por lo que la medición de la cantidad de tiempo de procesamiento asignados a la tarea ociosa proporciona una indicación clara de cuánto tiempo de procesamiento no está siendo utilizado).
- Colocar el procesador en un modo de bajo consumo - proporcionar un método automático de ahorro de energía siempre que no haya procesamiento de la aplicación a realizar.

Limitaciones en la aplicación de las funciones del Enlace de Tarea Ociosa:

Las funciones del enlace de tarea ociosa deben cumplir con las siguientes reglas:

1. Nunca deben intentar bloquear o suspender. La tarea ociosa sólo se ejecutará cuando no haya otra tarea capaz de hacerlo (a menos que las tareas de aplicación comparten la prioridad de inactividad). El bloqueo de la tarea ociosa podría provocar un escenario en el que no haya tareas disponibles para entrar en el estado de ejecución.

2. Si la aplicación hace uso de la función API `vTaskDelete()` entonces el enlace de tarea ociosa debe siempre retornar a la función desde donde se llamó en un plazo de tiempo razonable. Esto se debe a la tarea ociosa es responsable de la limpieza de los recursos del kernel después que se ha suprimido una tarea. Si la tarea ociosa permanece permanentemente en la función de enlace entonces no puede realizar esta limpieza.

Las funciones de enlace de tarea ociosa deben tener el nombre y el prototipo mostrado por el Listado 17.

```
void vApplicationIdleHook( void );
```

Listado 17 – Nombre y Prototipo de la función de Enlace de Tarea Ociosa.

Ejemplo 7. Definiendo una función de Enlace de Tarea Ociosa:

El uso de la función bloqueante `vTaskDelay()` en el Ejemplo 4 creaba una gran cantidad de tiempo de inactividad, es decir, momentos en que la tarea ociosa se ejecuta debido a que ambas tareas se encontraban en estado bloqueado. En este ejemplo se hace uso de este tiempo de inactividad a través del empleo de la función de Enlace de Tarea Ociosa, la fuente para esto se muestra en el Listado 18.

```
/* Declaro una variable que será incrementada por la función de Enlace. */
unsigned long ullIdleCycleCount = 0UL;

/* La función de Enlace debe ser llamada a través de vApplicationIdleHook(), no tiene parámetros, y
no retorna ningún valor. */
void vApplicationIdleHook( void )
{
    /* Esta función de enlace no hace más que incrementar un contador. */
    ullIdleCycleCount++;
}
```

Listing 18 – Función simple de Enlace de Tarea Ociosa.

`configUSE_IDLE_HOOK` debe establecerse en 1 dentro `FreeRTOSConfig.h` para que la función de enlace pueda ser llamada.

La función que implementa las tareas creadas se modifica ligeramente para imprimir el valor de `ullIdleCycleCount`, como se muestra en el Listado 19.

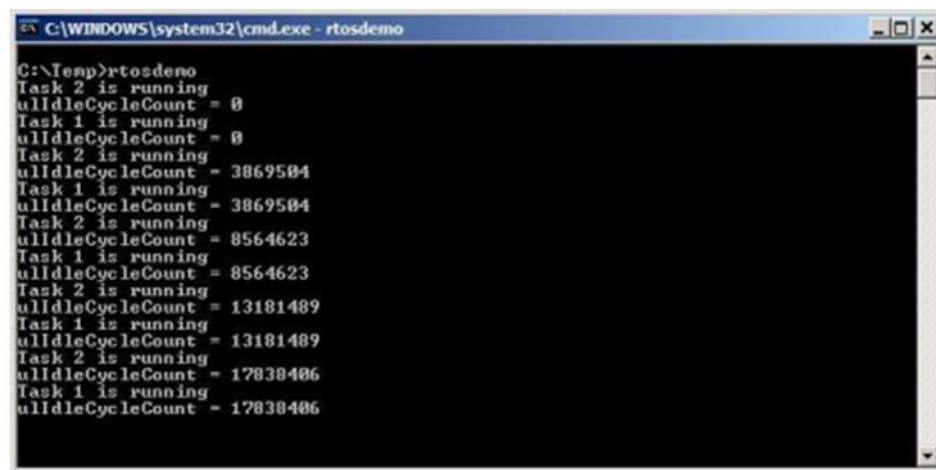
```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;

    /* La cadena a imprimir es pasada a través del parámetro. Se castea como puntero de carácter. */
    pcTaskName = ( char * ) pvParameters;
```

```
/* Como siempre, se implementa la tarea dentro de un lazo infinito. */
for(;;)
{
    /* Imprime el nombre de la tarea y el número de veces que ha sido aumentada la
    variable ullIdleCycleCount . */
    vPrintStringAndNumber( pcTaskName, ullIdleCycleCount );
    /* Retardo por un periodo de 250 milisegundos.*/
    vTaskDelay( 250 / portTICK_RATE_MS );
}
}
```

Listado 19 – Fuente del código para el ejemplo.

La salida producida por el ejemplo 7 se muestra en la Figura 13 y muestra que (en mi equipo) la función de enlace de tarea se llama (muy) aproximadamente 4,5 millones de veces entre cada iteración del tareas de la aplicación.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 2 is running
ullIdleCycleCount = 0
Task 1 is running
ullIdleCycleCount = 0
Task 2 is running
ullIdleCycleCount = 3869584
Task 1 is running
ullIdleCycleCount = 3869584
Task 2 is running
ullIdleCycleCount = 8564623
Task 1 is running
ullIdleCycleCount = 8564623
Task 2 is running
ullIdleCycleCount = 13181489
Task 1 is running
ullIdleCycleCount = 13181489
Task 2 is running
ullIdleCycleCount = 17838406
Task 1 is running
ullIdleCycleCount = 17838406
```

Figura 13 – Salida producida cuando se ejecuta el Ejemplo 7.

1.8 Cambiando la Prioridad de una Tarea:

Función API `vTaskPrioritySet()`:

Esta función puede ser usada para cambiar la prioridad de cualquier tarea luego que el scheduler ha iniciado.

```
void vTaskPrioritySet( xTaskHandle pxTask, unsignedportBASE_TYPE uxNewPriority );
```

Listado 20 – Prototipo de la función API `vTaskPrioritySet()`.

Tabla 4 Parámetros de `vTaskPrioritySet()`.

Nombre del parámetro	Descripción
<code>pxTask</code>	El manipulador de la tarea cuya prioridad está siendo modificada (la tarea sujeto), ver el parámetro <code>pxCreatedTask</code> del <code>xTaskCreate()</code> para obtener información sobre la obtención manipuladores de tareas. Una tarea puede cambiar su propia prioridad pasando NULL en lugar de un identificador de tarea válida.
<code>uxNewPriority</code>	La prioridad a la que la tarea sujeto se va a establecer. Esto tendrá un tope de forma automática a la prioridad máxima disponible de <code>configMAX_PRIORITIES (- 1)</code> , donde <code>configMAX_PRIORITIES</code> es una opción establecida en el <code>FreeRTOSConfig.h</code> .

Función API `uxTaskPriorityGet()`:

Esta función se puede usar para consultar la prioridad de una tarea.

```
unsigned portBASE_TYPE uxTaskPriorityGet( xTaskHandle pxTask );
```

Listado 21 – Prototipo de la Función API `uxTaskPriorityGet()`.

Tabla 5 vTaskPrioritySet() Parámetros y valores de retorno.

Nombre del parámetro	Descripción
pxTask	El manipulador de la tarea cuya prioridad está siendo consultada (la tarea sujeto), ver el parámetro pxCreatedTask del xTaskCreate () para obtener información sobre la obtención manipuladores de tareas. Una tarea puede consultar su propia prioridad pasando NULL en lugar de un identificador de tarea válida.
Returned value	La prioridad actual asignada a la tarea consultada

Ejemplo 8. Cambiando las prioridades de tareas:

El scheduler siempre seleccionará a la tarea disponible con más alto nivel de prioridad como aquella que debe entrar en el estado de ejecución. El ejemplo 8 demuestra esto mediante el uso de la función vTaskPrioritySet () API para cambiar la prioridad de dos tareas relativas entre sí.

Dos tareas se crean en dos prioridades diferentes. Ninguna tarea hace ningún llamado a alguna función API haga que se bloqueen, por lo tanto, siempre se encuentran disponibles para el scheduler. La tarea con la más alta prioridad relativa será siempre la tarea seleccionada por el scheduler para estar en el estado de ejecución.

Ejemplo 8 se comporta de la siguiente manera:

- La Tarea 1 (Listado 22) se crea con la más alta prioridad, de manera que se garantiza que se ejecutará en primer lugar. La Tarea 1 imprime un par de cadenas antes de levantar la prioridad de la Tarea 2 (Listado 23) por encima de su propia prioridad.
- La Tarea 2 comienza a ejecutarse tan pronto como tenga la más alta prioridad relativa. Sólo una tarea puede estar en estado de ejecución en un momento así que cuando la Tarea 2 se ejecuta, la Tarea 1 pasa a estar en estado disponible.
- La Tarea 2 imprime un mensaje antes de establecer nuevamente su propia prioridad por debajo de la de Tarea 1.
- El ajuste de la prioridad de la Tarea 2 por debajo de la Tarea 1, significa que nuevamente la Tarea 1 es la tarea de mayor prioridad, por lo que vuelve a entrar en el estado de ejecución, obligando a la Tarea 2 a entrar de nuevo en el estado disponible.

```
void vTask1( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;
```

```
    /* Esta tarea se ejecutará siempre antes que la Tarea 2, ya que es creada con la más alta
    prioridad. Ni la Tarea 1 ni la 2 bloquean, por lo tanto siempre estarán ejecutándose o dis-
    ponibles para ejecutarse.
```

```
    Consulta la prioridad a la que esta tarea está en ejecución. */
    uxPriority = uxTaskPriorityGet( NULL );
```

```
    for( ;; )
```

```

    {
        /* Imprime el nombre de la tarea. */
        vPrintString( "Task1 is running\r\n" );

        /* Establecer la prioridad de la Tarea 2 por encima de la prioridad de la Tarea 1
        causará que la Tarea 2 de inmediato comience a ejecutarse. Observe el uso del
        manipulador de Tarea 2 (xTask2Handle) en la llamada a vTaskPrioritySet (). El Lis-
        tado 24 muestra cómo se obtuvo el manipulador. */
        vPrintString( "About to raise the Task2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );
        /* La Tarea 1 sólo se ejecutará cuando tenga una prioridad más alta que la Tarea
        2. Por lo tanto, para que esta tarea llegue a este punto, la Tarea 2 ya debe haber
        ejecutado y establecido su prioridad nuevamente por debajo de la prioridad de
        esta tarea. */
    }
}

```

Listado 22 - Implementación de la Tarea 1 en el Ejemplo 8.

```

void vTask2( void *pvParameters )
{
    unsigned portBASE_TYPE uxPriority;
    /* La Tarea 1 siempre se ejecutará antes que esta, dado que es creada mayor prioridad. */
    uxPriority = uxTaskPriorityGet( NULL );

    for( ;; )
    {
        /* Para que esta tarea llegue a este punto, la Tarea 1 ya debe haber ejecutado y
        establecido la prioridad de esta tarea por encima de la propia.
        Imprime el nombre de esta tarea. */
        vPrintString( "Task2 is running\r\n" );

        /* Establecer nuevamente la prioridad de esta tarea a su valor original. Pasando
        NULL como manipulador de tarea significa "cambiar mi prioridad". Ajustar la
        prioridad a una inferior a la de la Tarea 1 causará que esta comience de inmediato
        a ejecutarse de nuevo. */
        vPrintString( "About to lower the Task2 priority\r\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );
    }
}

```

Listado 23 – Implementación de la Tarea 2 del Ejemplo 8.

Cada tarea puede tanto consultar como establecer su propia prioridad, sin el uso de un identificador de tarea válido (NULL utilizado en su lugar). Un manipulador de tarea sólo es necesario cuando una tarea desea hacer referencia a otra tarea. Por ejemplo para que la Tarea 1 le cambie la prioridad a la Tarea 2, el manipulador de la tarea 2 se obtiene y se guarda cuando se crea esta tarea, como se destaca en las observaciones en el Listado 24.


```
/* Se declara una variable que es usada para guardar el manipulador de la Tarea 2. */
xTaskHandle xTask2Handle;

int main( void )
{
    /* Se crea la primer tarea con prioridad 2. El parámetro de tarea y el manipulador no se
    usan, por lo que se establecen como NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

    /* La Tarea se crea con prioridad 2. */

    /* Se crea la segunda tarea con prioridad 1. Nuevamente no se usa el parámetro de tarea
    por lo que se establece como NULL, pero ahora si se usará el manipulador de tarea. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );
    /* El manipulador de tarea es el último parámetro. */

    /* Inicio el Scheduler para que last areas comiencen a ejecutarse. */
    vTaskStartScheduler();

    /*Si todo anda bien, el main() nunca debería alcanzar este punto, dado que el Scheduler va
    a estar siempre ejecutando las tareas. Si el main() alcanza este punto, entonces el probable
    que la memoria disponible no era suficiente para crear la tarea ociosa. El capítulo 5 provee
    más información sobre el manejo de memoria. */
    for( ;; );
}
```

Listado 24 – Implementación del main() para el Ejemplo 8.

La Figura 14 muestra la secuencia en la cual se ejecutan las tareas del Ejemplo 8, con la salida mostrada en la Figura 15.

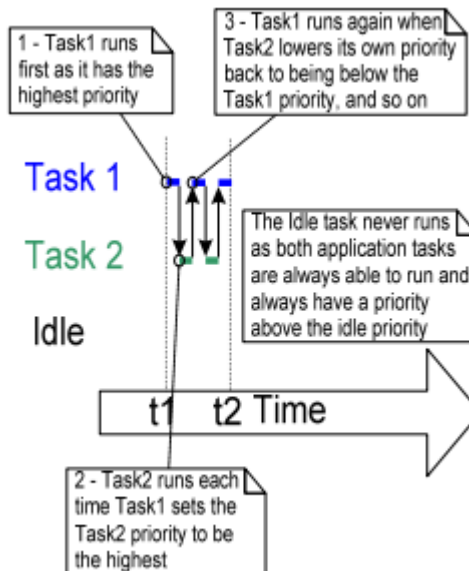


Figura 14 – Secuencia de ejecución de las tareas del Ejemplo 8.

```
DOSBox 0.72, Cpu Cycles: 3000, Frameskip: 0, Program: RTOSDEMO
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
```

Figura 15 – Salida producida cuando se ejecuta el Ejemplo 8.

1.9 Eliminación de tareas:

Función API vTaskDelete():

Una tarea puede utilizar la función API vTaskDelete () para eliminarse a sí misma o a cualquier otra tarea.

Las tareas borradas ya no existen, y no pueden entrar en el estado de ejecución nuevamente.

Es responsabilidad de la tarea ociosa liberar memoria que se asignó a las tareas que fueron eliminadas. Por ello es importante que las aplicaciones que hacen uso de la (función API vTaskDelete) permitan que se ejecute por momentos la tarea ociosa.

Tenga en cuenta también que sólo la memoria que se asigna a una tarea por el propio kernel se liberará automáticamente cuando se elimina la tarea. Cualquier memoria u otro recurso que la aplicación se haya asignado deben ser liberados de forma explícita.

```
void vTaskDelete( xTaskHandle pxTaskToDelete );
```

Listado 25 – Prototipo de la Función API vTaskDelete().

Tabla 6 Parámetros de la función vTaskDelete().

Nombre del parámetro	Descripción
pxTaskToDelete	El manipulador de la tarea que se desea borrar (la tarea sujeto), ver el parámetro pxCreatedTask del xTaskCreate () para obtener información sobre la obtención manipuladores de

	tareas. Una tarea puede borrarse a sí misma pasando NULL en lugar de un identificador de tarea válida.
--	--

Ejemplo 9. Eliminación de tareas:

Este es un ejemplo muy sencillo que se comporta de la siguiente manera:

- La Tarea 1 es creada por el main() con prioridad 1. Cuando se ejecuta crea la Tarea 2 con prioridad 2. La Tarea 2 es ahora la tarea de mayor prioridad por lo que comenzará a ejecutarse inmediatamente. La fuente de main() es mostrada en el Listado 26, y para la Tarea 1 en el Listado 27.
- La Tarea 2 no hace más que eliminarse a sí misma. Se podría borrar pasando NULL a vTaskDelete (), pero para fines puramente demostrativos en su lugar utiliza su propio manipulador de tarea. La fuente de la Tarea 2 se muestra en el Listado 28.
- Cuando la Tarea 2 se ha suprimido, la Tarea 1 volverá a ser la tarea de más alta prioridad, por lo que continuará ejecutándose, en cuyo punto se llama vTaskDelay () para bloquearla durante un corto período.
- La tarea de ociosa se ejecutará mientras la Tarea 1 está en el estado bloqueado y libera la memoria que había sido asignada a la Tarea 2.
- Cuando la Tarea 1 abandona el estado bloqueado será una vez más la tarea en estado disponible con mayor prioridad por lo que comienza a ejecutarse nuevamente, creando de nuevo a la Tarea 2, y así se repite el ciclo.

```
int main( void )
{
    /* Se crea la primer tarea con prioridad 1. El parámetro de tarea y el manipulador de tarea no serán
    utilizados, por lo que se los deja NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

    /* Inicio el scheduler para que las tareas comiencen a ejecutarse. */
    vTaskStartScheduler();

    /* El main() nunca debería alcanzar este punto, ya que el scheduler estará siempre ejecutándose. */
    for( ;; );
}
```

Listado 26 – Implementación del main() del Ejemplo 9.

```
void vTask1( void *pvParameters )
{
    const portTickType xDelay100ms = 100 / portTICK_RATE_MS;
    for( ;; )
    {
        /* Imprime el nombre de esta tarea. */
        vPrintString( "Task1 is running\r\n" );
```

```

/* Se crea la Tarea 2 con una prioridad más alta (2) Nuevamente no se utilizará el parámetro de tarea, por lo que se lo deja NULL. Pero esta vez sí se utilizará el manipulador, dado que esta tarea será borrada utilizándolo. */
xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

/* La Tarea 2 tenía la prioridad más alta, entonces para que la Tarea 1 alcance este punto la Tarea 2 ya debe haberse ejecutado y borrado a sí misma. */
vTaskDelay( xDelay100ms );
/* Retardo 100 milisegundos.*/
}
}

```

Listado 27 – Implementación de la Tarea 1 para el Ejemplo 9.

```

void vTask2( void *pvParameters )
{
    /* La Tarea 2 no hace más que eliminarse a sí misma. Para ello se podría llamar vTaskDelete () usando NULL como parámetro, pero en su lugar y puramente con fines demostrativos, le pasa a vTaskDelete () su propio identificador de tarea. */
    vPrintString( "Task2 is running and about to delete itself\r\n" );
    vTaskDelete( xTask2Handle );
}

```

Listado 28 - Implementación de la Tarea 2 para el Ejemplo 9.

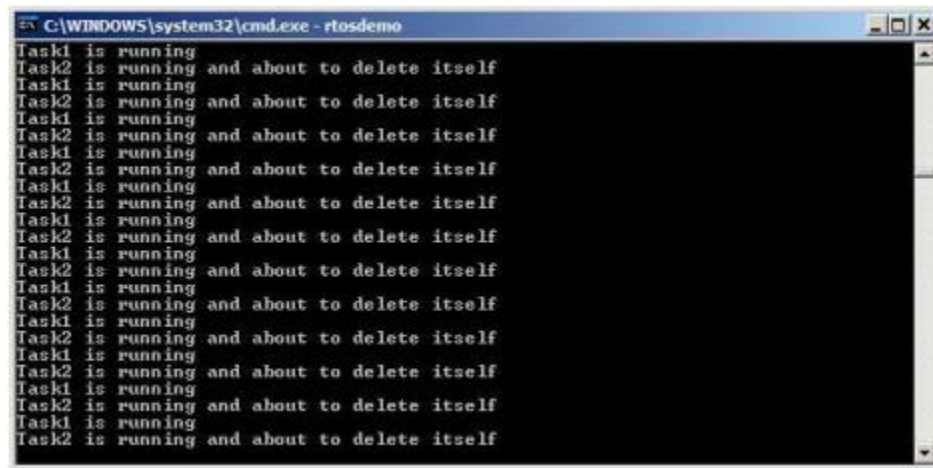


Figura 16 – La salida producida cuando el Ejemplo 9 se está ejecutando.

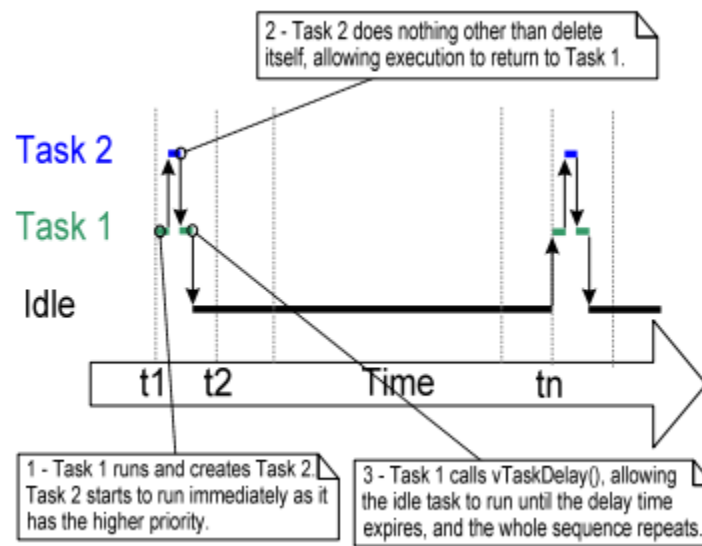


Figura 17 – Secuencia de Ejecución para el Ejemplo 9.

1.10 El algoritmo de planificación – Resumen

Programación con Prioridad fija y derecho preferente:

Los ejemplos de este capítulo han ilustrado cómo y cuándo FreeRTOS selecciona qué tarea debe ser ejecutada:

- A cada tarea se le asigna una prioridad.
- Cada tarea puede existir en uno de varios estados.
- Sólo una tarea puede existir en estado de ejecución en un momento dado.
- El scheduler siempre selecciona la tarea disponible de más alto nivel de prioridad para entrar en el estado de ejecución.

Este tipo de esquema se llama “Programación con Prioridad fija y derecho preferente”. “Prioridad Fija”, ya que a cada tarea se le asigna una prioridad que no se altera por el propio kernel (sólo las tareas pueden cambiar las prioridades). “Derecho preferente” porque una tarea que entra en el estado disponible o que su

prioridad sea alterada será siempre elegida para ser ejecutada si su prioridad resulta mayor que el resto de las tareas en estados disponibles o ejecutándose.

Las tareas pueden esperar en el estado bloqueado para un evento y serán trasladados automáticamente al estado disponible cuando se produce el evento. Los eventos temporales ocurren en un momento determinado, por ejemplo cuando un periodo de tiempo expira. Se utilizan generalmente para implementar periodos o tiempos de espera. Los eventos de sincronización se producen cuando una tarea o servicios de interrupción de rutina envían información a la cola o uno de los muchos tipos de semáforos. Se utilizan generalmente para indicar la actividad asincrónica, tales como la llegada de datos a un periférico.

La Figura 18 demuestra todo este comportamiento, ilustrando el patrón de ejecución de una hipotética solicitud.

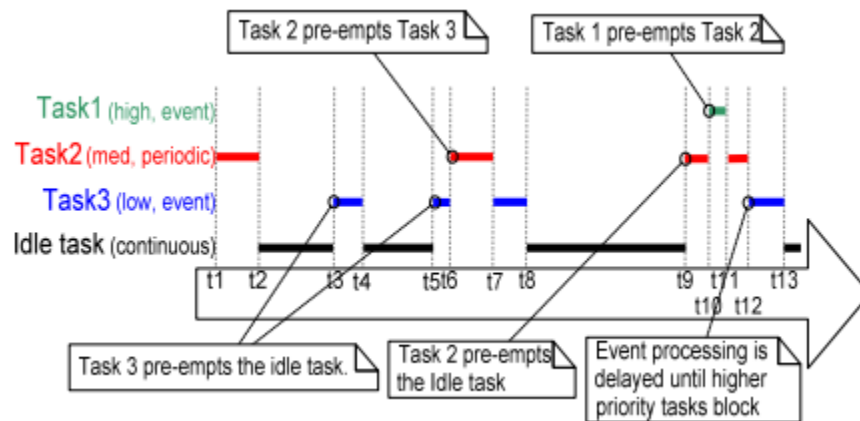


Figura 18 – Patrón de Ejecución con puntos de derecho preferente remarcados.

Refiriéndose a la Figura 18:

1. Tarea Ociosa:

La tarea ociosa se ejecuta a la prioridad más baja, por lo que deja de ejecutarse cada vez que una tarea de mayor prioridad entra en el estado disponible, por ejemplo, en t3, t5 y T9.

2. Tarea 3:

La Tarea 3 es una tarea de evento impulsado, que se ejecuta con una prioridad relativamente baja, pero mayor que la prioridad de la tarea ociosa. Pasa la mayor parte de su tiempo en el estado bloqueado esperando el evento de interés, transicionando desde el estado bloqueado al estado disponible cada vez que se produce el evento. Todos los mecanismos de comunicación inter-tarea de FreeRTOS (colas, semáforos, etc.) pueden utilizarse para señalar eventos y desbloquear las tareas de esta manera.

Los eventos ocurren en los instantes t3, t5, y también en algún lugar entre T9 y T12. Los eventos que ocurren en tiempos t3 y t5 se procesan inmediatamente dado que en estos tiempos la Tarea 3 es la tarea

Usando el Kernel de Tiempo Real FreeRTOS – Cap 1 / Manejo de Tareas

con más alta prioridad que esta disponible. El evento que se produce en algún lugar entre los tiempos t_9 y t_{12} no se procesa hasta t_{12} porque hasta entonces las tareas 1 y 2, que son de mayor prioridad, se siguen ejecutando. Es sólo en t_{12} que tanto las tareas 1 y 2 están en el estado bloqueado, haciendo a la Tarea 3 la de más alta prioridad y disponible para ejecutarse.

3. Tarea 2:

La Tarea 2 es una tarea periódica que se ejecuta con una prioridad por encima de la prioridad de la Tarea 3, pero por debajo de la prioridad de la Tarea 1. El intervalo de tiempo significa que la Tarea 2 quiere ejecutarse en los instantes t_1 , t_6 y t_9 .

En el tiempo t_6 , la Tarea 3 está en estado de ejecución, pero la Tarea 2 tiene la más alta prioridad, por lo que pasa a ejecutarse y la Tarea 3 pasa a estado disponible. La Tarea 2 completa su procesamiento y reingresa en el estado bloqueado en t_7 , en cuyo punto la Tarea 3 puede volver a entrar en el estado de ejecución para completar su procesamiento. La Tarea 3 se autobloquea en el instante t_8 .

4. Tarea 1:

La Tarea 1 es también una tarea periódica de evento impulsado. Se ejecuta con la prioridad más alta de todas por lo que puede adelantarse a cualquier otra tarea en el sistema. El único evento de la Tarea 1 que se muestra tiene lugar en el tiempo t_{10} , momento en el cual la Tarea 1 se adelanta a la tarea Tarea 2, es decir, se comienza a ejecutar la Tarea 1 y la Tarea 2 pasa a estado disponible. La Tarea 2 sólo puede completar su ejecución después que la Tarea 1 ha vuelto a entrar en el estado bloqueado, en instante t_{11} .

Selección de Prioridad de tareas:

La figura 18 muestra la importancia de la asignación de prioridades, dado que determinan por completo el funcionamiento de una aplicación.

Como regla general, a las tareas que implementan funciones de tiempo real duro se le asignan prioridades por encima de los que implementan las funciones de tiempo real suave. Sin embargo, otras características como los tiempos de ejecución y la utilización del procesador también debe tenerse en cuenta para garantizar que la aplicación nunca pierda el tiempo de respuesta que un sistema de tiempo real duro exige.

El Schedulling de Tasa monótona (RMS) es una técnica de asignación de prioridad que dicta un único nivel de prioridad común para todas aquellas tareas que se ejecuten con una misma frecuencia, es decir, se asignará la prioridad de cada tarea de acuerdo con la tasa de ejecución periódica de esa tarea. El más alto nivel de prioridad se le asigna a la tarea que tiene la mayor frecuencia de ejecución periódica. La prioridad más baja se asigna a la tarea con la frecuencia más baja de la ejecución periódica. La asignación de prioridades de acuerdo a este método ha demostrado que logra maximizar la planificación de todas las aplicaciones, pero las variaciones en los tiempos de ejecución y el hecho de que no todas las tareas son periódicas hace que los cálculos se tornen bastante complejos.

Planificación Cooperativa:

Este libro se centra en la planificación de derecho preferente. FreeRTOS también puede utilizarse en la opción de planificación cooperativa

Cuando se utiliza un scheduler cooperativo puro, un cambio de contexto sólo se producirá cuando una tarea en ejecución pasa a estado bloqueado, o una tarea en estado de ejecución hace un llamado explícito a `taskYIELD()`. Las tareas nunca se adelantaran, y las tareas de igual prioridad no compartirán automáticamente tiempo de procesamiento. En este sentido, la planificación cooperativa es más simple pero puede potencialmente dar lugar a un sistema menos sensible.

Docente: Ing. Marín Ariel

Año: 2024

Auxiliar: Ing. Fabián Massotto

Apunte: Td3 – Clase 1 (Teórica)

Usando el Kernel de Tiempo Real FreeRTOS – Cap 1 / Manejo de Tareas

Un esquema híbrido también es posible, donde se utilizan rutinas de servicio de interrupción para causar explícitamente un cambio de contexto. Esto permite eventos de sincronización, pero no eventos temporales. Esto puede ser deseable debido a su eficiencia y dado que tiene una configuración común del scheduler.