3.6 Medición del tiempo y la interrupción de ticks

La Sección 3.12, Algoritmos de planificación, describe una característica opcional llamada "fracción de tiempo". La división de tiempo se utilizó en los ejemplos presentados hasta ahora, y es el comportamiento observado en la salida que produjeron. En los ejemplos, ambas tareas se crearon con la misma prioridad y ambas tareas siempre se pudieron ejecutar. Por lo tanto, cada tarea se ejecutó durante un "período de tiempo", ingresando al estado de Ejecución al comienzo de un período de tiempo y saliendo del estado de Ejecución al final de un período de tiempo. En la Figura 11, el tiempo entre t1 y t2 es igual a un solo intervalo de tiempo.

Para poder seleccionar la siguiente tarea a ejecutar, el propio planificador debe ejecutarse al final de cada intervalo de tiempo. Para este propósito, se utiliza una interrupción periódica, llamada "interrupción de tick". La duración del intervalo de tiempo se establece de manera efectiva mediante la frecuencia de interrupción de ticks, que se configura mediante la constante de configuración de tiempo de compilación configTICK_RATE_HZ definida por la aplicación dentro de FreeRTOSConfig.h. Por ejemplo, si configTICK_RATE_HZ se establece en 100 (Hz), el intervalo de tiempo será de 10 milisegundos. El tiempo entre dos interrupciones de tick se denomina "período de tick". Un intervalo de tiempo equivale a un período de tick.

La figura 11 se puede ampliar para mostrar la ejecución del planificador mismo en la secuencia de ejecución. Esto se muestra en la Figura 12, en la que la línea superior muestra cuándo se está ejecutando el planificador, y las flechas finas muestran la secuencia de ejecución desde una tarea hasta la interrupción del tick, luego desde la interrupción del tick, regresa a una tarea diferente.

El valor óptimo para configTICK_RATE_HZ depende de la aplicación que se esté desarrollando, aunque un valor de 100 es típico.

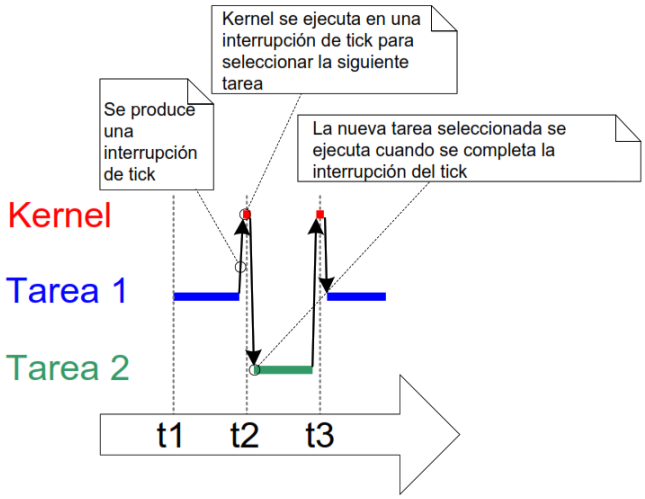

Fig. 12 La secuencia de ejecución ampliada para mostrar la ejecución de la interrupción tick

Las llamadas a la API de FreeRTOS siempre especifican el tiempo en múltiplos de períodos de ticks, que a menudo se denominan simplemente "ticks". La macro pdMS_TO_TICKS() convierte un tiempo especificado en milisegundos en un tiempo especificado en ticks. La resolución disponible depende de la frecuencia de marca definida, y pdMS_TO_TICKS() no se puede usar si la frecuencia de marca está por encima de 1 KHz (si configTICK_RATE_HZ es mayor que

1000). El Listado 20 muestra cómo usar pdMS_TO_TICKS() para convertir un tiempo especificado como 200 milisegundos en un tiempo equivalente especificado en ticks.

```
/* pdMS_TO_TICKS() toma un tiempo en milisegundos como su único parámetro y se evalúa como el tiempo equivalente en períodos de ticks. Este ejemplo muestra que xTimeInTicks se establece en la cantidad de períodos de ticks que equivalen a 200 milisegundos. */

TickType_t xTimeInTicks = pdMS_TO_TICKS (200);
```

Listado 20. Uso de la macro pdMS_TO_TICKS() para convertir 200 milisegundos en un tiempo equivalente en períodos de tic

Nota: No se recomienda especificar tiempos en ticks directamente dentro de la aplicación, sino usar la macro pdMS_TO_TICKS() para especificar tiempos en milisegundos y, al hacerlo, asegurarse de que los tiempos especificados dentro de la aplicación no cambien si la frecuencia de ticks es cambió.

El valor de "recuento de ticks" es el número total de interrupciones de ticks que se han producido desde que se inició el programador, suponiendo que el recuento de ticks no se haya desbordado. Las aplicaciones de usuario no tienen que considerar los desbordamientos al especificar períodos de retraso, ya que FreeRTOS administra internamente la consistencia del tiempo. La Sección 3.12, Algoritmos de planificación, describe las constantes de configuración que afectan cuándo el programador seleccionará una nueva tarea para ejecutar y cuándo se ejecutará una interrupción de tick.

Ejemplo 3. Experimentando con prioridades

El planificador siempre se asegurará de que la tarea de mayor prioridad que se pueda ejecutar sea la tarea seleccionada para ingresar al estado En ejecución. En nuestros ejemplos hasta ahora, se han creado dos tareas con la misma prioridad, por lo que ambas entraron y salieron del estado En ejecución a la vez. Este ejemplo analiza lo que sucede cuando se cambia la prioridad de una de las dos tareas creadas en el Ejemplo 2. Esta vez, la primera tarea se creará con prioridad 1 y la segunda con prioridad 2. El código para crear las tareas se muestra en el Listado 21. La única función que implementa ambas tareas no ha cambiado; todavía simplemente imprime una cadena periódicamente, usando un bucle nulo para crear un retraso.

```
/* Defina las cadenas que se pasarán como parámetros de la tarea. Estos se definen de forma constante y no en la pila para garantizar que sigan siendo válidos cuando se ejecutan las tareas. */

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

int main( void )
{
    /* Cree la primera tarea en la prioridad 1. La prioridad es el penúltimo parámetro. */

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

    /* Cree la segunda tarea con la prioridad 2, que es superior a la prioridad 1. La prioridad es el penúltimo parámetro. */
```

```
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );
    /* Inicie el planificador para que las tareas comiencen a ejecutarse. */
    vTaskStartScheduler();
    /* No llegará hasta aquí. */
     return 0;
}
```

Listado 21. Creación de dos tareas con diferentes prioridades

La salida producida por el Ejemplo 3 se muestra en la Figura 13.

El planificador siempre seleccionará la tarea de mayor prioridad que pueda ejecutarse. La tarea 2 tiene mayor prioridad que la tarea 1 y siempre puede ejecutarse; por lo tanto, la tarea 2 es la única que entra en el estado de ejecución. Como la tarea 1 nunca entra en el estado de ejecución, nunca imprime su cadena. Se dice que la tarea 1 está "privada" de tiempo de procesamiento por la tarea 2.

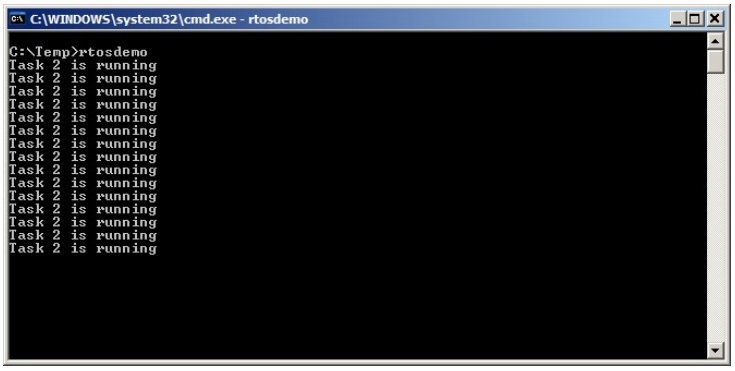

Figura 13. Ejecución de ambas tareas con diferentes prioridades

La tarea 2 siempre es capaz de ejecutarse porque nunca tiene que esperar nada, ya que o bien circula por un bucle nulo, o bien imprime en la terminal. La Figura 14 muestra la secuencia de ejecución del Ejemplo 3.

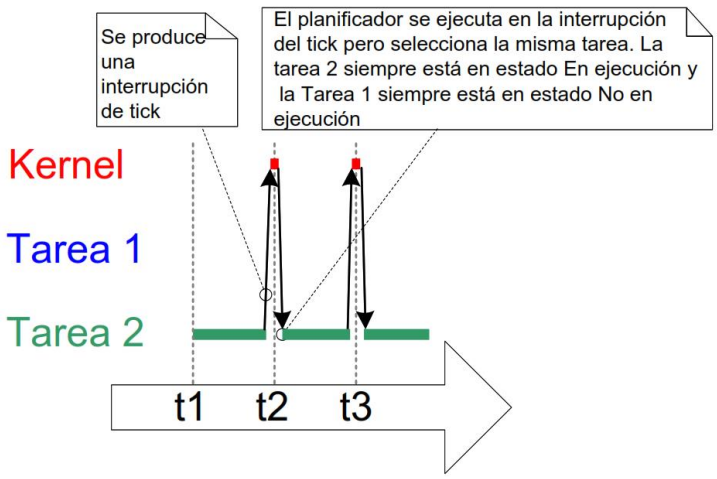

Fig. 14 El modelo de ejecución cuando una tarea tiene mayor prioridad que la otra

3.7 Expansión del estado 'No en ejecución'

Hasta ahora, las tareas creadas siempre han tenido procesamiento que realizar y nunca han tenido que esperar nada; como nunca tienen que esperar nada, siempre pueden entrar en el estado de ejecución. Este tipo de tareas de *"procesamiento continuo"* tiene una utilidad limitada, porque sólo pueden crearse con la prioridad más baja. Si se ejecutan con cualquier otra prioridad, impedirán que se ejecuten tareas de menor prioridad.

Para que las tareas sean útiles, se deben reescribir para que estén basadas en eventos. Una tarea controlada por eventos tiene trabajo (procesamiento) que realizar solo después de que ocurra el evento que la desencadena, y no puede ingresar al estado En ejecución antes de que ocurra ese evento. El Planificador siempre selecciona la tarea de mayor prioridad que pueda ejecutarse. El hecho de que las tareas de alta prioridad no puedan ejecutarse significa que el Planificador no puede seleccionarlas y debe, en su lugar, seleccionar una tarea de menor prioridad que sí pueda ejecutarse. Por lo tanto, el uso de tareas basadas en eventos significa que las tareas pueden ser creadas con diferentes prioridades sin que las tareas de mayor prioridad priven de tiempo de procesamiento a todas las tareas de menor prioridad.

El estado de bloqueo

Una tarea que está a la espera de un evento se dice que está en el estado "Bloqueado", que es un subestado del estado de No Ejecución.

Las tareas pueden entrar en el estado de bloqueo para esperar dos tipos diferentes de eventos:

1. Eventos temporales (relacionados con el tiempo): el evento es un período de retraso que expira o un tiempo absoluto que se alcanza. Por ejemplo, una tarea puede entrar en el estado de Bloqueo para esperar que pasen 10 milisegundos.

2. Eventos de sincronización: cuando los eventos se originan en otra tarea o interrupción. Por ejemplo, una tarea puede entrar en el estado de Bloqueo para esperar que lleguen datos a una cola. Los eventos de sincronización cubren una amplia gama de tipos de eventos.

Las colas de FreeRTOS, los semáforos binarios, los semáforos de conteo, las exclusiones mutuas, las exclusiones mutuas recursivas, los grupos de eventos y las notificaciones directas a tareas se pueden usar para crear eventos de sincronización. Todas estas características se tratan en capítulos futuros de este libro.

Es posible que una tarea se bloquee en un evento de sincronización con un tiempo de espera, bloqueando efectivamente ambos tipos de eventos simultáneamente. Por ejemplo, una tarea puede optar por esperar un máximo de 10 milisegundos para que los datos lleguen a una cola. La tarea dejará el estado Bloqueado si los datos llegan dentro de los 10 milisegundos o si pasan 10 milisegundos sin que llegue ningún dato.

El Estado Suspendido

“Suspendido" es también un subestado de No ejecutado. Las tareas en estado Suspendido no están disponibles para el Planificador. La única manera de entrar en el estado Suspendido es a través de una llamada a la función API vTaskSuspend(), la única manera de salir es a través de

una llamada a las funciones API vTaskResume() o xTaskResumeFromISR(). La mayoría de las aplicaciones no utilizan el estado de suspensión.

El Estado Listo

Se dice que las tareas que están en el estado No en ejecución pero que no están Bloqueadas o Suspendidas están en el estado Listo. Pueden ejecutarse y, por lo tanto, están "listos" para ejecutarse, pero actualmente no se encuentran en el estado En ejecución.

Completar el diagrama de transición de estados

La Figura 15 amplía el diagrama de estado anterior, excesivamente simplificado, para incluir todos los subestados de No en ejecución descritos en esta sección. Las tareas creadas en los ejemplos hasta ahora no han utilizado los estados Bloqueado o Suspendido; sólo han hecho la transición entre el estado Listo y el estado En Ejecución -resaltados por las líneas en negrita en la Figura 15.

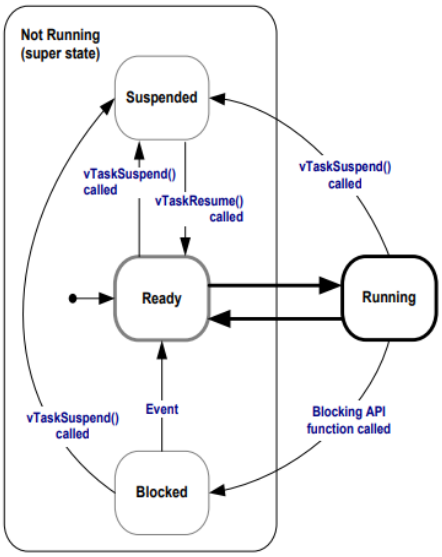

Figura 15. Estructura de estado de la tarea completa

Ejemplo 4. Uso del estado Bloqueado para crear un delay

Todas las tareas creadas en los ejemplos presentados hasta ahora han sido "periódicas": se retrasaron durante un período e imprimieron su cadena, antes de retrasarse una vez más, y así sucesivamente. El retardo se ha generado de forma muy burda utilizando un bucle nulo: la tarea ha sondeado de forma efectiva un contador de bucle incremental hasta que ha alcanzado un valor fijo. El ejemplo 3 demostró claramente la desventaja de este método. La tarea de mayor prioridad permaneció en el estado En ejecución mientras ejecutaba el bucle nulo, "privando" de tiempo de procesamiento a la tarea de menor prioridad.

Cualquier forma de sondeo tiene otras desventajas, entre las que destaca su ineficiencia. Durante el sondeo, la tarea no tiene realmente ningún trabajo que hacer, pero sigue utilizando el máximo tiempo de procesamiento, por lo que desperdicia ciclos de procesador. El ejemplo 4 corrige este comportamiento sustituyendo el bucle nulo de sondeo por una llamada a la función de la API vTaskDelay(), cuyo prototipo se muestra en el listado 22. La nueva definición de la tarea se muestra en el Listado 23. Tenga en cuenta que la función de la API vTaskDelay() sólo está disponible cuando INCLUDE_vTaskDelay se establece en 1 en FreeRTOSConfig.h.

vTaskDelay() coloca la tarea de llamada en el estado Bloqueado por un número fijo de interrupciones de tick. La tarea no usa ningún tiempo de procesamiento mientras está en estado Bloqueado, por lo que la tarea solo usa el tiempo de procesamiento cuando realmente hay trabajo por hacer.

```
void vTaskDelay( TickType_t xTicksToDelay );
```

xTicksToDelay: El número de interrupciones de ticks en las que la tarea de llamada permanecerá en el estado Bloqueado antes de volver al estado Listo.

Por ejemplo, si una tarea se llama vTaskDelay (100) cuando el recuento de ticks era 10.000, entraría inmediatamente en el estado Bloqueado y permanecería en el estado Bloqueado hasta que el recuento de ticks llegara a 10.100.

La macro pdMS_TO_TICKS() se puede usar para convertir un tiempo especificado en milisegundos en un tiempo especificado en ticks. Por ejemplo, llamar a vTaskDelay(pdMS_TO_TICKS(100)) hará que la tarea de llamada permanezca en el estado Bloqueado durante 100 milisegundos.

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

    /* La cadena a imprimir se pasa a través del parámetro. Convierte esto en un puntero de
    carácter. */
    pcTaskName = ( char * ) pvParameters;

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

        /* Delay (Retraso) durante un periodo. Esta vez se utiliza una llamada a vTaskDelay() que
        pone la tarea en estado de Bloqueo hasta que el periodo de Delay haya expirado. El
        parámetro toma un tiempo especificado en 'ticks', y se utiliza la macro pdMS_TO_TICKS()
        (donde se declara la constante xDelay250ms) para convertir 250 milisegundos en un
        tiempo equivalente en ticks. */
        vTaskDelay( xDelay250ms );
    }
}
```

Listado 23. El código fuente de la tarea de ejemplo después de que el Delay del bucle null haya sido sustituido por una llamada a vTaskDelay()

Aunque las dos tareas se siguen creando con diferentes prioridades, ahora ambas se ejecutarán. La salida del Ejemplo 4, que se muestra en la Figura 16, confirma el comportamiento esperado.

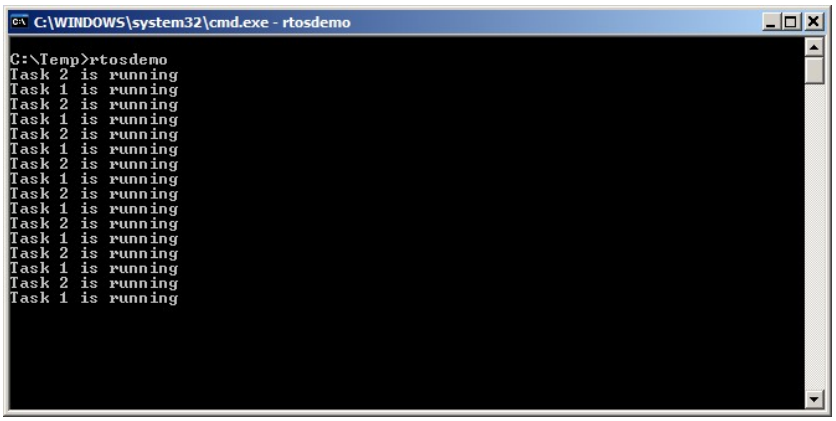

Figura 16. La salida producida cuando se ejecuta el Ejemplo 4

La secuencia de ejecución que se muestra en la Figura 17 explica por qué se ejecutan ambas tareas, aunque se hayan creado con diferentes prioridades. La ejecución del Planificador en sí se omite para simplificar.

La tarea inactiva se crea automáticamente cuando se inicia el Planificador, para garantizar que siempre haya al menos una tarea que pueda ejecutarse (al menos una tarea en estado Listo). La sección 3.8, La tarea inactiva y el gancho de la tarea inactiva, describe la tarea inactiva con más detalle.

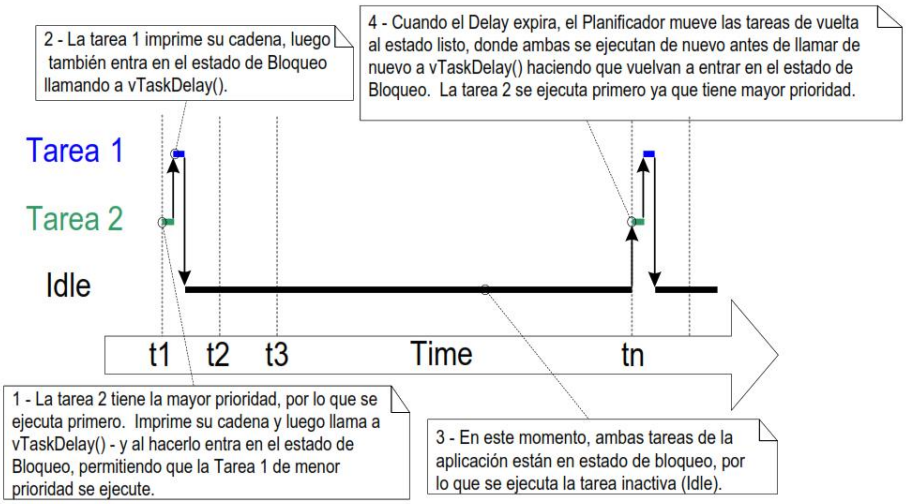

Figure 17. La secuencia de ejecución cuando las tareas utilizan vTaskDelay() en lugar del bucle NULL

Sólo ha cambiado la implementación de las dos tareas, no su funcionalidad. La comparación de la Figura 17 con la Figura 12 demuestra claramente que esta funcionalidad se consigue de forma mucho más eficiente.

La figura 12 muestra el modelo de ejecución cuando las tareas utilizan un bucle nulo para crear un Delay, por lo que siempre pueden ejecutarse y, en consecuencia, utilizan el cien por cien del tiempo de procesador disponible entre ellas. La figura 17 muestra el patrón de ejecución cuando las tareas entran en el estado de Bloqueo durante todo su periodo de Delay, por lo que utilizan el tiempo del procesador sólo cuando realmente tienen trabajo que realizar (en este caso simplemente un mensaje que hay que imprimir), y como resultado sólo utilizan una pequeña fracción del tiempo de procesamiento disponible.

En el esquema de la Figura 17, cada vez que las tareas salen del estado de Bloqueo se ejecutan durante una fracción de tick antes de volver a entrar en el estado de Bloqueo. La mayor parte del tiempo no hay tareas de aplicación que puedan ejecutarse (no hay tareas de aplicación en el estado Listo) y, por lo tanto, no hay tareas de aplicación que puedan seleccionarse para entrar en el estado Ejecutado. Mientras este sea el caso, la tarea inactiva se ejecutará. La cantidad de tiempo de procesamiento asignado a la tarea inactiva es una medida de la capacidad de procesamiento disponible en el sistema. El uso de un RTOS puede aumentar significativamente la capacidad de procesamiento sobrante simplemente permitiendo que una aplicación sea completamente dirigida por eventos.

Las líneas en negrita de la Figura 18 muestran las transiciones realizadas por las tareas del Ejemplo 4, con cada una de ellas pasando por el estado de Bloqueo antes de volver al estado de Listo.

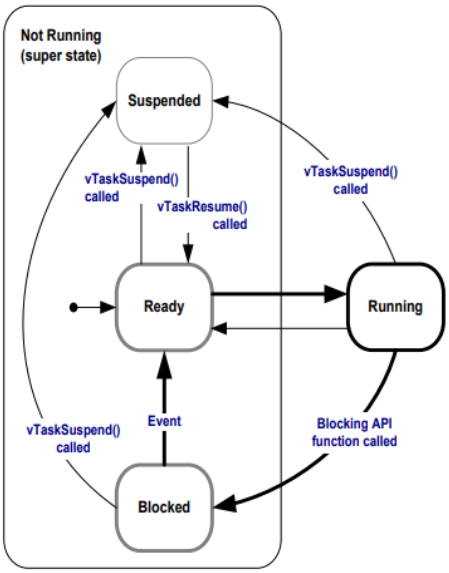

Figure 18. Bold lines indicate the state transitions performed by the tasks in Example 4

La función API vTaskDelayUntil()

vTaskDelayUntil() es similar a vTaskDelay(). Como se acaba de demostrar, el parámetro vTaskDelay() especifica la cantidad de interrupciones de ticks que deben ocurrir entre una tarea que llama a vTaskDelay() y la misma tarea que vuelve a salir del estado Bloqueado. El tiempo que la tarea permanece en estado bloqueado se especifica mediante el parámetro vTaskDelay(), pero el momento en que la tarea deja el estado bloqueado es relativo al momento en que se llamó a vTaskDelay().

Los parámetros de vTaskDelayUntil() especifican, en cambio, el valor exacto de conteo de ticks en el que la tarea que realiza la llamada debe pasar del estado Bloqueado al estado Listo. vTaskDelayUntil() es la función API que se debe usar cuando se requiere un período de ejecución fijo (donde desea que su tarea se ejecute periódicamente con una frecuencia fija), ya que el momento en que se desbloquea la tarea que llama es absoluto, en lugar de relativo a cuando se llamó a la función (como es el caso de vTaskDelay()).

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

pxPreviousWakeTime Este parámetro recibe su nombre asumiendo que vTaskDelayUntil() se usa para implementar una tarea que se ejecuta periódicamente y con una frecuencia fija. En este caso, pxPreviousWakeTime mantiene la hora a la que la tarea dejó por última vez el estado Bloqueado (fue "activada"). Este tiempo se usa como punto de referencia para calcular el tiempo en el que la tarea debe salir del estado Bloqueado.

La variable a la que apunta pxPreviousWakeTime se actualiza automáticamente dentro de la función vTaskDelayUntil(); normalmente no sería modificado por el código de la aplicación, pero debe inicializarse con el conteo de ticks actual antes de que se use por primera vez. El Listado 25 demuestra cómo se realiza la inicialización.

xTimeIncrement Este parámetro también recibe su nombre suponiendo que vTaskDelayUntil() se usa para implementar una tarea que se ejecuta periódicamente y con una frecuencia fija; la frecuencia la establece el valor de xTimeIncrement.

xTimeIncrement se especifica en 'ticks'. La macro pdMS_TO_TICKS() se puede usar para convertir un tiempo especificado en milisegundos en un tiempo especificado en ticks.

Ejemplo 5. Conversión de las tareas de ejemplo para utilizar vTaskDelayUntil()

Las dos tareas creadas en el Ejemplo 4 son tareas periódicas, pero el uso de vTaskDelay() no garantiza que la frecuencia con la que se ejecutan sea fija, ya que el momento en el que las tareas abandonan el estado de Bloqueo es relativo al momento en el que llaman a vTaskDelay(). Convertir las tareas para que usen vTaskDelayUntil() en lugar de vTaskDelay() resuelve este problema potencial.

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
TickType_t xLastWakeTime;
    /* La cadena a imprimir se pasa a través del parámetro. Convierte esto en un puntero de carácter. */
    pcTaskName = ( char * ) pvParameters;

    /* La variable xLastWakeTime necesita ser inicializada con la cuenta de tics actual. Tenga en cuenta que esta es la única vez que la variable se escribe explícitamente. Después de esto xLastWakeTime se actualiza automáticamente dentro de vTaskDelayUntil (). */
    xLastWakeTime = xTaskGetTickCount();

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

        /* Esta tarea debe ejecutarse cada 250 milisegundos exactamente. Según la función vTaskDelay(), el tiempo se mide en ticks, y la macro pdMS_TO_TICKS() se utiliza para convertir los milisegundos en ticks. xLastWakeTime se actualiza automáticamente dentro de vTaskDelayUntil(), por lo que no se actualiza explícitamente por la tarea. */
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );
    }
```

```
}
```

Listado 25. La implementación de la tarea de ejemplo utilizando vTaskDelayUntil()

Ejemplo 6. Combinación de tareas bloqueantes y no bloqueantes

Los ejemplos anteriores han examinado el comportamiento de las tareas de sondeo y de bloqueo de forma aislada. Este ejemplo refuerza el comportamiento esperado del sistema demostrando una secuencia de ejecución cuando se combinan los dos esquemas, como sigue:

1. Se crean dos tareas con prioridad 1. Estas no hacen nada más que imprimir continuamente una cadena.
   Estas tareas nunca hacen ninguna llamada a la función de la API que pueda hacer que entren en el estado de bloqueo, por lo que siempre están en el estado de listo o en el de ejecución. Las tareas de esta naturaleza se denominan tareas de "procesamiento continuo", ya que siempre tienen trabajo que hacer (aunque sea un trabajo bastante trivial, en este caso). La fuente de las tareas de procesamiento continuo se muestra en el Listado 26.

2. A continuación, se crea una tercera tarea con prioridad 2, es decir, por encima de la prioridad de las otras dos tareas. La tercera tarea también sólo imprime una cadena, pero esta vez periódicamente, por lo que utiliza la función de la API vTaskDelayUntil() para colocarse en el estado de Bloqueo entre cada iteración de impresión.

La función de la tarea periódica se muestra en el listado 27.

```
void vContinuousProcessingTask( void *pvParameters )
{
char *pcTaskName;
    /* La cadena a imprimir se pasa a través del parámetro. Convierte esto en un puntero de
    carácter. */
    pcTaskName = ( char * ) pvParameters;

    /* Como la mayoría de las tareas, esta tarea se implementa en un bucle infinito. */
    for( ;; )
    {
        /* Imprime el nombre de esta tarea. Esta tarea sólo hace esto repetidamente sin
        bloquear ni retrasar nunca. */
        vPrintString( pcTaskName );
    }
}
```

Listado 26. La tarea de procesamiento continúo utilizada en el Ejemplo 6

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

    /* La variable xLastWakeTime necesita ser inicializada con la cuenta de tics actual. Tenga
    en cuenta que esta es la única vez que la variable se escribe explícitamente. Después de
    esto xLastWakeTime es gestionado automáticamente por la función de la API
    vTaskDelayUntil(). */
    xLastWakeTime = xTaskGetTickCount();
```

```
        /* Como en la mayoría de las tareas, esta tarea se implementa en un bucle infinito. */
        for( ;; )
        {
            /* Imprime el nombre de esta tarea. */
            vPrintString( "Periodic task is running\r\n" );

            /* La tarea debe ejecutarse cada 3 milisegundos exactamente - ver la declaración de
            xDelay3ms en esta función. */
            vTaskDelayUntil( &xLastWakeTime, xDelay3ms );
        }
}
```

Listado 27. La tarea periódica utilizada en el Ejemplo 6

La Figura 19 muestra la salida producida por el Ejemplo 6, con una explicación del comportamiento observado dada por la secuencia de ejecución mostrada en la Figura 20.

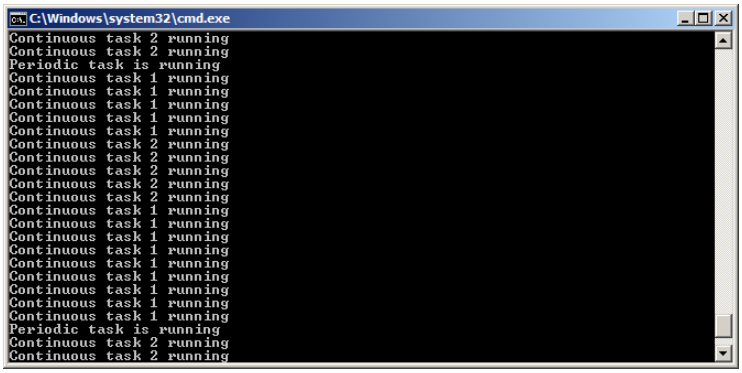

Figura 19. La salida producida cuando se ejecuta el Ejemplo 6

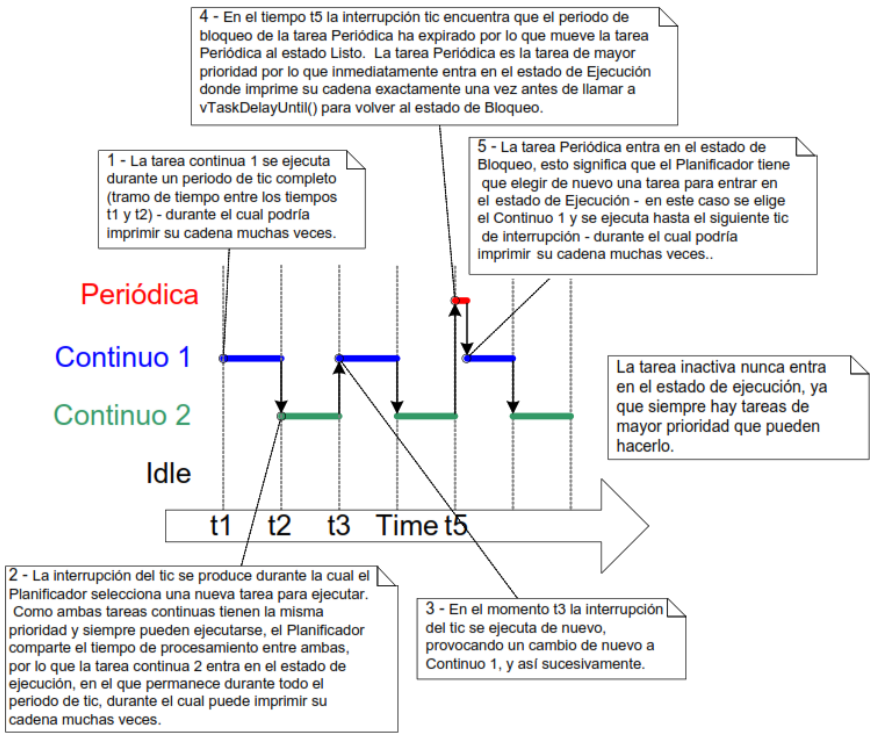

Figure 20. El patrón de ejecución del ejemplo 6

3.9 Cambio de la prioridad de una tarea

Función de la API vTaskPrioritySet()

La función de la API vTaskPrioritySet() se puede utilizar para cambiar la prioridad de cualquier tarea después de que el Planificador se haya iniciado. Tenga en cuenta que la función de la API vTaskPrioritySet() sólo está disponible cuando INCLUDE_vTaskPrioritySet se establece como 1 en FreeRTOSConfig.h.

```
void vTaskPrioritySet( TaskHandle_t pxTask, UBaseType_t uxNewPriority );
```

Listado 31. Prototipo de la función API vTaskPrioritySet()

pxTask El manejador de la tarea cuya prioridad se está modificando (la tarea en cuestión) - véase el parámetro pxCreatedTask de la función API xTaskCreate() para obtener información sobre los manejadores de las tareas.

Una tarea puede cambiar su propia prioridad pasando NULL en lugar de un manejador de tarea válido.

uxNewPriority La prioridad a la que se va a establecer la tarea en cuestión. Esto se limita automáticamente a la máxima prioridad disponible de (configMAX_PRIORITIES – 1), donde configMAX_PRIORITIES es una constante de tiempo de compilación establecida en el archivo de encabezado FreeRTOSConfig.h.

Función de la API uxTaskPriorityGet()

La función de la API uxTaskPriorityGet() puede utilizarse para consultar la prioridad de una tarea. Tenga en cuenta que la función de la API uxTaskPriorityGet() sólo está disponible cuando INCLUDE_uxTaskPriorityGet se establece como 1 en FreeRTOSConfig.h.

```
UBaseType_t uxTaskPriorityGet( TaskHandle_t pxTask );
```

Listado 32. Prototipo de la función API uxTaskPriorityGet()

pxTask El manejador de la tarea cuya prioridad se está modificando (la tarea en cuestión) - véase el parámetro pxCreatedTask de la función API xTaskCreate() para obtener información sobre los manejadores de las tareas.

Una tarea puede cambiar su propia prioridad pasando NULL en lugar de un manejador de tarea válido.

Returned value La prioridad asignada actualmente a la tarea que se está consultando.

Ejemplo 8. Cambiar las prioridades de las tareas

El Planificador siempre seleccionará la tarea más alta del estado Listo como la tarea que entrará en el estado de Ejecución. El ejemplo 8 lo demuestra utilizando la función de la API vTaskPrioritySet() para cambiar la prioridad de dos tareas entre sí.

El ejemplo 8 crea dos tareas con dos prioridades diferentes. Ninguna de las tareas realiza ninguna llamada a la función de la API que pueda hacer que entre en el estado Bloqueado, por lo que ambas están siempre en el estado Listo o en ejecución. Por lo tanto, la tarea con la prioridad relativa más alta siempre será la tarea seleccionada por el planificador para estar en estado En ejecución.

El ejemplo 8 se comporta de la siguiente manera:

1. La tarea 1 (Listado 33) se crea con la prioridad más alta, por lo que se garantiza que se ejecutará primero. La Tarea 1 imprime un par de cadenas antes de elevar la prioridad de la Tarea 2 (Listado 34) por encima de su propia prioridad.

2. La tarea 2 comienza a ejecutarse (ingresa al estado En ejecución) tan pronto como tiene la prioridad relativa más alta. Solo una tarea puede estar en estado En ejecución a la vez, por lo que cuando la Tarea 2 está en estado En ejecución, la Tarea 1 está en estado Lista.

3. La Tarea 2 imprime un mensaje antes de volver a establecer su propia prioridad por debajo de la de la Tarea 1.

4. Si la Tarea 2 vuelve a bajar su prioridad, significa que la Tarea 1 vuelve a ser la tarea de mayor prioridad, por lo que la Tarea 1 vuelve a entrar en el estado En ejecución, obligando a la Tarea 2 a volver al estado Listo.

```
void vTask1( void *pvParameters )
{
UBaseType_t uxPriority;
    /* Esta tarea siempre se ejecutará antes que la Tarea 2, ya que se crea con mayor
    prioridad. Ni la Tarea 1 ni la Tarea 2 se bloquean nunca, por lo que ambas estarán siempre
    en estado En ejecución o Listo.
    Consulte la prioridad con la que se ejecuta esta tarea: pasar NULL significa "devolver la
    prioridad de la tarea que llama". */
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* Imprime el nombre de esta tarea. */
        vPrintString( "Task 1 is running\r\n" );

        /* Establecer la prioridad de la Tarea 2 por encima de la prioridad de la Tarea 1 hará
        que la Tarea 2 comience a ejecutarse inmediatamente (ya que entonces la Tarea 2
        tendrá la prioridad más alta de las dos tareas creadas). Tenga en cuenta el uso del
        identificador de la tarea 2 (xTask2Handle) en la llamada a vTaskPrioritySet(). El
        Listado 35 muestra cómo se obtuvo el mango. */
        vPrintString( "About to raise the Task 2 priority\r\n" );
        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* La Tarea 1 solo se ejecutará cuando tenga una prioridad mayor que la Tarea 2. Por
        lo tanto, para que esta tarea llegue a este punto, la Tarea 2 ya debe haberse
        ejecutado y establecer su prioridad nuevamente por debajo de la prioridad de esta
        tarea. */
    }
}
```

Listado 33. La implementación de la Tarea 1 en el Ejemplo 8

```
void vTask2( void *pvParameters )
{
UBaseType_t uxPriority;
```

```
    /* La Tarea 1 siempre se ejecutará antes que esta tarea, ya que la Tarea 1 se crea con
    mayor prioridad. Ni la Tarea 1 ni la Tarea 2 se bloquean nunca, por lo que siempre estarán
    en estado En ejecución o Listo. Consulta la prioridad con la que se ejecuta esta tarea:
    pasar NULL significa "devolver la prioridad de la tarea que llama"*/
    uxPriority = uxTaskPriorityGet( NULL );
    for( ;; )
    {
        /* Para que esta tarea llegue a este punto, la Tarea 1 ya debe haberse ejecutado y
        establecer la prioridad de esta tarea más alta que la suya.
         Imprime el nombre de esta tarea.. */
        vPrintString( "Task 2 is running\r\n" );

        /* Vuelva a establecer la prioridad de esta tarea en su valor original. Pasar NULL
        como identificador de tarea significa "cambiar la prioridad de la tarea de llamada".
        Establecer la prioridad por debajo de la Tarea 1 hará que la Tarea 1 comience a
        ejecutarse de inmediato nuevamente, anticipándose a esta tarea.. */
        vPrintString( "About to lower the Task 2 priority\r\n" );
        vTaskPrioritySet( NULL, ( uxPriority - 2 ) );

    }
}
```

Listado 34. La implementación de la Tarea 2 en el Ejemplo 8

Cada tarea puede consultar y establecer su propia prioridad sin necesidad de utilizar un manejador de tarea válido, simplemente utilizando NULL, en su lugar. Un manejador de tarea es necesario sólo cuando una tarea desea hacer referencia a otra tarea que no sea ella misma, como cuando la Tarea 1 cambia la prioridad de la Tarea 2. Para permitir que la Tarea 1 haga esto, el manejador de la Tarea 2 se obtiene y se guarda cuando se crea la Tarea 2, como se destaca en los comentarios del Listado 35.

```
/* Declarar una variable que se utiliza para mantener el manejador de la Tarea 2. */
TaskHandle_t xTask2Handle = NULL; int main( void )
{
    /* Crea la primera tarea con prioridad 2. El parámetro de la tarea no se utiliza y se
    establece como NULL. El manejador de la tarea tampoco se utiliza, por lo que también se
    establece como NULL. */
    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );
    /* La tarea se crea con prioridad 2 ______^. */

    /* Cree la segunda tarea con prioridad 1, que es inferior a la prioridad otorgada a la Tarea
    1. Nuevamente, el parámetro de la tarea no se usa, por lo que se establece en NULL, PERO
    esta vez se requiere el manejador de la tarea, por lo que la dirección de xTask2Handle se
    pasa en el último parámetro. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );
    /* El manejador de la tarea es el último parámetro _____^^^^^^^^^^^^ */

    /* Iniciar el Planificador para que las tareas comiencen a ejecutarse. */
    vTaskStartScheduler();

    /* Si todo está bien, main() nunca llegará aquí, ya que el programador ahora ejecutará las
    tareas. Si main() llega aquí, es probable que no haya suficiente memoria en el montón
```

```
    /* disponible para crear la tarea inactiva. El Capítulo 2 proporciona más información sobre la
    gestión de la memoria en montón. */
    for( ;; );
}
```

Listado 35. La implementación de main() para el Ejemplo 8

La Figura 22 demuestra la secuencia en la que se ejecutan las tareas del Ejemplo 8, con el resultado resultante que se muestra en la Figura 23.

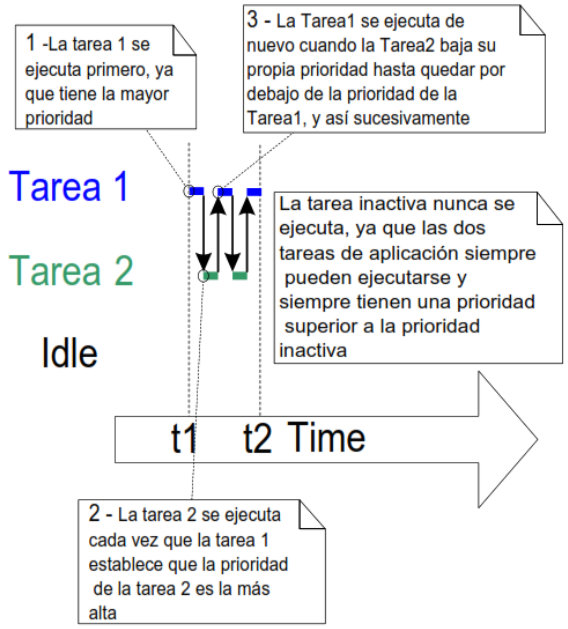

Figura 22. La secuencia de ejecución de tareas al ejecutar el Ejemplo 8

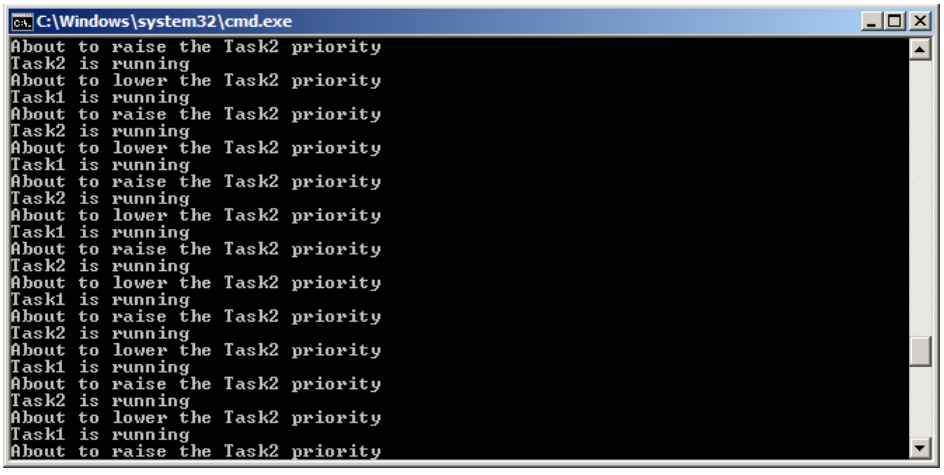

Figura 23. La salida producida cuando se ejecuta el Ejemplo 8