
Kernel FreeRTOS

Guía para desarrolladores

Amazon's trademarks and trade dress may not be used in connection with any product or service that is not Amazon's, in any manner that is likely to cause confusion among customers, or in any manner that disparages or discredits Amazon. All other trademarks not owned by Amazon are the property of their respective owners, who may or may not be affiliated with, connected to, or sponsored by Amazon.

Table of Contents

Acerca del kernel FreeRTOS	1
Propuesta de valor	1
Nota acerca de la terminología	1
Razones para utilizar un kernel en tiempo real	2
Características del kernel FreeRTOS	3
Licencias	4
Archivos y proyectos originales incluidos	4
Distribución del kernel FreeRTOS	5
Descripción de la distribución del kernel FreeRTOS	5
Creación del kernel FreeRTOS	5
FreeRTOSConfig.h	5
La distribución del kernel FreeRTOS oficial	5
Los principales directorios de la distribución de FreeRTOS	6
Archivos de origen de FreeRTOS comunes para todos los puertos	6
Archivos de origen de FreeRTOS específicos de un puerto	7
Archivos de encabezado	8
Aplicaciones de demostración	9
Creación de una proyecto de FreeRTOS	10
Creación de un proyecto nuevo a partir de cero	11
Guía de tipos de datos y estilo de codificación	12
Nombres de variables	12
Nombre de funciones	12
Formato	12
Nombres de macro	12
Lógica del exceso de conversiones de tipos	13
Administración de la memoria en montón	14
Requisitos previos	14
Asignación de memoria dinámica y su relevancia para FreeRTOS	14
Opciones para la asignación de memoria dinámica	15
Métodos de asignación de memoria de ejemplo	15
Heap_1	15
Heap_2	16
Heap_3	18
Heap_4	18
Configuración de una dirección de inicio para la matriz usada por Heap_4	20
Heap_5	20
La función de API vPortDefineHeapRegions()	20
Funciones de utilidades relacionadas con el montón	24
La función de API xPortGetFreeHeapSize()	24
La función de API xPortGetMinimumEverFreeHeapSize()	25
Funciones de enlace erróneas de malloc	25
Administración de tareas	26
Las funciones de las tareas	26
Los estados de las tareas de nivel superior	27
Creación de tareas	28
La función de API xTaskCreate()	28
Creación de tareas (ejemplo 1)	29
Uso del parámetro de tarea (ejemplo 2)	32
Prioridades de las tareas	34
La medida del tiempo y la interrupción de ciclo	35
Experimentación con prioridades (ejemplo 3)	36
Ampliación del estado No está en ejecución	38
El estado Bloqueado	38
El estado suspendido	39

El estado Listo	39
Cumplimentación del Diagrama de transición de estado	39
Uso del estado Bloqueado para crear un retraso (ejemplo 4)	40
La función de API vTaskDelayUntil()	44
Conversión de la tareas de ejemplo para utilizar vTaskDelayUntil() (ejemplo 5)	45
Combinación de tareas de bloqueo y tareas que no bloquean (ejemplo 6)	46
La tarea de inactividad y el enlace de tareas de inactividad	48
Funciones de enlace de tareas de inactividad	49
Limitaciones en la Implementación de funciones de enlace de tareas de inactividad	49
Definición de una función de enlace de tarea de inactividad (ejemplo 7)	50
Cambio de la prioridad de una tarea	51
Función de API vTaskPrioritySet()	51
Función de API uxTaskPriorityGet()	52
Cambio de prioridades de tareas (ejemplo 8)	52
Eliminación de una tarea	55
Función de API vTaskDelete()	55
Eliminación de tareas (ejemplo 9)	56
Programación de algoritmos	58
Un resumen de estados de tareas y eventos	58
Configuración del algoritmo de programación	59
Programación de reemplazo prioritario con intervalos de tiempo	59
Programación de reemplazo prioritario (sin intervalos de tiempo)	62
Programación cooperativa	64
Administración de colas	67
Características de una cola	67
Almacenamiento de datos	67
Acceso de varias tareas	69
Bloqueo en las lecturas de cola	69
Bloqueo en las escrituras de cola	69
Bloqueo en varias colas	70
Uso de una cola	70
Función de API xQueueCreate()	70
Funciones de API xQueueSendToBack() y xQueueSendToFront()	71
Función de API xQueueReceive()	72
Función de API uxQueueMessagesWaiting()	74
Bloqueo al recibir de una cola (Ejemplo 10)	74
Recepción de datos de varios orígenes	78
Bloqueo al enviar a una cola y envío de estructuras en una cola (Ejemplo 11)	79
Uso de datos de tamaño grande o variable	84
Colocación de punteros en la cola	84
Uso de una cola para enviar distintos tipos y longitudes de datos	86
Recepción desde varias colas	89
Conjuntos de colas	89
Función de API xQueueCreateSet()	90
Función de API xQueueAddToSet()	91
Función de API xQueueSelectFromSet()	91
Uso de un conjunto de colas (Ejemplo 12)	92
Casos de uso de conjuntos de colas más realistas	96
Uso de una cola para crear un buzón de correo	98
Función de API xQueueOverwrite()	99
Función de API xQueuePeek()	100
Administración del temporizador de software	102
Las funciones de devolución de llamada del temporizador de software	102
Atributos y estados de un temporizador de software	103
Periodo de un temporizador de software	103
Temporizadores de una activación y temporizadores de recarga automática	103
Estados de los temporizadores de software	103

El contexto de los temporizadores de software	105
Tarea de demonio RTOS (servicio de temporizador)	105
La cola de comandos del temporizador	105
Programación de tareas de demonio	106
Creación e inicio de un temporizador de software	108
La función de API xTimerCreate()	108
La función de API xTimerStart()	109
Creación de temporizadores de una activación y temporizadores de recarga automática (ejemplo 13)	111
El ID de los temporizadores	114
La función de API vTimerSetTimerID()	114
La función de API pvTimerGetTimerID()	115
Uso del parámetro de función de devolución de llamada y el ID del temporizador de software (ejemplo 14)	115
Cambio de los periodos de un temporizador	117
La función de API xTimerChangePeriod()	117
Restablecimiento de un temporizador de software	120
La función de API xTimerReset()	121
Restablecimiento de un temporizador de software (ejemplo 15)	122
Administración de interrupciones	125
API a prueba de interrupciones	125
Ventajas de utilizar una API a prueba de interrupciones independiente	126
Desventajas de utilizar una API a prueba de interrupciones independiente	126
Parámetro xHigherPriorityTaskWoken	127
Macros portYIELD_FROM_ISR() y portEND_SWITCHING_ISR()	128
Procesamiento diferido de interrupciones	128
Semáforos binarios utilizados para la sincronización	130
Función de API xSemaphoreCreateBinary()	133
Función de API xSemaphoreTake()	133
Función de API xSemaphoreGiveFromISR()	134
Uso de un semáforo binario para sincronizar una tarea con una interrupción (Ejemplo 16)	135
Mejora de la implementación de la tarea utilizada en el Ejemplo 16	139
Semáforos de recuento	143
Función de API xSemaphoreCreateCounting()	145
Uso de un semáforo de recuento para sincronizar una tarea con una interrupción (Ejemplo 17)	146
Diferir el trabajo en la tarea de demonio de RTOS	147
Función de API xTimerPendFunctionCallFromISR()	148
Procesamiento de interrupciones diferido centralizado (Ejemplo 18)	149
Uso de colas en una rutina del servicio de interrupciones	152
Funciones de API xQueueSendToFrontFromISR() y xQueueSendToBackFromISR()	152
Aspectos a tener en cuenta al usar una cola desde una ISR	153
Envío y recepción en una cola desde una interrupción (Ejemplo 19)	153
Anidamiento de interrupciones	158
Usuarios de Cortex-M y GIC de ARM	159
Administración de recursos	161
Exclusión mutua	163
Secciones críticas y suspensión del programador	163
Secciones críticas básicas	163
Suspensión (o bloqueo) del programador	166
Función de API vTaskSuspendAll()	166
Función de API xTaskResumeAll()	166
Mutex (y semáforos binarios)	167
Función de API xSemaphoreCreateMutex()	167
Reescritura de vPrintString() para utilizar un semáforo (Ejemplo 20)	168
Inversión de prioridades	171
Herencia de prioridades	172
Interbloqueo (o abrazo mortal)	173

Mutex recursivos	174
Mutex y programación de tareas	175
Tareas de guardián	179
Reescritura de vPrintString() para utilizar una tarea de guardián (Ejemplo 21)	179
Grupos de eventos	184
Características de un grupo de eventos	184
Grupos de eventos, marcas de eventos y bits de eventos	184
Más acerca del tipo de datos EventBits_t	185
Acceso de varias tareas	69
Ejemplo práctico del uso de un grupo de eventos	185
Administración de eventos mediante grupos de eventos	186
Función de API xEventGroupCreate()	186
Función de API xEventGroupSetBits()	186
Función de API xEventGroupSetBitsFromISR()	187
Función de API xEventGroupWaitBits()	188
Experimentación con grupos de eventos (Ejemplo 22)	192
Sincronización de tareas mediante un grupo de eventos	196
Función de API xEventGroupSync()	199
Sincronización de tareas (Ejemplo 23)	201
Notificaciones de tareas	204
Comunicación a través de objetos intermediarios	204
Notificaciones de tareas: comunicación directa a la tarea	204
Beneficios y limitaciones de las notificaciones de tareas	205
Limitaciones de las notificaciones de tareas	205
Uso de notificaciones de tareas	206
Opciones de API de notificación de tareas	206
La función de API xTaskNotifyGive()	207
La función de API vTaskNotifyGiveFromISR()	207
La función de API ulTaskNotifyTake()	208
Método 1 para usar una tarea notificación en lugar de un semáforo (ejemplo 24)	209
Método 2 para usar una tarea notificación en lugar de un semáforo (ejemplo 25)	212
Funciones de API xTaskNotify() y xTaskNotifyFromISR()	214
La función de API xTaskNotifyWait()	216
Notificaciones de tarea usadas en controladores de dispositivos periféricos: ejemplo de UART	218
Notificaciones de tarea usadas en controladores de dispositivos periféricos: ejemplo de ADC	224
Notificaciones de tarea usadas directamente dentro de una aplicación	226
Developer Support	231
configASSERT()	231
Definiciones de ejemplo de configASSERT()	231
Tracealyzer	232
Funciones de enlace relacionadas con la depuración (devolución de llamadas)	235
Visualización de la información de estado de tareas y tiempo de ejecución	235
Estadísticas del tiempo de ejecución de tareas	235
Reloj de estadísticas de tiempo de ejecución	236
Configuración de una aplicación para recopilar estadísticas de tiempo de ejecución	236
Función de API uxTaskGetSystemState()	237
Función auxiliar vTaskList()	239
Función auxiliar vTaskGetRunTimeStats()	241
Generación y visualización de estadísticas en tiempo de ejecución: ejemplo de trabajo	242
Macros de enlace de seguimiento	244
Macros de enlace de seguimiento disponibles	244
Definición de macros de enlace de seguimiento	246
Complementos de depurador compatibles con FreeRTOS	246
Solución de problemas	248
Introducción y ámbito del capítulo	248
Prioridades de interrupción	248
Desbordamiento de pila	249

La función de API uxTaskGetStackHighWaterMark()	249
Información general de la comprobación de pila en tiempo de ejecución	250
Método 1 para comprobar la pila en tiempo de ejecución	250
Método 2 para comprobar la pila en tiempo de ejecución	250
Uso incorrecto de printf() y sprintf().	251
Printf-stdarg.c	251
Otros errores habituales	251
Síntoma: si se añade una tarea sencilla a una demostración esta se bloquea.	251
Síntoma: si se usa una función de la API en una interrupción la aplicación se bloquea.	252
Síntoma: en ocasiones la aplicación se bloquea en una rutina de servicio de interrupción.	252
Síntoma: el programador se bloquea al intentar comenzar la primera tarea.	252
Síntoma: las interrupciones se quedan desactivadas de forma inesperada o bien hay secciones críticas que no se anidan correctamente	253
Síntoma: la aplicación se bloquea incluso antes de que se inicie el programador.	253
Síntoma: si se llama a funciones de API mientras el programador está suspendido o desde dentro de una sección crítica, la aplicación se bloquea.	253

Acerca del kernel FreeRTOS

El kernel FreeRTOS es un software de código abierto que mantiene Amazon.

El kernel FreeRTOS es ideal para aplicaciones en tiempo real profundamente integradas que utilizan microcontroladores o pequeños microprocesadores. Normalmente, este tipo de aplicaciones incluyen una combinación de requisitos "duros" y "blandos" en tiempo real.

Los requisitos "blandos" en tiempo real son los que establecen un plazo de tiempo, pero aunque se incumpla el plazo, el sistema puede seguir funcionando. Por ejemplo, si la respuesta a las pulsaciones de teclas es muy lenta, es posible que ese problema haga que el sistema sea incómodo de usar, pero sigue funcionando.

Los requisitos "duros" en tiempo real son los que establecen un plazo de tiempo y, si se incumple ese plazo, se generaría un error absoluto del sistema. Por ejemplo, si un airbag de conductor responde con demasiada lentitud a las entradas de los sensores de choque, podría hacer más mal que bien.

El kernel FreeRTOS es un kernel en tiempo real (o un programador en tiempo real) sobre el que se pueden construir aplicaciones integradas para satisfacer los requisitos "duros" en tiempo real. Permite organizar las aplicaciones como una colección de subprocesos de ejecución independientes. En un procesador que solo tiene un núcleo, solo se puede ejecutar un subproceso al mismo tiempo. El kernel decide qué subproceso se debe ejecutar examinando la prioridad que ha asignado a cada subproceso el diseñador de la aplicación. En el caso más sencillo, el diseñador de la aplicación podría asignar prioridades más altas a los subprocesos que implementan los requisitos "duros" en tiempo real y las prioridades más bajas a los subprocesos que implementan requisitos "blandos" en tiempo real. Esto garantizaría que los subprocesos "duros" en tiempo real se ejecuten siempre antes que los subprocesos "blandos" en tiempo real, aunque las decisiones sobre la asignación de prioridades no son siempre tan simplistas.

No se preocupe si aún no comprende del todo los conceptos del apartado anterior. En esta guía, se explican detalladamente y se proporcionan muchos ejemplos que le ayudarán a entender cómo utilizar un kernel en tiempo real y el kernel FreeRTOS en particular.

Propuesta de valor

El éxito global sin precedentes del kernel FreeRTOS se debe a su atractiva propuesta de valor. El kernel FreeRTOS está desarrollado con profesionalidad, se ha sometido a un control de calidad estricto, es robusto, tiene soporte, no contiene ambigüedades sobre la propiedad intelectual y es realmente de uso gratuito en aplicaciones comerciales sin el requisito de exponer el código fuente propietario. Puede comercializar un producto con el kernel FreeRTOS sin pagar ninguna tasa, y miles de personas ya lo están haciendo. Si, en cualquier momento, desea recibir soporte adicional o si su equipo jurídico requiere garantías adicionales o una cláusula de indemnización por escrito, hay una ruta de actualización comercial muy sencilla de bajo costo. Puede trabajar con la tranquilidad de saber que puede utilizar esa ruta comercial en el momento que lo desee.

Nota acerca de la terminología

En el kernel FreeRTOS, cada subproceso de ejecución se denomina tarea. Aunque no existe un consenso sobre la terminología en la comunidad integrada, el término subproceso puede tener un significado más específico en algunos campos de aplicación.

Razones para utilizar un kernel en tiempo real

Existen muchas técnicas bien establecidas para escribir software integrado de buena calidad sin el uso de un kernel y, si el sistema que se va a desarrollar es sencillo, esas técnicas pueden ser la solución más adecuada. En casos más complejos, es preferible utilizar un kernel, pero el límite entre unos casos y otros siempre es algo subjetivo.

La priorización de tareas ayuda a garantizar que una aplicación cumpla sus plazos de procesamiento, pero un kernel puede tener también otros beneficios que no son tan evidentes:

- Abstracción de la información de tiempo

El kernel es responsable del tiempo de ejecución y proporciona una API relacionada con el tiempo a la aplicación. Esto permite que la estructura del código de la aplicación sea más sencilla y que tamaño general del código sea más reducido.

- Capacidad de mantenimiento/ampliación

Al abstraer los detalles del tiempo, se obtienen menos interdependencias entre los módulos y el software puede evolucionar de una forma controlada y predecible. Además, el kernel es responsable del tiempo, por lo que el rendimiento de la aplicación es menos susceptible a sufrir cambios en el hardware subyacente.

- Modularidad

Las tareas son módulos independientes, cada uno de los cuales debe tener un objetivo bien definido.

- Desarrollo en equipo

Las tareas también deben tener interfaces bien definidas, lo que facilita el desarrollo a los equipos.

- Pruebas más fáciles

Si las tareas son módulos independientes bien definidos con interfaces claras, se pueden probar de manera aislada.

- Reutilización del código

Gracias a la mayor modularidad y la menor cantidad de interdependencias, el código se puede reutilizar con menos esfuerzo.

- Mayor eficacia

El uso de un kernel permite que el software pueda basarse por completo en eventos, por lo que no se desperdicia ningún tiempo de procesamiento sondeando eventos que no se han producido. El código solo se ejecuta cuando hay algo que debe hacerse.

Algo que afecta negativamente a la eficacia es la necesidad de procesar la interrupción de ciclos de RTOS y cambiar la ejecución de una tarea a otra. Sin embargo, las aplicaciones que no utilizan un RTOS suelen incluir algún tipo de interrupción de ciclos.

- Tiempo de inactividad

La tarea de inactividad se crea automáticamente cuando se inicia el programador. Se ejecuta cuando no hay tareas de la aplicación que deseen ejecutarse. La tarea de inactividad se puede utilizar para medir la capacidad de procesamiento libre, para realizar comprobaciones en segundo plano o simplemente para poner el procesador en un modo de bajo consumo.

- Administración de energía

La mejora de la eficiencia gracias al uso de un RTOS permite que el procesador esté más tiempo en un modo de bajo consumo.

El consumo de energía puede reducirse de manera significativa al poner el procesador en un estado de bajo consumo cada vez que se ejecuta la tarea de inactividad. El kernel FreeRTOS también tiene un modo sin ciclos que permite al procesador entrar y permanecer en el modo de bajo consumo durante más tiempo.

- Control flexible de interrupciones

Los controladores de interrupciones pueden hacerse muy cortos retrasando el procesamiento a una tarea creada por el programador de aplicaciones o a una tarea de demonio de FreeRTOS.

- Combinación de requisitos de procesamiento

Si se utilizan patrones de diseño sencillos, se puede conseguir una combinación de procesamiento periódico, continuos y basados en eventos dentro de una aplicación. Además, se pueden cumplir los requisitos "duros" y "blandos" en tiempo real mediante las prioridades de tareas e interrupciones adecuadas.

Características del kernel FreeRTOS

El kernel FreeRTOS tiene las siguientes características estándar:

- Funcionamiento prioritario o colaborativo
- Asignación muy flexible de prioridades de las tareas
- Mecanismo de notificación de tareas flexible, rápido y ligero
- Colas
- Semáforos binarios
- Semáforos de recuento
- Mutex
- Mutex recursivos
- Temporizadores de software
- Grupos de eventos
- Funciones de enlace de ciclos
- Funciones de enlace de inactividad
- Comprobación de desbordamiento de pila
- Registro de seguimiento
- Recopilación de estadísticas del tiempo de ejecución de las tareas
- Soporte y licencias comerciales opcionales
- Modelo anidado de interrupciones completo (para algunas arquitecturas)
- Capacidad sin ciclos para aplicaciones de bajo consumo extremo

- Pila de interrupciones administrada por software cuando es apropiado (esto puede ayudar a ahorrar RAM)

Licencias

El kernel FreeRTOS está disponible para los usuarios en virtud de los términos de la licencia MIT.

Archivos y proyectos originales incluidos

En un archivo zip adjunto, se proporciona el código fuente, los archivos del proyecto preconfigurados y las instrucciones de compilación completas para todos los ejemplos presentados. Puede descargar el archivo zip de <http://www.FreeRTOS.org/Documentation/code> si no ha recibido una copia con el libro. Es posible que el archivo zip no incluya la versión más reciente del kernel FreeRTOS.

Las capturas de pantalla incluidas en este libro se realizaron ejecutando los ejemplos en un entorno de Microsoft Windows, utilizando el puerto de Windows de FreeRTOS. El proyecto que utiliza el puerto de Windows de FreeRTOS está preconfigurado para compilarse con la edición Express gratuita de Visual Studio, que se puede descargar de <https://www.visualstudio.com/vs/community/>. Aunque el puerto de Windows de FreeRTOS proporciona una plataforma de evaluación, prueba y desarrollo muy cómoda, no refleja el verdadero comportamiento en tiempo real.

Distribución del kernel FreeRTOS

El kernel FreeRTOS se distribuye como un archivo zip único que contiene todos los puertos de kernel FreeRTOS oficiales y un gran número de aplicaciones de demostración preconfiguradas.

Descripción de la distribución del kernel FreeRTOS

El kernel FreeRTOS puede construirse con aproximadamente 20 compiladores diferentes y se puede ejecutar en más de 30 arquitecturas de procesador diferentes. Se considera que cada combinación de compilador y procesador compatible es un puerto FreeRTOS independiente.

Creación del kernel FreeRTOS

Se puede pensar en FreeRTOS como una biblioteca que proporciona capacidades de multitarea a lo que de lo contrario sería una aplicación sin sistema operativo.

FreeRTOS se suministra como un conjunto de archivos de origen C. Algunos de los archivos de origen son comunes para todos los puertos, mientras que otros son específicos de un puerto. Compile los archivos de origen como parte de su proyecto para que la API de FreeRTOS esté disponible para su aplicación. Para facilitarle esta tarea, todos los puertos FreeRTOS oficiales se proporcionan con una aplicación de demostración. La aplicación de demostración está preconfigurada para compilar los archivos de origen correctos e incluir los archivos de encabezado correctos.

Las aplicaciones de demostración deben compilarse tal y como se entregan. Si, por el contrario, se efectúa un cambio en las herramientas de compilación desde que se lanzó la demostración, pueden generarse problemas. Para obtener más información, consulte Aplicaciones de demostración más adelante en este tema.

FreeRTOSConfig.h

FreeRTOS se configura con un archivo de encabezado llamado FreeRTOSConfig.h.

FreeRTOSConfig.h se utiliza para adaptar FreeRTOS para usarlo en una aplicación específica. Por ejemplo, FreeRTOSConfig.h contiene constantes como configUSEPREEMPTION, cuya configuración define si se usará el algoritmo de programación de reemplazo o programación cooperativa. FreeRTOSConfig.h contiene definiciones específicas de aplicaciones, por lo que debería estar en un directorio que forme parte de la aplicación que se crea, no en un directorio que contenga el código fuente de FreeRTOS.

Se proporciona una aplicación de demostración para cada puerto FreeRTOS y cada aplicación de demostración contiene un archivo FreeRTOSConfig.h. Por lo tanto, no necesitará nunca crear un archivo FreeRTOSConfig.h desde cero. En su lugar, le recomendamos que comience con el FreeRTOSConfig.h que usa la aplicación de demostración proporcionada y lo adapte para el puerto FreeRTOS en uso.

La distribución del kernel FreeRTOS oficial

FreeRTOS se distribuye en un solo archivo zip. El archivo zip contiene código fuente para todos los puertos y los archivos de proyecto de FreeRTOS de todas las aplicaciones de demostración de FreeRTOS.

También contiene una selección de componentes del ecosistema de FreeRTOS+ y una selección de aplicaciones de demostración del ecosistema de FreeRTOS+.

No se preocupe por el número de archivos de la distribución de FreeRTOS. Solo se necesita un pequeño número de archivos en cualquier aplicación.

Los principales directorios de la distribución de FreeRTOS

Los directorios de primer y segundo nivel de la distribución de FreeRTOS se muestran y se describen aquí.

FreeRTOS

| |

| └─Directorio de origen que contiene los archivos de origen de FreeRTOS.

| |

| └─Directorio de demostración que contiene proyectos de demostración de FreeRTOS específicos de puertos y configurados previamente.

|

FreeRTOS-Plus

|

| └─Directorio de origen que contiene código fuente para algunos componentes del ecosistema FreeRTOS+.

|

| └─Directorio de demostración que contiene proyectos de demostración para componentes del ecosistema FreeRTOS+.

El archivo zip contiene solo una copia de los archivos de origen de FreeRTOS, todos los proyectos de demostración de FreeRTOS y todos los proyectos de demostración de FreeRTOS+. Debe encontrar los archivos de origen de FreeRTOS en el directorio FreeRTOS/Source. Puede que los archivos no se compilen si la estructura de directorios ha cambiado.

Archivos de origen de FreeRTOS comunes para todos los puertos

El código fuente de FreeRTOS básico se encuentra en tan solo dos archivos C que son comunes a todos los puertos FreeRTOS. Se denominan `tasks.c` y `list.c`. Se encuentran directamente en el directorio FreeRTOS/Source. Los archivos de origen siguientes están en el mismo directorio:

- `queue.c`

`queue.c` proporciona servicios de cola y de semáforo. `queue.c` es casi siempre obligatorio.

- `timers.c`

`timers.c` proporciona una funcionalidad de temporizador de software. Debe incluirlo en la compilación solo si se van a usar temporizadores de software.

- eventgroups.c

eventgroups.c proporciona una funcionalidad de grupos de eventos. Debe incluirlo en la compilación solo si se van a usar grupos de eventos.

- croutine.c

croutine.c implementa la funcionalidad de corutina de FreeRTOS. Debe incluirlo en la compilación solo si se van a usar corutinas. Las corutinas estaban pensadas para utilizarse con microcontroladores muy pequeños. Ahora se usan en raras ocasiones y, por lo tanto, no se mantienen en el mismo nivel que otras características de FreeRTOS. Las corutinas no están cubiertas en esta guía.

FreeRTOS

|

└─Origen

|

└─Archivo de origen tasks.c de FreeRTOS: obligatorio siempre

└─Archivo de origen list.c de FreeRTOS: obligatorio siempre

└─Archivo de origen queue.c de FreeRTOS: obligatorio prácticamente siempre

└─Archivo de origen timers.c de FreeRTOS: opcional

└─Archivo de origen eventgroups.c de FreeRTOS: opcional

└─Archivo de origen croutine.c de FreeRTOS: opcional

Se ha demostrado que los nombres de archivo podrían dar lugar a problemas por el espacio de nombres, ya que muchos proyectos ya incluirán archivos con los mismos nombres. Sin embargo, cambiar los nombres de los archivos ahora sería problemático, ya que se acabaría con la compatibilidad de miles de proyectos que utilizan FreeRTOS, así como con herramientas de automatización y complementos de IDE.

Archivos de origen de FreeRTOS específicos de un puerto

Los archivos de origen específicos de un puerto FreeRTOS se encuentran en el directorio FreeRTOS/Source/portable. El directorio portable está organizado siguiendo una jerarquía, en primer lugar por compilador y después por arquitectura de procesador.

Si ejecuta FreeRTOS en un procesador con la arquitectura "arquitectura" y el compilador "compilador", tendrá que compilar no solo los archivos de origen de FreeRTOS de núcleo, sino también los archivos que se encuentran en el directorio FreeRTOS/Source/portable/[compilador]/[arquitectura].

Como se describe en el capítulo 2, Administración de la memoria en montón, FreeRTOS también considera que la asignación de memoria en montón forma parte de la capa portátil. Los proyectos que utilizan una versión de FreeRTOS posterior a la V9.0.0 tienen que incluir un administrador de memoria en montón. A partir de FreeRTOS V9.0.0, solo se necesita un administrador de memoria en montón si configSUPPORTDYNAMICALLY se establece en 1 en FreeRTOSConfig.h, o si configSUPPORTDYNAMICALLY se deja sin definir.

FreeRTOS proporciona cinco métodos de asignación de montón de ejemplo. Los cinco métodos reciben nombres correlativos entre heap1 y heap5, y los implementan respectivamente los archivos de origen heap1.c a heap5.c. Los métodos de asignación de montón de ejemplo se encuentran en el directorio

FreeRTOS/Source/portable/MemMang. Si ha configurado FreeRTOS para utilizar una asignación de memoria dinámica, es necesario compilar uno de estos cinco archivos de origen en su proyecto, a menos que su aplicación proporcione una implementación alternativa.

En la figura siguiente se muestran los archivos de origen específicos de los puertos en el árbol de directorios de FreeRTOS.

FreeRTOS

|

└─Origen

|

└─Directorio portátil que contiene todos los archivos de origen específicos de puerto

|

└─Directorio MemMang que contiene los 5 archivos de origen de asignación de montón alternativos

|

└─[compilador 1] Directorio que contiene archivos de puerto específicos del compilador 1

| |

| └─[arquitectura 1] Contiene archivos para el puerto compilador 1 arquitectura 1

| └─[arquitectura 2] Contiene archivos para el puerto compilador 1 arquitectura 2

| └─[arquitectura 3] Contiene archivos para el puerto compilador 1 arquitectura 3

|

└─[compilador 2] Directorio que contiene archivos de puerto específicos del compilador 2

|

└─[arquitectura 1] Contiene archivos para el puerto compilador 2 arquitectura 1

└─[arquitectura 2] Contiene archivos para el puerto compilador 2 arquitectura 2

└─[etc.]

Rutas de inclusión

FreeRTOS requiere que se incluyan tres directorios en la ruta de inclusión del compilador:

1. La ruta a los archivos de encabezado de FreeRTOS principales, que es siempre FreeRTOS/Source/include.
2. La ruta a los archivos de origen que son específicas del puerto FreeRTOS en uso. Como se ha descrito anteriormente, se trata de FreeRTOS/Source/portable/[compilador]/[arquitectura].
3. Una ruta al archivo de encabezado FreeRTOSConfig.h.

Archivos de encabezado

Un archivo de origen que utiliza la API FreeRTOS debe incluir "FreeRTOS.h", seguido del archivo de encabezado que contiene el prototipo para la función de API que se usa, ya sea "task.h", "queue.h", "semphr.h", "timers.h" o "eventgroups.h".

Aplicaciones de demostración

Cada puerto FreeRTOS incluye como mínimo una aplicación de demostración que debe compilarse sin errores ni advertencias, aunque algunas demostraciones son más antiguas que otras y, en ocasiones, un cambio en las herramientas de compilación creadas desde el lanzamiento de la demostración pueda provocar un problema.

Nota para usuarios de Linux: FreeRTOS se desarrolla y prueba en un host de Windows. En ocasiones esto crea errores de compilación cuando se compilan proyectos de demostración en un host de Linux. Los errores de compilación casi siempre están relacionados con el uso de mayúsculas o minúsculas en las letras que se utilizan para hacer referencia a los nombres de archivos, o la dirección de la barra inclinada que se usa en las rutas de archivos. Utilice el formulario de contacto de FreeRTOS (<http://www.FreeRTOS.org/contact>) para alertarnos de cualquiera de estos errores.

La aplicación de demostración tiene varias finalidades:

- Proporcionar un ejemplo de un proyecto configurado previamente que funciona que tiene los archivos correctos incluidos y el conjunto de opciones de compilador correcto.
- Permitir una experimentación directa con una configuración o conocimientos previos mínimos.
- Como demostración de cómo la API de FreeRTOS puede utilizarse.
- Como base a partir de la cual pueden crearse aplicaciones reales.

Cada proyecto de demostración se encuentra en un subdirectorio único en el directorio/FreeRTOS/Demo. El nombre del subdirectorio indica el puerto al que se refiere el proyecto de demostración.

Cada aplicación de demostración también se describe en una página web en el sitio web FreeRTOS.org. La página web proporciona información acerca de:

- Cómo localizar el archivo de proyecto para la demostración dentro de la estructura de directorios de FreeRTOS.
- Qué hardware tiene configurado el proyecto para utilizarlo.
- Cómo configurar el hardware para ejecutar la demostración.
- Cómo compilar la demostración.
- Cómo se espera que se comporte la demostración.

Todos los proyectos de demostración crean un subconjunto de tareas de demostración comunes, cuya implementación se encuentra en el directorio FreeRTOS/Demo/Common/Minimal. Las tareas de demostración comunes existen únicamente para demostrar la forma en que se puede utilizar la API de FreeRTOS; no implementan ninguna funcionalidad especialmente útil.

Los proyectos de demostración más recientes también pueden crear un proyecto "blinky" para principiantes. Los proyectos blinky son muy básicos. Por lo general crean solo dos tareas y una cola.

Todos los proyectos de demostración incluyen un archivo llamado main.c. Esto contiene la función main(), desde donde se crean las tareas de la aplicación de demostración. Consulte los comentarios de los archivos main.c individuales para recibir información específica de esa demostración.

A continuación se muestra la jerarquía de directorios de FreeRTOS/Demo.

FreeRTOS

|

└─Directorio de demostración que contiene todos los proyectos de demostración

|

- └─[Demo x] Contiene el archivo de proyecto que compila la demostración "x".
|
- └─[Demo y] Contiene el archivo de proyecto que compila la demostración "y".
|
- └─[Demo z] Contiene el archivo de proyecto que compila la demostración "z".
|
- └─Common Contiene archivos que todas las aplicaciones de demostración compilan.

Creación de una proyecto de FreeRTOS

Todos los puertos FreeRTOS incluyen al menos una aplicación de demostración configurada previamente que en principio se compila sin errores ni advertencias. Se recomienda que los nuevos proyectos se creen adaptando uno de estos proyectos ya existentes. De este modo, el proyecto tendrá los archivos correctos incluidos, los controladores de interrupción correctos instalados y el conjunto de opciones de compilador correcto.

Para iniciar una nueva aplicación a partir de un proyecto de demostración existente:

1. Abra el proyecto de demostración suministrado y asegúrese de que se compila y ejecuta según lo previsto.
2. Elimine los archivos de origen que definen las tareas de demostración. Cualquier archivo que se encuentre en el directorio Demo/Common se puede quitar del proyecto.
3. Elimine todas las llamadas de función de `main()`, excepto `prvSetupHardware()` y `vTaskStartScheduler()`, tal y como se muestra en la lista 1.
4. Compruebe si el proyecto se sigue compilando.

Seguir estos pasos creará un proyecto que incluye los archivos de origen de FreeRTOS correctos, pero no definirá ninguna funcionalidad.

```
int main( void )  
  
{  
  
    /* Perform any hardware setup necessary. */  
  
    prvSetupHardware();  
  
    /* --- APPLICATION TASKS CAN BE CREATED HERE --- */  
  
    /* Start the created tasks running. */  
  
    vTaskStartScheduler();  
  
    /* Execution will only reach here if there was insufficient heap to start the scheduler. */  
  
    for( ;; );  
  
    return 0;  
  
}
```

Creación de un proyecto nuevo a partir de cero

Como ya se ha mencionado, se recomienda que se creen nuevos proyectos a partir de un proyecto de demostración ya existente. Si esto no es deseable, se puede crear un nuevo proyecto utilizando el procedimiento siguiente:

1. Utilizando la cadena de herramientas que prefiera, cree un nuevo proyecto que todavía no incluya archivos de origen de FreeRTOS.
2. Asegúrese de que el nuevo proyecto se pueda compilar, descargarse en el hardware de destino y ejecutarse.
3. Solo cuando esté seguro del proyecto de trabajo es funcional, añada los archivos de origen de FreeRTOS que se detallan en la tabla 1 del proyecto.
4. Copie el archivo de encabezado FreeRTOSConfig.h utilizado por el proyecto de demostración que se proporciona para el puerto en uso en el directorio del proyecto.
5. Añada los directorios siguientes a la ruta donde el proyecto realizará búsquedas para localizar archivos de encabezado:

- FreeRTOS/Source/include
- FreeRTOS/Source/portable/[compilador]/[arquitectura] (donde
[compilador] y [arquitectura] son correctos para el puerto elegido)
- El directorio que contiene el archivo de encabezado FreeRTOSConfig.h.

1. Copie la configuración del compilador del proyecto de demostración pertinente.
2. Instale todos los controladores de interrupción de FreeRTOS que sean necesarios. Utilice la página web que describe el puerto en uso y el proyecto de demostración proporcionado para el puerto en uso, como referencia.

En la tabla siguiente se muestra una lista de los archivos de origen de FreeRTOS que se van a incluir en el proyecto.

Archivo	Ubicación
tasks.c	FreeRTOS/Source
queue.c	FreeRTOS/Source
list.c	FreeRTOS/Source
timers.c	FreeRTOS/Source
eventgroups.c	FreeRTOS/Source
Todos los archivos C y de ensamblador	FreeRTOS/Source/portable/[compilador]/ [arquitectura]
heapn.c	FreeRTOS/Source/portable/MemMang, donde n es 1, 2, 3, 4 o 5. Este archivo pasará a ser opcional a partir de FreeRTOS V9.0.0.

Los proyectos que utilizan una versión de FreeRTOS anterior a la V9.0.0 deben compilar uno de los archivos heapn.c. A partir de FreeRTOS V9.0.0, solo se necesita un archivo heapn.c

si `configSUPPORTDYNAMICALLYALLOCATION` se establece en 1 en `FreeRTOSConfig.h` o si `configSUPPORTDYNAMICALLYALLOCATION` se deja sin definir. Consulte el capítulo 2, Administración de memoria en montón, para obtener más información.

Guía de tipos de datos y estilo de codificación

Cada puerto de FreeRTOS tiene un archivo de encabezado `portmacro.h` que contiene (entre otras cosas) definiciones para dos tipos de datos específicos de puerto: `TickType_t` y `BaseType_t`.

En la siguiente tabla se muestran los tipos de datos que FreeRTOS utiliza.

Algunos compiladores convierten a todas las variables `char` no cualificadas en variables sin signo, mientras que otros las convierten en variables con signo. Por este motivo, el código fuente de FreeRTOS califica explícitamente todos los usos de `char` con la mención "signed" o "unsigned", a menos que el `char` se use para almacenar un carácter ASCII o un puntero a `char` se use para apuntar a una cadena.

Los tipos `int` estándar no se utilizan nunca.

Nombres de variables

Las variables llevan un prefijo que indica su tipo: "c" para `char`, "s" para `int16_t` (breve), "l" para `int32_t` (largo) y "x" para `BaseType_t` y cualquier otro tipo no estándar (estructuras, controladores de tareas, controladores de cola, etc.).

Si una variable no tiene signo, se le asigna el prefijo "u". Si una variable es un puntero, se le asigna el prefijo "p". Por ejemplo, una variable de tipo `uint8_t` llevará el prefijo "uc" y una variable de tipo puntero a `char` llevará el prefijo "pc".

Nombre de funciones

Las funciones tienen un prefijo que indica tanto el tipo que devuelven como el archivo donde están definidas. Por ejemplo:

- `vTaskPrioritySet()` devuelve un vacío y está definida en `task.c`.
- `xQueueReceive()` devuelve una variable del tipo `BaseType_t` y está definida en `queue.c`.
- `pvTimerGetTimerID()` devuelve un puntero a `void` y está definida en `timers.c`.

Las funciones de ámbito de archivo (privadas) llevan el prefijo "prv".

Formato

Una tabulación se establece siempre en cuatro espacios.

Nombres de macro

La mayoría de las macros están escritas en mayúsculas y llevan un prefijo en minúsculas que indica dónde está definida la macro.

En la siguiente tabla se muestra una lista de los prefijos de macro.

Prefijo	Ubicación de la definición de la macro
port (por ejemplo, portMAXDELAY)	portable.h o portmacro.h
task (por ejemplo, taskENTERCRITICAL())	task.h
pd (por ejemplo, pdTRUE)	projdefs.h
config (por ejemplo, configUSEPREEMPTION)	FreeRTOSConfig.h
err (por ejemplo, errQUEUEFULL)	projdefs.h

La API de semáforo está escrita prácticamente toda como un conjunto de macros, pero sigue la convención de nomenclatura de funciones en lugar de la convención de nomenclatura de macros.

En la tabla siguiente se muestra una lista de las macros usadas en el código fuente de FreeRTOS.

Macro	Valor
pdTRUE	1
pdFALSE	0
pdPASS	1
pdFAIL	0

Lógica del exceso de conversiones de tipos

El código fuente de FreeRTOS se puede compilar con muchos compiladores diferentes, que se diferencian en el modo y el momento en que generan advertencias. En concreto varios compiladores quieren que la conversión se use de formas diferentes. Por este motivo el código fuente de FreeRTOS contiene más conversiones de tipos de lo que normalmente se garantizaría.

Administración de la memoria en montón

A partir de la versión V9.0.0, las aplicaciones de FreeRTOS pueden asignarse de forma totalmente estática; es decir, no es necesario incluir un administrador de memoria en montón.

En esta sección se explica lo siguiente:

- Cuándo FreeRTOS asigna RAM.
- Los cinco métodos de asignación de memoria de ejemplo incluidos con FreeRTOS.
- Los casos de uso de cada método de asignación de memoria.

Requisitos previos

Necesita tener buenos conocimientos de programación en C para usar FreeRTOS. En concreto tiene que estar familiarizado con:

- Cómo se crea un proyecto de C, incluidas las fases de compilación y vinculación.
- Los conceptos de pila y montón.
- Las funciones estándar `malloc()` y `free()` de la biblioteca de C.

Asignación de memoria dinámica y su relevancia para FreeRTOS

En esta guía se introducen los objetos de kernel como tareas, colas, semáforos y grupos de eventos. Para facilitar al máximo el uso de FreeRTOS, dichos objetos de kernel no se asignan de forma estática durante la compilación, sino que se asignan dinámicamente durante el tiempo de ejecución. FreeRTOS asigna RAM cada vez que se crea un objeto de kernel y libera RAM cada vez que se elimina un objeto de kernel. Gracias a esta política, disminuye el trabajo de diseño y planificación, se simplifica la API y se reduce la huella de RAM.

La asignación de memoria dinámica es un concepto de programación en C. No se trata de un concepto específico de FreeRTOS ni de multitarea. Es relevante para FreeRTOS porque los objetos de kernel se asignan dinámicamente y los métodos de asignación de memoria dinámica que proporcionan los compiladores de uso general no siempre son adecuados para aplicaciones en tiempo real.

La memoria se puede asignar utilizando las funciones de biblioteca de C estándar `malloc()` y `free()`, pero puede darse el caso de que no sean adecuadas o apropiadas por uno o varios de los motivos siguientes:

- No están siempre disponibles en pequeños sistemas integrados.
- Su implementación puede ocupar un espacio relativamente grande y tomar de esta manera espacio de código valioso.
- En raras ocasiones están libres de subprocesos.
- No son deterministas. La cantidad de tiempo que se tarda en ejecutar las funciones variará según la llamada.
- Pueden sufrir de fragmentación. Se considera que el montón está fragmentado si la RAM libre del montón está desglosada en bloques pequeños independientes entre sí. Si el montón está fragmentado,

todo intento de asignar un bloque generará error si ningún bloque libre del montón es lo suficientemente grande para contener el bloque, incluso si el tamaño total de todos los bloques libres independientes del montón es varias veces mayor que el volumen del bloque que no se puede asignar.

- Pueden complicar la configuración del vinculador.
- Pueden ser una fuente de errores difíciles de depurar si se permite que el espacio del montón crezca en la memoria que usan otras variables.

Opciones para la asignación de memoria dinámica

En las versiones anteriores de FreeRTOS se usaba un método de asignación de grupos de memoria en el que se asignaban previamente, en el momento de la compilación, bloques de memoria de diferentes tamaños, que después las funciones de asignación de memoria devolvían. Aunque se trata de un método que se utiliza habitualmente en los sistemas en tiempo real, generaba una gran cantidad de solicitudes de soporte. Se acabó abandonando este método porque no podía utilizar RAM con la suficiente eficiencia para que fuera realmente viable para sistemas integrados realmente pequeños.

Ahora FreeRTOS trata la asignación de memoria como parte de la capa portátil (en lugar de tratarla como parte de la base de código principal). Con esto se reconocen los requisitos variables de tiempo y asignación de memoria dinámica de los sistemas integrados. Un único algoritmo de asignación de memoria dinámica es adecuado solo para un subconjunto de aplicaciones. Además, la eliminación de la asignación de memoria dinámica de la base de código principal permite que los programadores de aplicaciones proporcionen sus propias implementaciones específicas cuando proceda.

Cuando FreeRTOS necesita RAM, llama a `pvPortMalloc()` en lugar de llamar a `malloc()`. Cuando se libera RAM, el kernel llama a `vPortFree()` en lugar de llamar a `free()`. `pvPortMalloc()` tiene el mismo prototipo que la función `malloc()` de biblioteca C estándar. `vPortFree()` tiene el mismo prototipo que la función `free()` de la biblioteca C estándar.

`pvPortMalloc()` y `vPortFree()` son funciones públicas, por lo que también se pueden llamar desde el código de aplicación.

FreeRTOS incluye cinco implementaciones de ejemplo de `pvPortMalloc()` y `vPortFree()`, que están documentadas aquí. Las aplicaciones de FreeRTOS pueden utilizar una de estas implementaciones de ejemplo o proporcionar la suya propia.

Los cinco ejemplos están definidos en los archivos de origen `heap_1.c`, `heap_2.c`, `heap_3.c`, `heap_4.c` y `heap_5.c` ubicados en el directorio `FreeRTOS/Source/portátil/MemMang`.

Métodos de asignación de memoria de ejemplo

Como las aplicaciones de FreeRTOS pueden asignarse totalmente de forma estática; no es necesario incluir un administrador de memoria en montón.

Heap_1

Es frecuente que los pequeños sistemas integrados dedicados creen las tareas y otros objetos de kernel solo antes de que se inicie el programador. El kernel asigna dinámicamente la memoria antes de que la aplicación comience a ejecutar las funcionalidades en tiempo real, y la memoria permanece asignada durante toda la vida útil de la aplicación. Esto significa que el método de asignación elegido no ha de tener en cuenta ninguno de los problemas de asignación de memoria más complejos, como la fragmentación y el determinismo. En su lugar, puede considerar atributos como el tamaño o la simplicidad del código.

`Heap_1.c` implementa una versión muy básica de `pvPortMalloc()`. No implementa `vPortFree()`. Las aplicaciones que nunca eliminan una tarea ni ningún otro objeto de kernel usan `heap_1`.

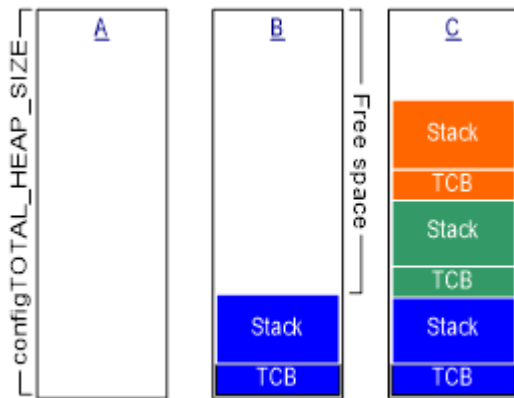
Algunos sistemas críticos desde el punto de vista comercial o de la seguridad que, de lo contrario, prohibirían el uso de la asignación de memoria dinámica, podrían usar heap_1. Los sistemas críticos a menudo prohíben la asignación de memoria dinámica debido a las incertidumbres que conllevan el no determinismo, la fragmentación de memoria y las asignaciones erróneas, pero heap_1 siempre es determinista y no puede fragmentar la memoria.

Cuando se realizan llamadas a `pvPortMalloc()`, el método de asignación de heap_1 subdivide una simple matriz en bloques más pequeños. La matriz se denomina el montón de FreeRTOS.

El tamaño total de la matriz (en bytes) se define con el `configTOTAL_HEAP_SIZE` de definición en `FreeRTOSConfig.h`. Al definir de esta manera una matriz de gran tamaño puede dar la sensación de que la aplicación consume una gran cantidad de memoria RAM incluso antes de que se asigne memoria desde la matriz.

Cada tarea que se crea requiere un bloque de control de la tarea (TCB) y una pila que debe asignarse desde el montón.

En la figura siguiente se muestra cómo heap_1 subdivide una matriz sencilla a medida que se crean tareas. La memoria RAM se asigna desde la matriz de heap_1 cada vez que se crea una tarea.



- A muestra la matriz antes de que se cree ninguna tarea. Toda la matriz está libre.
- B muestra la matriz después de que se haya creado una tarea.
- C muestra la matriz después de que se hayan creado tres tareas.

Heap_2

Heap_2 se incluye en la distribución de FreeRTOS para la compatibilidad con versiones anteriores. No se recomienda usar heap_2 con diseños nuevos. Considere la posibilidad de utilizar heap_4 en su lugar, ya que aporta más funcionalidades.

Heap_2.c también funciona subdividiendo una matriz cuya dimensión se define con `configTOTAL_HEAP_SIZE`. Para asignar memoria usa un algoritmo de mejor opción. A diferencia de heap_1, no permite liberar memoria. También en este caso la matriz se declara estáticamente, por lo que puede dar la sensación de que la aplicación consume una gran cantidad de memoria RAM incluso antes de que se asigne memoria desde la matriz.

El algoritmo de mejor opción garantiza que `pvPortMalloc()` utilice el bloque de memoria libre cuyo tamaño sea el más próximo al número de bytes solicitado. Por ejemplo, piense en una situación en la que:

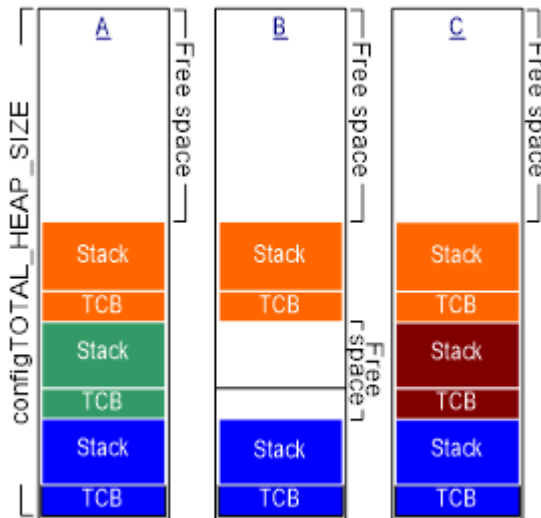
- El montón contenga tres bloques de memoria libre cuyo tamaño respectivo es de 5 bytes, 25 bytes y 100 bytes.

- Se llama a `pvPortMalloc()` para solicitar 20 bytes de RAM.

El bloque de RAM libre más pequeño en el que cabe el número de bytes solicitados es el bloque de 25 bytes, así que `pvPortMalloc()` dividirá el bloque de 25 bytes en un bloque de 20 bytes y otro de 5 bytes antes de devolver un puntero al bloque de 20 bytes. (En este caso hemos simplificado en exceso, ya que `heap_2` almacena información sobre los tamaños de bloque en el área de montón, por lo que la suma de los dos bloques divididos será inferior a 25). El nuevo bloque de 5 bytes permanecerá disponible para futuras llamadas a `pvPortMalloc()`.

A diferencia de `heap_4`, `heap_2` no combina bloques libres adyacentes en un único bloque de mayor tamaño. Por este motivo, es más sensible a la fragmentación. Sin embargo, la fragmentación no es un problema si los bloques que se asignan y posteriormente se liberan siempre tienen el mismo tamaño. `Heap_2` es adecuado para una aplicación que crea y elimina tareas de forma repetida, siempre y cuando el tamaño de la pila asignada a las tareas creadas no cambie.

En la figura siguiente se muestra la asignación y la liberación de RAM de la matriz de `heap_2` a medida que se crean y se eliminan tareas.



En la figura se muestra cómo funciona el algoritmo de mejor opción cuando se crea, se elimina y luego se vuelve a crear una tarea.

- A muestra la matriz después de que se hayan creado tres tareas. Queda un bloque grande libre en la parte superior de la matriz.

- B muestra la matriz después de que se haya eliminado una de las tareas. El

bloque libre de gran tamaño de la parte superior de la matriz permanece. Ahora hay también dos bloques libres más pequeños que se han asignado previamente al TCB y la pila de la tarea eliminada.

- C muestra la matriz después de que se haya creado otra tarea. Crear

la tarea ha generado dos llamadas a `pvPortMalloc()`: una para asignar un nuevo TCB y otra para asignar la pila de la tarea. Las tareas se crean mediante la función de API `xTaskCreate()`, que se describe en [Creación de tareas \(p. 28\)](#). Las llamadas a `pvPortMalloc()` se producen internamente en `xTaskCreate()`.

Cada TCB tiene exactamente el mismo tamaño, por lo que el algoritmo de mejor opción garantiza que el bloque de RAM asignado anteriormente al TCB de la tarea eliminada se vuelve a usar para asignar el TCB de la nueva tarea.

El tamaño de la pila asignada a la tarea que acaba de crear es igual al tamaño de la pila asignada a la tarea eliminada anteriormente, por lo que el algoritmo de mejor opción garantiza que el bloque de RAM asignado previamente a la pila de la tarea eliminada se vuelva a usar para asignar la pila de la nueva tarea.

El bloque sin asignar más grande de la parte superior de la matriz permanece intacto.

Heap_2 no es determinista, pero es más rápido que la mayoría de las implementaciones de biblioteca estándar de malloc() y free().

Heap_3

Heap_3.c utiliza las funciones malloc() y free() de biblioteca estándar, por lo que el tamaño del montón se define mediante la configuración del vinculador. El valor de configTOTAL_HEAP_SIZE no tiene ningún efecto.

Heap_3 hace que malloc() y free() estén a salvo de subprocesos ya que suspende temporalmente el programador de FreeRTOS. Para obtener información sobre la seguridad de los subprocesos y la suspensión del programador, consulte la sección [Administración de recursos \(p. 161\)](#).

Heap_4

Al igual que heap_1 y heap_2, heap_4 subdivide una matriz en bloques más pequeños. La matriz se declara estáticamente y su dimensión se indica mediante configTOTAL_HEAP_SIZE, por lo que puede dar la sensación de que la aplicación consume una gran cantidad de memoria RAM incluso antes de que se asigne memoria desde la matriz.

Heap_4 usa un algoritmo de primera opción adecuada para asignar memoria. A diferencia de heap_2, combina (fusiona) bloques libres adyacentes de memoria en un único bloque más grande. De este modo se minimiza el riesgo de fragmentación de la memoria.

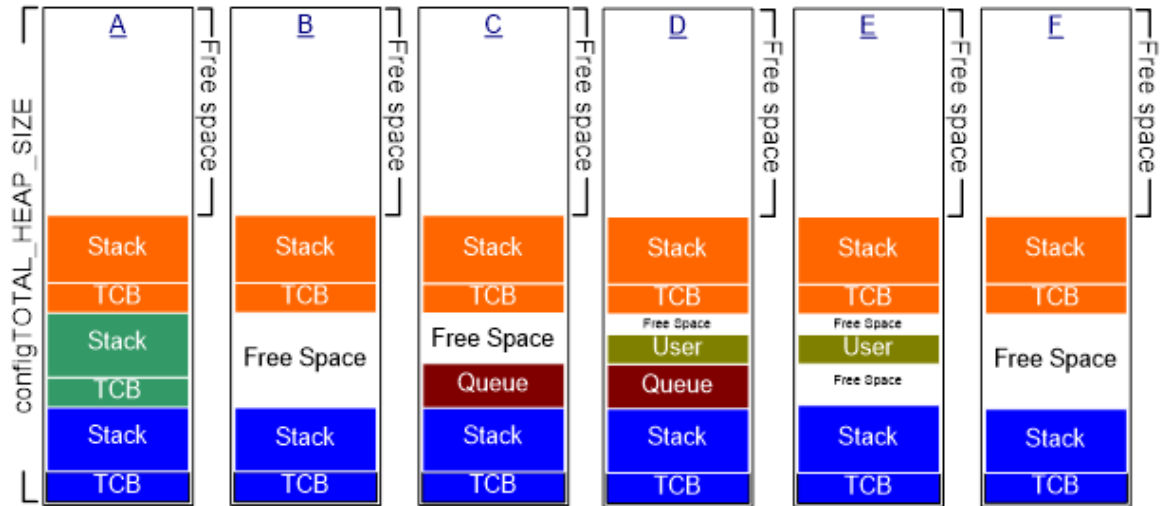
El algoritmo de primera opción adecuada garantiza que pvPortMalloc() utilice el primer bloque de memoria libre cuyo tamaño sea suficiente para contener el número de bytes solicitado. Por ejemplo, piense en una situación en la que:

- El montón contiene tres bloques de memoria libre. Aparecen en este orden en la matriz: 5 bytes, 200 bytes y 100 bytes.
- Se llama a pvPortMalloc() para solicitar 20 bytes de RAM.

El primer bloque de RAM libre en el que cabe el número de bytes solicitados es el bloque de 200 bytes, así que pvPortMalloc() dividirá el bloque de 200 bytes en un bloque de 20 bytes y otro de 180 bytes antes de devolver un puntero al bloque de 20 bytes. (En este caso hemos simplificado en exceso, ya que heap_4 almacena información sobre los tamaños de bloques en el área de montón, por lo que la suma de los dos bloques divididos será inferior a 200 bytes). El nuevo bloque de 180 bytes permanecerá disponible para futuras llamadas a pvPortMalloc().

Heap_4 combina (fusiona) bloques libres adyacentes en un único bloque más grande, lo que minimiza el riesgo de fragmentación. Heap_4 es adecuado para aplicaciones que asignan y liberan repetidamente bloques de tamaños diferentes de RAM.

En la figura siguiente se muestra la asignación y liberación de RAM de la matriz de heap_4. Muestra cómo funciona el algoritmo de primera opción con fusión de memoria de heap_4, a medida que se asigna y libera memoria.



A muestra la matriz después de que se hayan creado tres tareas. Queda un bloque libre de gran tamaño en la parte superior de la matriz.

B muestra la matriz después de que se haya eliminado una de las tareas. El bloque libre de gran tamaño de la parte superior de la matriz se conserva. También hay un bloque libre que anteriormente tenía asignados el TCB y la pila de la tarea eliminada. Tanto la memoria liberada al eliminar el TCB como la que se ha liberado al eliminar la pila no permanecen como dos bloques libres independientes, sino que se combinan para crear un único bloque libre y más grande.

C muestra la matriz después de que se haya creado una cola de FreeRTOS. Las colas se crean usando la función de API `xQueueCreate()`, que se describe en [Uso de una cola \(p. 70\)](#). `xQueueCreate()` llama a `pvPortMalloc()` para asignar la RAM que la cola usa. Dado que heap_4 usa un algoritmo de primera opción adecuada, `pvPortMalloc()` asigna RAM del primer bloque de RAM libre cuyo tamaño permita almacenar la cola. En la figura vemos que se trata de la RAM liberada al eliminar la tarea. La cola no consume toda la RAM del bloque libre, por lo que este se divide en dos. La parte sin utilizar permanecerá disponible para futuras llamadas a `pvPortMalloc()`.

D muestra la matriz después de que se haya llamado directamente a `pvPortMalloc()` desde el código de la aplicación en lugar de llamarlo indirectamente mediante una función de API de FreeRTOS. El bloque asignado por el usuario era lo suficientemente pequeño para caber en el primer bloque libre, que es el bloque que está entre la memoria asignada a la cola y la memoria asignada al TCB siguiente. Ahora la memoria liberada al eliminar la tarea se ha dividido en tres bloques independientes. El primer bloque contiene la cola. El segundo bloque contiene la memoria asignada por el usuario. La tercera sigue estando libre.

E muestra la matriz después de que se haya eliminado la cola, lo que libera automáticamente la memoria que se había asignado a la cola eliminada. Ahora hay memoria libre a ambos lados del bloque asignado por el usuario.

F muestra la matriz después de que también se haya liberado la memoria asignada por el usuario. La memoria que había usado el bloque asignado por el usuario se ha combinado con la memoria libre de ambos lados para crear un bloque libre más grande.

Heap_4 no es determinista, pero es más rápido que la mayoría de las implementaciones de biblioteca estándar de `malloc()` y `free()`.

Configuración de una dirección de inicio para la matriz usada por Heap_4

Nota: Esta sección contiene información avanzada. No es necesario leer esta sección para utilizar heap_4.

En ocasiones el programador de aplicaciones tiene que colocar la matriz que heap_4 utiliza en una dirección de memoria específica. Por ejemplo, la pila que una tarea de FreeRTOS usa se asigna desde el montón, por lo que puede ser necesario asegurarse de que el montón esté en una memoria interna rápida en lugar de estar en memoria externa lenta.

De forma predeterminada, la matriz que usa heap_4 se declara dentro del archivo de origen heap_4.c. Su dirección de inicio se establece automáticamente mediante el vinculador. Sin embargo, si la constante de configuración del tiempo de compilación `configAPPLICATION_ALLOCATED_HEAP` se establece en 1 en `FreeRTOSConfig.h`, la aplicación que usa FreeRTOS tendrá que declarar la matriz. Si la matriz se declara como parte de la aplicación, el programador de la aplicación puede establecer su dirección de inicio.

Si se establece `configAPPLICATION_ALLOCATED_HEAP` en 1 en `FreeRTOSConfig.h`, tendrá que declararse una matriz `uint8_t` llamada `ucHeap`, con las dimensiones configuradas en `configTOTAL_HEAP_SIZE`, en uno de los archivos de origen de la aplicación.

La sintaxis necesaria para introducir una variable en una dirección de memoria específica depende del compilador. Para obtener más información, consulte la documentación del compilador.

A continuación damos ejemplos de dos compiladores.

A continuación se muestra la sintaxis requerida para que el compilador GCC declare la matriz y la ponga en una asignación de memoria llamada `.my_heap`:

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] __attribute__ ( (section( ".my_heap" ) ) );
```

A continuación se muestra la sintaxis requerida para que el compilador IAR declare la matriz y la ponga en la dirección de memoria absoluta `0x20000000`:

```
uint8_t ucHeap[ configTOTAL_HEAP_SIZE ] @ 0x20000000;
```

Heap_5

El algoritmo que usa heap_5 para asignar y liberar memoria es idéntico al que usa heap_4. A diferencia de heap_4, heap_5 no se limita a asignar memoria desde una sola matriz declarada estáticamente. Heap_5 puede asignar memoria desde varios espacios de memoria independientes. Heap_5 es útil cuando la RAM proporcionada por el sistema en el que se ejecuta FreeRTOS no se muestra como un único bloque contiguo (sin espacios) en el mapa de memoria del sistema.

Heap_5 es el único de los métodos de asignación de memoria proporcionados que tiene que inicializarse para poder llamar a `pvPortMalloc()`. Se inicializa con la función de API `vPortDefineHeapRegions()`. Cuando se utiliza heap_5, hay que llamar a `vPortDefineHeapRegions()` antes de crear objetos de kernel (tareas, colas, semáforos, etc.).

La función de API `vPortDefineHeapRegions()`

`vPortDefineHeapRegions()` se utiliza para especificar la dirección de inicio y el tamaño de cada área de memoria independiente. Juntas constituyen la memoria total que heap_5 utiliza.

```
void vPortDefineHeapRegions( const HeapRegion_t * const pxHeapRegions );
```

Cada área de memoria independiente se describe mediante una estructura del tipo `HeapRegion_t`. Se pasa una descripción de todas las áreas de memoria disponibles a `vPortDefineHeapRegions()` como una matriz de estructuras `HeapRegion_t`.

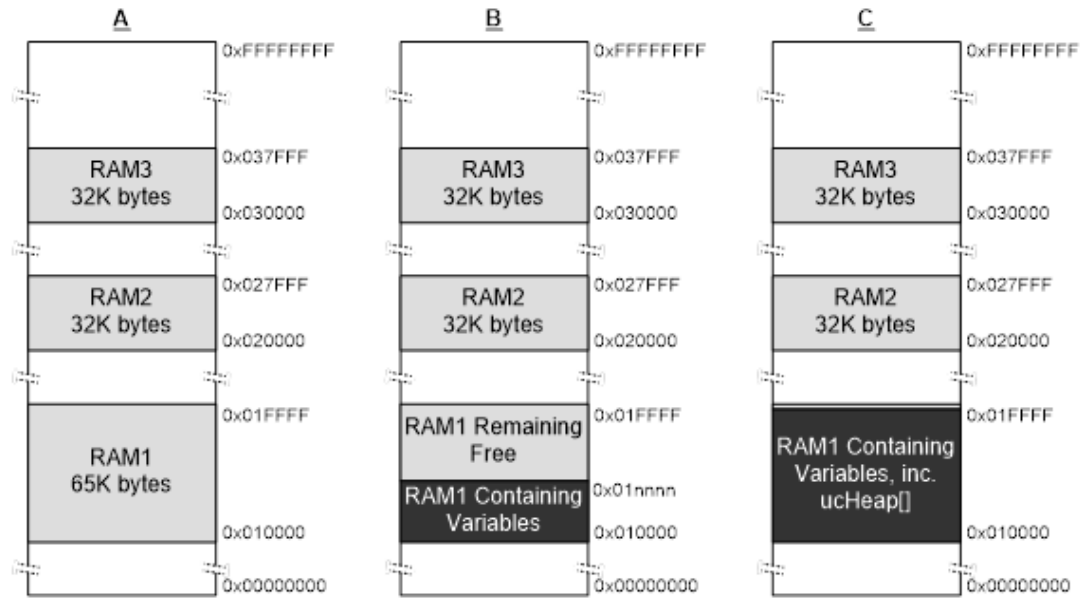
```
typedef struct HeapRegion
{
    /* The start address of a block of memory that will be part of the heap.*/
    uint8_t *pucStartAddress;

    /* The size of the block of memory in bytes. */
    size_t xSizeInBytes;
} HeapRegion_t;
```

En la tabla siguiente se enumeran los parámetros de `vPortDefineHeapRegions()`.

Nombre de parámetro / Valor devuelto	Descripción
<code>pxHeapRegions</code>	Puntero al inicio de una matriz de estructuras <code>HeapRegion_t</code> . Cada estructura de la matriz describe la dirección de inicio y la longitud de un área de memoria que formarán parte del montón cuando se use <code>heap_5</code> . Las estructuras de <code>HeapRegion_t</code> de la matriz tienen que ordenarse en función de la dirección de inicio. La estructura <code>HeapRegion_t</code> que describe el área de memoria con la dirección de inicio más baja tiene que ser la primera estructura de la matriz y la estructura <code>HeapRegion_t</code> que describe el área de memoria con la dirección de inicio más alta tiene que ser la última estructura de la matriz. El final de la matriz se marca con una estructura <code>HeapRegion_t</code> cuyo miembro <code>pucStartAddress</code> está establecido en <code>NULL</code> .

Tomemos, por ejemplo, el mapa de memoria hipotético que se muestra en la figura siguiente, que contiene tres bloques de RAM independientes: RAM1, RAM2 y RAM3. El código ejecutable se pone en memoria de solo lectura, pero no se muestra.



El código siguiente muestra una matriz de estructuras HeapRegion_t. Juntas describen los tres bloques de RAM.

```
/* Define the start address and size of the three RAM regions. */

#define RAM1_START_ADDRESS ( ( uint8_t * ) 0x00010000 )

#define RAM1_SIZE ( 65 * 1024 )

#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )

#define RAM2_SIZE ( 32 * 1024 )

#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )

#define RAM3_SIZE ( 32 * 1024 )

/* Create an array of HeapRegion_t definitions, with an index for each of the three RAM
regions, and terminating the array with a NULL address. The HeapRegion_t structures must
appear in start address order, with the structure that contains the lowest start address
appearing first. */

const HeapRegion_t xHeapRegions[] =
{
    { RAM1_START_ADDRESS, RAM1_SIZE },
    { RAM2_START_ADDRESS, RAM2_SIZE },
    { RAM3_START_ADDRESS, RAM3_SIZE },
    { NULL, 0 } /* Marks the end of the array. */
};

int main( void )
{
```

```
/* Initialize heap_5. */  
  
vPortDefineHeapRegions( xHeapRegions );  
  
/* Add application code here. */  
  
}
```

Aunque el código describe correctamente la RAM, no es un ejemplo útil, ya que asigna toda la RAM al montón y no deja RAM libre para que la usen otras variables.

Cuando se crea un proyecto, la fase de vinculación del proceso de compilación asigna una dirección de RAM a cada variable. La RAM que está disponible para que la use el vinculador normalmente se describe mediante un archivo de configuración del vinculador, como un script de vinculador. En B, en la figura anterior, se presupone que el script del vinculador incluía información sobre RAM1, pero no sobre RAM2 o RAM3. El vinculador, por lo tanto, ha colocado variables en RAM1 y ha dejado solo la parte de RAM1 por encima de la dirección 0x0001nnnn disponible para que heap_5 la use. El valor real de 0x0001nnnn dependerá del tamaño combinado de todas las variables incluidas en la aplicación que se está vinculando. El vinculador ha dejado todo RAM2 y RAM3 sin utilizar, así que están disponibles para que los use heap_5.

Si se usase el código anterior, la RAM asignada a la dirección 0x0001nnnn siguiente de heap_5 se solaparía con la RAM usada para almacenar variables. Para evitar que esto se produzca, la primera estructura HeapRegion_t de la matriz xHeapRegions[] podría utilizar una dirección de inicio 0x0001nnnn en vez de usar 0x00010000.

No se recomienda esta solución porque:

- La dirección de inicio podría no ser fácil de determinar.
- La cantidad de RAM que utiliza el vinculador podría cambiar en compilaciones posteriores y sería preciso actualizar a la dirección de inicio usada en la estructura HeapRegion_t.
- Las herramientas de compilación no conocerán al programador de la aplicación y, por lo tanto, no podrán advertirle si la RAM que usa heap_5 se solapa.

El código siguiente muestra un ejemplo más cómodo y mantenible. Declara una matriz denominada ucHeap. ucHeap es una variable normal, por lo que se convierte en parte de los datos asignados a RAM1 por el vinculador. La primera estructura HeapRegion_t de la matriz xHeapRegions describe la dirección de inicio y el tamaño de ucHeap, por lo que ucHeap se convierte en parte de la memoria que heap_5 administra. El tamaño de ucHeap puede aumentarse hasta que la RAM que utiliza el vinculador consuma toda la RAM1, tal y como se muestra en C, en la figura anterior.

```
/* Define the start address and size of the two RAM regions not used by the linker. */  
  
#define RAM2_START_ADDRESS ( ( uint8_t * ) 0x00020000 )  
  
#define RAM2_SIZE ( 32 * 1024 )  
  
#define RAM3_START_ADDRESS ( ( uint8_t * ) 0x00030000 )  
  
#define RAM3_SIZE ( 32 * 1024 )  
  
/* Declare an array that will be part of the heap used by heap_5. The array will be placed  
   in RAM1 by the linker. */  
  
#define RAM1_HEAP_SIZE ( 30 * 1024 )  
  
static uint8_t ucHeap[ RAM1_HEAP_SIZE ];  
  
/* Create an array of HeapRegion_t definitions. Whereas in previous code listing, the first  
   entry described all of RAM1, so heap_5 will have used all of RAM1, this time the first
```

```
entry only describes the ucHeap array, so heap_5 will only use the part of RAM1 that
contains the ucHeap array. The HeapRegion_t structures must still appear in start address
order, with the structure that contains the lowest start address appearing first. */

const HeapRegion_t xHeapRegions[] =

{

    { ucHeap, RAM1_HEAP_SIZE },

    { RAM2_START_ADDRESS, RAM2_SIZE },

    { RAM3_START_ADDRESS, RAM3_SIZE },

    { NULL, 0 } /* Marks the end of the array. */

};
```

En el código anterior, una matriz de estructuras HeapRegion_t describe toda la memoria RAM2, toda la memoria RAM3, pero solo una parte de RAM1.

Las ventajas de la técnica que se muestra aquí son las siguientes:

- No es necesario utilizar una dirección de inicio codificada de forma rígida.
- El vinculador establecerá automáticamente la dirección que se utiliza en la estructura HeapRegion_t, por lo que siempre será correcta, incluso si la cantidad de RAM que utiliza el vinculador cambia en versiones posteriores.
- No es posible que la RAM asignada a heap_5 se solape con los datos que el vinculador incluye en RAM1.
- La aplicación no se vinculará si ucHeap es demasiado grande.

Funciones de utilidades relacionadas con el montón

La función de API xPortGetFreeHeapSize()

La función de API xPortGetFreeHeapSize() devuelve el número de bytes libres que hay en el montón en el momento en que se solicita la función. Se puede utilizar para optimizar el tamaño del montón. Por ejemplo, si xPortGetFreeHeapSize() devuelve 2000 después de que se hayan creado todos los objetos de kernel, el valor de configTOTAL_HEAP_SIZE puede reducirse en 2000.

xPortGetFreeHeapSize() no está disponible cuando se utiliza heap_3.

El prototipo de la función de API xPortGetFreeHeapSize()

```
size_t xPortGetFreeHeapSize( void );
```

En la siguiente tabla se muestra el valor de retorno de xPortGetFreeHeapSize().

Nombre de parámetro / Valor devuelto	Descripción
Valor devuelto	El número de bytes que permanecen sin asignar en el montón, en el momento en que se llama a xPortGetFreeHeapSize().

La función de API xPortGetMinimumEverFreeHeapSize()

La función de API xPortGetMinimumEverFreeHeapSize() devuelve el número de bytes más pequeño sin asignar que ha habido en el montón desde que comenzó a ejecutarse la aplicación de FreeRTOS.

El valor que xPortGetMinimumEverFreeHeapSize() devuelve permite hacerse una idea de hasta qué punto la aplicación ha estado a punto de quedarse sin espacio de montón. Por ejemplo, si xPortGetMinimumEverFreeHeapSize() devuelve 200, en algún momento desde que la aplicación comenzó a ejecutarse llegó a 200 bytes de quedarse sin espacio de montón.

xPortGetMinimumEverFreeHeapSize() está disponible solo con heap_4 o heap_5.

En la siguiente tabla se muestra el valor de retorno de xPortGetMinimumEverFreeHeapSize().

Nombre de parámetro / Valor devuelto	Descripción
Valor devuelto	El número de bytes más pequeño sin asignar que ha habido en el montón desde que comenzó a ejecutarse la aplicación de FreeRTOS.

Funciones de enlace erróneas de malloc

pvPortMalloc() se puede llamar directamente desde el código de la aplicación. También se puede llamar desde archivos de origen de FreeRTOS cada vez que se crea un objeto de kernel. Los objetos de kernel incluyen tareas, colas, semáforos y grupos de eventos, que se describen más adelante.

Al igual que la función malloc() estándar de biblioteca, si pvPortMalloc() no puede devolver un bloque de RAM porque no existe un bloque del tamaño solicitado, devolverá el valor NULL. Si se ejecuta pvPortMalloc() porque el programador de la aplicación está creando un objeto de kernel y la llamada a pvPortMalloc() devuelve NULL, no se creará el objeto de kernel.

Todos los métodos de asignación de montón de ejemplo se pueden configurar para llamar a una función de enlace (o devolución de llamada) si una llamada a pvPortMalloc() devuelve NULL.

Si configUSE_MALLOC_FAILED_HOOK se establece en 1 en FreeRTOSConfig.h, la aplicación tiene que proporcionar una función de enlace errónea de malloc cuyo nombre y prototipo se muestran aquí. La función se puede implementar en cualquier modo que sea adecuado para la aplicación.

```
void vApplicationMallocFailedHook( void );
```


Administración de tareas

Los conceptos que se introducen en esta sección son fundamentales para comprender cómo usar FreeRTOS y como se comportan las aplicaciones de FreeRTOS. En esta sección se explica lo siguiente:

- Cómo FreeRTOS asigna el tiempo de procesamiento a cada tarea de una aplicación.
- Cómo FreeRTOS elige qué tarea se debe ejecutar en un momento dado.
- Cómo la prioridad relativa de cada tarea influye en el comportamiento del sistema.
- Los estados en que puede existir una tarea.

En esta sección también aprenderá lo siguiente:

- Cómo implementar tareas.
- Cómo crear una o varias instancias de una tarea.
- Cómo usar el parámetro de tareas.
- Cómo cambiar la prioridad de una tarea que ya se ha creado.
- Cómo eliminar una tarea.
- Cómo implementar un procesamiento periódico usando una tarea. Para obtener más información, consulte `software_tier_management`.
- Cuándo se ejecutará la tarea de inactividad y cómo se puede utilizar.

Las funciones de las tareas

Las tareas se implementan como funciones de C. Solo se distinguen por su prototipo, que tiene que volver vacío y tomar un parámetro de puntero vacío, tal y como se muestra aquí.

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

Cada tarea es un pequeño programa en sí mismo. Tiene un punto de entrada, se ejecuta normalmente de forma continua en un bucle infinito y no se cierra.

No debe permitirse que las tareas de FreeRTOS vuelvan de ningún modo desde su función de implementación. No deben contener una instrucción "return" ni debe permitirse que se ejecuten al finalizar la función. Si una tarea ya no es necesaria, debe eliminarse explícitamente.

Se puede utilizar una única definición de función de tarea para crear tantas tareas como se quiera. Cada tarea creada es una instancia de ejecución independiente, con su propia pila y su propia copia de todas las variables (pila) automáticas definidas en la tarea misma.

A continuación mostramos la estructura de una tarea típica.

```
void ATaskFunction( void *pvParameters )
{
    /* Variables can be declared just as per a normal function. Each instance of a task created
    using this example function will have its own copy of the lVariableExample variable. This
```

```
would not be true if the variable was declared static, in which case only one copy of the
variable would exist, and this copy would be shared by each created instance of the task.
(The prefixes added to variable names are described in Data Types and Coding Style Guide.)
*/

int32_t lVariableExample = 0;

/* A task will normally be implemented as an infinite loop. */

for( ;; )

{

/* The code to implement the task functionality will go here. */

}

/* Should the task implementation ever break out of the above loop, then the task must be
deleted before reaching the end of its implementing function. The NULL parameter passed to
the vTaskDelete() API function indicates that the task to be deleted is the calling (this)
task. The convention used to name API functions is described in Data Types and Coding
Style Guide. */

vTaskDelete( NULL );

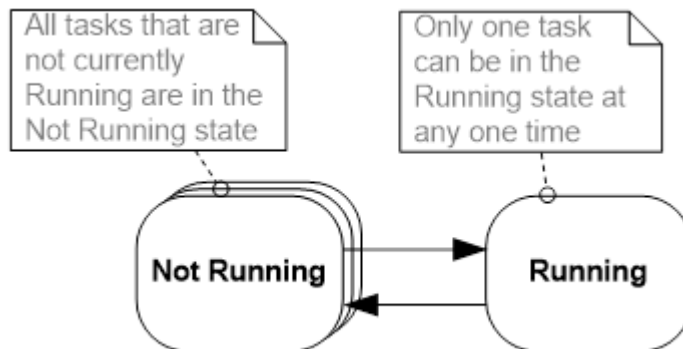
}
```

Los estados de las tareas de nivel superior

Una aplicación puede contener numerosas tareas. Si el procesador que ejecuta la aplicación contiene un único núcleo, solo se puede ejecutar una sola tarea en un momento dado. Esto supone que una tarea puede existir en uno de los dos estados siguientes: En ejecución o No está en ejecución. Tenga en cuenta que esto es una simplificación excesiva. Más adelante, en esta sección, verá que el estado No está en ejecución en realidad contiene varios subestados.

Cuando una tarea se encuentra en el estado En ejecución, el procesador está ejecutando el código de la tarea. Cuando una tarea no se encuentra en el estado En ejecución, significa que está latente; es decir, su estado se ha guardado y está lista para reanudar la ejecución la siguiente vez que el programador decida que debe entrar en el estado En ejecución. Cuando una tarea reanuda su ejecución, lo hace desde la instrucción que estaba a punto de ejecutar en el momento en que dejó de estar en el estado En ejecución.

En la figura siguiente se muestran las transiciones y los estados de las tareas de nivel superior.



Se considera que una tarea que ha pasado del estado No está en ejecución al estado En ejecución se ha cambiado o intercambiado. Por el contrario, se considera que una tarea que ha pasado del estado En

ejecución al estado No está en ejecución se ha desactivado o ha finalizado. El programador de FreeRTOS es la única entidad que puede activar o desactivar una tarea.

Creación de tareas

La función de API xTaskCreate()

FreeRTOS V9.0.0 también incluye la función xTaskCreateStatic(), que asigna la memoria necesaria para crear una tarea de forma estática durante la compilación. Las tareas se crean utilizando la función de API xTaskCreate() de FreeRTOS. Se trata probablemente de la más compleja de todas las funciones de API. Como las tareas son los componentes más básicos de un sistema multitarea, es preciso dominarlas primero. En esta guía encontrará numerosos ejemplos de la función xTaskCreate().

Para obtener más información acerca de los tipos de datos y las convenciones de nomenclatura usados, consulte la guía de tipos de datos y estilo de codificación en la sección de [distribución de kernel de FreeRTOS](#) (p. 5).

El código siguiente muestra el prototipo de la función de API xTaskCreate().

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

A continuación se muestra una lista de los parámetros de xTaskCreate() y los valores de retorno.

- **pvTaskCode:** las tareas son simplemente funciones C que nunca finalizan y, en este sentido, normalmente se implementan en bucle infinito. El parámetro pvTaskCode es simplemente un puntero que señala a la función que implementa la tarea (en efecto, solo el nombre de la función).
- **pcName:** nombre descriptivo de la tarea. FreeRTOS no lo utiliza en modo alguno. Se incluye solo como ayuda para la depuración. Es más sencillo identificar una tarea con un nombre legible que intentar identificarla por su controlador. La constante definida por la aplicación configMAX_TASK_NAME_LEN define la longitud máxima del nombre de una tarea, incluido el terminador NULL. Si suministra una cadena más larga que este valor máximo la cadena se truncará silenciosamente.
- **usStackDepth:** cada tarea tiene su propia pila exclusiva que el kernel asigna a la tarea cuando esta se crea. El valor usStackDepth indica al kernel el tamaño que ha de tener la pila. El valor especifica el número de palabras que la pila puede contener, no su número de bytes. Por ejemplo, si la pila tiene 32 bits de ancho y se pasa usStackDepth con un valor de 100, se asignarán 400 bytes de espacio de pila (100* 4 bytes). La profundidad de la pila multiplicada por el ancho de la pila no debe superar el valor máximo que puede contenerse en una variable de tipo uint16_t. El tamaño de la pila que utiliza la tarea de inactividad se define con la constante configMINIMAL_STACK_SIZE definida por la aplicación. Esta es la única forma de que el código fuente de FreeRTOS use la configuración configMINIMAL_STACK_SIZE. La constante también se utiliza dentro de las aplicaciones de demostración para poder transportar la demostración entre varias arquitecturas de procesador. El valor asignado a esta constante en la aplicación de demostración FreeRTOS para la arquitectura del procesador que se usa es el valor mínimo recomendado para cualquier tarea. Si la tarea utiliza una gran cantidad de espacio de la pila, es preciso asignar un valor más grande. No es fácil determinar el espacio de pila que necesita una tarea. Es posible calcularlo, pero la mayoría de los usuarios simplemente asignan un valor que consideran razonable y luego usan las características de FreeRTOS para asegurarse de que el espacio asignado sea realmente suficiente y que no se desperdicie RAM. Para obtener información acerca de cómo consultar el espacio máximo de pila que se ha utilizado para una tarea, consulte la sección sobre [desbordamiento de pila](#) (p. 249) en la sección de resolución de problemas.
- **pvParameters:** las funciones de tareas aceptan un parámetro del tipo puntero a void (void*). El valor que se asigna a pvParameters es el valor que se pasa a la tarea. En esta guía se mencionan varios ejemplos que demuestran cómo se puede usar el parámetro.

- `uxPriority`: define la prioridad de ejecución de la tarea. Se pueden asignar prioridades desde 0, que es la prioridad más baja, hasta `(configMAX_PRIORITIES - 1)`, que es la máxima prioridad. `configMAX_PRIORITIES` es una constante definida por el usuario que se describe en [Prioridades de las tareas \(p. 34\)](#). Si se transfiere un valor de `uxPriority` por encima de `(configMAX_PRIORITIES - 1)`, la prioridad asignada a la tarea se rebajará silenciosamente al valor máximo legítimo.
- `pxCreatedTask`: este parámetro se puede utilizar para pasar un controlador a la tarea que se está creando. A continuación, el controlador se puede utilizar para hacer referencia a la tarea en llamadas a la API que, por ejemplo, cambien la prioridad de la tarea o la eliminen. Si la aplicación no necesita el controlador de la tarea, `pxCreatedTask` puede establecerse en `NULL`.

Existen dos posibles valores de devolución: `pdPASS`, que indica que la tarea se ha creado correctamente y `pdFAIL`, que indica que la tarea no se ha creado porque no hay bastante memoria en montón disponible para que FreeRTOS asigne RAM suficiente para contener estructuras de datos de la tarea y una pila. Para obtener más información, consulte [Administración de memoria en montón \(p. 14\)](#).

Creación de tareas (ejemplo 1)

En este ejemplo se muestran los pasos necesarios para crear dos tareas sencillas y comenzar a ejecutarlas. Las tareas simplemente imprimen una cadena periódicamente, mediante un bucle nulo bruto para crear el plazo del período. Ambas tareas se crean con la misma prioridad y son idénticas, a excepción de la cadena que imprimen. El código siguiente muestra la implementación de la primera tarea que se utiliza en el ejemplo 1.

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

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

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

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

El código siguiente muestra la implementación de la segunda tarea que se utiliza en el ejemplo 1.

```
void vTask2( void *pvParameters )
```

```
{

    const char *pcTaskName = "Task 2 is running\r\n";

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )

    {

        /* Print out the name of this task. */

        vPrintString( pcTaskName );

        /* Delay for a period. */

        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )

        {

            /* This loop is just a very crude delay implementation. There is nothing to do
            in here. Later examples will replace this crude loop with a proper delay/sleep function.
            */

        }

    }

}
```

La función main() crea las tareas antes de iniciar el programador.

```
int main( void )

{

    /* Create one of the two tasks. Note that a real application should check the return
    value of the xTaskCreate() call to ensure the task was created successfully. */

    xTaskCreate( vTask1, /* Pointer to the function that implements the task. */ "Task 1", /*
    * Text name for the task. This is to facilitate debugging only. */ 1000, /* Stack depth -
    small microcontrollers will use much less stack than this. */ NULL, /* This example does
    not use the task parameter. */ 1, /* This task will run at priority 1. */ NULL ); /* This
    example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks. If main() does reach here then it is likely that there was insufficient
    heap memory available for the idle task to be created. For more information, see Heap
    Memory Management. */

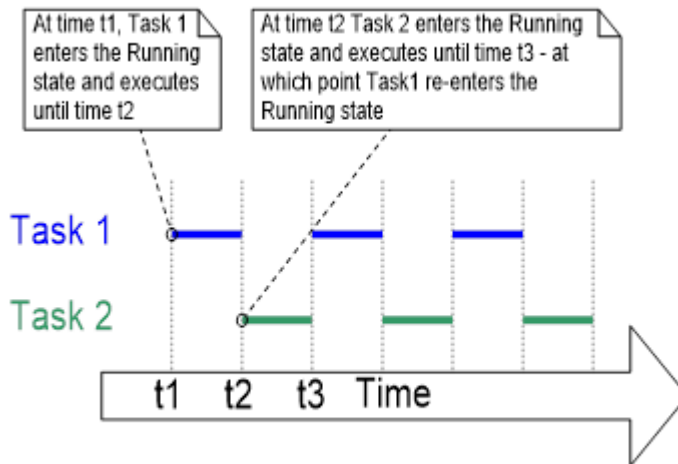
    for( ;; );

}
```

El resultado se muestra aquí.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
```

Las dos tareas parecen ejecutarse simultáneamente. Sin embargo, dado que ambas tareas se ejecutan en el mismo núcleo de procesador, no puede ser así. En realidad ambas tareas entran y salen rápidamente del estado En ejecución. Las dos tareas se ejecutan con la misma prioridad, por lo que comparten tiempo en el mismo núcleo de procesador. Su patrón de ejecución se muestra en la siguiente figura.



La flecha que recorre la parte inferior de esta figura muestra el paso del tiempo desde t1 en adelante. Las líneas en color muestran qué tarea se está ejecutando en cada momento (por ejemplo, la tarea 1 se ejecuta entre los tiempos t1 y t2).

Solo puede haber una tarea en el estado En ejecución en un momento dado, por lo que cuando una tarea entra en el estado En ejecución (la tarea se activa), la otra entra en el estado No está en ejecución (la tarea se desactiva).

Ambas tareas se crean desde main(), antes de iniciar el programador. También es posible crear una tarea desde otra tarea. Por ejemplo, la tarea 2 se puede haber creado desde tarea 1.

El código siguiente muestra una tarea creada desde otra tarea después de que el programador se haya iniciado.

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

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* If this task code is executing then the scheduler must already have been started.
    Create the other task before entering the infinite loop. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

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

        vPrintString( pcTaskName );

        /* Delay for a period. */

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

        }
    }
}
```

Uso del parámetro de tarea (ejemplo 2)

Las dos tareas creadas en el ejemplo 1 son prácticamente idénticas. La única diferencia radica en la cadena de texto que imprimen. Esta duplicación se puede eliminar creando dos instancias de una única implementación de tareas. El parámetro de tareas se puede utilizar para transferir a cada tarea la cadena que debe imprimir.

A continuación se muestra el código de la función de tarea única (vTaskFunction). Esta función única sustituye las dos funciones de tareas (vTask1 y vTask2) utilizadas en el ejemplo 1. El parámetro de tarea se convierte a char * para obtener la cadena que la tarea debe imprimir.

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

    volatile uint32_t ul; /* volatile to ensure ul is not optimized away.*/

    /* The string to print out is passed in via the parameter. Cast this to a character
    pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* As per most tasks, this task is implemented in an infinite loop. */
}
```

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

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

    }
}
}
```

Aunque ahora solo hay una implementación de tarea (vTaskFunction), se puede crear más de una instancia de la tarea definida. Cada instancia creada se ejecutará de forma independiente bajo el control del programador FreeRTOS.

El código siguiente muestra cómo el parámetro pvParameters de la función xTaskCreate() se utiliza para pasar la cadena de texto a la tarea.

```
/* Define the strings that will be passed in as the task parameters. These are defined
const and not on the stack to ensure they remain valid when the tasks are executing. */

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

int main( void )
{
    /* Create one of the two tasks. */

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

    /* Create the other task in exactly the same way. Note this time that multiple tasks
    are being created from the SAME task implementation (vTaskFunction). Only the value passed
    in the parameter is different. Two instances of the same task are being created. */

    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks. If main() does reach here then it is likely that there was insufficient
```



```
heap memory available for the idle task to be created. For more information, see Heap  
Memory Management. */  
  
    for( ;; );  
  
}
```

Prioridades de las tareas

El parámetro `uxPriority` de la función de API `xTaskCreate()` asigna una prioridad inicial a la tarea que se está creando. Puede cambiar la prioridad después de que el programador se haya iniciado usando la función de API `vTaskPrioritySet()`.

El número máximo de prioridades disponibles se establece mediante la constante de configuración de tiempo de configuración `configMAX_PRIORITIES` definida por la aplicación, dentro de `FreeRTOSConfig.h`. Los valores de baja prioridad numérica denotan tareas de prioridad baja, donde la prioridad 0 es la prioridad más baja posible. Por lo tanto, el rango de prioridades disponibles es de 0 a (`configMAX_PRIORITIES - 1`). Cualquier número de tareas puede compartir la misma prioridad, lo que garantiza una máxima flexibilidad de diseño.

El programador de FreeRTOS puede optar entre dos métodos para decidir qué tarea estará en el estado En ejecución. El valor máximo en que puede establecerse `configMAX_PRIORITIES` depende del método utilizado:

1. Método genérico

Este método se implementa en C y se puede utilizar en todos los puertos de arquitectura de FreeRTOS.

Cuando se usa el método genérico, FreeRTOS no limita el valor máximo en el que puede establecerse `configMAX_PRIORITIES`. Sin embargo, le recomendamos que mantenga el valor de `configMAX_PRIORITIES` en el mínimo requerido, ya que cuánto más elevado sea el valor, más RAM se consumirá y más durará el tiempo de ejecución en el peor de los casos.

Se usará este método si `configUSE_PORT_OPTIMISED_TASK_SELECTION` se establece en 0 en `FreeRTOSConfig.h` o si `configUSE_PORT_OPTIMISED_TASK_SELECTION` se deja sin definir, o si el método genérico es el único método proporcionado para el puerto FreeRTOS en uso.

2. Método optimizado para arquitectura

Este método utiliza una pequeña cantidad de código de ensamblador y es más rápido que el método genérico. El valor `configMAX_PRIORITIES` no influye en el tiempo de ejecución en el peor de los casos.

Si utiliza este método, `configMAX_PRIORITIES` no puede ser mayor que 32. Al igual que ocurre con el método genérico, debe tener `configMAX_PRIORITIES` en el valor mínimo requerido, ya que cuánto más alto sea el valor, más RAM se consumirá.

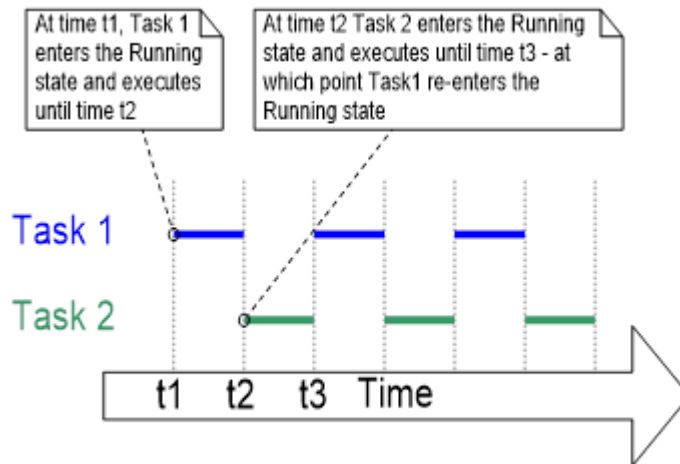
Este método se utiliza si establece `configUSE_PORT_OPTIMISED_TASK_SELECTION` en 1 en `FreeRTOSConfig.h`.

No todos los puertos de FreeRTOS proporcionan un método optimizado para arquitectura.

El programador de FreeRTOS siempre se asegura de que la tarea de máxima prioridad que puede ejecutarse sea la tarea seleccionada para entrar en el estado En ejecución. Cuando se puede ejecutar más de una tarea con la misma prioridad, el programador pasa por turnos cada tarea al estado En ejecución y la saca.

La medida del tiempo y la interrupción de ciclo

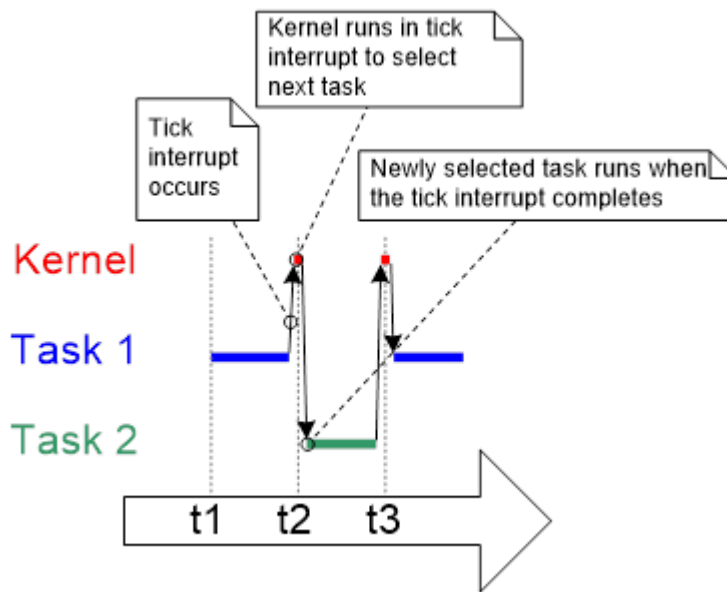
La sección [Algoritmos de programación](#) (p. 58) describe una característica opcional que se conoce como intervalos de tiempo. Los intervalos de tiempo se han utilizado en los ejemplos presentados hasta ahora. Es el comportamiento observado en el resultado que producen. En los ejemplos ambas tareas se han creado con la misma prioridad y ambas tareas siempre han podido ejecutarse. Por lo tanto, cada tarea ejecutada durante un intervalo de tiempo entra en el estado En ejecución al principio del intervalo de tiempo y sale del estado En ejecución cuando termina un intervalo de tiempo. En este gráfico el tiempo entre t_1 y t_2 equivale a un único intervalo de tiempo.



Para poder seleccionar la siguiente tarea que debe ejecutarse, el mismo programador tiene que ejecutarse al final de cada intervalo de tiempo. El final de un intervalo de tiempo no es el único lugar donde el programador puede seleccionar una nueva tarea para ejecutarla. El programador también seleccionará una nueva tarea para ejecutarla inmediatamente después de que la tarea que se está ejecutando actualmente entre en el estado Bloqueado, o cuando una interrupción mueva una tarea de mayor prioridad al estado Listo. Con este fin se usa una interrupción periódica denominada interrupción de ciclo. La longitud del intervalo de tiempo se establece de forma efectiva con la frecuencia de interrupción del ciclo, que se configura con la constante de configuración de tiempo de compilación `configTICK_RATE_HZ` definida por la aplicación, dentro de `FreeRTOSConfig.h`. Por ejemplo, si se establece `configTICK_RATE_HZ` en 100 (Hz), el intervalo de tiempo será de 10 milisegundos. El tiempo entre dos interrupciones de ciclo se denomina el período de ciclo. Un intervalo de tiempo equivale a un período de ciclo.

En la siguiente figura se muestra la secuencia de ejecución ampliada para mostrar la ejecución de la interrupción del ciclo. La línea superior muestra cuándo se ejecuta el programador. Las flechas delgadas muestran la secuencia de ejecución desde una tarea hasta la interrupción del ciclo y después desde la interrupción del ciclo a otra tarea.

El valor óptimo de `configTICK_RATE_HZ` depende de la aplicación que se está desarrollando, aunque un valor de 100 es normal.



Las llamadas a la API de FreeRTOS siempre especifican el tiempo en múltiplos de periodos de ciclos que por lo general se denominan simplemente ciclos. La macro `pdMS_TO_TICKS()` convierte un tiempo especificado en milisegundos en un tiempo especificado en ciclos. La resolución disponible depende de la frecuencia de ciclos definida. `pdMS_TO_TICKS()` no se puede utilizar si la frecuencia de ciclos es superior a 1 KHz (si `configTICK_RATE_HZ` es superior a 1000). El código siguiente muestra cómo utilizar `pdMS_TO_TICKS()` para convertir un tiempo especificado como 200 milisegundos en un tiempo equivalente especificado en ciclos.

```
/* pdMS_TO_TICKS() takes a time in milliseconds as its only parameter, and evaluates to the
equivalent time in tick periods. This example shows xTimeInTicks being set to the number
of tick periods that are equivalent to 200 milliseconds. */

TickType_t xTimeInTicks = pdMS_TO_TICKS( 200 );
```

Nota: No recomendamos que especifique los tiempos en ciclos directamente en la aplicación. En su lugar utilice la macro `pdMS_TO_TICKS()` para especificar los tiempos en milisegundos. Esto garantiza que los tiempos especificados en la aplicación no cambien si se modifica la frecuencia de ciclos.

El valor de recuento de ciclos es el número total de interrupciones de ciclo que se han producido desde que se inició el programador, suponiendo que el recuento de ciclos no se haya desbordado. Las aplicaciones de usuario no han de tener en cuenta los desbordamientos cuando especifican periodos de retraso porque FreeRTOS administra internamente la coherencia de tiempo.

Para obtener información acerca de las constantes de configuración que afectan a cuándo el programador seleccionará una nueva tarea para ejecutarla y cuándo se ejecutará una interrupción de ciclo consulte [Programación de algoritmos \(p. 58\)](#).

Experimentación con prioridades (ejemplo 3)

El programador siempre se asegura de que la tarea de máxima prioridad que puede ejecutarse sea la tarea seleccionada para entrar en el estado En ejecución. En los ejemplos utilizados hasta el momento, se han creado dos tareas con la misma prioridad, por lo que ambas entran y salen del estado En ejecución por turnos. En este ejemplo se muestra qué ocurre cuando la prioridad de una de las dos tareas creadas en el ejemplo 2 cambia. Esta vez, la primera tarea se crea con prioridad 1 y la segunda con prioridad 2.

Aquí se ve el código de muestra utilizado para crear las tareas con diferentes prioridades. La función única que implementa ambas tareas no ha cambiado. Sigue imprimiendo sencillamente una cadena de forma periódica mediante un bucle nulo para crear un plazo.

```
/* Define the strings that will be passed in as the task parameters. These are defined
   const and not on the stack to ensure they remain valid when the tasks are executing. */

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

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

int main( void )
{
    /* Create the first task at priority 1. The priority is the second to last parameter.
    */

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

    /* Create the second task at priority 2, which is higher than a priority of 1. The
    priority is the second to last parameter. */

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

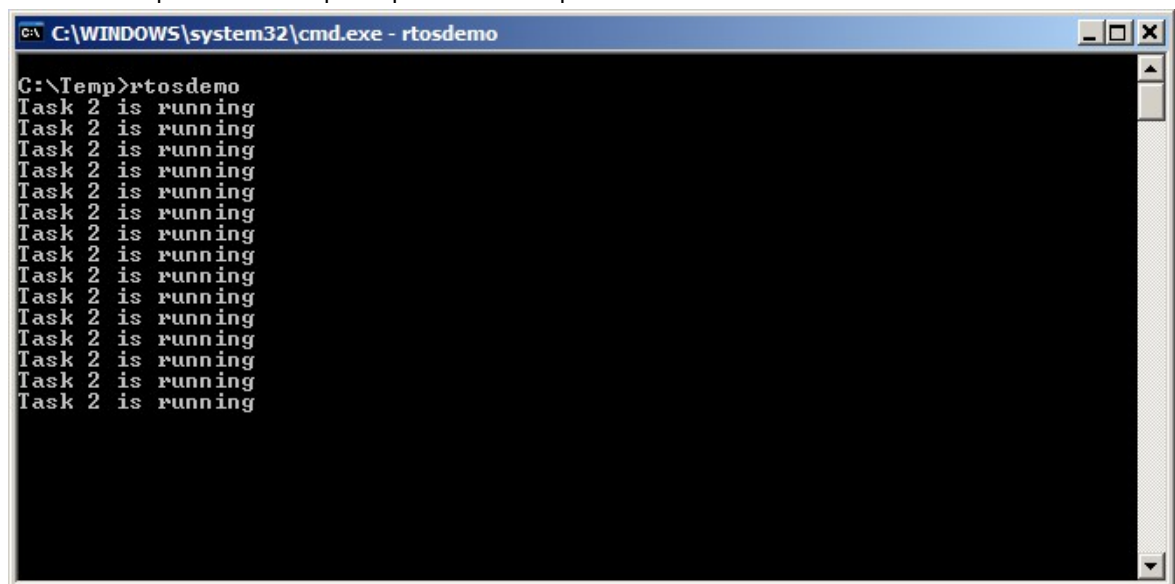
    /* Start the scheduler so the tasks start executing. */

    vTaskStartScheduler();

    /* Will not reach here. */

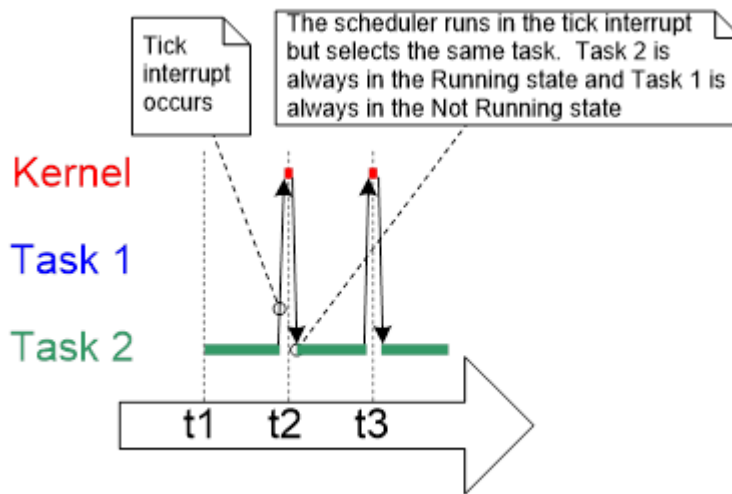
    return 0;
}
```

La salida producida por este ejemplo se muestra aquí. El programador siempre seleccionará la tarea de máxima prioridad que puede ejecutarse. La tarea 2 tiene una prioridad más alta que la tarea 1 y siempre puede ejecutarse. Por lo tanto, la tarea 2 es la única tarea que entrará en el estado En ejecución. Dado que la tarea 1 nunca entra en el estado En ejecución, nunca imprime su cadena. Se dice, pues, que la tarea 1 se ve privada de tiempo de procesamiento por la tarea 2.



La tarea 2 siempre puede ejecutarse, ya que nunca tiene que esperar a nada. Está reproduciendo un bucle nulo o imprimiendo en el terminal.

En la figura siguiente se muestra la secuencia de ejecución del código de muestra anterior.



Ampliación del estado No está en ejecución

Hasta ahora, las tareas creadas siempre han tenido procesamiento para ejecutar y nunca han tenido que esperar por nada. Dado que nunca han tenido que esperar por nada, siempre pueden entrar en el estado En ejecución. Este tipo de tarea de procesamiento continuo tiene una utilidad limitada, ya que las tareas solo se pueden crear con la prioridad más baja. Si se ejecutan con cualquier otra prioridad, impedirán por completo que las tareas de menor prioridad puedan ejecutarse.

Para que las tareas sean útiles, es preciso volver a escribirlas para que dependan de los eventos. Un tarea que depende de un evento ejecuta un trabajo (procesamiento) solo después de que se produzca el evento que la desencadena, y no puede entrar en el estado En ejecución antes de que se produzca el evento. El programador siempre selecciona la tarea de máxima prioridad que puede ejecutarse. Si las tareas de prioridad alta no se pueden ejecutar significa que el programador no puede seleccionarla y que, en su lugar, tiene que seleccionar una tarea de menor prioridad que pueda ejecutarse. Por lo tanto, el uso de tareas que dependen de eventos implica que se pueden crear tareas con diferentes prioridades sin que las tareas de máxima prioridad agoten el tiempo de procesamiento para las tareas de menor prioridad.

El estado Bloqueado

Se dice que una tarea que está a la espera de un evento se encuentra en estado Bloqueado, que es un subestado del estado No está en ejecución.

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

1. Eventos temporales (relacionados con el tiempo), donde los eventos son la caducidad de un plazo o que se llega a un tiempo absoluto. Por ejemplo, una tarea puede entrar en estado Bloqueado a esperar que pasen 10 milisegundos.
2. Eventos de sincronización, donde los eventos se originan en otra tarea o interrupción. Por ejemplo, una tarea puede entrar en el estado Bloqueado a esperar a que lleguen datos a una cola. Los eventos de sincronización cubren una amplia gama de tipos de eventos.

Se pueden usar colas de FreeRTOS, semáforos binarios, semáforos de recuento, exclusiones mutuas, exclusiones mutuas recursivas, grupos de eventos y notificaciones directas en la tarea para crear eventos de sincronización.

Es posible que una tarea se pueda bloquear en un evento de sincronización con un tiempo de espera, lo que bloquea efectivamente ambos tipos de evento de forma simultánea. Por ejemplo, una tarea puede elegir esperar un máximo de 10 milisegundos para que lleguen datos a una cola. La tarea abandonará el estado Bloqueado si llegan los datos en un plazo de 10 milisegundos o si pasan 10 milisegundos y no llegan datos.

El estado suspendido

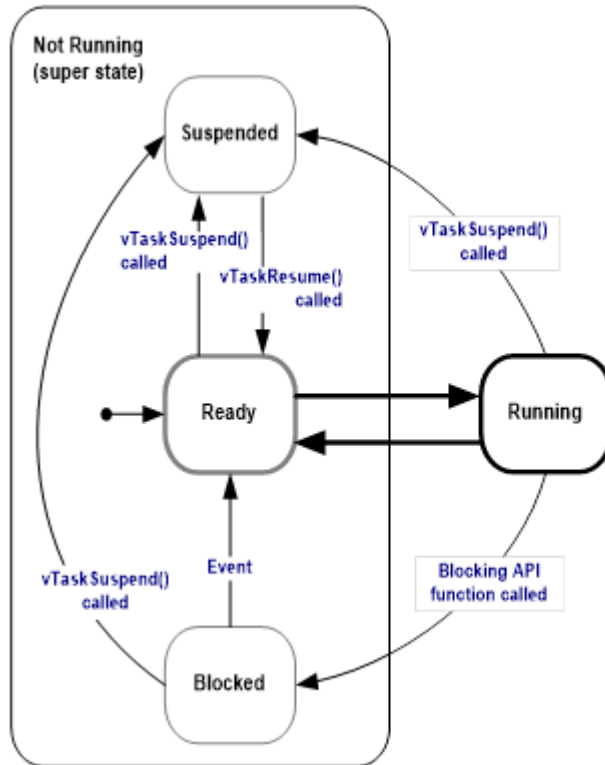
El estado suspendido es también un subestado de No está en ejecución. Las tareas que se encuentran en estado suspendido no están disponibles para el programador. La única forma de entrar en el estado suspendido es mediante una llamada a la función de API `vTaskSuspend()`. La única manera de salir del estado suspendido es mediante una llamada a las funciones de API `vTaskResume()` o `xTaskResumeFromISR()`. La mayoría de las aplicaciones no utilizan el estado suspendido.

El estado Listo

Se dice que las tareas que no se encuentran en el estado No está en ejecución, pero no están bloqueadas ni suspendidas se encuentran en el estado Listo. Pueden ejecutarse y, por lo tanto, están listas para ejecutarse, pero no están actualmente en el estado En ejecución.

Cumplimentación del Diagrama de transición de estado

La siguiente figura amplía el diagrama de estado, demasiado simplificado, para incluir todos los subestados No está en ejecución descritos en esta sección. Hasta ahora, las tareas creadas en los ejemplos no han utilizado los estados Bloqueado o Suspendido. Solo han migrado entre el estado Listo y el estado En ejecución, tal y como muestran las líneas en negrita aquí.



Uso del estado Bloqueado para crear un retraso (ejemplo 4)

Hasta ahora todas las tareas creadas en los ejemplos presentados han sido periódicas. Se han retrasado durante un periodo y han impreso su cadena antes de retrasarse una vez más, etc. El retraso se ha generado muy directamente utilizando un bucle nulo. La tarea sondeó efectivamente un contador de bucles que se iba incrementando hasta que alcanzó un valor fijo. En el ejemplo 3 se demuestra claramente las desventajas de usar este método. La tarea de más prioridad permanece en el estado En ejecución mientras ejecuta el bucle nulo, dejando sin tiempo de procesamiento a la tarea de menos prioridad.

Usar cualquier forma de sondeo presenta muchas desventajas, y su poca eficiencia no es la menor. Durante el sondeo, la tarea en realidad no tiene ningún trabajo que hacer, pero sigue usando un tiempo de procesamiento máximo, derrochando de esta manera ciclos de procesador. En el ejemplo 4 se corrige este comportamiento reemplazando el bucle nulo de sondeo por una llamada a la función de API `vTaskDelay()`. La función de API `vTaskDelay()` solo está disponible cuando `INCLUDE_vTaskDelay` está establecido en 1 en `FreeRTOSConfig.h`.

`vTaskDelay()` pone la tarea de llamada en el estado Bloqueado durante un número fijo de interrupciones de ciclos. La tarea no utiliza tiempo de procesamiento mientras se encuentra en el estado Bloqueado, por lo que solo utiliza tiempo de procesamiento cuando es realmente necesario llevar a cabo el trabajo.

Aquí se muestra el prototipo de la función de API `vTaskDelay()`.

```
void vTaskDelay( TickType_t xTickToDelay );
```

En la tabla siguiente se muestra el parámetro `vTaskDelay()`.

Nombre del parámetro	Descripción
----------------------	-------------

xTicksToDelay

El número de interrupciones de ciclos durante los cuales la tarea de llamada permanecerá en el estado Bloqueado antes de pasar al estado Listo.

Por ejemplo, si una tarea llama a `vTaskDelay(100)` cuando el recuento de ciclos es de 10 000, entra inmediatamente en el estado Bloqueado y permanece en dicho estado hasta que el recuento de ciclos alcance los 10 100.

La macro `pdMS_TO_TICKS()` se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos. Por ejemplo, llamar a `vTaskDelay(pdMS_TO_TICKS(100))` hace que la tarea de llamada permanezca en el estado Bloqueado durante 100 milisegundos.

En el código siguiente la tarea de ejemplo posterior al retraso del bucle nulo se ha sustituido con una llamada a `vTaskDelay()`. Este código muestra la nueva definición de tarea.

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

    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );

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

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

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

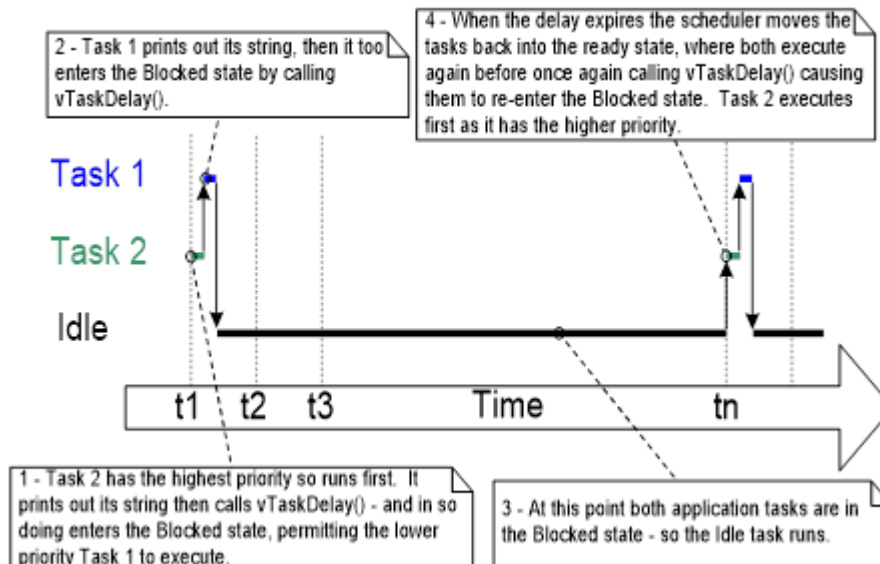
Aunque ambas tareas se siguen creando con diferentes prioridades, ahora ambas se ejecutan. El resultado confirma el comportamiento esperado.


```
C:\WINDOWS\system32\cmd.exe - rtosdemo
C:\Temp>rtosdemo
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
Task 2 is running
Task 1 is running
```

La secuencia de ejecución que se muestra en la figura siguiente explica por qué ambas tareas se ejecutan aunque se hayan creado con diferentes prioridades. La ejecución del programador en sí se omite para mayor simplicidad.

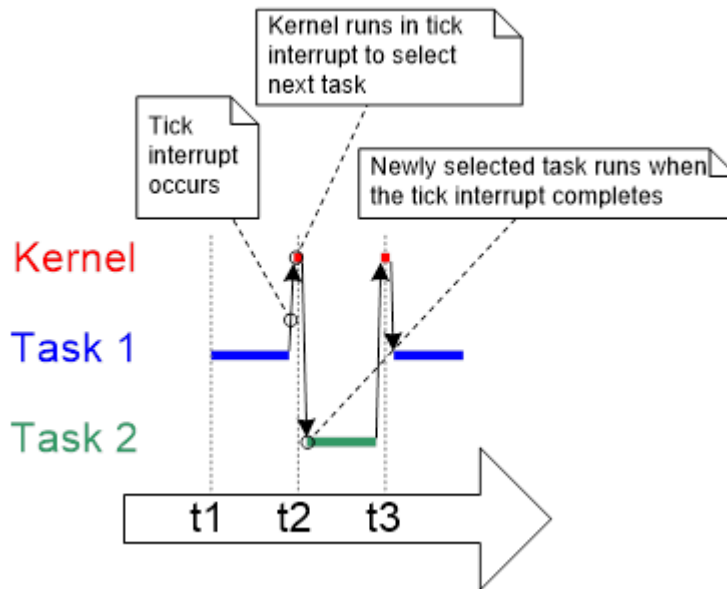
La tarea de inactividad se crea automáticamente cuando el programador se inicia para garantizar que siempre haya al menos una tarea que pueda ejecutarse (al menos una tarea en el estado Listo). Para obtener más información acerca de esta tarea, consulte [La tarea de inactividad y el enlace de tareas de inactividad](#) (p. 48).

Esta figura muestra la secuencia de ejecución cuando las tareas utilizan `vTaskDelay()` en lugar del bucle NULL.



Solo ha cambiado la implementación de ambas tareas, no su funcionalidad. Si compara esta cifra con la cifra de [Medida del tiempo e interrupción de ciclo](#) (p. 35), verá que esta funcionalidad se consigue de una manera mucho más eficiente.

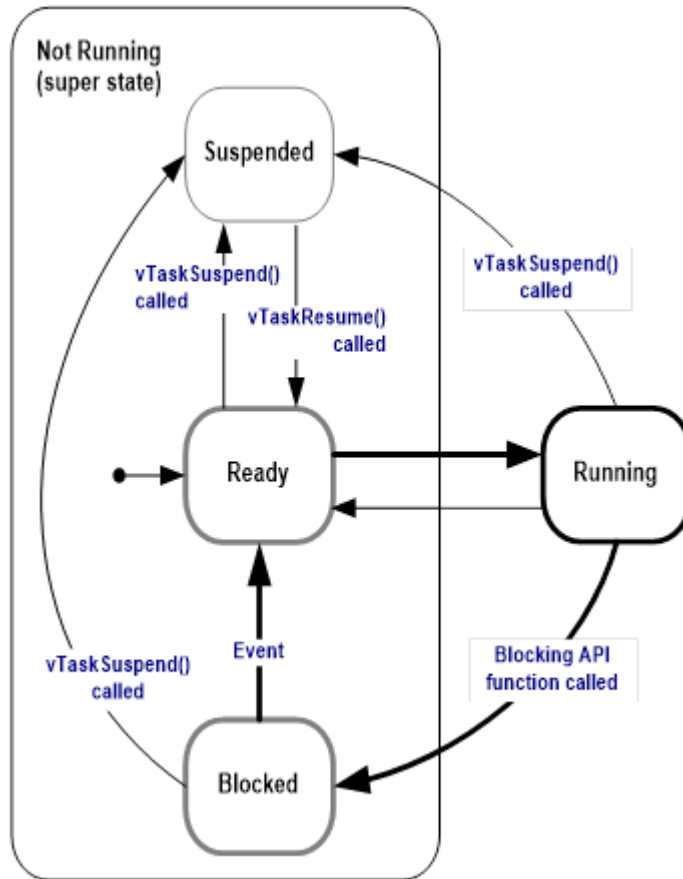
Esta figura muestra el patrón de ejecución cuando las tareas entran en el estado Bloqueado durante todo el periodo de retraso, por lo que utilizan tiempo del procesador solo cuando en realidad tienen trabajo que debe realizarse (en este caso simplemente imprimir un mensaje).



Cada vez que las tareas salen del estado Bloqueado, se ejecutan durante una fracción de un periodo de ciclo antes de volver a entrar en el estado Bloqueado. En la mayoría de los casos no hay tareas de aplicación que puedan ejecutarse (sin tareas de aplicación en el estado Listo) y, por lo tanto, no hay tareas de aplicación que puedan seleccionarse para entrar en el estado En ejecución. Mientras este sea el caso, la tarea de inactividad se ejecutará. La cantidad de tiempo de procesamiento asignado a la tarea de inactividad es una medida de la capacidad de procesamiento libre en el sistema. El uso de un RTOS puede aumentar significativamente la capacidad de procesamiento libre simplemente permitiendo que una aplicación dependa completamente de un evento.

La líneas en negrita de la siguiente figura muestran las transiciones realizadas por las tareas en el ejemplo 4; ahora con una transición de cada una por el estado Bloqueado antes de volver al estado Listo.

Las líneas en negrita indican las transiciones de estado realizadas por las tareas.



La función de API vTaskDelayUntil()

vTaskDelayUntil() es similar a vTaskDelay(). El parámetro vTaskDelay() especifica el número de interrupciones de ciclos que deben producirse entre una tarea que llama a vTaskDelay() y la misma tarea saliendo una vez más en transición del estado Bloqueado. El periodo de tiempo que la tarea permanece en el estado Bloqueado se especifica mediante el parámetro vTaskDelay(), pero el momento en que la tarea sale del estado Bloqueado está indicada en relación con el momento en que se llama a vTaskDelay().

Los parámetros para vTaskDelayUntil() especifican el valor de recuento de ciclos exacto en el que la tarea de llamada debe pasar del estado Bloqueado al estado Listo. vTaskDelayUntil() es la función de API que debe utilizarse cuando necesita un periodo de ejecución fijo (en el que quiera que su tarea se ejecute periódicamente con una frecuencia fija), ya que la hora a la que se desbloquea la tarea de llamada es absoluta, en vez de ser en relación con la hora en que se llamó a la función (como es el caso con vTaskDelay()).

Aquí se muestra el prototipo de la función de API vTaskDelayUntil().

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

En la tabla siguiente se enumeran los parámetros de vTaskDelayUntil().

Nombre del parámetro	Descripción
pxPreviousWakeTime	Este parámetro se llama con el supuesto de que vTaskDelayUntil() se utiliza para implementar una

	<p>tarea que se ejecuta periódicamente y con una frecuencia fija. En este caso pxPreviousWakeTime contiene el momento en que la tarea abandonó el estado Bloqueado (se despertó). Esta hora se utiliza como punto de referencia para calcular la siguiente vez que la tarea tiene que dejar el estado Bloqueado.</p> <p>La variable a la que pxPreviousWakeTime señala se actualiza automáticamente dentro de la función vTaskDelayUntil(). Normalmente el código de la aplicación no la modificaría, pero debe inicializarse al recuento de ciclos actual antes de que se utilice por primera vez.</p>
xTimeIncrement	<p>Este parámetro también se llama con el supuesto de que vTaskDelayUntil() se utiliza para implementar una tarea que se ejecuta periódicamente y con una frecuencia fija. La frecuencia se establece con el valor de xTimeIncrement.</p> <p>xTimeIncrement se especifica en ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos.</p>

Conversión de la tareas de ejemplo para utilizar vTaskDelayUntil() (ejemplo 5)

Las dos tareas creadas en el ejemplo 4 son tareas periódicas, pero el uso de vTaskDelay() no garantiza que la frecuencia con que se ejecutan sea fija, ya que cuándo las tareas dejan el estado Bloqueado depende de cuándo llamaron a vTaskDelay(). Si convierte las tareas para que utilicen vTaskDelayUntil() en lugar de vTaskDelay() solucionará este posible problema.

El código siguiente muestra la implementación de la tarea de ejemplo usando vTaskDelayUntil().

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

    TickType_t xLastWakeTime;

    /* The string to print out is passed in by the parameter. Cast this to a character
    pointer. */

    pcTaskName = ( char * ) pvParameters;

    /* The xLastWakeTime variable needs to be initialized with the current tick count.
    This is the only time the variable is written to explicitly. After this, xLastWakeTime is
    automatically updated within vTaskDelayUntil(). */

    xLastWakeTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */
```

```
for( ;; )  
{  
    /* Print out the name of this task. */  
    vPrintString( pcTaskName );  
  
    /* This task should execute every 250 milliseconds exactly. As per the  
    vTaskDelay() function, time is measured in ticks, and the pdMS_TO_TICKS() macro is  
    used to convert milliseconds into ticks. xLastWakeTime is automatically updated within  
    vTaskDelayUntil(), so is not explicitly updated by the task. */  
    vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ) );  
}  
}
```

Combinación de tareas de bloqueo y tareas que no bloquean (ejemplo 6)

En los ejemplos anteriores se ha examinado el comportamiento de las tareas de sondeo y de bloqueo por separado. En este ejemplo se refuerza el comportamiento del sistema previsto indicando demostrando una secuencia de ejecución cuando se combinan los dos esquemas.

1. Se crean dos tareas con la prioridad 1. No hacen nada, salvo imprimir continuamente una cadena.

Estas tareas nunca realizan llamadas a funciones de API que pudieran hacerles entrar en el estado Bloqueado, por lo que siempre se encuentran en el estado Listo o en Ejecución. Las tareas de este tipo se denominan tareas de procesamiento continuo, ya que siempre tienen trabajo que hacer (aunque sea trivial como en este caso).

2. A continuación se crea otra tarea, esta vez con prioridad 2, por lo que la prioridad es superior a la de las otras dos tareas. La tercera tarea también imprime solamente una cadena, aunque esta vez periódicamente, por lo que utiliza la función de API `vTaskDelayUntil()` para colocarse a sí misma en el estado Bloqueado entre cada iteración de impresión.

El código siguiente muestra la tarea de procesamiento continuo.

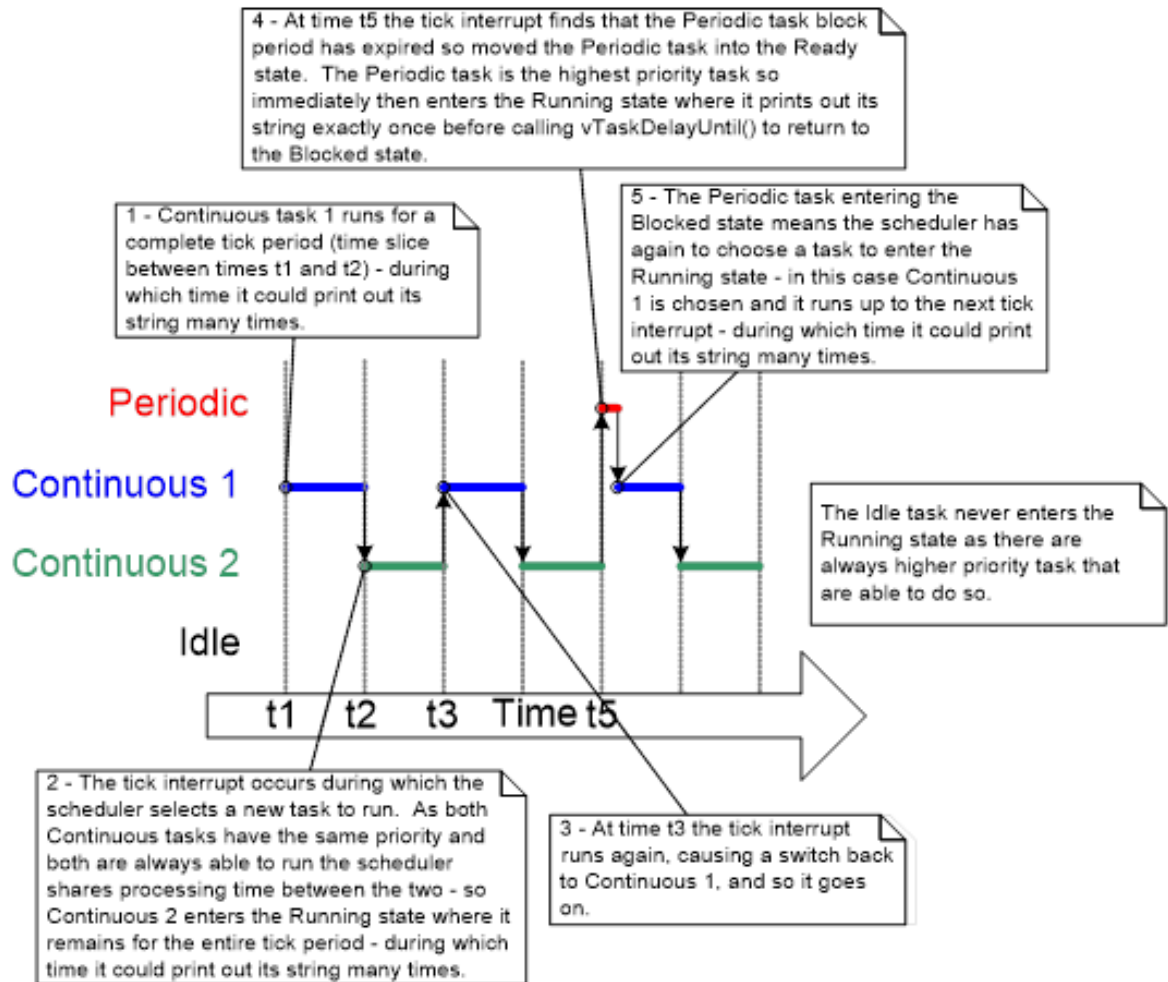
```
void vContinuousProcessingTask( void *pvParameters )  
{  
    char *pcTaskName;  
  
    /* The string to print out is passed in by the parameter. Cast this to a character  
    pointer. */  
    pcTaskName = ( char * ) pvParameters;  
  
    /* As per most tasks, this task is implemented in an infinite loop. */  
    for( ;; )  
    {  
        /* Print out the name of this task. This task just does this repeatedly without  
        ever blocking or delaying. */  
    }  
}
```

```
        vPrintString( pcTaskName );  
    }  
}
```

El código siguiente muestra la tarea periódica.

```
void vPeriodicTask( void *pvParameters )  
{  
    TickType_t xLastWakeTime;  
  
    const TickType_t xDelay3ms = pdMS_TO_TICKS( 3 );  
  
    /* The xLastWakeTime variable needs to be initialized with the current tick count. Note  
    that this is the only time the variable is explicitly written to. After this xLastWakeTime  
    is managed automatically by the vTaskDelayUntil() API function. */  
  
    xLastWakeTime = xTaskGetTickCount();  
  
    /* As per most tasks, this task is implemented in an infinite loop. */  
  
    for( ;; )  
    {  
        /* Print out the name of this task. */  
  
        vPrintString( "Periodic task is running\r\n" );  
  
        /* The task should execute every 3 milliseconds exactly. See the declaration of  
        xDelay3ms in this function. */  
  
        vTaskDelayUntil( &xLastWakeTime, xDelay3ms );  
    }  
}
```

La figura siguiente muestra la secuencia de ejecución.



La tarea de inactividad y el enlace de tareas de inactividad

Las tareas creadas en el ejemplo 4 están la mayor parte de su tiempo en el estado Bloqueado. Mientras se encuentran en este estado, no pueden ejecutarse, por lo que el programador no puede seleccionarlás.

Tiene que haber siempre al menos una tarea que pueda entrar en el estado En ejecución. Este es el caso incluso cuando se usan las características de bajo consumo especiales de FreeRTOS. El microcontrolador en el que se ejecuta FreeRTOS se pondrá en modo de bajo consumo si ninguna de las tareas creadas por la aplicación se puede ejecutar.

Para asegurarse de que al menos una tarea pueda entrar en el estado En ejecución, el programador crea una tarea de inactividad cuando se llama a vTaskStartScheduler(). La tarea de inactividad apenas hace algo más que estar en bucle de modo que, al igual que las tareas del primer ejemplo, siempre puede ejecutarse.

La tarea de inactividad tiene la prioridad más baja posible (prioridad cero) para asegurarse de que nunca impida que una aplicación de más prioridad entre en el estado En ejecución. Sin embargo, no hay nada que le impida crear tareas con la prioridad de las tareas de inactividad y, por lo tanto, de compartirlas.

Se puede usar la constante de configuración de tiempo de compilación `configIDLE_SHOULD_YIELD` de `FreeRTOSConfig.h` para evitar que la tarea de inactividad consuma tiempo de procesamiento que sería más productivo si se asignase a tareas de aplicaciones. Para obtener más información acerca de `configIDLE_SHOULD_YIELD`, consulte [Programación de algoritmos \(p. 58\)](#).

La ejecución con la prioridad más baja garantiza que la tarea de inactividad salga del estado En ejecución tan pronto como la tarea de más prioridad entre en el estado Listo. Esto se puede ver en el momento tn del gráfico de la secuencia de ejecución del ejemplo 4, donde la tarea de inactividad se intercambia inmediatamente para permitir que la tarea 2 se ejecute tan pronto como abandone el estado Bloqueado. Se dice que la tarea 2 ha reemplazado la tarea de inactividad. El reemplazo se produce automáticamente y sin el conocimiento de la tarea que se está reemplazando.

Nota: Si una aplicación utiliza la función de API `vTaskDelete()`, es fundamental que la tarea de inactividad no se quede sin tiempo de procesamiento. Esto se debe a que la tarea de inactividad es responsable de limpiar los recursos de kernel después de que se elimine una tarea.

Funciones de enlace de tareas de inactividad

Puede agregar una funcionalidad específica para aplicaciones directamente en la tarea de inactividad usando una función de enlace de inactividad (o una devolución de llamada inactiva). Se trata de una función a la que la tarea de inactividad llama automáticamente, una vez por iteración del bucle de la tarea de inactividad.

Los usos habituales de un enlace de tareas de inactividad son:

- Ejecutar funcionalidades de baja prioridad, en segundo plano o continuas.
- Medir la cantidad de capacidad de procesamiento libre. (La tarea de inactividad se ejecutará únicamente cuando todas las tareas de aplicación de más prioridad ya no tengan trabajo que realizar, por lo que medir la cantidad de tiempo de procesamiento asignado a la tarea de inactividad proporciona una indicación clara de la cantidad de tiempo de procesamiento libre).
- Poner el procesador en modo de bajo consumo, lo que proporciona un método automático y sencillo de ahorrar energía cuando no es necesario procesar aplicaciones. (Puedes ahorrar más energía mediante el modo inactivo sin ciclos).

Limitaciones en la Implementación de funciones de enlace de tareas de inactividad

Las funciones de enlace de tareas de inactividad tienen que cumplir las siguientes reglas.

1. Una función de enlace de tarea de inactividad nunca debe intentar bloqueos ni suspensiones.

Nota: Bloquear la tarea de inactividad, sea cual sea la forma en que se realice, puede provocar una situación en la que no haya ninguna tarea disponible para entrar en el estado En ejecución.

2. Si la aplicación usa la función de API `vTaskDelete()`, el enlace de la tarea de inactividad siempre tiene que volver a su intermediario en un período de tiempo razonable. Esto se debe a que la tarea de inactividad es responsable de limpiar los recursos de kernel después de que se elimine una tarea. Si la tarea de inactividad permanece de forma permanente en la función de enlace de inactividad, no puede producirse esta limpieza.

Las funciones de enlace de tarea de inactividad deben tener el nombre y el prototipo que se muestran aquí.

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


Definición de una función de enlace de tarea de inactividad (ejemplo 7)

El uso de llamadas a la API `vTaskDelay()` de bloqueo realizado en el ejemplo 4 ha creado una gran cantidad de tiempo libre, que la tarea de inactividad aprovecha para ejecutarse ya que ambas tareas de aplicación se encuentran en el estado Bloqueado. En el ejemplo 7 se usa este tiempo de inactividad añadiendo una función de enlace de inactividad, cuyo origen se muestra aquí.

En el código siguiente se describe una función de enlace de inactividad muy sencilla.

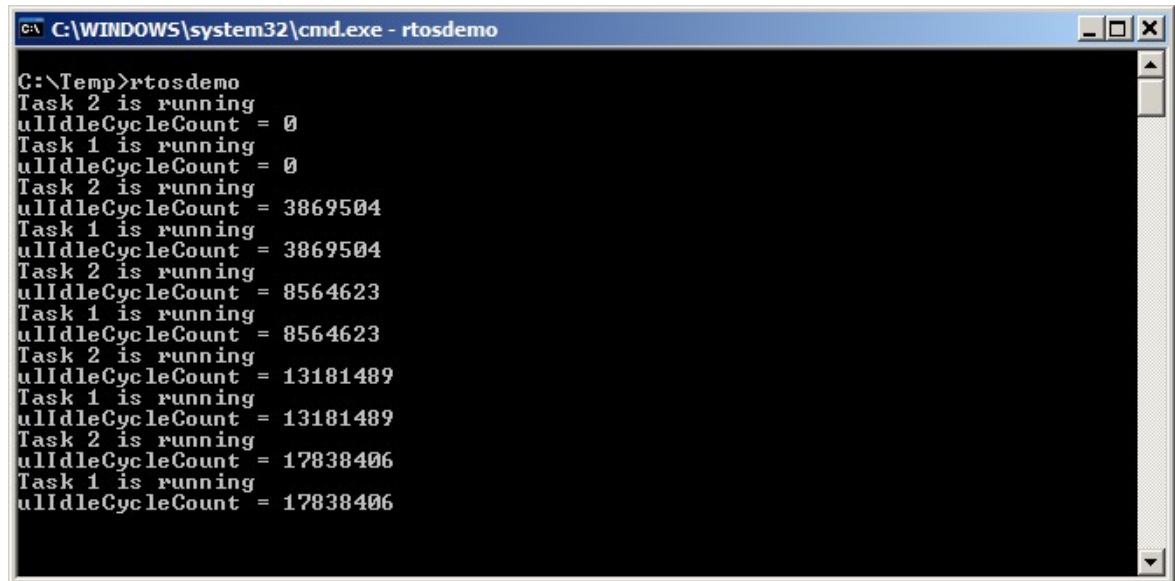
```
/* Declare a variable that will be incremented by the hook function.*/  
  
volatile uint32_t ulIdleCycleCount = 0UL;  
  
/* Idle hook functions MUST be called vApplicationIdleHook(), take no parameters, and  
return void. */  
  
void vApplicationIdleHook( void )  
{  
  
    /* This hook function does nothing but increment a counter. */  
  
    ulIdleCycleCount++;  
  
}  
  
To call the idle  
hook function, called configUSE_IDLE_HOOK must be set to 1 in FreeRTOSConfig.h.
```

La función que implementa las tareas creadas se modifica ligeramente para imprimir el valor `ulIdleCycleCount`, tal y como se muestra aquí. En el código siguiente se muestra cómo imprimir el valor `ulIdleCycleCount`.

```
void vTaskFunction( void *pvParameters )  
{  
  
    char *pcTaskName;  
  
    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );  
  
    /* The string to print out is passed in by the parameter. Cast this to a character  
pointer. */  
  
    pcTaskName = ( char * ) pvParameters;  
  
    /* As per most tasks, this task is implemented in an infinite loop. */  
  
    for( ;; )  
    {  
  
        /* Print out the name of this task AND the number of times ulIdleCycleCount has  
been incremented. */  
  
        vPrintStringAndNumber( pcTaskName, ulIdleCycleCount );  
  
        vTaskDelay( xDelay250ms );  
    }  
}
```

```
/* Delay for a period of 250 milliseconds. */  
  
vTaskDelay( xDelay250ms );  
  
}  
  
}
```

La salida producida por el ejemplo 7 se muestra aquí. Verás que se llama a la función de enlace de la tarea de inactividad alrededor de 4 millones de veces entre cada iteración de las tareas de aplicación. (El número de iteraciones depende de la velocidad del hardware a la que se ejecuta la demostración).



```
C:\Temp>rtosdemo  
Task 2 is running  
ulIdleCycleCount = 0  
Task 1 is running  
ulIdleCycleCount = 0  
Task 2 is running  
ulIdleCycleCount = 3869504  
Task 1 is running  
ulIdleCycleCount = 3869504  
Task 2 is running  
ulIdleCycleCount = 8564623  
Task 1 is running  
ulIdleCycleCount = 8564623  
Task 2 is running  
ulIdleCycleCount = 13181489  
Task 1 is running  
ulIdleCycleCount = 13181489  
Task 2 is running  
ulIdleCycleCount = 17838406  
Task 1 is running  
ulIdleCycleCount = 17838406
```

Cambio de la prioridad de una tarea

Función de API vTaskPrioritySet()

La función de API vTaskPrioritySet() puede usarse para cambiar la prioridad de cualquier tarea después de que el programador se haya iniciado. La función de API vTaskPrioritySet() solo está disponible cuando INCLUDE_vTaskPrioritySet está establecido en 1 en FreeRTOSConfig.h.

El código siguiente muestra el prototipo de la función de API vTaskPrioritySet().

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

En la tabla siguiente se enumeran los parámetros de vTaskPrioritySet().

Nombre del parámetro	Descripción
pxTask	El controlador de la tarea cuya prioridad se está modificando (la tarea del asunto). Para obtener información acerca de cómo obtener controladores para las tareas, consulte el parámetro pxCreatedTask de la función de API xTaskCreate().

	Una tarea puede cambiar su propia prioridad pasando NULL en lugar de un controlador de tarea válido.
uxNewPriority	La prioridad en la que debe establecerse la tarea del asunto. Este valor 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 API uxTaskPriorityGet()

La función de API uxTaskPriorityGet() se puede utilizar para consultar la prioridad de una tarea. La función de API uxTaskPriorityGet() solo está disponible cuando INCLUDE_uxTaskPriorityGet está establecido en 1 en FreeRTOSConfig.h.

El código siguiente muestra el prototipo de la función de API uxTaskPriorityGet().

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

En la siguiente tabla se indica el parámetro uxTaskPriorityGet() y el valor de retorno.

Nombre de parámetro/Valor devuelto	Descripción
pxTask	<p>El controlador de la tarea cuya prioridad se está consultando (la tarea del asunto). Para obtener información acerca de cómo obtener controladores para las tareas, consulte el parámetro pxCreatedTask de la función de API xTaskCreate().</p> <p>Una tarea puede consultar su propia prioridad pasando NULL en lugar de un controlador de tarea válido.</p>
Valor devuelto	La prioridad asignada actualmente a la tarea que se está consultando.

Cambio de prioridades de tareas (ejemplo 8)

El programador siempre seleccionará la tarea de estado Listo de nivel más alto como la tarea que tiene que entrar en el estado En ejecución. En el ejemplo 8 se demuestra esto usando la función de API vTaskPrioritySet() para cambiar la prioridad de dos tareas entre sí.

El código de muestra utilizado aquí crea dos tareas con dos diferentes prioridades. Ninguna tarea realiza ninguna llamada de función de API que pueda hacer que entre en estado Bloqueado, por lo que ambas tareas siempre están en el estado Listo o el estado En ejecución. Por lo tanto, la tarea de máxima prioridad relativa siempre será la tarea que el programador seleccione para entrar en el estado En ejecución.

1. La tarea 1 (que se muestra en el código que está justo abajo) se crea con la máxima prioridad, por lo que está garantizado que se ejecute en primer lugar. La tarea 1 imprime un par de cadenas antes de aumentar la prioridad de la tarea 2 (mostrada en segundo lugar) por encima de su propia prioridad.

2. La tarea 2 comienza a ejecutarse (entra en el estado En ejecución) tan pronto como adquiera la mayor prioridad relativa. Solo una tarea puede encontrarse en el estado En ejecución en un momento dado, de manera que cuando la tarea 2 se encuentra en el estado En ejecución, la tarea 1 está en el estado Listo.
3. La tarea 2 imprime un mensaje antes de volver a establecer su propia prioridad de nuevo por debajo de la prioridad de la tarea 1.
4. Cuando la tarea 2 establece vuelve a bajar su prioridad, una vez más la tarea 1 vuelve a ser la tarea de máxima prioridad. La tarea 1 vuelve a entrar en el estado En ejecución, lo que obliga a la tarea 2 a retomar el estado Listo.

```
/*The implementation of Task 1*/
void vTask1( void *pvParameters )
{
    UBaseType_t uxPriority;

    /* This task will always run before Task 2 because it is created with the higher
    priority. Task 1 and Task 2 never block, so both will always be in either the Running or
    the Ready state. Query the priority at which this task is running. Passing in NULL means
    "return the calling task's priority". */

    uxPriority = uxTaskPriorityGet( NULL );

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

        vPrintString( "Task 1 is running\r\n" );

        /* Setting the Task 2 priority above the Task 1 priority will cause Task 2 to
        immediately start running (Task 2 will have the higher priority of the two created tasks).
        Note the use of the handle to Task 2 (xTask2Handle) in the call to vTaskPrioritySet(). The
        code that follows shows how the handle was obtained. */

        vPrintString( "About to raise the Task 2 priority\r\n" );

        vTaskPrioritySet( xTask2Handle, ( uxPriority + 1 ) );

        /* Task 1 will only run when it has a priority higher than Task 2. Therefore, for
        this task to reach this point, Task 2 must already have executed and set its priority back
        down to below the priority of this task. */

    }
}
```

```
/*The implementation of Task 2 */
void vTask2( void *pvParameters )
{
    UBaseType_t uxPriority;

    /* Task 1 will always run before this task because Task 1 is created with the higher
    priority. Task 1 and Task 2 never block so they will always be in either the Running or
    the Ready state. Query the priority at which this task is running. Passing in NULL means
    "return the calling task's priority". */
```

```
uxPriority = uxTaskPriorityGet( NULL );

for( ;; )

{
    /* For this task to reach this point Task 1 must have already run and set the
    priority of this task higher than its own. Print out the name of this task. */

    vPrintString( "Task 2 is running\r\n" );

    /* Set the priority of this task back down to its original value. Passing in NULL
    as the task handle means "change the priority of the calling task". Setting the priority
    below that of Task 1 will cause Task 1 to immediately start running again, preempting this
    task. */

    vPrintString( "About to lower the Task 2 priority\r\n" );

    vTaskPrioritySet( NULL, ( uxPriority - 2 ) );

}

}
```

Cada tarea puede consultar y establecer su propia prioridad sin usar un controlador de tarea válido, usando sencillamente NULL. Se necesita un controlador de tarea solo cuando una tarea hace referencia a una tarea que no sea ella misma, como cuando la tarea 1 cambia la prioridad de la tarea 2. Para que la tarea 1 pueda efectuar esta operación, se obtiene y se guarda el controlador de la tarea 2 cuando se crea la tarea 2, tal y como se muestra en los comentarios del código aquí.

```
/* Declare a variable that is used to hold the handle of Task 2. */
TaskHandle_t xTask2Handle = NULL;

int main( void )
{
    /* Create the first task at priority 2. The task parameter is not used and set to NULL.
    The task handle is also not used so is also set to NULL. */

    xTaskCreate( vTask1, "Task 1", 1000, NULL, 2, NULL );

    /* The task is created at priority 2 _____. */

    /* Create the second task at priority 1, which is lower than the priority given to
    Task 1. Again, the task parameter is not used so it is set to NULL, but this time the task
    handle is required so the address of xTask2Handle is passed in the last parameter. */

    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, &xTask2Handle );

    /* The task handle is the last parameter _____^^^^^^^^^^^^^^^^ */

    /* Start the scheduler so the tasks start executing. */

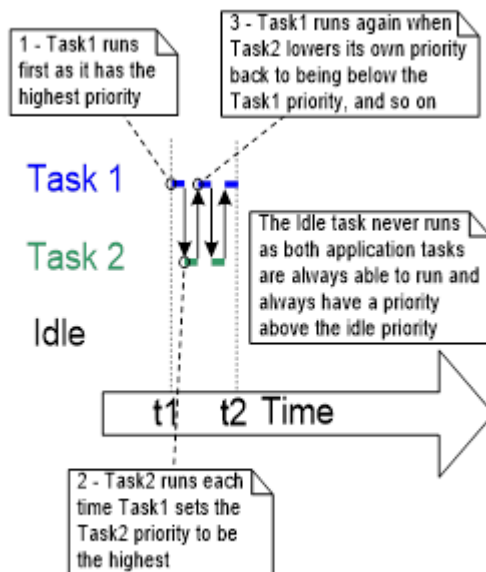
    vTaskStartScheduler();

    /* If all is well, then main() will never reach here because the scheduler will now be
    running the tasks. If main() does reach here, then it is likely there was insufficient
    heap memory available for the idle task to be created. For information, see Heap Memory
    Management. */

    for( ;; );
}
```

}

La siguiente figura muestra la secuencia de ejecución de las tareas.



El resultado se muestra aquí.

```
C:\Windows\system32\cmd.exe
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
Task2 is running
About to lower the Task2 priority
Task1 is running
About to raise the Task2 priority
```

Eliminación de una tarea

Función de API vTaskDelete()

Una tarea puede utilizar la función de API vTaskDelete() para eliminarse a sí misma o a cualquier otra tarea. La función de API vTaskDelete() solo está disponible cuando INCLUDE_vTaskDelete está establecido en 1 en FreeRTOSConfig.h.

Las tareas eliminadas ya no existen y no pueden volver a entrar en el estado En ejecución.

Es responsabilidad de la tarea de inactividad liberar memoria asignada a tareas que ya han sido eliminadas. Por lo tanto, es importante que las aplicaciones que usen la función de API `vTaskDelete()` no agoten el tiempo de procesamiento para la tarea de inactividad.

Nota: Solo la memoria que haya sido asignada a una tarea por el kernel se liberará automáticamente cuando se elimine la tarea. Toda memoria u otro recurso que la implementación de la tarea haya asignado deberá liberarse explícitamente.

Aquí se muestra el prototipo de la función de API `vTaskDelete()`.

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

En la tabla siguiente se muestra el parámetro `vTaskDelete()`.

Nombre de parámetro/Valor devuelto	Descripción
<code>pxTaskToDelete</code>	El controlador de la tarea que debe eliminarse (la tarea del asunto). Para obtener información acerca de cómo obtener controladores para las tareas, consulte el parámetro <code>pxCreatedTask</code> de la función de API <code>xTaskCreate()</code> . Una tarea puede eliminarse a sí misma pasando <code>NULL</code> en lugar de un controlador de tarea válido.

Eliminación de tareas (ejemplo 9)

A continuación se muestra un ejemplo muy sencillo.

1. La tarea 1 se crea con `main()` con prioridad 1. Cuando se ejecuta, crea la tarea 2 con una prioridad de 2. Ahora la tarea 2 es la tarea de máxima prioridad, por lo que comienza a ejecutarse de forma inmediata. El origen de `main()` se muestra en la primera lista de códigos. El origen de la tarea 1 se muestra en la segunda lista de códigos.
2. La tarea 2 se limita a eliminarse a sí misma. Podría eliminarse pasando `NULL` a `vTaskDelete()`, pero en su lugar, para fines de demostración, utiliza su propio controlador de tarea. El origen de la tarea 2 se muestra en la tercera lista de códigos.
3. Una vez que se ha eliminado la tarea 2, la tarea 1 vuelve a ser la tarea de máxima prioridad, por lo que sigue ejecutándose, momento en el cual llama a `vTaskDelay()` para bloquearse durante un breve periodo.
4. La tarea de inactividad se ejecuta mientras la tarea 1 se encuentra en estado bloqueado y libera la memoria que se ha asignado a la tarea 2 que ahora está eliminada.
5. Cuando la tarea 1 sale del estado de bloqueo, se convierte de nuevo en la tarea Listo de máxima prioridad, por lo que reemplaza a la tarea de inactividad. Cuando entra en el estado En ejecución, vuelve a crear la tarea 2, y así sucesivamente.

```
int main( void )  
{  
  
    /* Create the first task at priority 1. The task parameter is not used so is set to  
    NULL. The task handle is also not used so likewise is set to NULL. */
```

```
xTaskCreate( vTask1, "Task 1", 1000, NULL, 1, NULL );

/* The task is created at priority 1 _____. */

/* Start the scheduler so the task starts executing. */

vTaskStartScheduler();

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

for( ;; );

}
```

```
TaskHandle_t xTask2Handle = NULL;

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

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

        vPrintString( "Task 1 is running\r\n" );

        /* Create task 2 at a higher priority. Again, the task parameter is not used so it
        is set to NULL, but this time the task handle is required so the address of xTask2Handle
        is passed as the last parameter. */

        xTaskCreate( vTask2, "Task 2", 1000, NULL, 2, &xTask2Handle );

        /* The task handle is the last parameter _____ */

        /* Task 2 has/had the higher priority, so for Task 1 to reach here, Task 2 must
        have already executed and deleted itself. Delay for 100 milliseconds. */

        vTaskDelay( xDelay100ms );

    }
}
```

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself. To do this, it could call vTaskDelete() using
    NULL as the parameter, but for demonstration purposes, it calls vTaskDelete(), passing its
    own task handle. */

    vPrintString( "Task 2 is running and about to delete itself\r\n" );

    vTaskDelete( xTask2Handle );
}
```

El resultado se muestra aquí.

Las tareas que no se están ejecutando pero no se encuentran en el estado Bloqueado o el estado Suspendido se encuentran en el estado Listo. Las tareas que se encuentran en el estado Listo están disponibles para que el programador las seleccione como tarea que debe entrar en el estado En ejecución. El programador siempre elegirá la tarea de estado Listo que tenga el nivel de prioridad más alto para que entre en el estado En ejecución.

Las tareas pueden esperar en el estado Bloqueado a que se produzca un evento y se muevan automáticamente al estado Listo cuando dicho evento se produce. Los eventos temporales se producen en un momento determinado (por ejemplo, cuando un tiempo de bloqueo vence) y se utilizan normalmente para implementar un comportamiento periódico o de tiempo de espera. Los eventos de sincronización se producen cuando una tarea o una rutina de servicio de interrupción envía información mediante una notificación de tarea, cola, grupo de eventos o uno de los muchos tipos de semáforo. Por lo general se utilizan para señalar actividad asíncrona de señales, por ejemplo, una llegada de datos a un periférico.

Configuración del algoritmo de programación

El algoritmo de programación es la rutina de software que decide qué tarea con el estado Listo pasará al estado En ejecución.

Todos los ejemplos mostrados hasta ahora utilizan el mismo algoritmo de programación, pero puede cambiarlo mediante las constantes de configuración `configUSE_PREEMPTION` y `configUSE_TIME_SLICING`. Ambas constantes están definidas en `FreeRTOSConfig.h`.

Una tercera constante de configuración, `configUSE_TICKLESS_IDLE` también afecta al algoritmo de programación. Su uso puede provocar que la interrupción de ciclos se desactive totalmente durante periodos extendidos. `configUSE_TICKLESS_IDLE` es una opción avanzada proporcionada específicamente para usarla en aplicaciones que deben minimizar su consumo energético. Las descripciones proporcionadas en esta sección presuponen que `configUSE_TICKLESS_IDLE` está establecido en 0, que es el valor predeterminado si la constante se deja sin definir.

En todas las configuraciones posibles, el programador FreeRTOS se asegurará de que las tareas que comparten la misma prioridad se seleccionen para entrar en el estado En ejecución por turnos. Esta política de asunción por turnos a menudo se denomina programación de turno rotativo. Un algoritmo de programación de turno rotativo no garantiza que el tiempo se comparta por igual entre todas las tareas que tengan la misma prioridad, sino que las tareas con el estado En ejecución que tengan la misma prioridad entrarán en el estado En ejecución por turnos.

Programación de reemplazo prioritario con intervalos de tiempo

La configuración que se muestra en la siguiente tabla establece el programador FreeRTOS para que utilice un algoritmo de programación denominado Programación de reemplazo prioritario fijo con intervalos de tiempo, que es el algoritmo de programación utilizado por la mayoría de las pequeñas aplicaciones RTOS y el algoritmo utilizado en todos los ejemplos presentados hasta ahora.

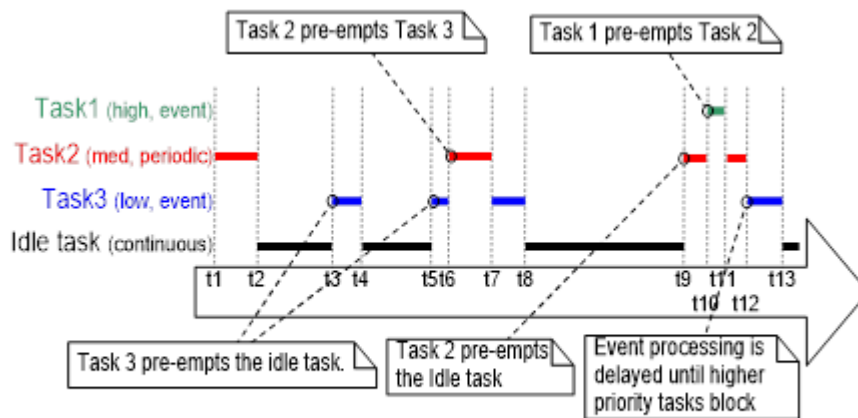
Constant	Valor
<code>configUSE_PREEMPTION</code>	1
<code>configUSE_TIME_SLICING</code>	1

A continuación se indica la terminología usada en el algoritmo:

- tareas de prioridad fija: los algoritmos de programación descritos como algoritmos de prioridad fija no cambian la prioridad asignada a las tareas en proceso de programación, pero tampoco impiden que las tareas mismas cambien su prioridad o la de otras tareas.

- estado preferente: los algoritmos de programación preferente sustituirán inmediatamente la tarea con el estado En ejecución si una tarea que tiene una mayor prioridad entra en el estado Listo. Sustituir inmediatamente por prioridad significa que una tarea sale involuntariamente (sin cesión ni bloqueo explícito) del estado En ejecución y entra en el estado Listo para que otra tarea entre en el estado En ejecución.
- intervalos de tiempo: los intervalos de tiempo se utilizan para compartir el tiempo de procesamiento entre tareas que tienen la misma prioridad, incluso cuando estas no ceden explícitamente al estado Bloqueado ni entran en dicho estado explícitamente. Los algoritmos de programación que usan intervalos de tiempo seleccionarán una nueva tarea para que entre en el estado En ejecución al final de cada intervalo de tiempo si hay otras tareas con el estado Listo que tienen la misma prioridad que la tarea que se está ejecutando. Un intervalo de tiempo es igual al tiempo transcurrido entre dos interrupciones de ciclos RTOS.

La siguiente figura muestra la secuencia de selección de las tareas para que entren en el estado En ejecución cuando todas las tareas de una aplicación tienen una prioridad única.



En esta figura se muestra el patrón de ejecución, donde se resalta la priorización y la preferencia de tareas en una hipotética aplicación en la que se ha asignado a cada tarea una prioridad única.

1. Tarea de inactividad

La tarea de inactividad se está ejecutando con la prioridad más baja por lo que se reemplaza cada vez que una tarea con más prioridad entra en el estado Listo (por ejemplo, en las horas t3, t5 y t9).

2. Tarea 3

La tarea 3 es una tarea basada en eventos que se ejecuta con una prioridad relativamente baja, aunque por encima de la prioridad de inactividad. Pasa la mayor parte de su tiempo en el estado Bloqueado a la espera de que se produzca su evento de interés, y se transfiere desde el estado Bloqueado al estado Listo cada vez que se produce el evento. Todos los mecanismos de comunicación entre tareas de FreeRTOS (notificaciones de tareas, colas, semáforos, grupos de eventos, etc.) se pueden utilizar para señalar eventos y desbloquear tareas de este modo.

Los eventos se producen en las horas t3 y t5, y en algún punto entre t9 y t12. Los eventos que se producen en las horas t3 y t5 se procesan inmediatamente, ya que en esas horas la tarea 3 es la tarea de máxima prioridad que puede ejecutarse. El evento que se produce en algún punto entre t9 y t12 no se procesa hasta t12, ya que hasta entonces las tareas de mayor prioridad, las tareas 1 y 2, siguen ejecutándose. Solo en el momento t12 ambas tareas (1 y 2) están en el estado Bloqueado, lo que convierte a la tarea 3 en la tarea de estado Listo con mayor prioridad.

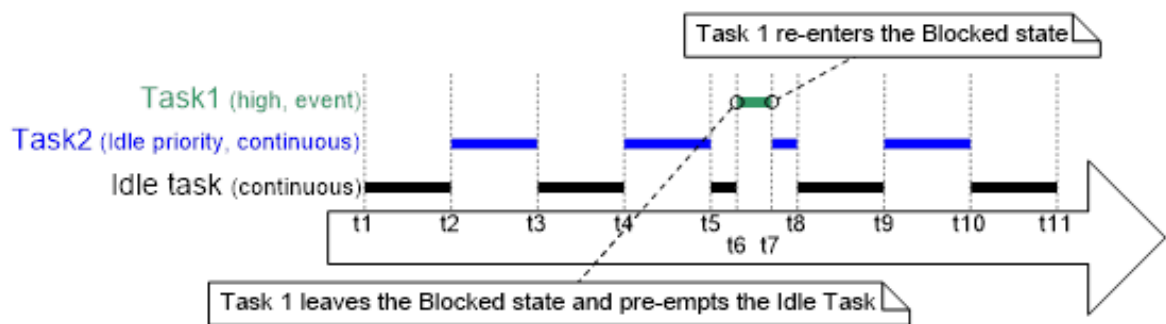
3. Tarea 2

La tarea 2 es una tarea periódica que se ejecuta con una prioridad superior a la prioridad de la tarea 3, pero por debajo de la prioridad de la tarea 1. El intervalo del periodo de la tarea significa que la tarea 2 quiere ejecutarse en las horas t1, t6 y t9. En t6, la tarea 3 se encuentra en el estado En ejecución, pero la tarea 2 tiene la mayor prioridad relativa por lo que reemplaza a la tarea 3 y comienza a ejecutarse de forma inmediata. La tarea 2 completa su procesamiento y vuelve a entrar en el estado Bloqueado en el momento t7, momento en el que la tarea 3 puede volver a entrar en el estado En ejecución para completar su procesamiento. La tarea 3 se bloquea en el t8.

4. Tarea 1

La tarea 1 también es una tarea basada en eventos. Se ejecuta con la prioridad más alta de todas las tareas, por lo que puede reemplazar a cualquier otra tarea del sistema. El único evento de tarea 1 que se muestra se produce en el momento t10, momento en el que la tarea 1 reemplaza a la tarea 2. La tarea 2 puede completar su procesamiento solo después de que la tarea 1 haya vuelto a entrar en el estado Bloqueado en el momento t11.

En el gráfico siguiente se muestra el patrón de ejecución; se resalta la priorización de tareas y los intervalos de tiempo en una hipotética aplicación en la que dos tareas se ejecutan con la misma prioridad.



1. Tarea de inactividad y tarea 2

La tarea de tiempo de inactividad y la tarea 2 son ambas tareas de procesamiento continuo y ambas tienen una prioridad de 0 (la prioridad más baja posible). El programador solo asigna el tiempo de procesamiento a tareas de prioridad 0 cuando no hay tareas de mayor prioridad que puedan ejecutarse, y comparte el tiempo que se asigna a las tareas de prioridad 0 mediante intervalos de tiempo. En cada interrupción de ciclo se inicia un intervalo de tiempo nuevo, que se produce en los momentos t1, t2, t3, t4, t5, t8, t9, t10 y t11.

La tarea de tiempo de inactividad y la tarea 2 entran en el estado En ejecución por turnos, lo que a su vez puede dar lugar a que ambas tareas se encuentren en el estado En ejecución durante parte del mismo intervalo de tiempo, tal y como ocurre entre las horas t5 y t8.

2. Tarea 1

La prioridad de la tarea 1 es superior a la de la prioridad de inactividad. La tarea 1 es una tarea gestionada por eventos que pasa la mayor parte de su tiempo en el estado Bloqueado a la espera de que se produzca su evento de interés, y se transfiere desde el estado Bloqueado al estado Listo cada vez que se produce el evento. El evento de interés se produce en el momento t6 cuando la tarea 1 pasa a ser la tarea de máxima prioridad que puede ejecutarse y, por lo tanto, la tarea 1 reemplaza a la parte de tarea de inactividad a través de un intervalo de tiempo. El procesamiento del evento se completa en t7, momento en el cual la tarea 1 vuelve a entrar en el estado Bloqueado.

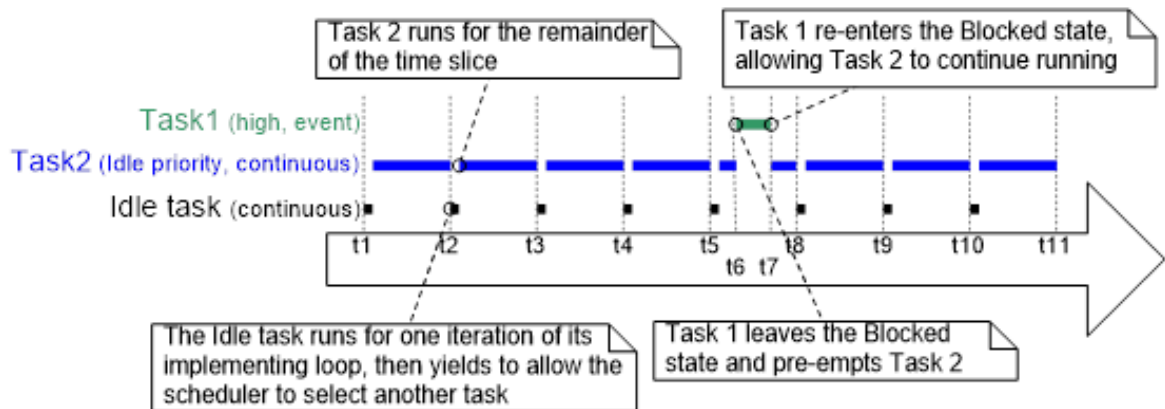
En el gráfico anterior se muestra el tiempo de procesamiento compartido de la tarea de inactividad con una tarea creada por un programador de aplicaciones. No es recomendable asignar tanto tiempo

de procesamiento a la tarea de inactividad si las tareas con prioridad de inactividad creadas por el programador tienen trabajo que hacer, pero la tarea de inactividad no tiene. Puede utilizar la constante de configuración de tiempo de compilación `configIDLE_SHOULD_YIELD` constante para cambiar la forma en que la tarea de inactividad se programa:

- Si `configIDLE_SHOULD_YIELD` está establecido en 0, la tarea de inactividad permanecerá en el estado En ejecución durante todo su intervalo de tiempo, a menos que lo reemplace otra tarea con mayor prioridad.
- Si `configIDLE_SHOULD_YIELD` se establece en 1, la tarea de inactividad cederá (abandonará voluntariamente el tiempo del intervalo de tiempo asignado que le quede) en cada iteración de su bucle si hay otras tareas de prioridad de inactividad en el estado Listo.

El patrón de ejecución que se muestra en el gráfico anterior es lo que se observaría cuando `configIDLE_SHOULD_YIELD` se establece en 0.

El patrón de ejecución que se muestra en el gráfico siguiente es lo que se observaría en la misma situación cuando `configIDLE_SHOULD_YIELD` se establece en 1. Muestra la secuencia de selección de las tareas para que entren en el estado En ejecución cuando dos tareas de una aplicación comparten una misma prioridad.



Esta figura también muestra que cuando `configIDLE_SHOULD_YIELD` se establece en 0, la tarea seleccionada para entrar en el estado En ejecución después de la tarea de inactividad no se ejecuta durante un intervalo de tiempo completo; en su lugar se ejecuta durante el intervalo de tiempo que queda en el que la tarea de inactividad abandonó el trabajo.

Programación de reemplazo prioritario (sin intervalos de tiempo)

La programación de reemplazo con prioridad sin intervalos de tiempo mantiene la misma selección de tareas y algoritmos de preferencia descritos en la sección anterior, pero no utiliza los intervalos de tiempo para compartir el tiempo de procesamiento entre tareas de una misma prioridad.

En la siguiente tabla se muestran los valores de `FreeRTOSConfig.h` que configuran el programador FreeRTOS para que use la programación de reemplazo prioritario sin intervalos de tiempo.

Constant	Valor
<code>configUSE_PREEMPTION</code>	1

configUSE_TIME_SLICING

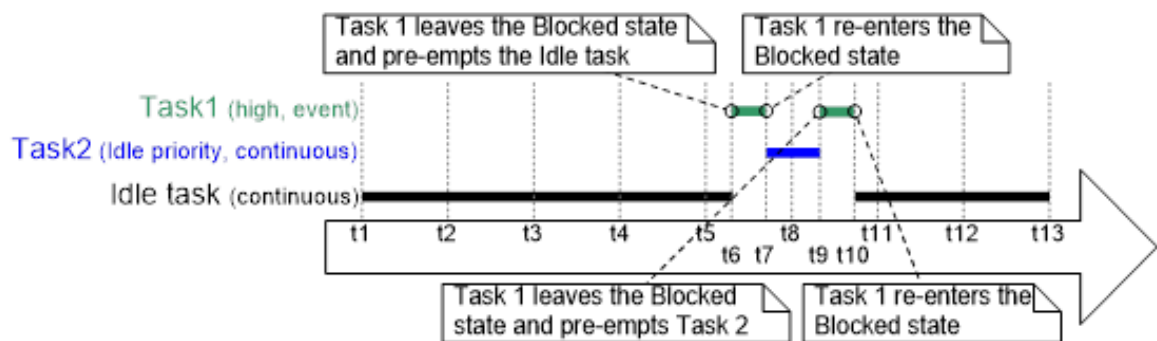
0

Si se usan los intervalos de tiempo y hay más de una tarea en el estado Listo con la prioridad más alta que puede ejecutarse, el programador seleccionará una nueva tarea para entrar en el estado En ejecución durante cada interrupción de ciclo RTOS (una interrupción de ciclo marca el final de un intervalo de tiempo). Si no se usan los intervalos de tiempo, el programador seleccionará una nueva tarea para entrar en el estado En ejecución solo si:

- Una tarea de mayor prioridad entra en el estado Listo.
- La tarea en el estado En ejecución entra en el estado Bloqueado o Suspendido.

Cuando no se utilizan los intervalos de tiempo, hay menos cambios de contexto de tareas. Por lo tanto, desactivar los intervalos de tiempo se traduce en una reducción de la sobrecarga de procesamiento del programador. Sin embargo, desactivar los intervalos de tiempo también puede dar lugar a que tareas que tienen la misma prioridad reciban cantidades de tiempo de procesamiento muy diferentes. Ejecutar el programador sin intervalos de tiempo se considera una técnica avanzada. Solo deberían utilizarla usuarios experimentados.

En este gráfico se muestra cómo tareas que tienen la misma prioridad pueden recibir cantidades muy diferentes de tiempo de procesamiento cuando no se usan los intervalos de tiempo.



En el gráfico anterior, configIDLE_SHOULD_YIELD se establece en 0.

1. Interrupciones de ciclo

Las interrupciones de ciclo se producen en t1, t2, t3, t4, t5, t8, t11, t12 y t13.

2. Tarea 1

La tarea 1 es una de alta prioridad basada en eventos que pasa la mayor parte de su tiempo en el estado Bloqueado a la espera de su evento de interés. La tarea 1 pasa del estado Bloqueado al estado Listo (y posteriormente, porque es la tarea con el estado Listo de prioridad más alta, al estado En ejecución) cada vez que se produce el evento. En el gráfico anterior se muestra a la tarea 1 procesando un evento entre las horas t6 y t7 y después, de nuevo, entre las horas t9 y t10.

3. Tarea de inactividad y tarea 2

El tarea de tiempo de inactividad y la tarea 2 son ambas tareas de procesamiento continuo y ambas tienen una prioridad de 0 (la prioridad de tiempo de inactividad). Las tareas de procesamiento continuo no entran en el estado Bloqueado.

Los intervalos de tiempo no se usan, por lo que una tarea de prioridad de inactividad que se encuentre en el estado En ejecución permanecerá en dicho estado hasta que la reemplace la tarea 1 con más prioridad.

En el gráfico anterior, la tarea de inactividad comienza a ejecutarse en el momento t1 y permanece en el estado En ejecución hasta que la reemplaza la tarea 1 en el momento t6, que está a más de cuatro períodos de ciclos completos después de entrar en el estado En ejecución.

La tarea 2 comienza a ejecutarse en el momento t7, que es cuando la tarea 1 vuelve a entrar en el estado Bloqueado para esperar otro evento. La tarea 2 permanece en el estado En ejecución hasta que la reemplaza la tarea 1 en el momento t9 que es menos de un periodo de ciclos después de que entre en el estado En ejecución.

En el momento t10, la tarea de inactividad vuelve a entrar en el estado En ejecución, a pesar de que ha recibido cuatro veces más tiempo de procesamiento que la tarea 2.

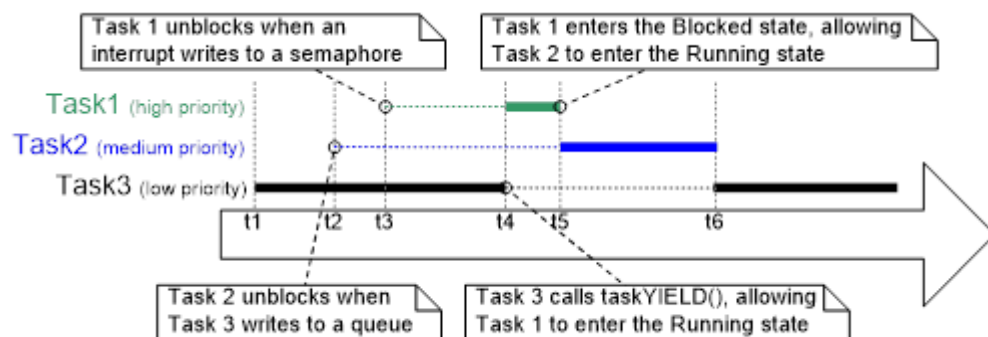
Programación cooperativa

FreeRTOS también puede utilizar la programación cooperativa. En la siguiente tabla se muestran los valores de FreeRTOSConfig.h que configuran el programador FreeRTOS para que use la programación cooperativa.

Constant	Valor
configUSE_PREEMPTION	0
configUSE_TIME_SLICING	Cualquier valor

Cuando se usa el programador cooperativo, se producirá un cambio de contexto solo cuando el estado En ejecución entre en el estado Bloqueado o la tarea en estado En ejecución ceda de forma explícita (solicite manualmente una reprogramación) llamando a `taskYIELD()`. Las tareas nunca se reemplazan, por lo que no se pueden utilizar los intervalos de tiempo.

En el siguiente gráfico se muestra el comportamiento del programador cooperativo. Las líneas discontinuas horizontales muestran cuándo una tarea se encuentra en el estado Listo.



1. Tarea 1

La tarea 1 tiene la máxima prioridad. Comienza en el estado Bloqueado, a la espera de un semáforo.

En el momento t3, una interrupción da el semáforo, lo que hace que la tarea 1 deje el estado Bloqueado y entre en el estado Listo. Para obtener más información acerca de cómo dar semáforos a partir de interrupciones, consulte la sección de [administración de interrupciones](#) (p. 125).

En el momento t3, la tarea 1 es la tarea en estado Listo que tiene la máxima prioridad y si se hubiera usado el programador de reemplazos, la tarea 1 se habría convertido en la tarea con el estado En

ejecución. Sin embargo, como se está utilizando el programador cooperativo, la tarea 1 permanece en el estado Listo hasta el momento t4, que es cuando la tarea con el estado En ejecución llama a taskYIELD().

2. Tarea 2

La prioridad de la tarea 2 está entre la de las tareas 1 y 3. Comienza en el estado Bloqueado, a la espera de que la tarea 3 le envíe un mensaje en el momento t2.

En el momento t2, la tarea 2 es la tarea en estado Listo que tiene la máxima prioridad y si se hubiera usado el programador de reemplazos, la tarea 2 se habría convertido en la tarea con el estado En ejecución. Sin embargo, como se está utilizando el programador cooperativo, la tarea 2 permanece en el estado Listo hasta que la tarea que está en el estado En ejecución entre en el estado En ejecución o llame a taskYIELD().

La tarea en el estado En ejecución llama a taskYIELD() en el momento t4, pero para entonces la tarea 1 es la tarea de estado Listo con la más alta prioridad, por lo que la tarea 2 no se convierte en realidad en la tarea con el estado En ejecución hasta que la tarea 1 vuelve a entrar en el estado Bloqueado en el momento t5.

En el momento t6, la tarea 2 vuelve a entrar en el estado Bloqueado para esperar al siguiente mensaje, momento en el cual la tarea 3 vuelve a ser la tarea en el estado Listo con la máxima prioridad.

En una aplicación multitarea, el programador debe asegurarse de que no acceden a un mismo recurso más de una tarea a la vez, ya que el acceso simultáneo puede dañar al recurso. Tenga en cuenta la situación siguiente en la que el recurso al que se accede es un UART (puerto serie). Dos tareas escriben cadenas en el UART. La tarea 1 está escribiendo "abcdefghijklmnp" y la tarea 2 está escribiendo "123456789":

1. La tarea 1 se encuentra en el estado En ejecución y comienza a escribir su cadena. Escribe "abcdefg" en el UART, pero sale del estado En ejecución antes de poder escribir más caracteres.
2. La tarea 2 entra en el estado En ejecución y escribe "123456789" en el UART, antes de dejar el estado En ejecución.
3. La tarea 1 vuelve a entrar en el estado En ejecución y escribe los caracteres restantes de su cadena en el UART.

En ese caso, lo que realmente se ha escrito en el UART es "abcdefg123456789hijklmnp". La cadena que ha escrito la tarea 1 no se ha escrito en el UART en una secuencia ininterrumpida como se pretendía. En lugar de ello ha resultado dañada, ya que la cadena que la tarea 2 ha escrito en el UART aparece dentro de ella.

Puede evitar los problemas causados por el acceso simultáneo usando el programador cooperativo. Los métodos para compartir recursos de forma segura entre tareas se abordan más adelante en esta guía. Los recursos proporcionados por FreeRTOS, como, por ejemplo, colas y semáforos, son siempre seguros y se pueden compartir entre tareas.

- Cuando se usa el programador de reemplazos, la tarea que tiene el estado En ejecución puede reemplazarse en cualquier momento, incluso cuando un recurso que comparte con otra tarea se encuentra en estado incoherente. Tal y como muestra el ejemplo del UART, dejar un recurso en un estado incoherente puede dar lugar a que los datos resulten dañados.
- Cuando se usa el programador cooperativo, el programador controla cuándo se puede producir un cambio a otra tarea. Por consiguiente, el programador puede asegurar que no se produzca un cambio a otra tarea mientras haya un recurso en estado incoherente.
- En el ejemplo del UART, el programador puede garantizar que la tarea 1 no salga del estado En ejecución hasta que su cadena completa se haya escrito en el UART y, al hacerlo, se elimina la posibilidad de que la cadena resulte dañada por la activación de otra tarea.

Los sistemas tendrán menos capacidad de respuesta cuando se utilice el programador cooperativo.

- Cuando se usa el programador de reemplazos, el programador comienza a ejecutar una tarea inmediatamente después de que la tarea se convierta en la tarea en el estado Listo con la máxima prioridad. Esto suele ser fundamental en los sistemas en tiempo real que deben responder a eventos de alta prioridad dentro de un periodo de tiempo definido.
- Cuando se usa el programador cooperativo, no se lleva a cabo un cambio a una tarea que se ha convertido en la tarea en el estado Listo con la máxima prioridad hasta que la tarea en estado En ejecución entre en el estado Bloqueado o llame a `taskYIELD()`.

Administración de colas

Las colas proporcionan mecanismo de comunicación de una a otra tarea, de una tarea a una interrupción y de una interrupción a una tarea. En esta sección, se explica la comunicación de una tarea a otra. Para obtener más información acerca de la comunicación de una tarea a una interrupción y de una interrupción a una tarea, consulte [Administración de interrupciones \(p. 125\)](#).

En esta sección se explica lo siguiente:

- Cómo crear una cola.
- Cómo administra una cola los datos que contiene.
- Cómo enviar datos a una cola.
- Cómo recibir datos de una cola.
- Qué significa bloquear en una cola.
- Cómo bloquear en varias colas.
- Cómo sobrescribir datos en una cola.
- Cómo borrar una cola.
- El efecto de las prioridades de tareas al escribir y leer en una cola.

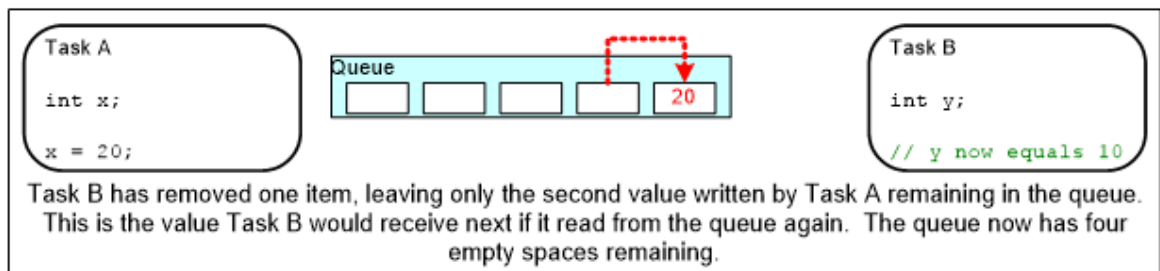
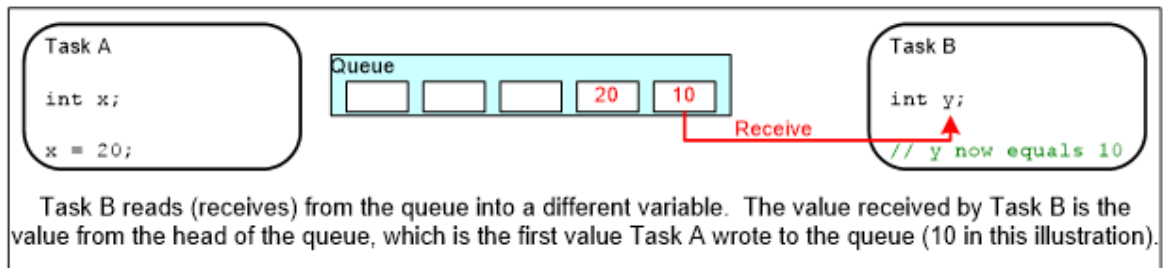
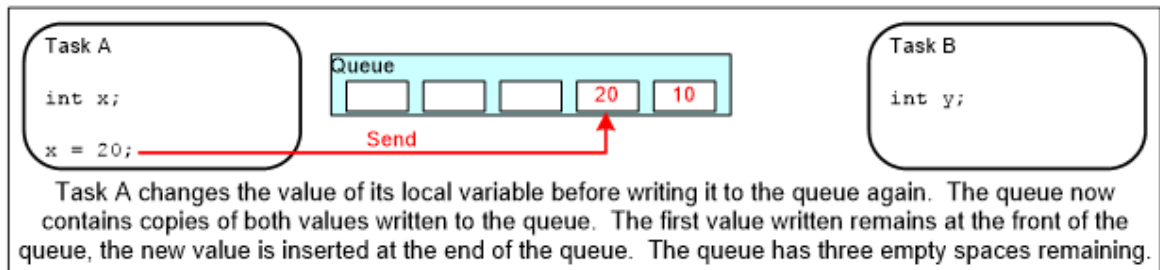
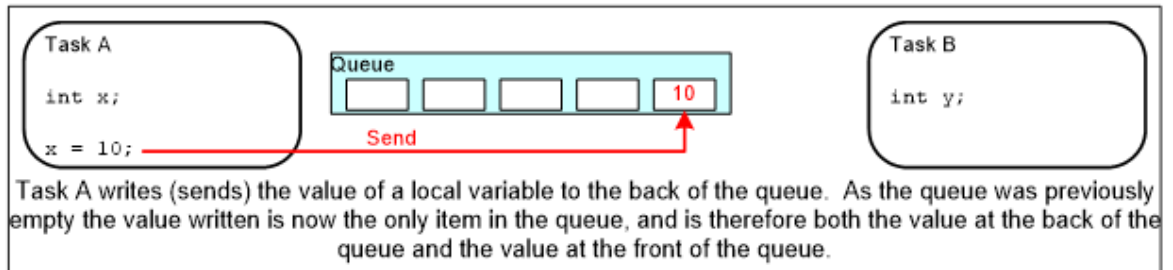
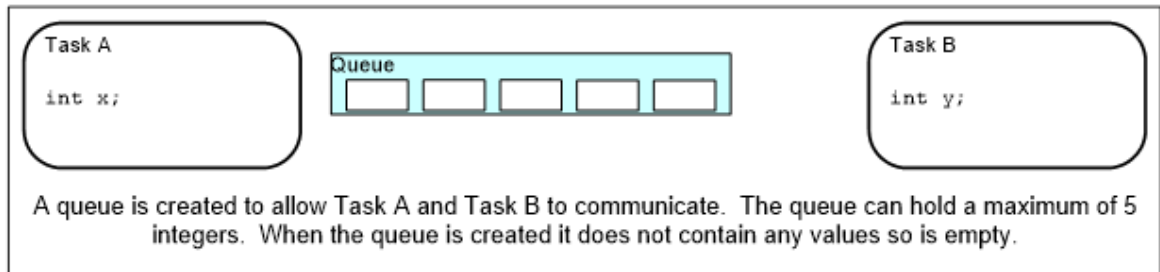
Características de una cola

Almacenamiento de datos

Una cola puede contener un número finito de elementos de datos de tamaño fijo. El número máximo de elementos que puede contener una cola es su longitud. Tanto la longitud como el tamaño de cada elemento de datos se establecen en el momento de crear la cola.

Normalmente, las colas se utilizan como búferes FIFO (primera en entrar, primera en salir), en las que los datos se escriben al final de la cola y se eliminan de la parte delantera de la cola.

En la siguiente figura, se muestra cómo se escriben y se leen datos de una cola que se está utilizando como FIFO. También es posible escribir en la parte delantera de una cola y sobrescribir los datos que ya se encuentran allí.



Hay dos formas de implementar el comportamiento de la cola:

1. Cola por copia

Los datos que se envían a la cola se copian byte por byte en la cola.

2. Cola por referencia

La cola contiene punteros a los datos que se envían a la cola, no los datos propiamente dichos.

FreeRTOS utiliza la cola por copia. Este método se considera más útil y más fácil de usar que la cola por referencia porque:

- Se puede enviar una variable de pila directamente a una cola, aunque la variable no exista después de que la función en la que se ha declarado haya salido.
- Los datos se pueden enviar a una cola sin asignar primero un búfer en el que almacenar los datos y, a continuación, copiar los datos en el búfer asignado.
- La tarea de envío puede reutilizar inmediatamente la variable o el búfer que se enviaron a la cola.
- La tarea de envío y la tarea de recepción están completamente desacopladas. No es necesario que los diseñadores de aplicaciones se preocupen de qué tarea tiene los datos o qué tarea es responsable de la liberación de los datos.
- La cola por copia no impide que la cola también se utilice para la cola por referencia. Por ejemplo, cuando el tamaño de los datos que se ponen en cola hace que sea poco práctico copiar los datos en la cola, se puede copiar un puntero a los datos en la cola en su lugar.
- RTOS asume toda la responsabilidad de asignar la memoria utilizada para almacenar datos.
- En un sistema con protección de memoria, la memoria RAM a la que puede acceder una tarea está restringida. En ese caso, la cola por referencia solo se puede utilizar si las tareas de envío y recepción de tareas pueden acceder a la RAM en la que se almacenan los datos. La cola por copia no impone dicha restricción. El kernel siempre se ejecuta con todos los privilegios, lo que permite usar una cola para pasar datos entre los límites de protección de memoria.

Acceso de varias tareas

Las colas son objetos a los que puede acceder cualquier tarea o registro del servicio de interrupciones (ISR) que conozca su existencia. Cualquier número de tareas puede escribir en la misma cola y cualquier número de tareas puede leer de la misma cola. Es muy común que una cola tenga varios escritores, pero mucho menos frecuente que una cola tenga varios lectores.

Bloqueo en las lecturas de cola

Cuando una tarea intenta leer de una cola, puede especificar un tiempo de bloqueo. Es el tiempo durante el que la tarea se mantendrá en el estado Bloqueado a la espera de que haya datos disponibles en la cola en caso de que esté vacía. Una tarea en el estado Bloqueado, a la espera de que haya disponibles datos en una cola, pasa automáticamente al estado Listo cuando otra tarea o interrupción coloca datos en la cola. La tarea también pasa automáticamente del estado Bloqueado al estado Listo si el tiempo de bloqueo especificado vence antes de que los datos estén disponibles.

Las colas pueden tener varios lectores, por lo que es posible que una única cola tenga bloqueada más de una tarea a la espera de datos. En ese caso, solo se desbloquea una tarea cuando los datos vuelven a estar disponibles. La tarea que se desbloquea siempre será la tarea de máxima prioridad que está a la espera de datos. Si las tareas bloqueadas tienen la misma prioridad, se desbloqueará la tarea que lleve esperando datos más tiempo.

Bloqueo en las escrituras de cola

De la misma forma que en las lecturas de colas, una tarea puede especificar un tiempo de bloqueo al escribir en una cola. En este caso, el tiempo de bloqueo es el tiempo máximo durante el que la tarea debe

mantenerse en el estado Bloqueado a la espera de que haya espacio disponible en la cola si la cola ya está llena.

Las colas pueden tener varios escritores, por lo que es posible que una cola llena tenga bloqueada más de una tarea a la espera de que se complete una operación de envío. En ese caso, solo se desbloquea una tarea cuando vuelve a haber espacio disponible en la cola. La tarea que se desbloquea siempre será la tarea de máxima prioridad que está a la espera de que se libere espacio. Si las tareas bloqueadas tienen la misma prioridad, se desbloqueará la tarea que lleve más tiempo esperando a que se libere espacio.

Bloqueo en varias colas

Las colas pueden agruparse en conjuntos, lo que permite que una tarea pase al estado Bloqueado a la espera de que haya disponibles datos en cualquiera de las colas del conjunto. Para obtener más información acerca de los conjuntos de colas, consulte [Recepción desde varias colas \(p. 89\)](#).

Uso de una cola

Función de API xQueueCreate()

Una cola se debe crear explícitamente antes de usarla.

A las colas se les hace referencia por medio de controladores, que son las variables de tipo `QueueHandle_t`. La función de API `xQueueCreate()` crea una cola y devuelve un `QueueHandle_t` que hace referencia a la cola que se ha creado.

FreeRTOS V9.0.0 también incluye la función `xQueueCreateStatic()`, que asigna la memoria necesaria para crear una cola estáticamente durante la compilación. FreeRTOS asigna RAM del montón de FreeRTOS cuando se crea una cola. La RAM se utiliza para almacenar las estructuras de datos de la cola y los elementos que se encuentran en la cola. `xQueueCreate()` devuelve NULL si no hay suficiente RAM del montón para la cola que se va a crear.

Aquí se muestra el prototipo de la función de API `xQueueCreate()`.

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

En la siguiente tabla se muestran los parámetros de `xQueueCreate()` y su valor de retorno.

Nombre del parámetro	Descripción
<code>uxQueueLength</code>	El número máximo de elementos que la cola que se está creando puede contener en un momento dado.
<code>uxItemSize</code>	El tamaño, en bytes, de cada elemento de datos que se puede almacenar en la cola.
Valor de retorno	<p>Si se devuelve NULL, la cola no se puede crear porque no hay suficiente memoria disponible en el montón para que FreeRTOS asigne las estructuras de datos y el área de almacenamiento de la cola.</p> <p>Si se devuelve un valor que no sea NULL, eso significa que la cola se ha creado correctamente.</p>

El valor devuelto debe almacenarse como el controlador en la cola creada.

Después de crear una cola, la función de API xQueueReset() se puede utilizar para devolver la cola a su estado vacío original.

Funciones de API xQueueSendToBack() y xQueueSendToFront()

xQueueSendToBack() se utiliza para enviar datos a la parte posterior (final) de una cola.
xQueueSendToFront() se utiliza para enviar datos a la parte delantera (cabeza) de una cola.

xQueueSend() es equivalente y exactamente igual que xQueueSendToBack().

Nota: No llame a xQueueSendToFront() ni a xQueueSendToBack() desde una rutina del servicio de interrupciones. Utilice las versiones a prueba de interrupciones, xQueueSendToFrontFromISR() y xQueueSendToBackFromISR() en su lugar.

Aquí se muestra el prototipo de la función de API xQueueSendToFront().

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

Aquí se muestra el prototipo de la función de API xQueueSendToBack().

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

En la siguiente tabla, se enumeran los parámetros de las funciones xQueueSendToFront() y xQueueSendToBack() y su valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xQueue	El controlador de la cola a la que se envían los datos (escritos). El controlador de cola se habrá devuelto desde la llamada a xQueueCreate() que se utiliza para crear la cola.
pvItemToQueue	Un puntero a los datos que se van a copiar en la cola. El tamaño de cada elemento que puede contener la cola se establece al crear la cola, por lo que esta cantidad de bytes se copiará desde pvItemToQueue en el área de almacenamiento de la cola.
xTicksToWait	El tiempo máximo durante el que la tarea debe mantenerse en el estado Bloqueado a la espera de

	<p>que haya espacio disponible en la cola si la cola ya está llena.</p> <p>xQueueSendToFront() y xQueueSendToBack() se devolverán inmediatamente si xTicksToWait es igual a cero y la cola ya está llena.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. Se puede usar la macro pdMS_TO_TICKS() para convertir un tiempo especificado en milisegundos en un tiempo especificado en ciclos.</p> <p>Al establecer xTicksToWait en portMAX_DELAY, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando INCLUDE_vTaskSuspend esté establecido en 1 en FreeRTOSConfig.h.</p>
Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <ol style="list-style-type: none">1. pdPASS Solo se devuelve si los datos se han enviado correctamente a la cola. Si se ha especificado un tiempo de bloqueo (xTicksToWait no es cero), es posible que la tarea que realizó la llamada entrara en el estado Bloqueado a la espera de que quedase espacio disponible en la cola antes de que se devolviera la función, pero que se escribieran datos correctamente en la cola antes de que el tiempo de bloqueo venciera.2. errQUEUE_FULL Se devuelve si los datos no se han podido escribir en la cola, porque la cola ya estaba llena. Si se ha especificado un tiempo de bloqueo (xTicksToWait no era cero), la tarea de la llamada se habrá puesto en el estado Bloqueado a la espera de que otra tarea o interrupción liberase espacio en la cola, pero el tiempo de bloqueo especificado ha vencido antes de que esto ocurriera.

Función de API xQueueReceive()

xQueueReceive() se utiliza para recibir (leer) un elemento de una cola. El elemento que se recibe se elimina de la cola.

Nota: No llame a xQueueReceive() desde una rutina del servicio de interrupciones. En su lugar, utilice la función de API a prueba de interrupciones xQueueReceiveFromISR().

Aquí se muestra el prototipo de la función de API xQueueReceive().

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

En la siguiente tabla, se muestran los parámetros de la función xQueueReceive() y sus valores de retorno.

Nombre de parámetro/Valor devuelto	Descripción
xQueue	El controlador de la cola desde la que se están recibiendo los datos (leídos). El controlador de cola se habrá devuelto desde la llamada a xQueueCreate() que se utiliza para crear la cola.
pvBuffer	<p>Puntero a la memoria en la que se copiarán los datos recibidos.</p> <p>El tamaño de cada elemento de datos que contiene la cola se establece en el momento de crear la cola. La memoria a la que apunta pvBuffer debe ser al menos lo suficientemente grande como para almacenar esos bytes.</p>
xTicksToWait	<p>El tiempo máximo durante el que la tarea debe mantenerse en el estado Bloqueado a la espera de que haya datos disponibles en la cola si la cola ya está vacía.</p> <p>Si xTicksToWait es cero, xQueueReceive() se devuelve de inmediato si la cola ya está vacía.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos.</p> <p>Al establecer xTicksToWait en portMAX_DELAY, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando INCLUDE_vTaskSuspend esté establecido en 1 en FreeRTOSConfig.h.</p>
Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <p>1. pdPASS</p> <p>Solo se devuelve si los datos se han leído correctamente en la cola.</p> <p>Si se ha especificado un tiempo de bloqueo (xTicksToWait no es cero), es posible que la tarea que realizó la llamada pasara al estado Bloqueado a la espera de que hubiera datos disponibles en la cola, pero que se hayan leído</p>

datos correctamente en la cola antes de que el tiempo de bloqueo venciera.

2. errQUEUE_EMPTY

Se devuelve si no se pueden leer datos en la cola, porque la cola ya está vacía.

Si se ha especificado un tiempo de bloqueo (xTicksToWait no era cero), la tarea de la llamada se habrá puesto en el estado Bloqueado a la espera de que otra tarea o interrupción envíe datos a la cola, pero el tiempo de bloqueo especificado ha vencido antes de que eso ocurriera.

Función de API uxQueueMessagesWaiting()

uxQueueMessagesWaiting() se utiliza para consultar el número de elementos que hay actualmente en una cola.

Nota: No llame a uxQueueMessagesWaiting() desde una rutina del servicio de interrupciones. En su lugar, utilice la versión a prueba de interrupciones, uxQueueMessagesWaitingFromISR().

Aquí se muestra el prototipo de la función de API uxQueueMessagesWaiting().

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

En la siguiente tabla se indica el parámetro de la función uxQueueMessagesWaiting() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xQueue	El controlador de la cola que se está consultando. El controlador de cola se habrá devuelto desde la llamada a xQueueCreate() que se utiliza para crear la cola.
Valor devuelto	El número de elementos que contiene la cola que se está consultando actualmente. Si se devuelve cero, eso significa que la cola está vacía.

Bloqueo al recibir de una cola (Ejemplo 10)

En este ejemplo, se demuestra cómo se crea una cola, cómo varias tareas envían datos a la cola y cómo se reciben datos de la cola. La cola se crea para almacenar elementos de datos de tipo int32_t. Las tareas que envían a la cola no especifican un tiempo de bloqueo, pero sí la tarea que recibe de la cola.

La prioridad de las tareas que envían a la cola es menor que la prioridad de la tarea que recibe de la cola. Esto significa que la cola no debería contener nunca más de un elemento, ya que, en el momento en el que los datos se envían a la cola, la tarea de recepción se desbloqueará, tendrá preferencia sobre la tarea de envío y eliminará los datos, dejando la cola vacía de nuevo.

El código siguiente muestra la implementación de la tarea que escribe en la cola. Se crean dos instancias de esta tarea: una que escribe de forma continua el valor 100 en la cola y otra que escribe de forma

continúa el valor 200 en la misma cola. El parámetro de la tarea se utiliza para pasar estos valores a cada instancia de la tarea.

```
static void vSenderTask( void *pvParameters )
{
    int32_t lValueToSend;

    BaseType_t xStatus;

    /* Two instances of this task are created so the value that is sent to the queue
    is passed in through the task parameter. This way, each instance can use a different
    value. The queue was created to hold values of type int32_t, so cast the parameter to the
    required type. */

    lValueToSend = ( int32_t ) pvParameters;

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )
    {
        /* Send the value to the queue. The first parameter is the queue to which data is
        being sent. The queue was created before the scheduler was started, so before this task
        started to execute. The second parameter is the address of the data to be sent, in this
        case the address of lValueToSend. The third parameter is the Block time, the time the task
        should be kept in the Blocked state to wait for space to become available on the queue
        should the queue already be full. In this case a block time is not specified because the
        queue should never contain more than one item, and therefore never be full. */

        xStatus = xQueueSendToBack( xQueue, &lValueToSend, 0 );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete because the queue was full. This must
            be an error because the queue should never contain more than one item! */

            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

El código siguiente muestra la implementación de la tarea que recibe datos de la cola. La tarea de recepción especifica un tiempo de bloqueo de 100 milisegundos, por lo que pasará al estado Bloqueado a la espera de que haya disponibles datos. Saldrá del estado Bloqueado cuando haya disponibles datos en la cola o pasen 100 milisegundos sin que haya disponible ningún dato. En este ejemplo, el tiempo de espera de 100 milisegundos no debía vencer nunca, porque hay dos tareas escribiendo de forma continua en la cola.

```
static void vReceiverTask( void *pvParameters )
{
    /* Declare the variable that will hold the values received from the queue. */

    int32_t lReceivedValue;
```

```
BaseType_t xStatus;

const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

/* This task is also defined within an infinite loop. */

for( ;; )

{
    /* This call should always find the queue empty because this task will immediately
    remove any data that is written to the queue. */

    if( uxQueueMessagesWaiting( xQueue ) != 0 )
    {
        vPrintString( "Queue should have been empty!\r\n" );
    }

    /* Receive data from the queue. The first parameter is the queue from which data is
    to be received. The queue is created before the scheduler is started, and therefore before
    this task runs for the first time. The second parameter is the buffer into which the
    received data will be placed. In this case the buffer is simply the address of a variable
    that has the required size to hold the received data. The last parameter is the block
    time, the maximum amount of time that the task will remain in the Blocked state to wait
    for data to be available should the queue already be empty. */

    xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );

    if( xStatus == pdPASS )
    {
        /* Data was successfully received from the queue, print out the received value.
        */
        vPrintStringAndNumber( "Received = ", lReceivedValue );
    }
    else
    {
        /* Data was not received from the queue even after waiting for 100ms. This must
        be an error because the sending tasks are free running and will be continuously writing to
        the queue. */
        vPrintString( "Could not receive from the queue.\r\n" );
    }
}
}
```

El siguiente código contiene la definición de la función main(). Simplemente, crea la cola y las tres tareas antes de iniciar el programador. La cola se crea para almacenar un máximo de cinco valores int32_t, aunque las prioridades de las tareas se establecen de forma que la cola no contendrá nunca más de un elemento a la vez.

```
/* Declare a variable of type QueueHandle_t. This is used to store the handle to the queue
that is accessed by all three tasks. */

QueueHandle_t xQueue;

int main( void )
{
    /* The queue is created to hold a maximum of 5 values, each of which is large enough to
    hold a variable of type int32_t. */

    xQueue = xQueueCreate( 5, sizeof( int32_t ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will send to the queue. The task parameter
        is used to pass the value that the task will write to the queue, so one task will
        continuously write 100 to the queue while the other task will continuously write 200 to
        the queue. Both tasks are created at priority 1. */

        xTaskCreate( vSenderTask, "Sender1", 1000, ( void * ) 100, 1, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, ( void * ) 200, 1, NULL );

        /* Create the task that will read from the queue. The task is created with priority
        2, so above the priority of the sender tasks. */

        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

        /* Start the scheduler so the created tasks start executing. */

        vTaskStartScheduler();

    }

    else
    {
        /* The queue could not be created. */

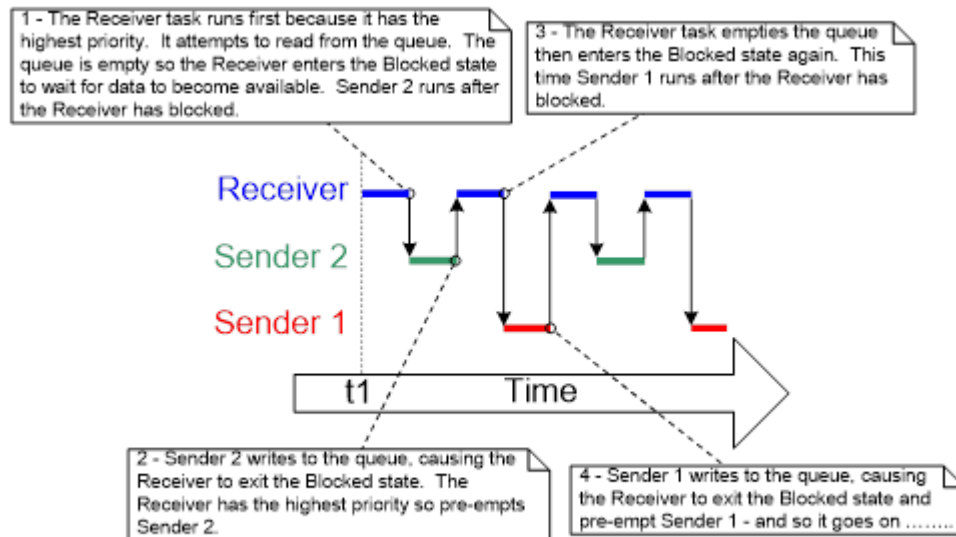
    }

    /* If all is well then main() will never reach here as the scheduler will now be
    running the tasks. If main() does reach here then it is likely that there was insufficient
    FreeRTOS heap memory available for the idle task to be created. For more information, see
    Heap Memory Management. */

    for( ;; );
}
```

Ambas tareas que envían a la cola tiene una prioridad idéntica. Esto hace que las dos tareas de envío envíen datos a la cola por turnos.

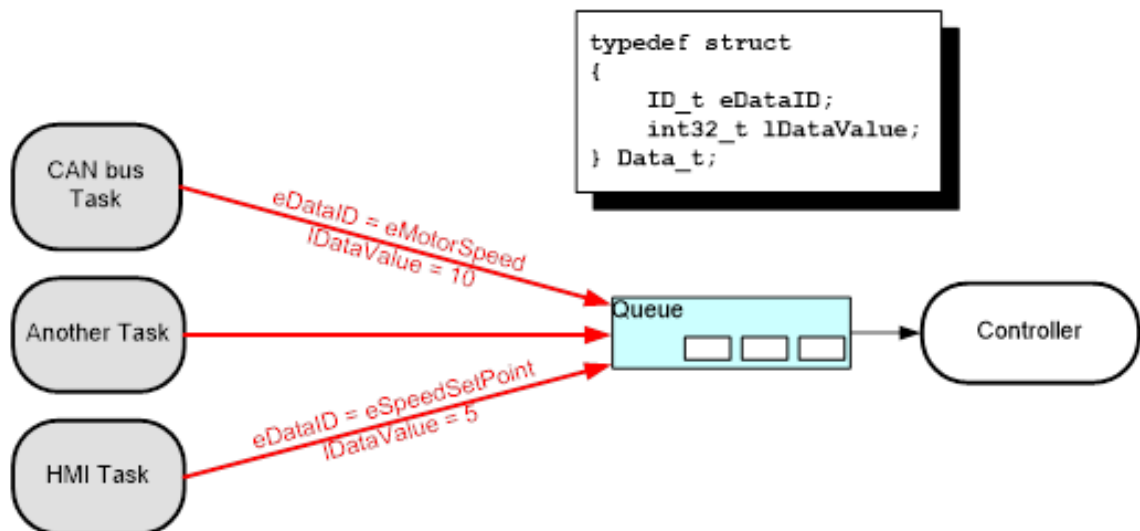
La figura siguiente muestra la secuencia de ejecución.



Recepción de datos de varios orígenes

En diseños de FreeRTOS, es frecuente que una tarea reciba datos de más de un origen. La tarea de recepción debe saber de dónde proceden los datos para determinar cómo se deben procesar. Una solución de diseño muy sencilla es utilizar una única cola para transferir estructuras con el valor de los datos y el origen de los datos contenidos en los campos de la estructura.

La siguiente figura muestra una situación de ejemplo en la que se envían estructuras en una cola.



- Se crea una cola que contiene estructuras de tipo `Data_t`. Los miembros de la estructura permiten enviar un valor de datos y un tipo enumerado a la cola en un mensaje. El tipo enumerado se utiliza para indicar qué significan los datos.
- Se utiliza una tarea de controlador central para realizar la función del sistema principal. Este tiene reaccionar a las entradas y los cambios en el estado del sistema que se le comunica en la cola.

- Se utiliza una tarea del bus CAN para encapsular la funcionalidad de interfaz del bus CAN. Cuando la tarea del bus CAN ha recibido y descodificado un mensaje, envía el mensaje ya descodificado a la tarea del controlador en una estructura Data_t. El miembro eDataID de la estructura transferida se utiliza para permitir que la tarea del controlador sepa cuáles son los datos (en este caso, un valor de velocidad del motor). El miembro lDataValue de la estructura transferida se utiliza para permitir que la tarea del controlador sepa el valor de velocidad del motor.
- Se utiliza una tarea de la interfaz hombre máquina (HMI) para encapsular toda la funcionalidad de HMI. Probablemente, el operario de la máquina puede introducir comandos y consultar valores de varias formas que se tienen que detectar e interpretar dentro de la tarea de HMI. Cuando se introduce un nuevo comando, la tarea de HMI envía el comando a la tarea del controlador en una estructura Data_t. El miembro eDataID de la estructura transferida se utiliza para hacer saber a la tarea del controlador cuáles son los datos (en este caso, un valor de punto de ajuste nuevo). El miembro lDataValue de la estructura transferida se utiliza para hacer saber a la tarea del controlador el valor del punto de ajuste.

Bloqueo al enviar a una cola y envío de estructuras en una cola (Ejemplo 11)

Este ejemplo es similar al ejemplo anterior, pero las prioridades de las tareas se invierten, por lo que la tarea de recepción tiene menor prioridad que las tareas de envío. Además, la cola se utiliza para transferir estructuras en lugar de números enteros.

El código siguiente muestra la definición de la estructura que se va a pasar en una cola y la declaración de dos variables.

```
/* Define an enumerated type used to identify the source of the data.*/  
  
typedef enum  
{  
    eSender1,  
    eSender2  
} DataSource_t;  
  
/* Define the structure type that will be passed on the queue. */  
  
typedef struct  
{  
    uint8_t ucValue;  
    DataSource_t eDataSource;  
} Data_t;  
  
/* Declare two variables of type Data_t that will be passed on the queue. */  
  
static const Data_t xStructsToSend[ 2 ] =  
{  
    { 100, eSender1 }, /* Used by Sender1. */  
    { 200, eSender2 } /* Used by Sender2. */  
}
```

```
};
```

En el ejemplo anterior, la tarea de recepción tiene la prioridad más alta, por lo que la cola nunca contiene más de un elemento. El resultado es que la tarea de recepción tiene preferencia sobre las tareas de envío en cuanto los datos se colocan en la cola. En el siguiente ejemplo, las tareas de envío tienen la mayor prioridad, por lo que normalmente la cola estará llena. Esto se debe a que, en cuanto la tarea de recepción elimina un elemento de la cola, una de las tareas de envío asume la preferencia, lo que rellena de inmediato la cola. A continuación, la tarea de envío vuelve a pasar al estado Bloqueado a la espera de que vuelva a haber espacio disponible en la cola.

El código siguiente muestra la implementación de la tarea de envío. La tarea de envío especifica un tiempo de bloqueo de 100 milisegundos, por lo que entra en el estado Bloqueado a la espera de que haya espacio disponible cada vez que se llena la cola. Sale del estado Bloqueado cuando hay disponible espacio en la cola o pasan 100 milisegundos sin que haya disponible espacio. En este ejemplo, el tiempo de espera de 100 milisegundos no debería vencer nunca, porque la tarea de recepción está liberando espacio continuamente eliminando elementos de la cola.

```
static void vSenderTask( void *pvParameters )
{
    BaseType_t xStatus;

    const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )
    {
        /* Send to the queue. The second parameter is the address of the structure being
        sent. The address is passed in as the task parameter so pvParameters is used directly. The
        third parameter is the Block time, the time the task should be kept in the Blocked state
        to wait for space to become available on the queue if the queue is already full. A block
        time is specified because the sending tasks have a higher priority than the receiving task
        so the queue is expected to become full. The receiving task will remove items from the
        queue when both sending tasks are in the Blocked state. */

        xStatus = xQueueSendToBack( xQueue, pvParameters, xTicksToWait );

        if( xStatus != pdPASS )
        {
            /* The send operation could not complete, even after waiting for 100 ms. This
            must be an error because the receiving task should make space in the queue as soon as both
            sending tasks are in the Blocked state. */

            vPrintString( "Could not send to the queue.\r\n" );
        }
    }
}
```

La tarea de recepción tiene la prioridad más baja, por lo que se ejecutará solo cuando las tareas de envío se encuentren en el estado Bloqueado. Las tareas de envío pasarán al estado Bloqueado solo cuando la cola esté llena, por lo que la tarea de recepción solo se ejecutará cuando la cola ya esté llena. Por lo tanto, siempre espera recibir datos incluso cuando no especifica un tiempo de bloqueo.

El código siguiente muestra la implementación de la tarea de recepción.

```
static void vReceiverTask( void *pvParameters )
{
    /* Declare the structure that will hold the values received from the queue. */
    Data_t xReceivedStructure;

    BaseType_t xStatus;

    /* This task is also defined within an infinite loop. */
    for( ;; )
    {
        /* Because it has the lowest priority, this task will only run when the sending
        tasks are in the Blocked state. The sending tasks will only enter the Blocked state when
        the queue is full so this task always expects the number of items in the queue to be equal
        to the queue length, which is 3 in this case. */

        if( uxQueueMessagesWaiting( xQueue ) != 3 )
        {
            vPrintString( "Queue should have been full!\r\n" );
        }

        /* Receive from the queue. The second parameter is the buffer into which the
        received data will be placed. In this case, the buffer is simply the address of a variable
        that has the required size to hold the received structure. The last parameter is the block
        time, the maximum amount of time that the task will remain in the Blocked state to wait
        for data to be available if the queue is already empty. In this case, a block time is not
        required because this task will only run when the queue is full. */

        xStatus = xQueueReceive( xQueue, &xReceivedStructure, 0 );

        if( xStatus == pdPASS )
        {
            /* Data was successfully received from the queue, print out the received value
            and the source of the value. */

            if( xReceivedStructure.eDataSource == eSender1 )
            {
                vPrintStringAndNumber( "From Sender 1 = ", xReceivedStructure.ucValue );
            }
            else
            {
                vPrintStringAndNumber( "From Sender 2 = ", xReceivedStructure.ucValue );
            }
        }
    }
}
```



```
        else
        {
            /* Nothing was received from the queue. This must be an error because this task
            should only run when the queue is full. */

            vPrintString( "Could not receive from the queue.\r\n" );

        }
    }
}
```

La función main() solo cambia ligeramente con respecto al ejemplo anterior. La cola se crea para almacenar tres estructuras Data_t y las prioridades de las tareas de envío y recepción se invierten. Aquí se muestra la implementación de main().

```
int main( void )
{
    /* The queue is created to hold a maximum of 3 structures of type Data_t. */
    xQueue = xQueueCreate( 3, sizeof( Data_t ) );

    if( xQueue != NULL )
    {
        /* Create two instances of the task that will write to the queue. The parameter
        is used to pass the structure that the task will write to the queue, so one task will
        continuously send xStructsToSend[ 0 ] to the queue while the other task will continuously
        send xStructsToSend[ 1 ]. Both tasks are created at priority 2, which is above the
        priority of the receiver. */

        xTaskCreate( vSenderTask, "Sender1", 1000, &(amp; xStructsToSend[ 0 ] ), 2, NULL );
        xTaskCreate( vSenderTask, "Sender2", 1000, &(amp; xStructsToSend[ 1 ] ), 2, NULL );

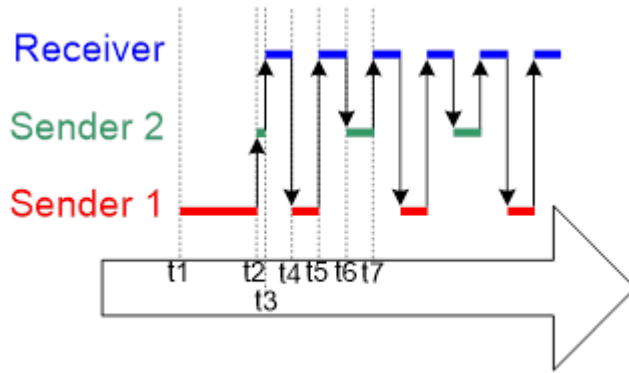
        /* Create the task that will read from the queue. The task is created with priority
        1, so below the priority of the sender tasks. */
        xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 1, NULL );

        /* Start the scheduler so the created tasks start executing. */
        vTaskStartScheduler();
    }
    else
    {
        /* The queue could not be created. */
    }

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

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

La siguiente figura muestra la secuencia de ejecución que se produce cuando la prioridad de las tareas de envío es superior a la prioridad de la tarea de recepción.



Esta tabla describe por qué los primeros cuatro mensajes provienen de la misma tarea.

Tiempo	Descripción
t1	Se ejecuta la tarea Sender 1 y envía tres elementos de datos a la cola.
t2	La cola está llena, por lo que Sender 1 entra en el estado Bloqueado a la espera de que se complete el siguiente envío. La tarea Sender 2 ahora tiene la máxima prioridad que puede ejecutarse, por lo que entra en el estado En ejecución.
t3	La tarea Sender 2 encuentra la cola llena, por lo que entra en el estado Bloqueado a la espera de que se complete el primer envío. La tarea Receiver es ahora la tarea de máxima prioridad que puede ejecutarse, por lo que entra en el estado En ejecución.
t4	Dos tareas que tienen una prioridad mayor que la tarea de recepción están esperando a que se libere espacio en la cola, lo que da como resultado que la tarea Receiver pierda la preferencia en cuanto ha eliminado un elemento de la cola. Las tareas Sender 1 y Sender 2 tienen la misma prioridad, por lo que el programador selecciona la tarea que lleva esperando más tiempo para entrar en el estado En ejecución (en este caso, es la tarea Sender 1).
t5	La tarea Sender 1 envía otro elemento de datos a la cola. Solo quedaba un espacio en la cola, por lo que la tarea Sender 1 entra en el estado Bloqueado a la espera de que se complete el siguiente envío. La tarea Receiver vuelve a ser la

	<p>tarea de máxima prioridad que puede ejecutarse, por lo que entra en el estado En ejecución.</p> <p>La tarea Sender 1 ya ha enviado cuatro elementos a la cola y la tarea Sender 2 sigue esperando para enviar su primer elemento a la cola.</p>
t6	<p>Dos tareas que tienen una prioridad mayor que la tarea de recepción están esperando a que se libere espacio en la cola, por lo que la tarea Receiver pierde la preferencia en cuanto ha eliminado un elemento de la cola. Esta vez, Sender 2 lleva esperando más tiempo que Sender 1, por lo que Sender 2 entra en el estado En ejecución.</p>
t7	<p>La tarea Sender 2 envía un elemento de datos a la cola. Solo quedaba un espacio en la cola, por lo que Sender 2 entra en el estado Bloqueado a la espera de que se complete el siguiente envío. Ambas tareas, Sender 1 y Sender 2, están esperando a que se libere espacio en la cola, por lo que tarea Receiver es la única que puede entrar en el estado En ejecución.</p>

Uso de datos de tamaño grande o variable

Colocación de punteros en la cola

Si el tamaño de los datos que se almacenan en la cola es grande, es mejor utilizar la cola para transferir punteros a los datos en lugar de copiar y sacar los datos de la cola byte por byte. La transferencia de punteros es más eficaz, tanto en lo que se refiere al tiempo de procesamiento como por la cantidad de RAM necesaria para crear la cola. Sin embargo, cuando coloque punteros en la cola, asegúrese de que:

- El propietario de la RAM a la que apunta está definido con claridad.

Cuando utilice un puntero para compartir memoria entre tareas, debe asegurarse de que ambas tareas no modifiquen el contenido de la memoria de forma simultánea ni realicen ninguna otra acción que pueda provocar que el contenido de la memoria no sea válido o sea incoherente. Lo ideal es que solo se permita el acceso a la memoria a la tarea de envío hasta que se ponga en cola un puntero a la memoria y solo se debe permitir el acceso a la memoria a la tarea de recepción después de que el puntero se haya recibido de la cola.

- La RAM a la que apunta sigue siendo válida.

Si la memoria a la que apunta se ha asignado de forma dinámica o se ha obtenido de un grupo de búferes preasignados, entonces una sola tarea debe ser responsable de liberar la memoria. Ninguna tarea debe intentar obtener acceso a la memoria después de que se haya liberado.

No debe utilizar nunca un puntero para obtener acceso a los datos que se han asignado en una pila de tareas. Los datos no serán válidos si el marco de la pila ha cambiado.

En los siguientes ejemplos de código, se muestra cómo usar una cola para enviar un puntero a un búfer de una tarea a otra.

El siguiente código crea una cola que puede contener hasta cinco punteros.

```
/* Declare a variable of type QueueHandle_t to hold the handle of the queue being created.
*/

QueueHandle_t xPointerQueue;

/* Create a queue that can hold a maximum of 5 pointers (in this case, character pointers).
*/

xPointerQueue = xQueueCreate( 5, sizeof( char * ) );
```

El siguiente código asigna un búfer, escribe una cadena en el búfer y, a continuación, envía un puntero al búfer a la cola.

```
/* A task that obtains a buffer, writes a string to the buffer, and then sends the address
of the buffer to the queue created in the previous listing. */

void vStringSendingTask( void *pvParameters )
{
    char *pcStringToSend;

    const size_t xMaxStringLength = 50;

    BaseType_t xStringNumber = 0;

    for( ;; )
    {
        /* Obtain a buffer that is at least xMaxStringLength characters big. The
        implementation of prvGetBuffer() is not shown. It might obtain the buffer from a pool of
        preallocated buffers or just allocate the buffer dynamically. */

        pcStringToSend = ( char * ) prvGetBuffer( xMaxStringLength );

        /* Write a string into the buffer. */

        snprintf( pcStringToSend, xMaxStringLength, "String number %d\r\n",
xStringNumber );
        /* Increment the counter so the string is different on each iteration of this task.
        */

        xStringNumber++;

        /* Send the address of the buffer to the queue that was created in the previous
        listing. The address of the buffer is stored in the pcStringToSend variable.*/

        xQueueSend( xPointerQueue, /* The handle of the queue. */ &pcStringToSend, /* The
        address of the pointer that points to the buffer. */ portMAX_DELAY );

    }
}
```

El siguiente código recibe un puntero a un búfer de la cola y luego imprime la cadena que contiene el búfer.

```
/* A task that receives the address of a buffer from the queue created in the first listing
and written to in the second listing. The buffer contains a string, which is printed out.
*/
```

```
void vStringReceivingTask( void *pvParameters )
{
    char *pcReceivedString;

    for( ;; )
    {
        /* Receive the address of a buffer. */

        xQueueReceive( xPointerQueue, /* The handle of the queue. */ &pcReceivedString, /*
Store the buffer's address in pcReceivedString. */ portMAX_DELAY );

        /* The buffer holds a string. Print it out. */

        vPrintString( pcReceivedString );

        /* The buffer is not required anymore. Release it so it can be freed or reused. */

        prvReleaseBuffer( pcReceivedString );
    }
}
```

Uso de una cola para enviar distintos tipos y longitudes de datos

El envío de estructuras a una cola y el envío de punteros a una cola son dos patrones de diseño muy útiles. Si se combinan estas técnicas, una tarea puede utilizar una única cola para recibir cualquier tipo de datos desde cualquier origen de datos. La implementación de la pila TCP/IP de FreeRTOS+TCP es un ejemplo práctico de todo esto.

La pila TCP/IP, que se ejecuta en su propia tarea, debe procesar eventos de diferentes orígenes. Cada tipo de evento diferente está asociado a diferentes tipos y longitudes de datos. Todos los eventos que se producen fuera de la tarea de TCP/IP se describen por medio de una estructura de tipo `IPStackEvent_t` y se envían a la tarea TCP/IP en una cola. El miembro `pvData` de la estructura `IPStackEvent_t` es un puntero que se puede utilizar para almacenar un valor directamente o para apuntar a un búfer.

Aquí se muestra la estructura `IPStackEvent_t` que se utiliza para enviar eventos a la tarea de la pila TCP/IP en FreeRTOS+TCP.

```
/* A subset of the enumerated types used in the TCP/IP stack to identify events. */
typedef enum
{
    eNetworkDownEvent = 0, /* The network interface has been lost or needs (re)connecting.
*/

    eNetworkRxEvent, /* A packet has been received from the network. */

    eTCPAcceptEvent, /* FreeRTOS_accept() called to accept or wait for a new client. */

    /* Other event types appear here but are not shown in this listing. */
}
```

```
} eIPEvent_t;  
  
/* The structure that describes events and is sent on a queue to the TCP/IP task. */  
typedef struct IP_TASK_COMMANDS  
{  
  
    /* An enumerated type that identifies the event. See the eIPEvent_t definition. */  
    eIPEvent_t eEventType;  
  
    /* A generic pointer that can hold a value or point to a buffer. */  
    void *pvData;  
  
} IPStackEvent_t;
```

Entre algunos ejemplos de eventos de TCP/IP y sus datos asociados se incluyen:

- eNetworkRxEvent: se ha recibido un paquete de datos de la red.

Los datos recibidos de la red se envían a la tarea de TCP/IP mediante una estructura de tipo IPStackEvent_t. El miembro eEventType de la estructura se establece en eNetworkRxEvent. El miembro pvData de la estructura se utiliza para apuntar al búfer que contiene los datos recibidos.

Este pseudocódigo demuestra cómo se utiliza una estructura IPStackEvent_t para enviar datos desde la red a la tarea de TCP/IP.

```
void vSendRxDataToTheTCPTask( NetworkBufferDescriptor_t    *pRxedData )  
{  
    IPStackEvent_t xEventStruct;  
  
    /* Complete the IPStackEvent_t structure. The received data is stored in pRxedData.  
    */  
    xEventStruct.eEventType = eNetworkRxEvent;  
    xEventStruct.pvData = ( void * ) pRxedData;  
  
    /* Send the IPStackEvent_t structure to the TCP/IP task. */  
    xSendEventStructToIPTask( &xEventStruct );  
}
```

- eTCPAcceptEvent: un socket debe aceptar o esperar una conexión de un cliente.

Los eventos de aceptación se envían desde la tarea que ha llamado a la tarea de TCP/IP FreeRTOS_accept() utilizando una estructura de tipo IPStackEvent_t. El miembro eEventType de la estructura se establece en eTCPAcceptEvent. El miembro pvData de la estructura se establece en el controlador del socket que acepta una conexión.

Este pseudocódigo demuestra cómo se utiliza una estructura IPStackEvent_t para enviar el controlador de un socket que acepta una conexión a la tarea de TCP/IP.

```
void vSendAcceptRequestToTheTCPTask( Socket_t xSocket )  
{
```

```
IPStackEvent_t xEventStruct;

/* Complete the IPStackEvent_t structure. */

xEventStruct.eEventType = eTCPAcceptEvent;

xEventStruct.pvData = ( void * ) xSocket;

/* Send the IPStackEvent_t structure to the TCP/IP task. */

xSendEventStructToIPTask( &xEventStruct );

}
```

- eNetworkDownEvent: la red tiene que conectarse o volver a conectarse.

Los eventos de caída de la red se envían a la interfaz de red mediante una tarea de TCP/IP utilizando una estructura de tipo IPStackEvent_t. El miembro eEventType de la estructura se establece en eNetworkDownEvent. Los eventos de caída de la red no están asociados a los datos, por lo que el miembro pvData de la estructura no se utiliza.

Este pseudocódigo demuestra cómo se utiliza una estructura IPStackEvent_t para enviar un evento de caída de la red a la tarea de TCP/IP.

```
void vSendNetworkDownEventToTheTCPTask( Socket_t xSocket )
{
    IPStackEvent_t xEventStruct;

    /* Complete the IPStackEvent_t structure. */

    xEventStruct.eEventType = eNetworkDownEvent;

    xEventStruct.pvData = NULL; /* Not used, but set to NULL for completeness. */

    /* Send the IPStackEvent_t structure to the TCP/IP task. */

    xSendEventStructToIPTask( &xEventStruct );

}
```

Aquí se muestra el código que recibe y procesa estos eventos dentro de la tarea TCP/IP. El miembro eEventType de las estructuras IPStackEvent_t que se reciben de la cola se utiliza para determinar cómo se va a interpretar el miembro pvData. Este pseudocódigo demuestra cómo se utiliza una estructura IPStackEvent_t para enviar una caída de la red a la tarea de TCP/IP.

```
IPStackEvent_t xReceivedEvent;

/* Block on the network event queue until either an event is received, or xNextIPSleep
ticks pass without an event being received. eEventType is set to eNoEvent in case the
call to xQueueReceive() returns because it timed out, rather than because an event was
received. */

xReceivedEvent.eEventType = eNoEvent;

xQueueReceive( xNetworkEventQueue, &xReceivedEvent, xNextIPSleep );

/* Which event was received, if any? */

switch( xReceivedEvent.eEventType )
```

```
{  
  
    case eNetworkDownEvent :  
  
        /* Attempt to (re)establish a connection. This event is not associated with any  
        data. */  
  
        prvProcessNetworkDownEvent();  
  
        break;  
  
    case eNetworkRxEvent:  
  
        /* The network interface has received a new packet. A pointer to the received data  
        is stored in the pvData member of the received IPStackEvent_t structure. Process the  
        received data. */  
  
        prvHandleEthernetPacket( ( NetworkBufferDescriptor_t * )  
        ( xReceivedEvent.pvData ) );  
  
        break;  
  
    case eTCPAcceptEvent:  
  
        /* The FreeRTOS_accept() API function was called. The handle of the socket that  
        is accepting a connection is stored in the pvData member of the received IPStackEvent_t  
        structure. */  
  
        pxSocket = ( FreeRTOS_Socket_t * ) ( xReceivedEvent.pvData );  
  
        xTCPCheckNewClient( pxSocket );  
  
        break;  
  
        /* Other event types are processed in the same way, but are not shown here. */  
  
}
```

Recepción desde varias colas

Conjuntos de colas

Con frecuencia, los diseños de las aplicaciones exigen que una única tarea reciba datos de diferentes tamaños, datos de significado diferente y datos de diferentes orígenes. En la sección anterior, se explicó cómo hacer esto de una forma eficiente y clara mediante una única cola que recibe estructuras. Sin embargo, es posible que a veces trabaje con restricciones que limitan las opciones de diseño, lo que requiere el uso de una cola independiente para algunos orígenes de datos. Por ejemplo, un código de terceros que se esté integrando en un diseño podría suponer que existe una cola especial. En estos casos, puede utilizar un conjunto de colas.

Los conjuntos de colas permiten que una tarea reciba datos de más de una cola sin sondear tareas por turnos en cada cola para determinar cuál contiene datos.

Un diseño que utilice un conjunto de colas para recibir datos de varios orígenes es menos claro y eficiente que un diseño que logra la misma funcionalidad mediante una única cola que recibe estructuras. Por este motivo, le recomendamos que solo utilice conjuntos de colas si es absolutamente necesario debido a las restricciones de diseño.

En las secciones siguientes se describe cómo:

1. Crear un conjunto de colas.
2. Añadir colas al conjunto.

También puede añadir semáforos a un conjunto de colas. Los semáforos se describen en [Administración de interrupciones \(p. 125\)](#).

3. Lea el conjunto de colas para determinar qué colas del conjunto contienen datos.

Cuando una cola que es miembro de un conjunto recibe datos, el controlador de la cola de recepción se envía al conjunto de colas y se devuelve cuando una tarea llama a una función que lee del conjunto de colas. Por lo tanto, si se devuelve un controlador de colas de un conjunto de colas, se sabe que la cola a la que hace referencia el controlador contiene datos, por lo que la tarea puede leer de la cola directamente.

Nota: Si una cola es miembro de un conjunto de colas, no lee datos de la cola a menos que el controlador de la cola se haya leído antes del conjunto de colas.

Para habilitar la funcionalidad del conjunto de colas, en FreeRTOSConfig.h, establezca la constante de configuración en tiempo de compilación configUSE_QUEUE_SETS en 1.

Función de API xQueueCreateSet()

Hay que crear explícitamente un conjunto de colas para poder usarlo.

A los conjuntos de colas se les hace referencia mediante controladores, que son variables de tipo QueueSetHandle_t. La función de API xQueueCreateSet() crea un conjunto de colas y devuelve un QueueSetHandle_t que hace referencia al conjunto de colas que se ha creado.

Aquí se muestra el prototipo de la función de API xQueueCreateSet().

```
QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength);
```

En la siguiente tabla se muestran los parámetros de xQueueCreateSet() y su valor de retorno.

Nombre del parámetro	Descripción
uxEventQueueLength	<p>Cuando una cola que es miembro de un conjunto de colas recibe datos, el controlador de la cola de recepción se envía al conjunto de colas. uxEventQueueLength define el número máximo de controladores de cola que puede contener en un momento dado el conjunto de colas que se está creando.</p> <p>Los controladores de colas solo se envían a un conjunto de colas cuando una cola del conjunto recibe datos. Una cola no puede recibir datos si está llena, por lo que no se pueden enviar controladores de cola al conjunto de colas si todas las colas del conjunto están llenas. Por lo tanto, el número máximo de elementos que contendrá el conjunto de colas en un momento dado es la suma de las longitudes de cada cola del conjunto.</p> <p>Por ejemplo, si hay tres colas vacías en el conjunto, y cada cola tiene una longitud de cinco,</p>

	<p>las colas del conjunto pueden recibir en total quince elementos (tres colas multiplicadas por cinco elementos cada una) antes de todas las colas del conjunto estén llenas. En ese ejemplo, uxEventQueueLength debe establecerse en quince para garantizar que el conjunto de colas pueda recibir todos los elementos que se le envían.</p> <p>También se pueden añadir semáforos a un conjunto de colas. Los semáforos binarios y de recuento se explican más adelante en esta guía. Para calcular uxEventQueueLength, la longitud de un semáforo binario es uno y la longitud de un semáforo de recuento es igual al valor de recuento máximo del semáforo.</p> <p>Como ejemplo adicional, si un conjunto de colas va a contener una cola que tiene una longitud de tres y un semáforo binario (que tiene una longitud de uno), uxEventQueueLength se debe establecer en cuatro (tres más uno).</p>
Valor de retorno	<p>Si se devuelve NULL, el conjunto de colas no se puede crear porque no hay suficiente memoria disponible en el montón para que FreeRTOS asigne las estructuras de datos y el área de almacenamiento del conjunto de colas.</p> <p>Si se devuelve un valor que no sea NULL, eso significa que el conjunto de colas se ha creado correctamente. El valor devuelto debe almacenarse como el controlador en el conjunto de colas creado.</p>

Función de API xQueueAddToSet()

xQueueAddToSet() añade una cola o un semáforo a un conjunto de colas. Para obtener más información sobre los semáforos, consulte [Administración de interrupciones](#) (p. 125).

Aquí se muestra el prototipo de la función de API xQueueAddToSet().

```
BaseType_t xQueueAddToSet( QueueSetMemberHandle_t xQueueOrSemaphore, QueueSetHandle_t  
xQueueSet );
```

En la siguiente tabla, se muestran los parámetros de xQueueAddToSet() y el valor de retorno.

Función de API xQueueSelectFromSet()

xQueueSelectFromSet() lee un controlador de cola en el conjunto de colas.

Cuando una cola o un semáforo que es miembro de un conjunto recibe datos, el controlador del semáforo o la cola de recepción se envía al conjunto de colas y se devuelve cuando una tarea llama a xQueueSelectFromSet(). Si se devuelve un controlador de una llamada a xQueueSelectFromSet(), se sabe que la cola o el semáforo al que hace referencia el controlador contiene datos y la tarea de llamada debe leer directamente de la cola o el semáforo.

Nota: No lea datos de una cola o semáforo que sea miembro de un conjunto a menos que el controlador de la cola o el semáforo se hayan devuelto antes de una llamada a `xQueueSelectFromSet()`. Solo lea un elemento de una cola o semáforo cada vez que se devuelva el controlador de cola o el controlador de semáforo desde una llamada a `xQueueSelectFromSet()`.

Aquí se muestra el prototipo de la función de API `xQueueSelectFromSet()`.

```
QueueSetMemberHandle_t xQueueSelectFromSet( QueueSetHandle_t xQueueSet, const TickType_t xTicksToWait );
```

En la siguiente tabla se muestran los parámetros de `xQueueSelectFromSet()` y su valor de retorno.

xQueueSet

El controlador del conjunto de colas desde el que se está recibiendo un controlador de cola o un controlador de semáforo (lectura). El controlador del conjunto de colas se habrá devuelto desde la llamada a `xQueueCreateSet()` que se utiliza para crear el conjunto de colas.

xTicksToWait

El tiempo máximo durante el que la tarea de llamada debe permanecer en el estado Bloqueado a la espera de recibir un controlador de cola o semáforo del conjunto de colas si todas las colas y el semáforo del conjunto están vacíos. Si `xTicksToWait` es cero, `xQueueSelectFromSet()` se devuelve de inmediato si todas las colas y semáforos del conjunto están vacíos. El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. La macro `pdMS_TO_TICKS()` se puede usar para convertir a ciclos un tiempo especificado en milisegundos. Al establecer `xTicksToWait` en `portMAX_DELAY`, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando `INCLUDE_vTaskSuspend` esté establecido en 1 en `FreeRTOSConfig.h`.

Valor de retorno

Un valor de retorno que no sea `NULL` será el controlador de una cola o semáforo que se sabe que contiene datos. Si se ha especificado un tiempo de bloqueo (`xTicksToWait` no es cero), es posible que la tarea que realizó la llamada entrara en el estado Bloqueado a la espera de que hubiera datos disponibles en una cola o semáforo del conjunto, pero un controlador se ha leído correctamente en el conjunto de colas antes de que el tiempo de bloqueo venciera. Los controladores se devuelven como un tipo `QueueSetMemberHandle_t`, que se puede convertir en el tipo `QueueHandle_t` o el tipo `SemaphoreHandle_t`.

Si el valor de retorno es `NULL`, eso significa que no se ha podido leer un controlador en el conjunto de colas. Si se había especificado un tiempo de bloqueo (`xTicksToWait` no era cero), la tarea de llamada se habrá puesto en el estado Bloqueado a la espera de que otra tarea o interrupción envíe datos a una cola o semáforo del conjunto, pero el tiempo de bloqueo ha vencido antes de que eso ocurriera.

Uso de un conjunto de colas (Ejemplo 12)

En este ejemplo, se crean dos tareas de envío y una tarea de recepción. Las tareas de envío envían datos a la tarea de recepción en dos colas separadas, una cola para cada tarea. Las dos colas se añaden a un conjunto de colas y la tarea de recepción lee el conjunto de colas para determinar cuál de las dos colas contiene datos.

Las tareas, las colas y el conjunto de colas se crean en la función `main()`.

```
/* Declare two variables of type QueueHandle_t. Both queues are added to the same queue set. */  
  
static QueueHandle_t xQueue1 = NULL, xQueue2 = NULL;
```

```
/* Declare a variable of type QueueSetHandle_t. This is the queue set to which the two
queues are added. */

static QueueSetHandle_t xQueueSet = NULL;

int main( void )
{
    /* Create the two queues, both of which send character pointers. The priority of the
    receiving task is above the priority of the sending tasks, so the queues will never have
    more than one item in them at any one time*/

    xQueue1 = xQueueCreate( 1, sizeof( char * ) );

    xQueue2 = xQueueCreate( 1, sizeof( char * ) );

    /* Create the queue set. Two queues will be added to the set, each of which can contain
    1 item, so the maximum number of queue handles the queue set will ever have to hold at one
    time is 2 (2 queues multiplied by 1 item per queue). */

    xQueueSet = xQueueCreateSet( 1 * 2 );

    /* Add the two queues to the set. */

    xQueueAddToSet( xQueue1, xQueueSet );

    xQueueAddToSet( xQueue2, xQueueSet );

    /* Create the tasks that send to the queues. */

    xTaskCreate( vSenderTask1, "Sender1", 1000, NULL, 1, NULL );

    xTaskCreate( vSenderTask2, "Sender2", 1000, NULL, 1, NULL );

    /* Create the task that reads from the queue set to determine which of the two queues
    contain data. */

    xTaskCreate( vReceiverTask, "Receiver", 1000, NULL, 2, NULL );

    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

    /* As normal, vTaskStartScheduler() should not return, so the following lines will
    never execute. */

    for( ;; );

    return 0;
}
```

La primera tarea de envío utiliza xQueue1 para enviar un puntero de carácter a la tarea de recepción cada 100 milisegundos. La segunda tarea de envío utiliza xQueue2 para enviar un puntero de carácter a la tarea de recepción cada 200 milisegundos. Se establecen punteros de caracteres que apuntan a una cadena que identifica a la tarea de envío. Aquí se muestra la implementación de ambas tareas de envío.

```
void vSenderTask1( void *pvParameters )
{
    const TickType_t xBlockTime = pdMS_TO_TICKS( 100 );
```

```
const char * const pcMessage = "Message from vSenderTask1\r\n";

/* As per most tasks, this task is implemented within an infinite loop. */

for( ;; )

{

    /* Block for 100ms. */

    vTaskDelay( xBlockTime );

    /* Send this task's string to xQueue1. It is not necessary to use a block time,
    even though the queue can only hold one item. This is because the priority of the task
    that reads from the queue is higher than the priority of this task. As soon as this task
    writes to the queue, it will be preempted by the task that reads from the queue, so the
    queue will already be empty again by the time the call to xQueueSend() returns. The block
    time is set to 0. */

    xQueueSend( xQueue1, &pcMessage, 0 );

}

}

/*-----*/

void vSenderTask2( void *pvParameters )

{

    const TickType_t xBlockTime = pdMS_TO_TICKS( 200 );

    const char * const pcMessage = "Message from vSenderTask2\r\n";

    /* As per most tasks, this task is implemented within an infinite loop. */

    for( ;; )

    {

        /* Block for 200ms. */

        vTaskDelay( xBlockTime );

        /* Send this task's string to xQueue2. It is not necessary to use a block time,
        even though the queue can only hold one item. This is because the priority of the task
        that reads from the queue is higher than the priority of this task. As soon as this task
        writes to the queue, it will be preempted by the task that reads from the queue, so the
        queue will already be empty again by the time the call to xQueueSend() returns. The block
        time is set to 0. */

        xQueueSend( xQueue2, &pcMessage, 0 );

    }

}
```

Las colas en las que escriben las tareas de envío son miembros del mismo conjunto de colas. Cada vez que una tarea envía a una de las colas, el controlador de la cola se envía al conjunto de colas. La tarea de recepción llama a `xQueueSelectFromSet()` para leer los controladores de cola del conjunto de colas. Una vez que la tarea de recepción ha recibido un controlador de cola del conjunto, sabe que la cola a la que hace referencia el controlador recibido contiene datos, por lo que lee los datos de la cola directamente. Los datos que lee de la cola son un puntero a una cadena, que la tarea de recepción imprime.

Si una llamada a `xQueueSelectFromSet()` agota el tiempo de espera, se devolverá `NULL`. En el código anterior, `xQueueSelectFromSet()` se llama con un tiempo de bloqueo indefinido, por lo que nunca se agota el tiempo de espera y solo puede devolver un controlador de cola válido. Por lo tanto, la tarea de recepción no necesita comprobar si `xQueueSelectFromSet()` ha devuelto `NULL` antes de que se use el valor de retorno.

`xQueueSelectFromSet()` solo devuelve un controlador de cola si la cola a la que se hace referencia mediante el controlador contiene datos, por lo que no es necesario usar un tiempo de bloqueo al leer de la cola.

Aquí se muestra la implementación de la tarea de recepción.

```
void vReceiverTask( void *pvParameters )
{
    QueueHandle_t xQueueThatContainsData;

    char *pcReceivedString;

    /* As per most tasks, this task is implemented within an infinite loop.*/
    for( ;; )
    {
        /* Block on the queue set to wait for one of the queues in the set to contain
        data. Cast the QueueSetMemberHandle_t value returned from xQueueSelectFromSet() to a
        QueueHandle_t because it is known all the members of the set are queues (the queue set
        does not contain any semaphores). */

        xQueueThatContainsData = ( QueueHandle_t ) xQueueSelectFromSet(xQueueSet,
        portMAX_DELAY );

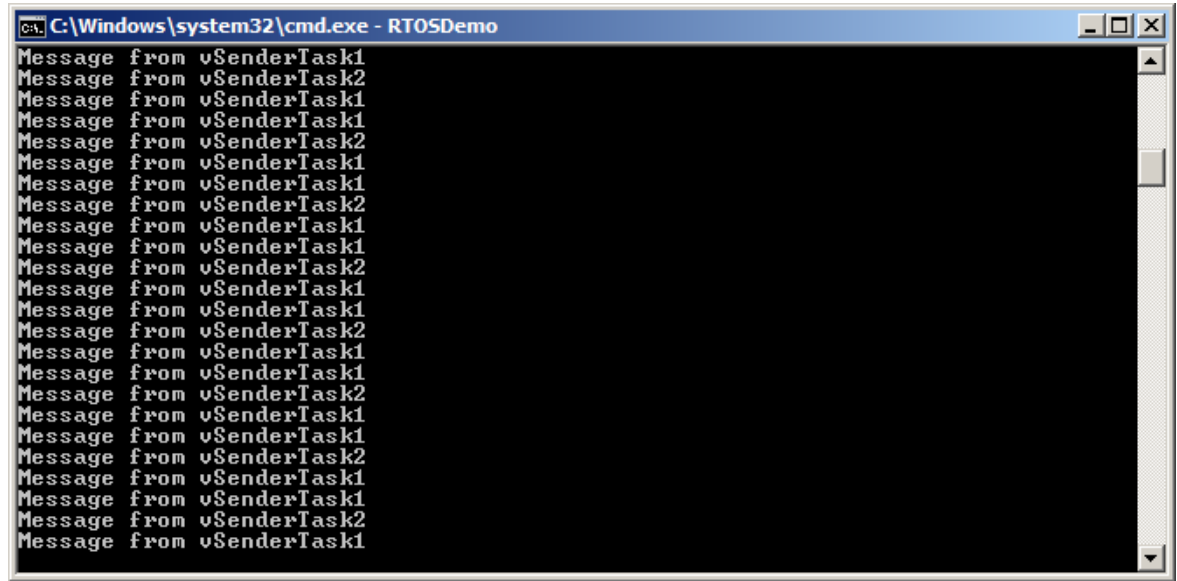
        /* An indefinite block time was used when reading from the queue set, so
        xQueueSelectFromSet() will not have returned unless one of the queues in the set contained
        data, and xQueueThatContainsData cannot be NULL. Read from the queue. It is not necessary
        to specify a block time because it is known the queue contains data. The block time is set
        to 0. */

        xQueueReceive( xQueueThatContainsData, &pcReceivedString, 0 );

        /* Print the string received from the queue. */

        vPrintString( pcReceivedString );
    }
}
```

El resultado se muestra aquí. La tarea de recepción recibe cadenas de ambas tareas de envío. El tiempo de bloqueo que utiliza `vSenderTask1()` es la mitad del tiempo de bloqueo que utiliza `vSenderTask2()`, lo que hace que las cadenas que ha enviado `vSenderTask1()` se impriman con el doble de frecuencia que las que ha enviado `vSenderTask2()`.



Casos de uso de conjuntos de colas más realistas

En el ejemplo anterior, el conjunto de colas solo contenía dos colas. Las colas se utilizaban para enviar un puntero de caracteres. En una aplicación real, un conjunto de colas podría contener colas y semáforos y es posible que no todas las colas contengan los mismos tipos de datos. Si es así, debe probar el valor que devuelve `xQueueSelectFromSet()` antes de utilizar el valor devuelto.

El código siguiente muestra cómo utilizar el valor devuelto desde `xQueueSelectFromSet()` cuando el conjunto cuenta con los siguientes miembros:

1. Un semáforo binario
2. Una cola desde la que se leen punteros de caracteres
3. Una cola desde la que se leen valores de `uint32_t`

En este código, se presupone que las colas y los semáforos ya se han creado y se han añadido al conjunto de colas.

```
/* The handle of the queue from which character pointers are received. */
QueueHandle_t xCharPointerQueue;

/* The handle of the queue from which uint32_t values are received.*/
QueueHandle_t xUInt32tQueue;

/* The handle of the binary semaphore. */
SemaphoreHandle_t xBinarySemaphore;

/* The queue set to which the two queues and the binary semaphore belong. */
QueueSetHandle_t xQueueSet;

void vAMoreRealisticReceiverTask( void *pvParameters )
{
```

```
QueueSetMemberHandle_t xHandle;

    char *pcReceivedString;

uint32_t ulRecievedValue;

const TickType_t xDelay100ms = pdMS_TO_TICKS( 100 );

for( ;; )

{
    /* Block on the queue set for a maximum of 100ms to wait for one of the members of
    the set to contain data. */

    xHandle = xQueueSelectFromSet( xQueueSet, xDelay100ms );

    /* Test the value returned from xQueueSelectFromSet(). If the returned value
    is NULL, then the call to xQueueSelectFromSet() timed out. If the returned value is
    not NULL, then the returned value will be the handle of one of the set's members. The
    QueueSetMemberHandle_t value can be cast to either a QueueHandle_t or a SemaphoreHandle_t.
    Whether an explicit cast is required depends on the compiler. */

    if( xHandle == NULL )
    {
        /* The call to xQueueSelectFromSet() timed out. */
    }

    else if( xHandle == ( QueueSetMemberHandle_t ) xCharPointerQueue )
    {
        /* The call to xQueueSelectFromSet() returned the handle of the queue that
        receives character pointers. Read from the queue. The queue is known to contain data, so a
        block time of 0 is used. */

        xQueueReceive( xCharPointerQueue, &pcReceivedString, 0 );

        /* The received character pointer can be processed here... */
    }

    else if( xHandle == ( QueueSetMemberHandle_t ) xUint32tQueue )
    {
        /* The call to xQueueSelectFromSet() returned the handle of the queue that
        receives uint32_t types. Read from the queue. The queue is known to contain data, so a
        block time of 0 is used. */

        xQueueReceive(xUint32tQueue, &ulRecievedValue, 0 );

        /* The received value can be processed here... */
    }

    else if( xHandle == ( QueueSetMemberHandle_t ) xBinarySemaphore )
    {

```



```
        /* The call to xQueueSelectFromSet() returned the handle of the binary
        semaphore. Take the semaphore now. The semaphore is known to be available, so a block time
        of 0 is used. */

        xSemaphoreTake( xBinarySemaphore, 0 );

        /* Whatever processing is necessary when the semaphore is taken can be
        performed here... */

    }

}

}
```

Uso de una cola para crear un buzón de correo

No hay consenso en la comunidad integrada sobre el significado del término buzón de correo. En esta guía, utilizamos el término para hacer referencia a una cola que tiene una longitud de uno. Una cola podría describirse como un buzón de correo, por la forma en que se utiliza en la aplicación, en lugar de porque tiene una diferencia funcional con respecto a una cola.

- Una cola se utiliza para enviar datos de una tarea a otra o desde una rutina del servicio de interrupciones a una tarea. El remitente coloca un elemento en la cola y el receptor saca el elemento de la cola. Los datos pasan a través de la cola desde el remitente al receptor.
- Un buzón de correo sirve para almacenar datos que puede leer cualquier tarea o rutina del servicio de interrupciones. Los datos no pasan a través del buzón de correo. En su lugar, permanecen en él hasta que se sobrescriben. El remitente sobrescribe el valor en el buzón de correo. El receptor lee el valor del buzón de correo, pero no saca el valor del buzón.

Las funciones de API `xQueueOverwrite()` y `xQueuePeek()` permiten utilizar una cola como un buzón de correo.

El código siguiente muestra cómo se crea una cola para usarla como un buzón de correo.

```
    /* A mailbox can hold a fixed-size data item. The size of the data item is set when the
    mailbox (queue) is created. In this example, the mailbox is created to hold an Example_t
    structure. Example_t includes a timestamp to allow the data held in the mailbox to note
    the time at which the mailbox was last updated. The timestamp used in this example is for
    demonstration purposes only. A mailbox can hold any data the application writer wants, and
    the data does not need to include a timestamp. */

typedef struct xExampleStructure
{
    TickType_t xTimeStamp;

    uint32_t ulValue;
} Example_t;

/* A mailbox is a queue, so its handle is stored in a variable of type
QueueHandle_t. */
QueueHandle_t xMailbox;

void vAFunction( void )
```

```
{  
  
    /* Create the queue that is going to be used as a mailbox. The queue has a length of 1  
    to allow it to be used with the xQueueOverwrite() API function, which is described below.  
    */  
  
    xMailbox = xQueueCreate( 1, sizeof( Example_t ) );  
  
}
```

Función de API xQueueOverwrite()

Al igual que la función de API xQueueSendToBack(), la función de API xQueueOverwrite() envía datos a una cola. A diferencia de xQueueSendToBack(), si la cola ya está llena, xQueueOverwrite() sobrescribirá los datos que ya se encuentran en la cola.

xQueueOverwrite() solo debe utilizarse con colas que tienen una longitud de uno. Esa restricción evita la necesidad de que la implementación de la función tome una decisión arbitraria respecto a qué elemento de la cola debe sobrescribir si la cola está llena.

Nota: No llame a xQueueOverwrite() desde una rutina del servicio de interrupciones. En su lugar, utilice la versión a prueba de interrupciones, xQueueOverwriteFromISR().

Aquí se muestra el prototipo de la función de API xQueueOverwrite().

```
BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void * pvItemToQueue );
```

En la siguiente tabla se muestran los parámetros de xQueueOverwrite() y su valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xQueue	El controlador de la cola a la que se envían los datos (escritos). El controlador de cola se habrá devuelto desde la llamada a xQueueCreate() que se utiliza para crear la cola.
pvItemToQueue	Un puntero a los datos que se van a copiar en la cola. El tamaño de cada elemento que puede contener la cola se establece al crear la cola, por lo que esta cantidad de bytes se copiará desde pvItemToQueue en el área de almacenamiento de la cola.
Valor devuelto	xQueueOverwrite() escribirá en la cola incluso cuando la cola esté llena, por lo que pdPASS es el único valor de retorno posible.

El código siguiente muestra cómo se usa xQueueOverwrite() para escribir en el buzón de correo (cola) que se ha creado anteriormente.

```
void vUpdateMailbox( uint32_t ulNewValue )  
{
```

```
/* Example_t was defined in the earlier code example. */  
  
Example_t xData;  
  
/* Write the new data into the Example_t structure.*/  
  
xData.ulValue = ulNewValue;  
  
/* Use the RTOS tick count as the timestamp stored in the Example_t structure. */  
  
xData.xTimeStamp = xTaskGetTickCount();  
  
/* Send the structure to the mailbox, overwriting any data that is already in the  
mailbox. */  
  
xQueueOverwrite( xMailbox, &xData );  
  
}
```

Función de API xQueuePeek()

xQueuePeek() se utiliza para recibir (leer) un elemento de una cola sin sacar el elemento de la cola. xQueuePeek() recibe los datos de la cabeza de la cola sin modificar los datos almacenados en la cola o el orden en que están almacenados en ella.

Nota: No llame a xQueuePeek() desde una rutina del servicio de interrupciones. En su lugar, utilice la versión a prueba de interrupciones, xQueuePeekFromISR().

xQueuePeek() tiene los mismos parámetros de función y valor de retorno que xQueueReceive().

El código siguiente muestra cómo se utiliza xQueuePeek() para recibir el elemento publicado en el buzón de correo (cola) creado en un ejemplo anterior.

```
BaseType_t vReadMailbox( Example_t *pxData )  
{  
  
    TickType_t xPreviousTimeStamp;  
  
    BaseType_t xDataUpdated;  
  
    /* This function updates an Example_t structure with the latest value received from the  
    mailbox. Record the timestamp already contained in *pxData before it gets overwritten by  
    the new data. */  
  
    xPreviousTimeStamp = pxData->xTimeStamp;  
  
    /* Update the Example_t structure pointed to by pxData with the data contained in  
    the mailbox. If xQueueReceive() was used here, then the mailbox would be left empty  
    and the data could not then be read by any other tasks. Using xQueuePeek() instead of  
    xQueueReceive() ensures the data remains in the mailbox. A block time is specified, so  
    the calling task will be placed in the Blocked state to wait for the mailbox to contain  
    data should the mailbox be empty. An infinite block time is used, so it is not necessary  
    to check the value returned from xQueuePeek(). xQueuePeek() will only return when data is  
    available. */  
  
    xQueuePeek( xMailbox, pxData, portMAX_DELAY );  
  
    /* Return pdTRUE if the value read from the mailbox has been updated since this  
    function was last called. Otherwise, return pdFALSE. */  
}
```

```
    if( pxData->xTimeStamp > xPreviousTimeStamp )
    {
        xDataUpdated = pdTRUE;
    }
    else
    {
        xDataUpdated = pdFALSE;
    }
    return xDataUpdated;
}
```

Administración del temporizador de software

En esta sección se explica lo siguiente:

- Las características de un temporizador de software en comparación con las características de una tarea.
- La tarea de demonio RTOS.
- La cola de comandos del temporizador.
- La diferencia entre un temporizador de software de una sola activación y un temporizador de software periódico.
- Cómo crear, comenzar, restablecer y cambiar el periodo de un temporizador de software.

Los temporizadores de software se utilizan para programar la ejecución de una función en un momento determinado del futuro o periódicamente con una frecuencia fija. La función que ejecuta el temporizador de software se denomina función de devolución de llamada del temporizador de software.

Los temporizadores de software se implementan mediante el control del kernel de FreeRTOS kernel y con el control de este. No requieren soporte de hardware. No están relacionados con los temporizadores de hardware ni con los contadores de hardware.

En consonancia con la filosofía de FreeRTOS de usar un diseño innovador para conseguir una máxima eficiencia, los temporizadores de software no utilizan tiempo de procesamiento a menos que se esté ejecutando una función de devolución de llamada de un temporizador de software.

La funcionalidad de temporizador de software es opcional. Para incluir la funcionalidad de temporizador de software:

1. Cree el archivo de origen de FreeRTOS, FreeRTOS/Source/timers.c, como parte de su proyecto.
2. En FreeRTOSConfig.h, establezca configUSE_TIMERS en 1.

Las funciones de devolución de llamada del temporizador de software

Las funciones de devolución de llamada del temporizador de software se implementan como funciones C. Solo se distinguen por su prototipo, que debe regresar vacío, y tomar un controlador a un temporizador de software como su único parámetro.

A continuación se muestra el prototipo de función de devolución de llamada.

```
void ATimerCallback( TimerHandle_t xTimer );
```

Las funciones de devolución de llamada del temporizador de software se ejecutan de principio a fin y se cierran de forma normal. Deben mantenerse breves y no tienen que entrar en el estado Bloqueado.

Nota: Las funciones de devolución de llamada del temporizador de software se ejecutan en el contexto de una tarea que se crea automáticamente cuando se inicia el programador de FreeRTOS. Por lo tanto, nunca deben llamar a funciones de API de FreeRTOS que hagan que la tarea de la llamada entre en el estado Bloqueado. Pueden llamar a funciones como xQueueReceive(), pero solo si el parámetro xTicksToWait de la función (que especifica el tiempo de bloqueo de la función) está establecido en 0. No pueden llamar a funciones como vTaskDelay() ya que esto siempre pondrá la tarea de llamada en el estado Bloqueado.

Atributos y estados de un temporizador de software

Periodo de un temporizador de software

Por periodo de temporizador de software se entiende el tiempo que transcurre entre el inicio del temporizador de software y la ejecución de su función de devolución de llamada.

Temporizadores de una activación y temporizadores de recarga automática

Existen dos tipos de temporizadores de software:

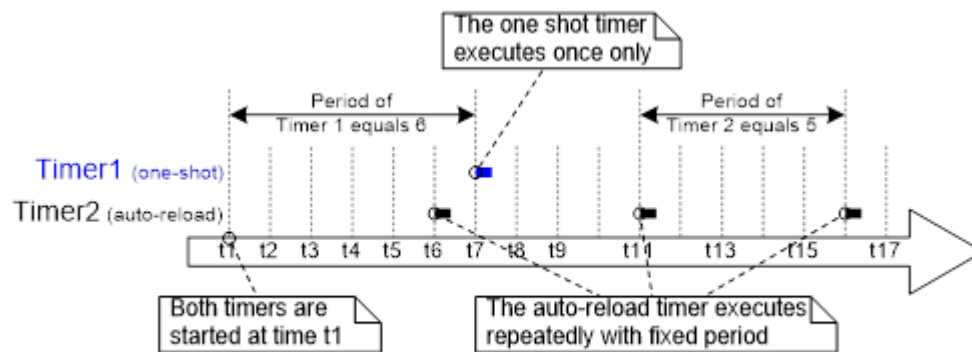
- Temporizadores de una activación

Cuando se inicia, el temporizador de una activación ejecuta su función de devolución de llamada una sola vez. Un temporizador de una activación se puede reiniciar manualmente, pero él mismo no se reiniciará.

- Temporizadores de recarga automática

Cuando se inicia, el temporizador de recarga automática se reinicia él mismo cada vez que vence, lo que se traduce en una ejecución periódica de su función de devolución de llamada.

En la figura siguiente se muestra la diferencia de comportamiento entre un temporizador de una activación y un temporizador de recarga automática. Las líneas verticales discontinuas marcan en qué momento se produce una interrupción de ciclo.



- El temporizador 1 es un temporizador de una sola activación que tiene un periodo de 6 ciclos. Comienza en el momento t1, por lo que su función de devolución de llamada se ejecuta 6 ciclos después, en el momento t7. Como el temporizador 1 es de una sola activación, su función de devolución de llamada no se vuelve a ejecutar.
- El temporizador 2 es un temporizador de recarga automática que tiene un periodo de 5 ciclos. Comienza en el momento t1, por lo que su función de devolución de llamada se ejecuta cada 5 ciclos después del momento t1. En la figura, dicha ejecución se produce en los momentos t6, t11 y t16.

Estados de los temporizadores de software

Un temporizador de software puede estar en uno de los dos estados siguientes:

- Latente

Hay un temporizador de software latente y se puede hacer referencia a él utilizando su controlador, pero no se está ejecutando, lo que significa que sus funciones de devolución de llamada no se ejecutan.

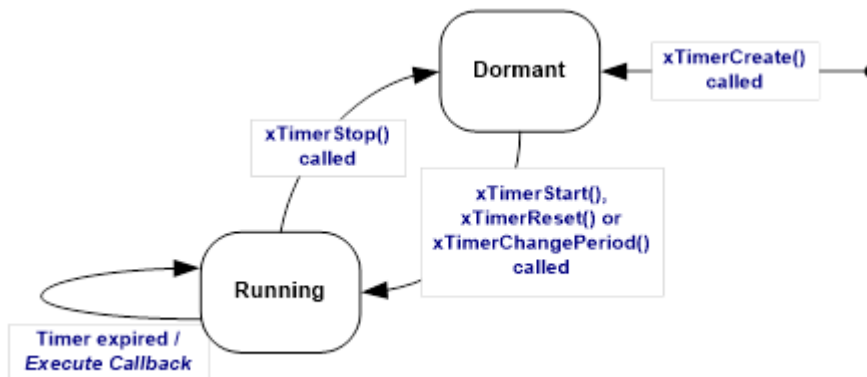
- En ejecución

Un temporizador de software que se encuentra en el estado En ejecución ejecuta su función de devolución de llamada cuando transcurre un tiempo equivalente a su periodo desde que el momento en que el temporizador de software entró en el estado En ejecución o desde el momento en que se restableció por última vez el temporizador de software.

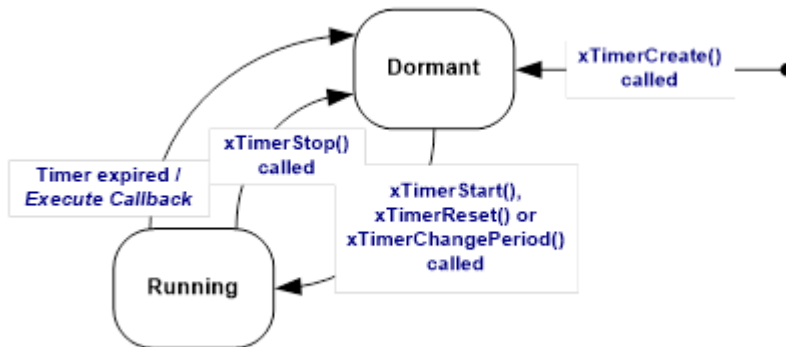
En las dos figuras siguientes se muestran las posibles transiciones entre los estados Latente y En ejecución de un temporizador de recarga automática y un temporizador de una activación. La diferencia clave entre las dos figuras radica en el estado en que entra el temporizador cuando caduca. El temporizador de recarga automática ejecuta su función de devolución de llamada y después vuelve a entrar en el estado En ejecución. El temporizador de una activación ejecuta su función de devolución de llamada y luego entra en el estado Latente.

La función de API `xTimerDelete()` elimina un temporizador. Los temporizadores se pueden eliminar en cualquier momento.

En la figura siguiente se muestran los estados y las transiciones del temporizador de software de recarga automática.



En la figura siguiente se muestran los estados y las transiciones del temporizador de software de una activación.



El contexto de los temporizadores de software

Tarea de demonio RTOS (servicio de temporizador)

Todas las funciones de devolución de llamadas del temporizador de software se ejecutan en el contexto de la misma tarea de demonio RTOS (o servicio de temporizador). (Esta tarea se solía llamar tarea de servicio del temporizador porque en un principio solo se usaba para ejecutar funciones de devolución de llamadas del temporizador de software. Ahora esa misma tarea se utiliza también para otros fines, por lo que se conoce con el nombre más general de tarea de demonio RTOS).

La tarea de demonio es una tarea de FreeRTOS estándar que se crea automáticamente cuando se inicia el programador. Su prioridad y el tamaño de pila se establecen con las constantes de configuración de tiempo de compilación `configTIMER_TASK_PRIORITY` y `configTIMER_TASK_STACK_DEPTH` respectivamente. Ambas constantes están definidas en `FreeRTOSConfig.h`.

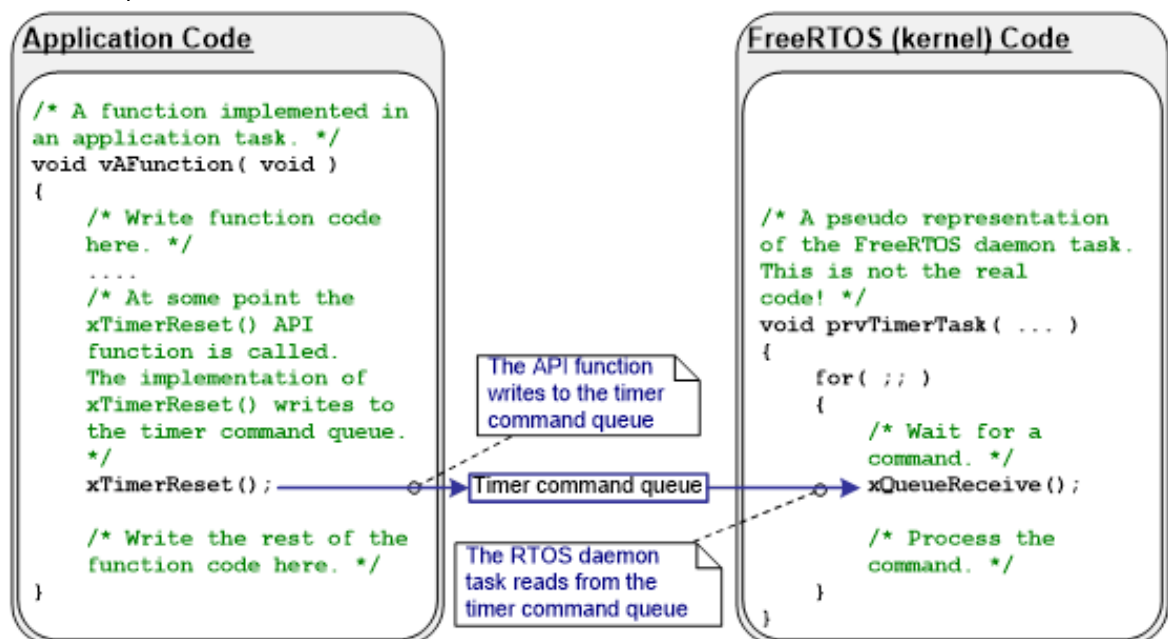
Las funciones de temporizador de software no deben llamar a funciones de API de FreeRTOS que hagan que la tarea de llamada entre en el estado Bloqueado, ya que si esto ocurre, la tarea de demonio también entrará en el estado Bloqueado.

La cola de comandos del temporizador

Las funciones de API del temporizador de software envían comandos desde la tarea de llamada a la tarea de demonio en una cola llamada la cola de comandos del temporizador. Como ejemplos de comandos se pueden citar "start a timer", "stop a timer" y "reset a timer".

La cola de comandos del temporizador es una cola de FreeRTOS estándar que se crea automáticamente cuando se inicia el programador. La longitud de la cola de comandos del temporizador se establece mediante la constante de configuración del tiempo de compilación `configTIMER_QUEUE_LENGTH` en `FreeRTOSConfig.h`.

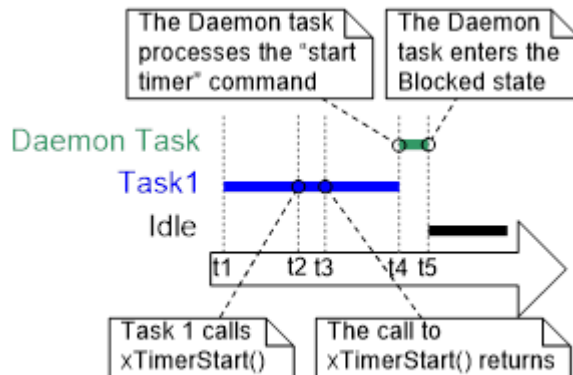
En la figura siguiente se muestra cómo función de API de temporizador de software usa la cola de comandos para comunicarse con la tarea de demonio RTOS.



Programación de tareas de demonio

Las tareas de demonio se programan igual que cualquier otra tarea de FreeRTOS. Procesan comandos o ejecutan funciones de devolución de llamada del temporizador solo cuando son la tarea de máxima prioridad que puede ejecutarse. En las siguientes figuras se muestra cómo `configTIMER_TASK_PRIORITY` influye en el patrón de ejecución.

En esta figura se muestra el patrón de ejecución que se genera cuando la prioridad de la tarea de demonio está por debajo de la prioridad de una tarea que llama a la función de API `xTimerStart()`.



1. En el momento t1

La tarea 1 se encuentra en el estado En ejecución, mientras que la tarea de demonio está en el estado Bloqueado.

La tarea de demonio abandonará el estado Bloqueado si se envía un comando a la cola de comandos del temporizador, en cuyo caso procesará el comando. Si un temporizador de software vence, ejecutará la función de devolución de llamada de dicho temporizador.

2. En el momento t2

La tarea 1 llama a `xTimerStart()`.

`xTimerStart()` envía un comando a la cola de comandos del temporizador, lo que hace que la tarea de demonio abandone el estado Bloqueado. La prioridad de la tarea 1 es superior a la prioridad de la tarea de demonio, por lo que esta última tarea no reemplaza a la tarea 1.

La tarea 1 sigue encontrándose en el estado En ejecución. La tarea de demonio ha abandonado el estado Bloqueado y se encuentra en el estado Listo.

3. En el momento t3

La tarea 1 acaba de ejecutar la función de API `xTimerStart()`. La tarea 1 ha ejecutado `xTimerStart()` desde el inicio hasta el final de la función sin salir del estado En ejecución.

4. En el momento t4

La tarea 1 llama a una función de API y esto hace que entre en el estado Bloqueado. Ahora la tarea de demonio es la tarea de máxima prioridad que se encuentra en el estado Listo, por lo que el programador selecciona la tarea de demonio como la tarea que tiene que entrar en el estado En ejecución. La tarea de demonio comienza a procesar el comando que la tarea 1 ha enviado a la cola de comandos del temporizador.

Nota: El momento en que expira el temporizador de software que se está iniciando se calcula a partir del momento en que se envía el comando "start a timer" a la cola de comandos del temporizador. No se

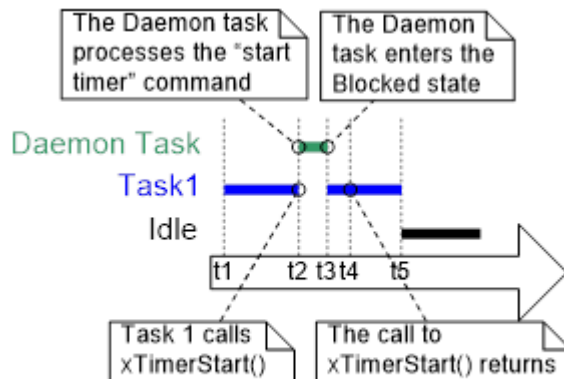
calcula a partir del momento en que la tarea de demonio recibe el comando "start a timer" de la cola de comandos del temporizador.

5. En el momento t5

La tarea de demonio ha acabado de procesar el comando que ha recibido de la tarea 1 e intenta recibir más datos desde la cola de comandos del temporizador. La cola de comandos del temporizador está vacía, por lo que la tarea de demonio vuelve a entrar en el estado Bloqueado. La tarea de demonio dejará de nuevo el estado Bloqueado si se envía un comando a la cola de comandos del temporizador o caduca un temporizador de software.

Ahora la tarea de inactividad es la tarea de máxima prioridad que se encuentra en el estado Listo, por lo que el programador selecciona la tarea de inactividad como la tarea que tiene que entrar en el estado En ejecución.

En la siguiente figura se muestra una situación similar a la que se enseña en la figura anterior. Esta vez la prioridad de la tarea de demonio supera la prioridad de la tarea que llama a xTimerStart().



1. En el momento t1

Igual que en la situación anterior, la tarea 1 se encuentra en el estado En ejecución, mientras que la tarea de demonio se encuentra en el estado Bloqueado.

2. En el momento t2

La tarea 1 llama a xTimerStart().

xTimerStart() envía un comando a la cola de comandos del temporizador, lo que hace que la tarea de demonio abandone el estado Bloqueado. La prioridad de la tarea de demonio es superior a la prioridad de la tarea 1, por lo que el programador selecciona la tarea de demonio como la tarea que debe entrar en el estado En ejecución.

La tarea de demonio ha reemplazado a la tarea 1 antes de que esta acabara de ejecutar la función xTimerStart() y ahora se encuentra en el estado Listo.

La tarea de demonio comienza a procesar el comando que la tarea 1 ha enviado a la cola de comandos del temporizador.

3. En el momento t3

La tarea de demonio ha acabado de procesar el comando que ha recibido de la tarea 1 e intenta recibir más datos desde la cola de comandos del temporizador. La cola de comandos del temporizador está vacía, por lo que la tarea de demonio vuelve a entrar en el estado Bloqueado.

Ahora la tarea 1 es la tarea de máxima prioridad que se encuentra en el estado Listo, por lo que el programador selecciona la tarea 1 como la tarea que debe entrar en el estado En ejecución.

4. En el momento t4

La tarea 1 ha sido reemplazada por la tarea de demonio antes de acabar de ejecutar la función xTimerStart() y sale (regresa) de xTimerStart() solo después de volver a entrar en el estado En ejecución.

5. En el momento t5

La tarea 1 llama a una función de API y esto hace que entre en el estado Bloqueado. Ahora la tarea de inactividad es la tarea de máxima prioridad que se encuentra en el estado Listo, por lo que el programador selecciona la tarea de inactividad como la tarea que tiene que entrar en el estado En ejecución.

En la figura de la primera situación, el tiempo transcurrido entre el momento en que la tarea 1 envía un comando a la cola de comandos del temporizador y el momento en que la tarea de demonio recibe y procesa el comando. La tarea de demonio ha recibido y procesado el comando que le ha enviado la tarea 1 antes de que dicha tarea regrese de la función que ha enviado el comando.

Los comandos enviados a la cola de comandos del temporizador contienen una marca de tiempo. La marca de tiempo se utiliza para tener en cuenta el tiempo transcurrido entre el momento en que una tarea de aplicación envía un comando y el momento en que lo procesa una tarea de demonio. Por ejemplo, si se envía un comando "start a timer" para iniciar un temporizador que tiene un periodo de 10 ciclos, la marca temporal se utiliza para garantizar que el temporizador que se está iniciando venza 10 ciclos después de haberse enviado el comando y no 10 ciclos después de que la tarea de demonio haya procesado el comando.

Creación e inicio de un temporizador de software

La función de API xTimerCreate()

FreeRTOS V9.0.0 también incluye la función xTimerCreateStatic(), que asigna la memoria necesaria para crear un temporizador estáticamente durante la compilación. Es preciso crear un temporizador de software explícitamente para poderlo usar.

La referencia a los temporizadores de software se lleva a cabo con variables del tipo TimerHandle_t. xTimerCreate() se utiliza para crear un temporizador de software y devuelve un TimerHandle_t para hacer referencia al temporizador de software que crea. Los temporizadores de software se crean en el estado Latente.

Los temporizadores de software se pueden crear antes de que el programador se ejecute o desde una tarea una vez que se haya iniciado el programador.

A continuación se muestra el prototipo de función de la API xTimerCreate().

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

En la siguiente tabla se muestran los parámetros de xTimerCreate() y su valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
--------------------------------------	-------------

pcTimerName	Nombre descriptivo del temporizador. FreeRTOS no lo utiliza. Se incluye solo como ayuda para la depuración. Es más sencillo identificar un temporizador con un nombre legible que intentar identificarlo mediante su controlador.
xTimerPeriodInTicks	El periodo del temporizador especificado en ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos.
uxAutoReload	Establezca uxAutoReload en pdTRUE para crear un temporizador de recarga automática. Establezca uxAutoReload en pdFALSE para crear un temporizador de una sola activación.
pvTimerID	<p>Cada temporizador de software tiene un valor de ID. El ID es un puntero vacío y el programador de la aplicación puede utilizarlo para cualquier objetivo. El ID es especialmente útil en los casos en que más de un temporizador de software use la misma función de devolución de llamada ya que se puede utilizar para proporcionar almacenamiento específico para el temporizador.</p> <p>pvTimerID establece un valor inicial para el ID de la tarea que se crea.</p>
pxCallbackFunction	Las funciones de devolución de llamada del temporizador de software son simplemente funciones C que se ajustan al prototipo que se muestra en Funciones de devolución de llamada del temporizador de software (p. 102). El parámetro pxCallbackFunction apunta a la función (de hecho, solo al nombre de la función) que se usará como función de devolución de llamada para el temporizador de software que se crea.
Valor devuelto	<p>Si se devuelve NULL, el temporizador de software no se puede crear porque no hay suficiente memoria en montón disponible para que FreeRTOS asigne la estructura de datos.</p> <p>Si se devuelve un valor que no es NULL, el temporizador de software se ha creado correctamente. El valor devuelto es el identificador del temporizador creado.</p>

La función de API xTimerStart()

xTimerStart() se utiliza para iniciar un temporizador de software que se encuentra en el estado Latente o restablecer (reiniciar) un temporizador que se encuentre en el estado En ejecución. xTimerStop() se utiliza para detener un temporizador de software que se encuentre en el estado En ejecución. Detener un temporizador de software es lo mismo que hacer que el temporizador pase al estado Latente.

Puede llamar a xTimerStart() antes de que el programador se inicie, pero el temporizador de software no se iniciará hasta el momento en que se inicie el programador.

Nota: No llame a xTimerStart() desde una rutina de servicio de interrupción. En su lugar, utilice la versión a prueba de interrupciones, xTimerStartFromISR().

A continuación se muestra el prototipo de la función de API xTimerStart().

```
TimerHandle_t xTimerStart( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

En la tabla siguiente se muestran los parámetros de xTimerStart() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTimer	El identificador del temporizador de software que se inicia o restablece. El identificador se habrá devuelto desde la llamada a xTimerCreate() que se utiliza para crear el temporizador de software.
xTicksToWait	<p>xTimerStart() utiliza la cola de comandos del temporizador para enviar el comando "start a timer" a la tarea de demonio. xTicksToWait especifica el periodo máximo de tiempo que la tarea debe permanecer en el estado Bloqueado a la espera de que quede espacio disponible en la cola de comandos del temporizador si la cola ya está completa.</p> <p>xTimerStart() volverá inmediatamente si xTicksToWait es igual a cero y la cola de comandos del temporizador ya está llena.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos.</p> <p>Si INCLUDE_vTaskSuspend se establece en 1 en FreeRTOSConfig.h, significa que si establece xTicksToWait portMAX_DELAY en portMAX_DELAY, la tarea que realiza la llamada se quedará indefinidamente en el estado Bloqueado (sin tiempo de espera agotado) a esperar a que quede espacio disponible en la cola de comandos del temporizador.</p> <p>Si se llama a xTimerStart() antes de que se haya iniciado el programador, el valor de xTicksToWait se pasará por alto y xTimerStart() se comportará como si xTicksToWait se hubiese establecido en cero.</p>
Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <ol style="list-style-type: none">pdPASS <p>pdPASS se devolverá solo si el comando "start a timer" se ha enviado correctamente a la cola de comandos del temporizador.</p>

Si la prioridad de la tarea de demonio es superior a la prioridad de la tarea que ha llamado a `xTimerStart()`, el programador se asegurará de que el comando de inicio se procese antes que regrese `xTimerStart()`. Esto se debe a que la tarea de demonio reemplazará a la tarea que llamó a `xTimerStart()` tan pronto como haya datos en la cola de comandos del temporizador.

Si se ha especificado un tiempo de bloqueo (`xTicksToWait` no es cero), es posible que la tarea que realizó la llamada entrara en el estado Bloqueado para esperar a que quedara espacio disponible en la cola de comandos del temporizador antes de que la función regresara, pero que se escribieran datos correctamente en la cola de comandos del temporizador antes de que el tiempo de bloqueo venciera.

2. `pdFALSE`

Se devolverá `pdFALSE` si el comando "start a timer" no se ha podido escribir en la cola de comandos del temporizador porque estaba llena.

Si se había especificado un tiempo de bloqueo (`xTicksToWait` no era cero) se habrá puesto la tarea de la llamada en el estado Bloqueado, a esperar a la tarea de demonio para liberar espacio en la cola de comandos del temporizador, pero el temporizador de bloqueo especificado ha caducado antes de que esto ocurra.

Creación de temporizadores de una activación y temporizadores de recarga automática (ejemplo 13)

En este ejemplo se crea y se inicia un temporizador de una sola activación y un temporizador de recarga automática.

```
/* The periods assigned to the one-shot and auto-reload timers are 3.333 second and half a
second, respectively. */

#define mainONE_SHOT_TIMER_PERIOD pdMS_TO_TICKS( 3333 )

#define mainAUTO_RELOAD_TIMER_PERIOD pdMS_TO_TICKS( 500 )

int main( void )
{
    TimerHandle_t xAutoReloadTimer, xOneShotTimer;

    BaseType_t xTimer1Started, xTimer2Started;
```

```
/* Create the one-shot timer, storing the handle to the created timer in xOneShotTimer.
*/

xOneShotTimer = xTimerCreate( /* Text name for the software timer - not
used by FreeRTOS. */ "OneShot", /* The software timer's period, in ticks. */
mainONE_SHOT_TIMER_PERIOD, /* Setting uxAutoReload to pdFALSE creates a one-shot software
timer. */ pdFALSE, /* This example does not use the timer ID. */ 0, /* The callback
function to be used by the software timer being created. */ prvOneShotTimerCallback );

/* Create the auto-reload timer, storing the handle to the created timer in
xAutoReloadTimer. */

xAutoReloadTimer = xTimerCreate( /* Text name for the software timer - not
used by FreeRTOS. */ "AutoReload", /* The software timer's period, in ticks. */
mainAUTO_RELOAD_TIMER_PERIOD, /* Setting uxAutoReload to pdTRUE creates an auto-reload
timer. */ pdTRUE, /* This example does not use the timer ID. */ 0, /* The callback
function to be used by the software timer being created. */ prvAutoReloadTimerCallback );

/* Check the software timers were created. */

if( ( xOneShotTimer != NULL ) && ( xAutoReloadTimer != NULL ) )
{
    /* Start the software timers, using a block time of 0 (no block time). The
scheduler has not been started yet so any block time specified here would be ignored
anyway. */

    xTimer1Started = xTimerStart( xOneShotTimer, 0 );

    xTimer2Started = xTimerStart( xAutoReloadTimer, 0 );

    /* The implementation of xTimerStart() uses the timer command queue, and
xTimerStart() will fail if the timer command queue gets full. The timer service task does
not get created until the scheduler is started, so all commands sent to the command queue
will stay in the queue until after the scheduler has been started. Check both calls to
xTimerStart() passed. */

    if( ( xTimer1Started == pdPASS ) && ( xTimer2Started == pdPASS ) )
    {
        /* Start the scheduler. */

        vTaskStartScheduler();

    }
}

/* As always, this line should not be reached. */

for( ;; );
}
```

Las funciones de devolución de llamada del temporizador se limitan a imprimir un mensaje cada vez que reciben una llamada. A continuación se muestra la implementación de la función de devolución de llamada del temporizador de una sola activación.

```
static void prvOneShotTimerCallback( TimerHandle_t xTimer )
{
}
```

```
TickType_t xTimeNow;

/* Obtain the current tick count. */

xTimeNow = xTaskGetTickCount();

/* Output a string to show the time at which the callback was executed. */

vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );

/* File scope variable. */

ulCallCount++;

}
```

A continuación se muestra la implementación de la función de devolución de llamada del temporizador de recarga automática.

```
static void prvAutoReloadTimerCallback( TimerHandle_t xTimer )

{

    TickType_t xTimeNow;

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

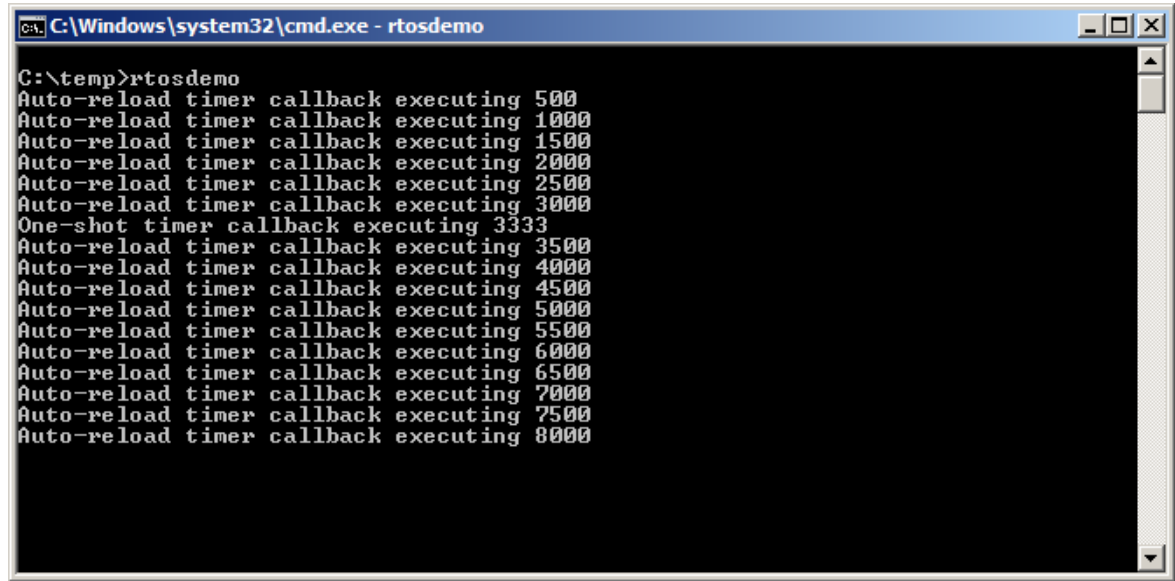
    /* Output a string to show the time at which the callback was executed. */

    vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow);

    ulCallCount++;

}
```

Si se ejecuta este ejemplo se generará el siguiente resultado. Muestra la función de devolución de llamada del temporizador de recarga automática ejecutándose con un periodo fijo de 500 ciclos (mainAUTO_RELOAD_TIMER_PERIOD se establece en 500) y la función de devolución de llamada del temporizador de una sola activación se ejecuta una sola vez, cuando el recuento de ciclos es 3333 (mainONE_SHOT_TIMER_PERIOD se establece en 3333).



```
C:\temp>rtdemo
Auto-reload timer callback executing 500
Auto-reload timer callback executing 1000
Auto-reload timer callback executing 1500
Auto-reload timer callback executing 2000
Auto-reload timer callback executing 2500
Auto-reload timer callback executing 3000
One-shot timer callback executing 3333
Auto-reload timer callback executing 3500
Auto-reload timer callback executing 4000
Auto-reload timer callback executing 4500
Auto-reload timer callback executing 5000
Auto-reload timer callback executing 5500
Auto-reload timer callback executing 6000
Auto-reload timer callback executing 6500
Auto-reload timer callback executing 7000
Auto-reload timer callback executing 7500
Auto-reload timer callback executing 8000
```

El ID de los temporizadores

Cada temporizador de software tiene un ID, que es un valor de etiqueta que puede utilizar el programador de aplicaciones para cualquier objetivo que desee. El ID se almacena en un puntero vacío (`void *`), para que pueda almacenar un valor entero directamente, apuntar a cualquier otro objeto o utilizarse como puntero de función.

Se asigna un valor inicial al ID cuando se crea el temporizador de software. El ID se puede actualizar mediante la función de API `vTimerSetTimerID()` y consultar con la función de API `pvTimerGetTimerID()`.

A diferencia de lo que ocurre con otras funciones de API del temporizador de software, `vTimerSetTimerID()` y `pvTimerGetTimerID()` acceden directamente al temporizador de software. No envían un comando a la cola de comandos del temporizador.

La función de API `vTimerSetTimerID()`

A continuación se muestra el prototipo de función de `vTimerSetTimerID()`.

```
void vTimerSetTimerID( const TimerHandle_t xTimer, void *pvNewID );
```

En la tabla siguiente se enumeran los parámetros de `vTimerSetTimerID()`.

Nombre de parámetro / Valor devuelto	Descripción
xTimer	El identificador del temporizador de software que se actualiza con un valor de ID nuevo. El identificador se habrá devuelto desde la llamada a <code>xTimerCreate()</code> que se utiliza para crear el temporizador de software.
pvNewID	El valor en el que se establecerá el ID del temporizador de software.

La función de API pvTimerGetTimerID()

A continuación se muestra el prototipo de función de pvTimerGetTimerID().

```
void *pvTimerGetTimerID( TimerHandle_t xTimer );
```

En la siguiente tabla se indica el parámetro pvTimerGetTimerID() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTimer	El identificador del temporizador de software que se consulta. El identificador se habrá devuelto desde la llamada a xTimerCreate() que se utiliza para crear el temporizador de software.
Valor devuelto	El ID del temporizador de software que se consulta.

Uso del parámetro de función de devolución de llamada y el ID del temporizador de software (ejemplo 14)

Se puede asignar una misma función de devolución de llamada a más de un temporizador de software. En dicho caso, se usa el parámetro de función de devolución de llamada para determinar qué temporizador de software ha caducado.

En el ejemplo 13 se han usado dos funciones de devolución de llamada independientes: una para el temporizador de una activación y otra para el temporizador de recarga automática. En este ejemplo se crea una funcionalidad similar, pero se asigna una única función de devolución de llamada a ambos temporizadores de software.

En el código siguiente se muestra cómo utilizar la misma función de devolución de llamada prvTimerCallback() para ambos temporizadores.

```
/* Create the one-shot software timer, storing the handle in xOneShotTimer. */  
  
xOneShotTimer = xTimerCreate( "OneShot", mainONE_SHOT_TIMER_PERIOD, pdFALSE, /* The  
timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */  
prvTimerCallback );  
  
/* Create the auto-reload software timer, storing the handle in xAutoReloadTimer */  
  
xAutoReloadTimer = xTimerCreate( "AutoReload", mainAUTO_RELOAD_TIMER_PERIOD, pdTRUE, /* The  
timer's ID is initialized to 0. */ 0, /* prvTimerCallback() is used by both timers. */  
prvTimerCallback );
```

prvTimerCallback() se ejecutará cuando cualquiera de los dos temporizadores caduque. La implementación de prvTimerCallback() utiliza el parámetro de la función para determinar si la han llamado porque ha caducado el temporizador de una activación o el temporizador de recarga automática.

prvTimerCallback() también muestra cómo utilizar el ID del temporizador de software como almacenamiento específico de temporizadores. Cada temporizador de software lleva en su propio ID un recuento del número de veces que ha caducado. El temporizador de recarga automática utiliza el recuento para detenerse la quinta vez que se ejecuta.

A continuación se muestra la implementación de prvTimerCallback().

```
static void prvTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow;

    uint32_t ulExecutionCount;

    /* A count of the number of times this software timer has expired is stored in the
    timer's ID. Obtain the ID, increment it, then save it as the new ID value. The ID is a
    void pointer, so is cast to a uint32_t. */

    ulExecutionCount = ( uint32_t ) pvTimerGetTimerID( xTimer );

    ulExecutionCount++;

    vTimerSetTimerID( xTimer, ( void * ) ulExecutionCount );

    /* Obtain the current tick count. */

    xTimeNow = xTaskGetTickCount();

    /* The handle of the one-shot timer was stored in xOneShotTimer when the timer was
    created. Compare the handle passed into this function with xOneShotTimer to determine if
    it was the one-shot or auto-reload timer that expired, then output a string to show the
    time at which the callback was executed. */

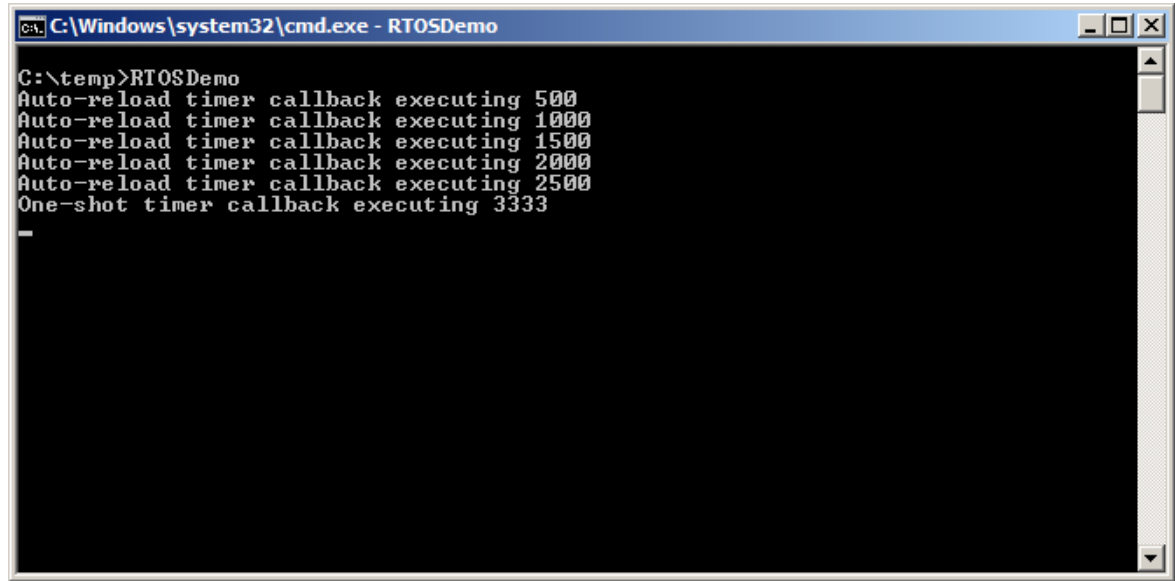
    if( xTimer == xOneShotTimer )
    {
        vPrintStringAndNumber( "One-shot timer callback executing", xTimeNow );
    }
    else
    {
        /* xTimer did not equal xOneShotTimer, so it must have been the auto-reload timer
        that expired. */

        vPrintStringAndNumber( "Auto-reload timer callback executing", xTimeNow );

        if( ulExecutionCount == 5 )
        {
            /* Stop the auto-reload timer after it has executed 5 times. This callback
            function executes in the context of the RTOS daemon task, so must not call any functions
            that might place the daemon task into the Blocked state. Therefore, a block time of 0 is
            used. */

            xTimerStop( xTimer, 0 );
        }
    }
}
```

El resultado se muestra aquí. El temporizador de recarga automática se ejecuta solo cinco veces.



```
C:\Windows\system32\cmd.exe - RTOSDemo
C:\temp>RTOSDemo
Auto-reload timer callback executing 500
Auto-reload timer callback executing 1000
Auto-reload timer callback executing 1500
Auto-reload timer callback executing 2000
Auto-reload timer callback executing 2500
One-shot timer callback executing 3333
```

Cambio de los periodos de un temporizador

Todo puerto FreeRTOS oficial se proporciona con uno o varios proyectos de ejemplo. La mayoría de los proyectos de ejemplo se autocomprueban. Se utiliza un LED para asignar comentarios visuales del estado del proyecto. Si las comprobaciones automáticas han aprobado siempre, el LED se alterna lentamente. Si alguna vez ha fallado una comprobación automática, el LED se alterna rápidamente.

Algunos proyectos de ejemplo realizan las comprobaciones automáticas de una tarea y utilizan la función `vTaskDelay()` para controlar la velocidad a la que se alterna el LED. Otros proyectos de ejemplo realizan las comprobaciones automáticas en una función de devolución de llamada del temporizador de software y utilizan el periodo del controlador para controlar la velocidad a la que se alterna el LED.

La función de API `xTimerChangePeriod()`

Puede utilizar la función `xTimerChangePeriod()` para cambiar el periodo de un temporizador de software.

Si se usa `xTimerChangePeriod()` para cambiar el periodo de un temporizador que ya se está ejecutando, el temporizador usará el nuevo valor de periodo para volver a calcular su periodo de caducidad. El tiempo de caducidad que se vuelve a calcular se indica en relación con el momento en que se llamó a `xTimerChangePeriod()`, no con el momento en que se inició originalmente el temporizador.

Si se usa `xTimerChangePeriod()` para cambiar el periodo de un temporizador que se encuentra en el estado Latente (un temporizador que no se está ejecutando), el temporizador calculará un tiempo de caducidad y pasará al estado En ejecución (el temporizador comenzará a ejecutarse).

Nota: No llame a `xTimerChangePeriod()` desde una rutina de servicio de interrupción. En su lugar, utilice la versión a prueba de interrupciones, `xTimerChangePeriodFromISR()`.

A continuación se muestra el prototipo de la función de API `xTimerChangePeriod()`.

```
BaseType_t xTimerChangePeriod( TimerHandle_t xTimer, TickType_t xNewTimerPeriodInTicks,
                               TickType_t xTicksToWait );
```

En la siguiente tabla se muestran los parámetros `xTimerChangePeriod()` y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTimer	El identificador del temporizador de software que se está actualizando con un nuevo valor de periodo. El identificador se habrá devuelto desde la llamada a xTimerCreate() que se utiliza para crear el temporizador de software.
xTimerPeriodInTicks	El nuevo periodo del temporizador de software, especificado en ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos.
xTicksToWait	<p>xTimerChangePeriod() utiliza la cola de comandos del temporizador para enviar el comando "change period" a la tarea de demonio. xTicksToWait especifica el periodo máximo de tiempo que la tarea debe permanecer en el estado Bloqueado a la espera de que quede espacio disponible en la cola de comandos del temporizador si la cola ya está completa.</p> <p>xTimerChangePeriod() volverá inmediatamente si xTicksToWait es igual a cero y la cola de comandos del temporizador ya está llena.</p> <p>La macro pdMS_TO_TICKS() se puede usar para convertir en ciclos una duración que se ha especificado en milisegundos.</p> <p>Si INCLUDE_vTaskSuspend se establece en 1 en FreeRTOSConfig.h, significa que si establece xTicksToWait portMAX_DELAY en portMAX_DELAY, la tarea que realiza la llamada se quedará indefinidamente en el estado Bloqueado (sin tiempo de espera agotado) a esperar a que quede espacio disponible en la cola de comandos del temporizador.</p> <p>Si se llama a xTimerChangePeriod() antes de que se haya iniciado el programador, el valor de xTicksToWait se pasará por alto y xTimerChangePeriod() se comportará como si xTicksToWait se hubiese establecido en cero.</p>
Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <p>1. pdPASS</p> <p>Se devolverá pdPASS solo si los datos se han enviado correctamente a la cola de comandos del temporizador.</p> <p>Si se ha especificado un tiempo de bloqueo (xTicksToWait no es cero), es posible que la tarea que realizó la llamada entrara en el estado Bloqueado para esperar a que quedara</p>

espacio disponible en la cola de comandos del temporizador antes de que la función regresara, pero que se escribieran datos correctamente en la cola de comandos del temporizador antes de que el tiempo de bloqueo venciera.

2. pdFALSE

Se devolverá pdFALSE si el comando "change period" no se ha podido escribir en la cola de comandos del temporizador porque ya estaba llena.

Si se había especificado un tiempo de bloqueo (xTicksToWait no era cero) se habrá puesto la tarea de la llamada en el estado Bloqueado, a esperar a que la tarea de demonio libere espacio en la cola, pero el temporizador de bloqueo especificado ha caducado antes de que esto ocurra.

El código siguiente muestra cómo la funcionalidad de autocomprobación de una función de devolución de llamada de temporizador de software puede utilizar xTimerChangePeriod() para aumentar la velocidad a la que un LED se alterna si falla una autocomprobación. El temporizador de software que realiza las autocomprobaciones se denomina temporizador de comprobación.

```
/* The check timer is created with a period of 3000 milliseconds, resulting in the LED
   toggling every 3 seconds. If the self-checking functionality detects an unexpected state,
   then the check timer's period is changed to just 200 milliseconds, resulting in a much
   faster toggle rate. */

const TickType_t xHealthyTimerPeriod = pdMS_TO_TICKS( 3000 );

const TickType_t xErrorTimerPeriod = pdMS_TO_TICKS( 200 );

/* The callback function used by the check timer. */

static void prvCheckTimerCallbackFunction( TimerHandle_t xTimer )
{
    static BaseType_t xErrorDetected = pdFALSE;

    if( xErrorDetected == pdFALSE )
    {
        /* No errors have yet been detected. Run the self-checking function again.
           The function asks each task created by the example to report its own status, and also
           checks that all the tasks are actually still running (and so able to report their status
           correctly). */

        if( CheckTasksAreRunningWithoutError() == pdFAIL )
        {
            /* One or more tasks reported an unexpected status. An error might have
               occurred. Reduce the check timer's period to increase the rate at which this callback
               function executes, and in so doing also increase the rate at which the LED is toggled.
               This callback function is executing in the context of the RTOS daemon task, so a block
               time of 0 is used to ensure the Daemon task never enters the Blocked state. */
```

```
        xTimerChangePeriod( xTimer, /* The timer being updated. */ xErrorTimerPeriod, /
* The new period for the timer. */ 0 ); /* Do not block when sending this command. */

    }

    /* Latch that an error has already been detected. */

    xErrorDetected = pdTRUE;

}

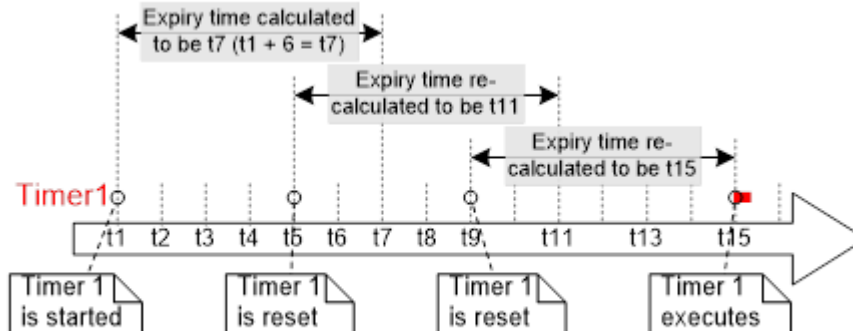
/* Toggle the LED. The rate at which the LED toggles will depend on how
often this function is called, which is determined by the period of the check
timer. The timer's period will have been reduced from 3000ms to just 200ms if
CheckTasksAreRunningWithoutError() has ever returned pdFAIL. */

ToggleLED();

}
```

Restablecimiento de un temporizador de software

Por restablecimiento de un temporizador de software se entiende reiniciar el temporizador. El tiempo de caducidad de un temporizador se vuelve a calcular para que se indique en relación con el momento en que se restableció el temporizador y no con el momento en se inició originalmente el temporizador. En la siguiente figura se muestra un temporizador cuyo periodo es de 6 y que se inicia y restablece dos veces antes de que caduque y ejecute su función de devolución de llamada.



- El temporizador 1 comienza en el momento t1. Su periodo es de 6, por lo que el momento en que se ejecute su función de devolución de llamada se calcula en un principio en t7, que equivale a 6 ciclos después del momento en que comenzó.
- El temporizador 1 se restablece antes de llegar a t7, es decir, antes de que expire y ejecute su función de devolución de llamada. El temporizador 1 se restablece en el momento t5, por lo que se vuelve a calcular el momento en que se ejecutará su función de devolución de llamada y se establece en t11, que equivale a 6 ciclos después del momento en que se restableció.
- El temporizador 1 se vuelve a restablecer antes de llegar a t11 es decir, antes de que expire y ejecute su función de devolución de llamada. El temporizador 1 se restablece en el momento t9, por lo que se vuelve a calcular el momento en que se ejecutará su función de devolución de llamada y se establece en t15, que equivale a 6 ciclos después del momento en que se restableció por última vez.
- El temporizador 1 no se vuelve a restablecer, por lo que caduca en el momento t15 y por tanto se ejecuta su función de devolución de llamada.

La función de API xTimerReset()

La función de API xTimerReset() se puede usar para restablecer un temporizador.

También puede usar xTimerReset() para iniciar un temporizador que se encuentra en el estado Latente.

Nota: No llame a xTimerReset() desde una rutina de servicio de interrupción. En su lugar, utilice la versión a prueba de interrupciones, xTimerResetFromISR().

A continuación se muestra el prototipo de la función de API xTimerReset().

```
BaseType_t xTimerReset( TimerHandle_t xTimer, TickType_t xTicksToWait );
```

En la tabla siguiente se muestran los parámetros de xTimerReset() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTimer	El identificador del temporizador de software que se restablece o inicia. El identificador se habrá devuelto desde la llamada a xTimerCreate() que se utiliza para crear el temporizador de software.
xTicksToWait	<p>xTimerChangePeriod() utiliza la cola de comandos del temporizador para enviar el comando "reset" a la tarea de demonio. xTicksToWait especifica el periodo máximo de tiempo que la tarea debe permanecer en el estado Bloqueado a la espera de que quede espacio disponible en la cola de comandos del temporizador si la cola ya está completa.</p> <p>xTimerReset() volverá inmediatamente si xTicksToWait es igual a cero y la cola de comandos del temporizador ya está llena.</p> <p>Si INCLUDE_vTaskSuspend se establece en 1 en FreeRTOSConfig.h, significa que si establece xTicksToWait portMAX_DELAY en portMAX_DELAY, la tarea que realiza la llamada se quedará indefinidamente en el estado Bloqueado (sin tiempo de espera agotado) a esperar a que quede espacio disponible en la cola de comandos del temporizador.</p>
Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <p>1. pdPASS</p> <p>Se devolverá pdPASS solo si los datos se han enviado correctamente a la cola de comandos del temporizador.</p> <p>Si se ha especificado un tiempo de bloqueo (xTicksToWait no es cero), es posible que la tarea que realizó la llamada entrara en el estado Bloqueado para esperar a que quedara espacio disponible en la cola de comandos del temporizador antes de que la función regresara,</p>

pero que se escribieran datos correctamente en la cola de comandos del temporizador antes de que el tiempo de bloqueo venciera.

2. pdFALSE

Se devolverá pdFALSE si el comando "reset" no se ha podido escribir en la cola de comandos del temporizador porque ya estaba llena.

Si se había especificado un tiempo de bloqueo (xTicksToWait no era cero) se habrá puesto la tarea de la llamada en el estado Bloqueado, a esperar a que la tarea de demonio libere espacio en la cola, pero el temporizador de bloqueo especificado ha caducado antes de que esto ocurra.

Restablecimiento de un temporizador de software (ejemplo 15)

En este ejemplo se simula el comportamiento de la iluminación de fondo de un teléfono móvil. La iluminación de fondo:

- Se activa cuando se pulsa una tecla.
- Permanece encendida siempre y cuando se vayan pulsando otras teclas durante un determinado periodo de tiempo.
- Se desactiva automáticamente si no se pulsa ninguna tecla en un determinado periodo de tiempo.

Para implementar este comportamiento se necesita un temporizador de software de una activación.

- La iluminación de fondo (simulada) se activa cuando se pulsa una tecla y se desactiva en la función de devolución de llamada del temporizador de software.
- El temporizador de software se reinicia cada vez que se pulsa una tecla.
- Por lo tanto, el periodo de tiempo durante el cual debe pulsarse una tecla para evitar que se desactive la iluminación de fondo es igual al periodo del temporizador de software. Si antes de que el temporizador caduque no se restablece el temporizador de software pulsando una tecla, la función de devolución de llamada del temporizador se ejecutará y se desactivará la iluminación de fondo.

La variable xSimulatedBacklightOn contiene el estado de la iluminación de fondo. Si se establece xSimulatedBacklightOn en pdTRUE, la iluminación de fondo está activa. Si se establece xSimulatedBacklightOn en pdFALSE, la iluminación de fondo está inactiva.

A continuación se muestra la función de devolución de llamada del temporizador de software.

```
static void prvBacklightTimerCallback( TimerHandle_t xTimer )
{
    TickType_t xTimeNow = xTaskGetTickCount();

    /* The backlight timer expired. Turn the backlight off. */

    xSimulatedBacklightOn = pdFALSE;
```

```
/* Print the time at which the backlight was turned off. */  
  
vPrintStringAndNumber("Timer expired, turning backlight OFF at time\t\t", xTimeNow );  
  
}
```

El uso de FreeRTOS permite controlar la aplicación mediante eventos. Los diseños controlados mediante eventos usan con eficiencia el tiempo. Solo se asigna tiempo en caso de que se produzca un evento. El tiempo de procesamiento no se malgasta realizando sondeos para buscar eventos que no se han producido. En el ejemplo no se ha podido aplicar un diseño de control mediante eventos porque no es práctico procesar las interrupciones de teclado cuando se usa el puerto de Windows FreeRTOS. Por este motivo ha sido preciso utilizar la técnica de sondeo mucho menos eficiente.

En el código siguiente se muestra la tarea de sondeo del teclado. Si fuera una rutina de interrupción de servicio, se utilizaría xTimerResetFromISR() en lugar de xTimerReset().

```
static void vKeyHitTask( void *pvParameters )  
{  
  
    const TickType_t xShortDelay = pdMS_TO_TICKS( 50 );  
  
    TickType_t xTimeNow;  
  
    vPrintString( "Press a key to turn the backlight on.\r\n" );  
  
    /* Ideally an application would be event-driven, and use an interrupt to process key  
    presses. It is not practical to use keyboard interrupts when using the FreeRTOS Windows  
    port, so this task is used to poll for a key press. */  
  
    for( ;; )  
    {  
  
        /* Has a key been pressed? */  
  
        if( _kbhit() != 0 )  
        {  
  
            /* A key has been pressed. Record the time. */  
  
            xTimeNow = xTaskGetTickCount();  
  
            if( xSimulatedBacklightOn == pdFALSE )  
            {  
  
                /* The backlight was off, so turn it on and print the time at which it was  
                turned on. */  
  
                xSimulatedBacklightOn = pdTRUE;  
  
                vPrintStringAndNumber( "Key pressed, turning backlight ON at time\t\t",  
xTimeNow );  
  
            }  
  
            else  
            {  
  
                /* The backlight was already on, so print a message to say the timer is  
                about to be reset and the time at which it was reset. */  
  

```

```
        vPrintStringAndNumber("Key pressed, resetting software timer at time\t\t",
xTimeNow );

    }

    /* Reset the software timer. If the backlight was previously off, then this
call will start the timer. If the backlight was previously on, then this call will restart
the timer. A real application might read key presses in an interrupt. If this function
was an interrupt service routine, then xTimerResetFromISR() must be used instead of
xTimerReset(). */

    xTimerReset( xBacklightTimer, xShortDelay );

    /* Read and discard the key that was pressed. It is not required by this simple
example. */

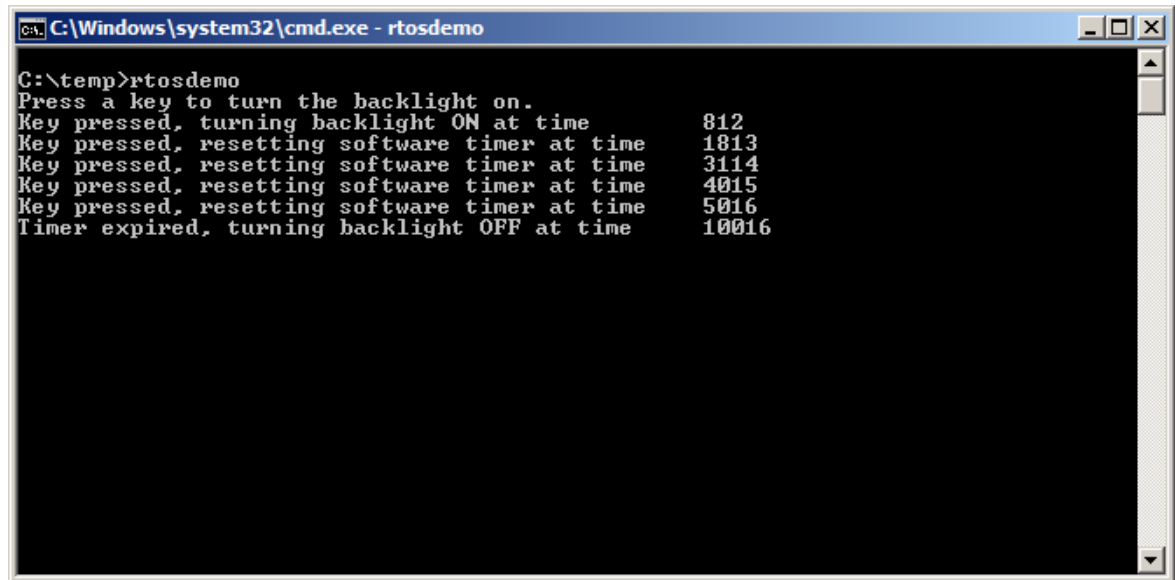
    ( void ) _getch();

}

}

}
```

El resultado se muestra aquí.



```
C:\temp>rtosdemo
Press a key to turn the backlight on.
Key pressed, turning backlight ON at time      812
Key pressed, resetting software timer at time  1813
Key pressed, resetting software timer at time  3114
Key pressed, resetting software timer at time  4015
Key pressed, resetting software timer at time  5016
Timer expired, turning backlight OFF at time  10016
```

- La primera tecla se pulsó cuando el recuento de ciclos era de 812. En ese momento se activó la iluminación de fondo y se inició el temporizador de una activación.
- Se volvieron a pulsar teclas cuando el recuento de ciclos era de 1813, 3114, 4015 y 5016. En todas esas pulsaciones de teclas el temporizador se restableció antes de caducar.
- El temporizador caducó cuando el recuento de ciclos era de 10016. En ese momento se desactivó la iluminación de fondo.

El temporizador tenía un periodo de 5000 ciclos. La iluminación de fondo se desactivó exactamente 5000 ciclos después de que se pulsara por última vez una tecla, es decir, 5000 ciclos después de que el temporizador se restableciera por última vez.

Administración de interrupciones

En esta sección se explica lo siguiente:

- Qué funciones de API de FreeRTOS se pueden utilizar desde una rutina del servicio de interrupciones
- Métodos para diferir el procesamiento de una interrupción a una tarea
- Cómo crear y utilizar semáforos binarios y semáforos de recuento
- Las diferencias entre los semáforos binarios y de recuento
- Cómo utilizar una cola para pasar datos dentro y fuera de una rutina del servicio de interrupciones
- El modelo de anidamiento de interrupciones disponible con algunos puertos de FreeRTOS

Los sistemas en tiempo real incorporados tienen que realizar acciones en respuesta a eventos que se originan en el entorno. Por ejemplo, es posible que un paquete que llega a un periférico de la Ethernet (el evento) necesite pasarse a una pila TCP/IP para procesarlo (la acción). Los sistemas no triviales tendrán que dar servicio a eventos que provienen de diversos orígenes, todos los cuales tendrán diferentes requisitos de sobrecarga de procesamiento y tiempo de respuesta. Piense en estas preguntas al elegir su estrategia de implementación del procesamiento de eventos:

1. ¿Cómo se debe detectar el evento? Las interrupciones se utilizan normalmente, pero las entradas también se pueden sondear.
2. Cuando se utilizan interrupciones, ¿qué cantidad de procesamiento debe realizarse dentro de la rutina del servicio de interrupciones y qué cantidad debe realizarse fuera? Se recomienda que cada rutina del servicio de interrupciones sea lo más corta posible.
3. ¿Cómo se comunican los eventos al código principal (no ISR) y cómo se puede estructurar este código para adaptarse mejor al procesamiento de casos potencialmente asíncronos?

FreeRTOS no impone ninguna estrategia de procesamiento de eventos al diseñador de aplicaciones, pero sí que ofrece características que le permiten implementar su estrategia seleccionada de una forma sencilla y posible de mantener.

Es importante establecer una distinción entre la prioridad de una tarea y la prioridad de una interrupción:

- Una tarea es una característica de software que no está relacionada con el hardware en el que se ejecuta FreeRTOS. La prioridad de una tarea la asigna en el software el programador de la aplicación. Un algoritmo de software (el programador) decide qué tarea estará en estado En ejecución.
- Aunque la rutina del servicio de interrupciones está escrita en el software, es una característica del hardware, porque el hardware controla qué rutina del servicio de interrupciones se ejecuta y cuándo. Las tareas solo se ejecutarán cuando no haya en ejecución ninguna rutina del servicio de interrupciones, de forma que la interrupción de menor prioridad interrumpirá a la tarea de mayor prioridad. No hay forma de que una tarea tenga prioridad sobre una rutina del servicio de interrupciones.

Todas las arquitecturas en las que se ejecuta FreeRTOS son capaces de procesar interrupciones, pero los detalles sobre la entrada de interrupciones y la asignación de prioridades a las interrupciones varían en cada arquitectura.

API a prueba de interrupciones

Con frecuencia, es necesario usar la funcionalidad que proporciona una función de API de FreeRTOS desde una rutina del servicio de interrupciones, pero muchas funciones de API de FreeRTOS realizan

acciones que no son válidas dentro de una rutina del servicio de interrupciones. La más evidente consiste en colocar la tarea que ha llamado a la función de API en el estado Bloqueado. Si una función de API se llama desde una rutina del servicio de interrupciones, eso significa que no se está llamando desde una tarea, por lo que no hay ninguna tarea de llamada que se pueda poner en el estado Bloqueado. Para resolver este problema, FreeRTOS proporciona dos versiones de algunas funciones de API: una versión que puede utilizarse desde tareas y otra que puede utilizarse desde rutinas del servicio de interrupciones. Al nombre de las funciones que están diseñadas para usarse desde ISR se le añade la palabra "FromISR".

Nota: En una ISR, no llame a una función de API de FreeRTOS que no tenga "FromISR" en su nombre.

Ventajas de utilizar una API a prueba de interrupciones independiente

Si tenemos una API independiente para usarla en interrupciones, la tarea y el código serán más eficientes y la entrada de interrupciones más sencilla. Piense en la posibilidad de usar la solución alternativa, que es proporcionar una única versión de cada función de API que se puede llamar desde una tarea y una ISR. Si es posible llamar a la misma versión de una función de API desde una tarea y una rutina del servicio de interrupciones:

- Las funciones de API necesitarían lógica adicional para determinar si se han llamado desde una tarea o una ISR. La lógica adicional introduciría nuevas rutas a través de la función, lo que haría que esas funciones fueran más largas, más complejas y más difíciles de probar.
- Algunos parámetros de funciones de API se quedarían obsoletas cuando una tarea llamara a la función, mientras que otras se quedarían obsoletas cuando una ISR llamara a la función.
- Cada puerto de FreeRTOS tendría que incluir un mecanismo para determinar el contexto de ejecución (tarea o ISR).
- Aquellas arquitecturas en las que no es fácil determinar el contexto de ejecución (tarea o ISR), necesitarían un código de entrada de interrupciones adicional, derrochador, más complejo y no estándar que permitiría que el software proporcionase el contexto de ejecución.

Desventajas de utilizar una API a prueba de interrupciones independiente

Al tener dos versiones de algunas funciones de API, las tareas y las ISR pueden ser más eficientes, pero se presenta un nuevo problema. En ocasiones, es necesario llamar a una función que no forma parte de la API de FreeRTOS, pero que la utiliza desde una tarea y una ISR.

Normalmente, este problema solo se da al integrar código de terceros, ya que es el único momento en que el diseño del software se sale del control del programador de la aplicación. Puede utilizar una de las siguientes técnicas:

1. Diferir el procesamiento de la interrupción a una tarea para que la función de API solo se llame desde el contexto de una tarea.
2. Si utiliza un puerto de FreeRTOS que admite el anidamiento de interrupciones, utilice la versión de la función de API que termina en "FromISR", porque esa versión se puede llamar desde tareas e ISR. (Lo contrario no es posible. No se debe llamar a funciones de API que no terminan en "FromISR" desde una ISR).
3. Normalmente, el código de terceros incluye una capa de abstracción de RTOS que se puede implementar para probar el contexto desde el que se llama a la función (tarea o interrupción) y, a continuación, llamar a la función de API que sea adecuada para el contexto.

Parámetro xHigherPriorityTaskWoken

Si una interrupción realiza un cambio de contexto, la tarea que se está ejecutando cuando la interrupción sale puede ser diferente a la tarea que se estaba ejecutando cuando se introdujo la interrupción. La interrupción tendrá interrumpida una tarea, pero se devuelve a otra tarea.

Algunas funciones de API de FreeRTOS pueden mover una tarea del estado Bloqueado al estado Listo. Esto ya ocurre con funciones como xQueueSendToBack(), que desbloquean una tarea si había una tarea en estado Bloqueado a la espera de que haya disponibles datos en la cola del asunto.

Si la prioridad de una tarea que desbloquea una función de API de FreeRTOS es mayor que la prioridad de la tarea que está en estado En ejecución, según la política de programación de FreeRTOS, se debe producir un cambio a la tarea de mayor prioridad. El momento en que se produce el cambio a la tarea de mayor prioridad depende de la función desde el que se llama a la función de API.

- Si la función de API se llama desde una tarea:

Si configUSE_PREEMPTION está establecido en 1 en FreeRTOSConfig.h, el cambio a la tarea de mayor prioridad se produce automáticamente en la función de API antes de que la función de API haya salido. Esto se ha demostrado en software_tier_management, donde al escribir en la cola de comandos del temporizador, se producía un cambio a la tarea de demonio de RTOS antes de que la función que escribió en la cola de comandos hubiera salido.

- Si la función de API se hubiera llamado desde una interrupción:

El cambio a una tarea de mayor prioridad no se produce de forma automática dentro de una interrupción. En su lugar, se establece una variable para informar al programador de la aplicación que debe realizarse un cambio de contexto. Las funciones de API a prueba de interrupciones (las que terminan en "FromISR") poseen un parámetro de puntero denominado pxHigherPriorityTaskWoken que se utiliza para este fin.

Si debe realizarse un cambio de contexto, la función de API a prueba de interrupciones establecerá *pxHigherPriorityTaskWoken en pdTRUE. Para saber cuándo ha sucedido esto, debe inicializarse la variable a la que apunta pxHigherPriorityTaskWoken en pdFALSE antes de utilizarla por primera vez.

Si el programador de la aplicación opta por no solicitar un cambio de contexto de la ISR, la tarea de mayor prioridad permanecerá en el estado Listo hasta que el programador vuelva a ejecutarse, que, en el peor de los casos, será durante la siguiente interrupción de ciclos.

Las funciones de API de FreeRTOS solo pueden establecer *pxHigherPriorityTaskWoken en pdTRUE. Si una ISR llama a más de una función de API de FreeRTOS, se puede pasar la misma variable como el parámetro pxHigherPriorityTaskWoken en cada llamada a la función de API. La variable debe inicializarse en pdFALSE solo antes de utilizarla por primera vez.

Existen varias razones por las que los cambios de contexto no se producen automáticamente dentro de la versión a prueba de interrupciones de una función de API:

1. Evitar cambios de contexto innecesarios

Una interrupción puede ejecutarse más de una vez antes de que sea necesario que una tarea realice algún procesamiento. Por ejemplo, supongamos que una tarea procesa una cadena que se ha recibido a través de un UART basado en interrupciones. Sería un desperdicio que la ISR de UART cambiara a la tarea cada vez que se recibe un carácter, ya que la tarea únicamente tendría que realizar algún procesamiento cuando haya recibido la cadena completa.

2. Controlar la secuencia de ejecución

Las interrupciones pueden producirse esporádicamente y en momentos impredecibles. Es posible que los usuarios avanzados de FreeRTOS deseen evitar un cambio impredecible a otra tarea en puntos

específicos de su aplicación. Para ello, puede usar el mecanismo de bloqueo del programador de FreeRTOS.

3. Portabilidad

Es el mecanismo más sencillo que se puede utilizar en todos los puertos de FreeRTOS.

4. Eficiencia

Los puertos que están pensados para arquitecturas de procesadores más pequeños solo permiten solicitar un cambio de contexto al final de una ISR. Para eliminar esa restricción, sería necesario disponer de código adicional más complejo. También permite realizar más de una llamada a una función de API de FreeRTOS en el mismo ISR sin generar más de una solicitud para un cambio de contexto dentro del mismo ISR.

5. Ejecución en la interrupción de ciclos de RTOS

Puede añadir código de aplicación a la interrupción de ciclos de RTOS. El resultado del intento de un cambio de contexto dentro de la interrupción de ciclos depende del puerto de FreeRTOS en uso. En el mejor de los casos, se producirá una llamada innecesaria al programador.

El uso del parámetro pxHigherPriorityTaskWoken es opcional. Si no es necesario, establezca pxHigherPriorityTaskWoken en NULL.

Macros portYIELD_FROM_ISR() y portEND_SWITCHING_ISR()

taskYIELD() es una macro que se puede llamar en una tarea para solicitar un cambio de contexto. portYIELD_FROM_ISR() y portEND_SWITCHING_ISR() son versiones a prueba de interrupciones de taskYIELD(). portYIELD_FROM_ISR() y portEND_SWITCHING_ISR() se utilizan de la misma forma y hacen lo mismo. Antiguamente, portEND_SWITCHING_ISR() era el nombre que se utilizaba en los puertos de FreeRTOS en los que era obligatorio utilizar un contenedor de código de ensamblado. portYIELD_FROM_ISR() era el nombre que se utilizaba en los puertos de FreeRTOS que permitían escribir todo el controlador de interrupciones en C. Algunos puertos de FreeRTOS solo incluyen una de las dos macros. Los puertos de FreeRTOS más modernos incluyen ambas macros.

```
portEND_SWITCHING_ISR( xHigherPriorityTaskWoken );
```

```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
```

El parámetro xHigherPriorityTaskWoken que se pasa a una función de API a prueba de interrupciones se puede utilizar directamente como parámetro en una llamada a portYIELD_FROM_ISR().

Si el parámetro xHigherPriorityTaskWoken de portYIELD_FROM_ISR() es pdFALSE (cero), no se solicita un cambio de contexto y la macro no tiene ningún efecto. Si el parámetro xHigherPriorityTaskWoken de portYIELD_FROM_ISR() no es pdFALSE (cero), se solicita un cambio de contexto y es posible que la tarea en estado En ejecución cambie. La interrupción siempre se devolverá a la tarea en estado En ejecución, incluso si esta ha cambiado mientras la interrupción estaba ejecutándose.

La mayoría de los puertos de FreeRTOS permiten llamar a portYIELD_FROM_ISR() desde cualquier lugar de una ISR. Algunos puertos de FreeRTOS (principalmente para arquitecturas más pequeñas) permiten llamar a portYIELD_FROM_ISR() solo al final de una ISR.

Procesamiento diferido de interrupciones

Lo más recomendable es mantener las ISR lo más cortas posibles porque:

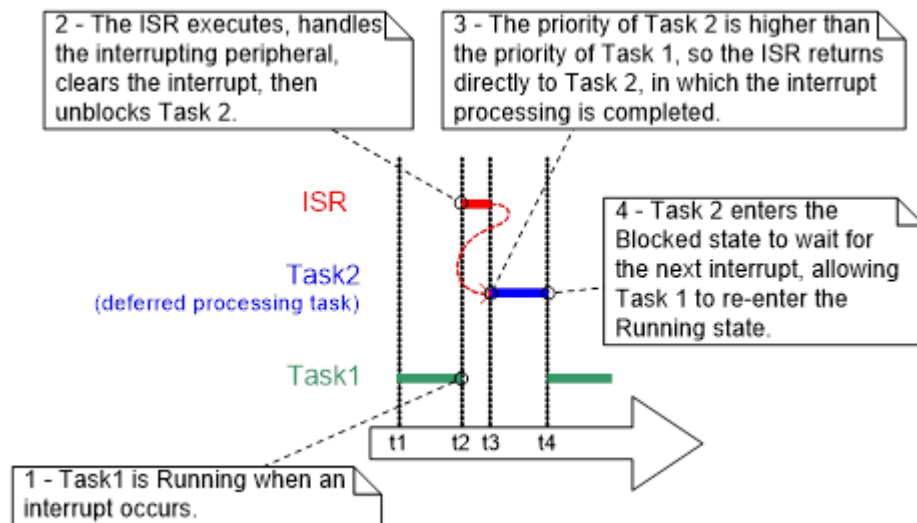
- Aunque las tareas tengan asignada una prioridad muy alta, solo se ejecutan si el hardware no está atendiendo ninguna interrupción.
- Las ISR pueden interrumpir (añadir fluctuación) al inicio y al tiempo de ejecución de una tarea.
- En función de la arquitectura en la que se esté ejecutando FreeRTOS, no sería posible aceptar nuevas interrupciones, o al menos un subconjunto de nuevas interrupciones, mientras que una ISR se está ejecutando.
- El programador de la aplicación debe tener en cuenta las consecuencias derivadas de que una tarea y una ISR accedan al mismo tiempo a recursos como variables, periféricos y búferes de memoria, por lo que debe evitarlo.
- Algunos puertos de FreeRTOS permiten el anidamiento de interrupciones, pero esto puede aumentar la complejidad y reducir la previsibilidad. Cuanto más corta sea una interrupción, menor será la probabilidad de que se anide.

Una rutina del servicio de interrupciones debe registrar la causa de la interrupción y borrarla. Con frecuencia, cualquier otro procesamiento que necesite la interrupción se puede realizar en una tarea, lo que permite que la rutina del servicio de interrupciones salga lo más rápido posible. Esto se denomina procesamiento diferido de interrupciones, porque el procesamiento que requiere la interrupción se difiere de la ISR a una tarea.

Diferir el procesamiento de interrupciones a una tarea también permite que el programador de aplicaciones priorice el procesamiento en relación con otras tareas de la aplicación y utilice todas las funciones de API de FreeRTOS.

Si la prioridad de la tarea a la que se va a diferir el procesamiento es mayor que la prioridad de cualquier otra tarea, el procesamiento se realizará de inmediato, como si se hubiera realizado en el propio ISR.

En la siguiente figura, se muestra una situación en la que la tarea 1 es una tarea de aplicación normal y la tarea 2 es la tarea a la que se difiere el procesamiento de interrupciones.



En esta figura, el procesamiento de interrupciones comienza en el momento t2 y finaliza de forma eficaz en el momento t4, pero en la ISR solo se pasa el periodo comprendido entre t2 y t3. Si no se hubiera utilizado el procesamiento diferido de interrupciones, se habría pasado en la ISR todo el periodo comprendido entre t2 y t4.

No existe ninguna regla que indique cuándo es mejor realizar todo el procesamiento que requiere una interrupción en la ISR y cuándo es mejor diferir parte del procesamiento a una tarea. Diferir el procesamiento a una tarea es más útil cuando:

- El procesamiento que requiere la interrupción no es trivial. Por ejemplo, si la interrupción solo está almacenando el resultado de una conversión de analógico a digital, es mejor realizarlo en la ISR, pero si el resultado de la conversión también debe pasarse a través de un filtro de software, podría ser mejor ejecutar el filtro en una tarea.
- Es conveniente que el procesamiento de la interrupción realice una acción que no se pueda realizar dentro de una ISR, como, por ejemplo, escribir en una consola o asignar memoria.
- El procesamiento de interrupciones no es determinista (es decir, no se sabe de antemano cuánto tiempo tardará el procesamiento).

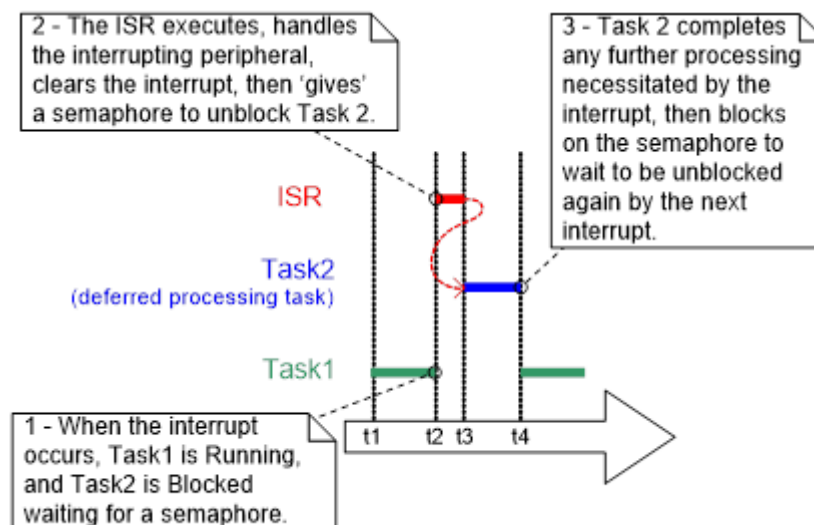
En las siguientes secciones, se describen y demuestran características de FreeRTOS que se pueden utilizar para implementar el procesamiento diferido de interrupciones.

Semáforos binarios utilizados para la sincronización

Puede utilizar la versión a prueba de interrupciones de la API de semáforos binarios para desbloquear una tarea cada vez que se produce una determinada interrupción, lo que, en la práctica, sincroniza la tarea con la interrupción. Esto permite implementar la mayor parte del procesamiento de eventos de interrupción en la tarea sincronizada, para que solo quede una parte muy rápida y corta directamente en la ISR. El semáforo binario se utiliza para diferir el procesamiento de interrupciones a una tarea. Es más eficaz utilizar una notificación directa a la tarea para desbloquear una tarea de una interrupción que va a utilizar un semáforo binario. Para obtener más información acerca de las notificaciones directas a la tarea, consulte [Notificaciones de tareas \(p. 204\)](#).

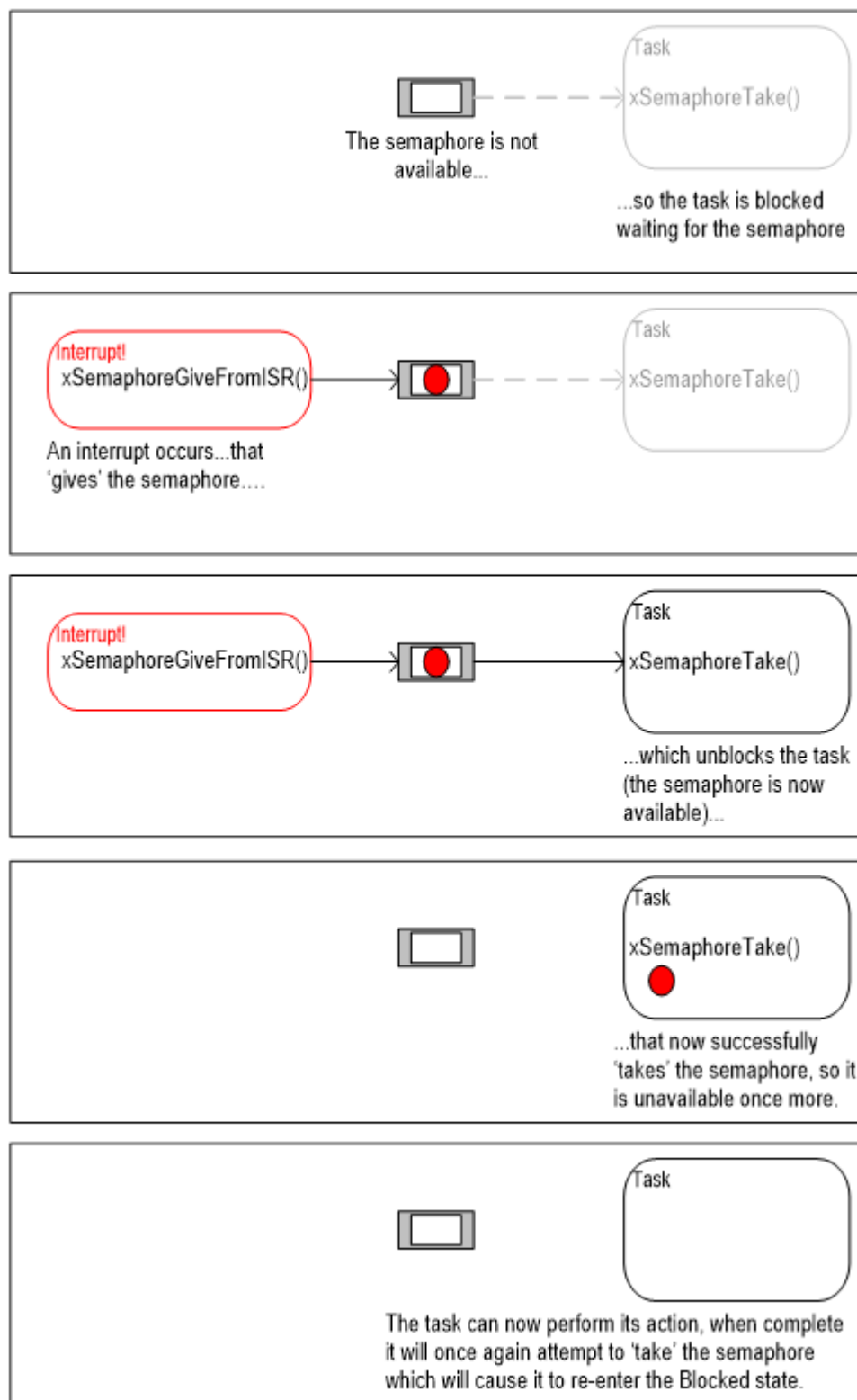
Si el tiempo del procesamiento de las interrupciones es especialmente crítico, la prioridad de la tarea de procesamiento diferido se puede configurar para garantizar que la tarea siempre tenga preferencia al resto de tareas en el sistema. Luego, se puede implementar la ISR para incluir una llamada a `portYIELD_FROM_ISR()`, lo que garantiza que la ISR vuelva directamente a la tarea cuyo procesamiento de interrupciones se está difiriendo. Esto garantiza que todo el procesamiento de eventos se ejecute de forma contigua (sin un descanso) y a tiempo, como si todo se hubiera implementado en el propio ISR.

En la siguiente figura, se utiliza el ejemplo de la figura anterior, pero se ha cambiado el texto para describir cómo se puede controlar la ejecución de la tarea de procesamiento diferido mediante un semáforo.



La tarea de procesamiento diferido utiliza una llamada "take" de bloqueo a un semáforo como medio para entrar en el estado Bloqueado a la espera de que se produzca el evento. Cuando se produce el evento, la ISR utiliza una operación "give" en el mismo semáforo para desbloquear la tarea, de modo que el procesamiento de eventos necesario pueda continuar.

Tomar un semáforo y dar un semáforo son conceptos que tienen distintos significados, en función de cada situación. En este ejemplo de sincronización de interrupciones, el semáforo binario puede considerarse conceptualmente como una cola con una longitud de uno. La cola puede contener un máximo de un elemento en un momento dado, por lo que siempre está llena o vacía (binario). Al llamar a `xSemaphoreTake()`, la tarea a la que se difiere el procesamiento de interrupciones intenta leer de la cola con un tiempo de bloqueo, lo que hace que la tarea pase al estado Bloqueado si la cola está vacía. Cuando se produce el evento, la ISR utiliza la función `xSemaphoreGiveFromISR()` para colocar un token (el semáforo) en la cola, lo que hace que la cola se llene. Esto hace que la tarea salga del estado Bloqueado y elimine el token, dejando la cola vacía una vez más. Cuando la tarea ha completado su procesamiento, intenta leer otra vez la cola y, al descubrir que está vacía, vuelve a pasar al estado Bloqueado a la espera del siguiente evento. La secuencia se muestra aquí.



Esta figura muestra cómo la interrupción da el semáforo, a pesar de que no lo ha tomado previamente y cómo la tarea toma el semáforo, pero nunca lo vuelve a dar. Esta es la razón por la que se dice que este caso es algo similar a escribir y leer de una cola. Suele causar confusión, ya que no sigue las mismas

reglas que otros casos de uso de semáforos, donde una tarea que toma un semáforo siempre debe volver a darlo, como en los ejemplos que se describen en la sección [Administración de recursos](#) (p. 161).

Función de API xSemaphoreCreateBinary()

FreeRTOS V9.0.0 también incluye la función xSemaphoreCreateBinaryStatic(), que asigna la memoria necesaria para crear un semáforo binario de forma estática durante la compilación. Los controladores de los distintos tipos de semáforos de FreeRTOS se almacenan en una variable de tipo SemaphoreHandle_t.

Para poder utilizar un semáforo, antes hay que crearlo. Para crear un semáforo binario, utilice la función de API xSemaphoreCreateBinary().

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

En la siguiente tabla se muestra el valor de retorno de xSemaphoreCreateBinary().

Nombre del parámetro	Descripción
Valor devuelto	<p>Si se devuelve NULL, el semáforo no se puede crear porque no hay suficiente memoria disponible en el montón para que FreeRTOS asigne las estructuras de datos del semáforo.</p> <p>Si se devuelve valor NULL, eso significa que el semáforo se ha creado correctamente. El valor devuelto debe almacenarse como el controlador en el semáforo creado.</p>

Función de API xSemaphoreTake()

Tomar un semáforo significa obtener o recibir el semáforo. Solo es posible tomar el semáforo si está disponible.

Todos los tipos de semáforos de FreeRTOS, excepto los mutex recursivos, se pueden tomar con la función xSemaphoreTake(). xSemaphoreTake() no debe utilizarse desde una rutina del servicio de interrupciones.

```
BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

En la siguiente tabla, se muestran los parámetros de xSemaphoreTake() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xSemaphore	<p>El semáforo que se está tomando.</p> <p>Una variable de tipo SemaphoreHandle_t hace referencia al semáforo. Se debe crear explícitamente antes de usarla.</p>
xTicksToWait	<p>El tiempo máximo que la tarea debe permanecer en el estado Bloqueado para esperar al semáforo si no está disponible en ese momento.</p>

	<p>Si xTicksToWait es cero, xSemaphoreTake() se devuelve de inmediato si el semáforo no está disponible.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir a ciclos un tiempo especificado en milisegundos.</p> <p>Al establecer xTicksToWait en portMAX_DELAY, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando INCLUDE_vTaskSuspend esté establecido en 1 en FreeRTOSConfig.h.</p>
Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <ol style="list-style-type: none">1. pdPASS <p>pdPASS solo se devuelve si la llamada a xSemaphoreTake() se ha realizado correctamente al obtener el semáforo.</p><p>Si se hubiera especificado un tiempo de bloqueo (xTicksToWait no era cero), es posible que la tarea que realiza la llamada se hubiera puesto en el estado Bloqueado para esperar al semáforo si no estaba disponible en ese momento, pero el semáforo dejó de estar disponible antes de que el tiempo de bloqueo venciera.</p>2. pdFALSE <p>El semáforo no está disponible.</p><p>Si se hubiera especificado un tiempo de bloqueo (xTicksToWait no era cero), la tarea que realiza la llamada se habría puesto en el estado Bloqueado para esperar a que el semáforo estuviera disponible, pero el temporizador de bloqueo se ha agotado antes de que esto ocurriera.</p>

Función de API xSemaphoreGiveFromISR()

Se pueden dar semáforos binarios y de recuento con la función xSemaphoreGiveFromISR().

xSemaphoreGiveFromISR() es la versión a prueba de interrupciones de xSemaphoreGive(), por lo que tiene el parámetro pxHigherPriorityTaskWoken.

```
BaseType_t xSemaphoreGiveFromISR( SemaphoreHandle_t xSemaphore, BaseType_t  
    *pxHigherPriorityTaskWoken );
```

A continuación, se muestran los parámetros de xSemaphoreGiveFromISR() y el valor de retorno.

xSemaphore

El semáforo que se está dando. Una variable de tipo SemaphoreHandle_t hace referencia al semáforo y debe crearse de forma explícita antes de usarse.

pxHigherPriorityTaskWoken

Es posible que un único semáforo tenga una o varias tareas bloqueadas a la espera de que el semáforo esté disponible. Al llamar a xSemaphoreGiveFromISR(), puede hacer que el semáforo esté disponible y, por tanto, que una tarea que estuviera esperando al semáforo salga del estado Bloqueado. Si al llamar a xSemaphoreGiveFromISR(), una tarea sale del estado Bloqueado y la prioridad de la tarea desbloqueada es mayor que la de la tarea que se está ejecutando en ese momento (la tarea que se interrumpió), xSemaphoreGiveFromISR() establecerá internamente *pxHigherPriorityTaskWoken en pdTRUE. Si xSemaphoreGiveFromISR() establece este valor en pdTRUE, normalmente debe efectuarse un cambio de contexto antes de salir de la interrupción. De este modo, se asegurará de que la interrupción vuelve directamente a la tarea con el estado Listo de máxima prioridad.

Hay dos valores de retorno posibles:

pdPASS

Se devuelve solo si la llamada a xSemaphoreGiveFromISR() se realiza correctamente.

pdFAIL

Si un semáforo ya está disponible, no es posible darlo y xSemaphoreGiveFromISR() devuelve pdFAIL.

Uso de un semáforo binario para sincronizar una tarea con una interrupción (Ejemplo 16)

En este ejemplo, se utiliza un semáforo binario para desbloquear una tarea desde una rutina del servicio de interrupciones, lo que en la práctica sincroniza la tarea con la interrupción.

Se utiliza una tarea periódica simple para generar una interrupción de software cada 500 milisegundos. La interrupción de software se utiliza por comodidad debido a la complejidad que supone enlazarse a una interrupción real en algunos entornos de destino.

El código siguiente muestra la implementación de la tarea periódica. La tarea imprime una cadena antes y después de que se genere la interrupción. Puede ver la secuencia de la ejecución en la salida.

```
/* The number of the software interrupt used in this example. The code shown is from the
   Windows project, where numbers 0 to 2 are used by the FreeRTOS Windows port itself, so 3
   is the first number available to the application. */

#define mainINTERRUPT_NUMBER 3

static void vPeriodicTask( void *pvParameters )
{
    const TickType_t xDelay500ms = pdMS_TO_TICKS( 500UL );

    /* As per most tasks, this task is implemented within an infinite loop.*/

    for( ;; )
    {
```

```
/* Block until it is time to generate the software interrupt again. */  
  
vTaskDelay( xDelay500ms );  
  
/* Generate the interrupt, printing a message both before and after the interrupt  
has been generated, so the sequence of execution is evident from the output. The syntax  
used to generate a software interrupt is dependent on the FreeRTOS port being used.  
The syntax used below can only be used with the FreeRTOS Windows port, in which such  
interrupts are only simulated.*/  
  
vPrintString( "Periodic task - About to generate an interrupt.\r\n" );  
  
vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );  
  
vPrintString( "Periodic task - Interrupt generated.\r\n\r\n\r\n\r\n" );  
};  
}
```

El código siguiente muestra la implementación de la tarea a la que se difiere el procesamiento de la interrupción, es decir, la tarea que se sincroniza con la interrupción del software mediante el uso de un semáforo binario. Una vez más, se imprime una cadena en cada iteración de la tarea. Verá la secuencia en la que la tarea y la interrupción se ejecutan en la salida.

Aunque este código sea suficiente como ejemplo de dónde genera el software las interrupciones, no es adecuado en los casos en los que las interrupciones las generen periféricos de hardware. Hay que cambiar la estructura del código para adecuarla a las interrupciones generadas por hardware.

```
static void vHandlerTask( void *pvParameters )  
{  
  
    /* As per most tasks, this task is implemented within an infinite loop.*/  
  
    for( ;; )  
    {  
  
        /* Use the semaphore to wait for the event. The semaphore was created before  
the scheduler was started, so before this task ran for the first time. The task blocks  
indefinitely, meaning this function call will only return once the semaphore has been  
successfully obtained so there is no need to check the value returned by xSemaphoreTake().  
*/  
  
        xSemaphoreTake( xBinarySemaphore, portMAX_DELAY );  
  
        /* To get here the event must have occurred. Process the event (in this case, just  
print out a message). */  
  
        vPrintString( "Handler task - Processing event.\r\n" );  
  
    }  
}
```

En el siguiente código se muestra la ISR. No hace mucho más que dar el semáforo para desbloquear la tarea en la que se ha diferido el procesamiento de la interrupción.

Se utiliza la variable `xHigherPriorityTaskWoken`. Se establece en `pdFALSE` antes de llamar a `xSemaphoreGiveFromISR()` y, a continuación, se utiliza como parámetro cuando se llama

a `portYIELD_FROM_ISR()`. Se solicitará un conmutador de contexto dentro de la macro `portYIELD_FROM_ISR()` si `xHigherPriorityTaskWoken` es igual a `pdTRUE`.

El prototipo de la ISR y la macro que se ha llamado para forzar un cambio de contexto son correctos para el puerto de Windows de FreeRTOS, pero es posible que no lo sean para otros puertos de FreeRTOS. Para encontrar la sintaxis necesaria para el puerto que está utilizando, consulte las páginas específicas de los puertos en la documentación del sitio web FreeRTOS.org y los ejemplos que se proporcionan en la descarga de FreeRTOS.

A diferencia de la mayoría de arquitecturas en las que se ejecuta FreeRTOS, el puerto de Windows de FreeRTOS necesita una ISR para devolver un valor. La implementación de la macro `portYIELD_FROM_ISR()` que se proporciona con el puerto de Windows incluye la instrucción de retorno, por lo que este código no muestra se devuelva un valor de forma explícita.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is
    required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Give the semaphore to unblock the task, passing in the address of
    xHigherPriorityTaskWoken as the interrupt-safe API function's pxHigherPriorityTaskWoken
    parameter. */

    xSemaphoreGiveFromISR( xBinarySemaphore, &xHigherPriorityTaskWoken );

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
    xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR(), then calling
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS
    ports, the Windows port requires the ISR to return a value. The return statement is inside
    the Windows version of portYIELD_FROM_ISR(). */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

La función `main()` crea el semáforo binario y las tareas, instala el controlador de interrupciones e inicia el programador. En el siguiente código, se muestra la implementación.

La sintaxis de la función que se llama para instalar un controlador de interrupciones es específica del puerto de Windows de FreeRTOS. Podría ser diferente para otros puertos de FreeRTOS. Para encontrar la sintaxis necesaria para el puerto que está utilizando, consulte la documentación específica de los puertos del sitio web FreeRTOS.org y los ejemplos que se proporcionan en la descarga de FreeRTOS.

```
int main( void )
{
    /* Before a semaphore is used, it must be explicitly created. In this example, a binary
    semaphore is created. */

    xBinarySemaphore = xSemaphoreCreateBinary();

    /* Check that the semaphore was created successfully. */
```



```
if( xBinarySemaphore != NULL )
{
    /* Create the 'handler' task, which is the task to which interrupt processing is
    deferred. This is the task that will be synchronized with the interrupt. The handler task
    is created with a high priority to ensure it runs immediately after the interrupt exits.
    In this case, a priority of 3 is chosen. */

    xTaskCreate( vHandlerTask, "Handler", 1000, NULL, 3, NULL );

    /* Create the task that will periodically generate a software interrupt. This is
    created with a priority below the handler task to ensure it will get preempted each time
    the handler task exits the Blocked state. */

    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, 1, NULL );

    /* Install the handler for the software interrupt. The syntax required to do this
    is depends on the FreeRTOS port being used. The syntax shown here can only be used with
    the FreeRTOS Windows port, where such interrupts are only simulated. */

    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

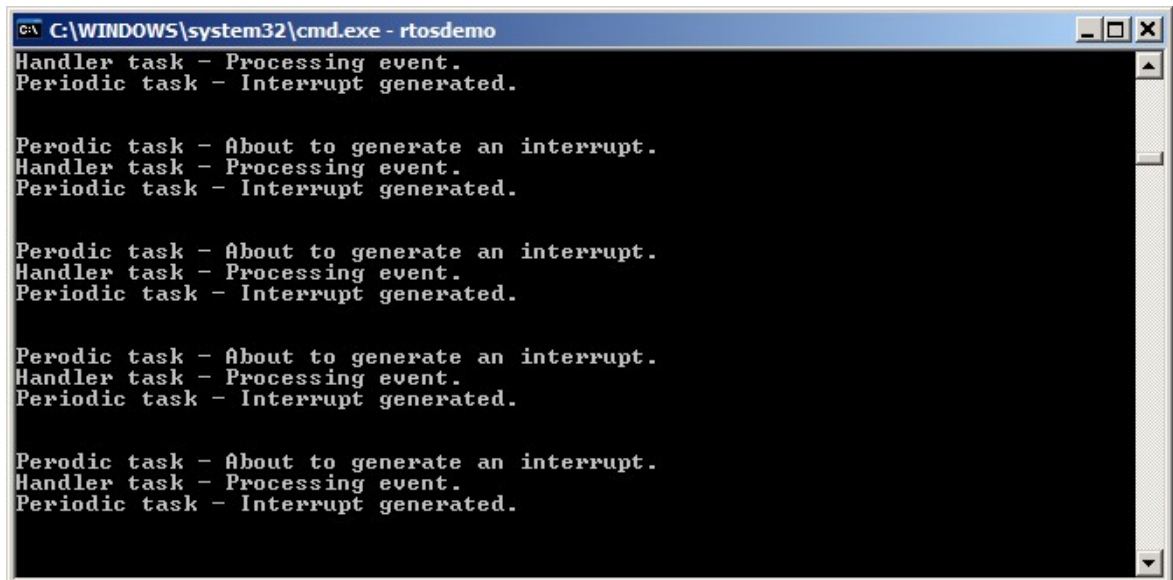
    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();
}

/* As normal, the following line should never be reached. */

for( ;; );
}
```

El código produce la siguiente salida. Como era de esperar, vHandlerTask() entra en el estado En ejecución en cuanto se genera la interrupción, por lo que la salida de la tarea divide la salida que produce la tarea periódica.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Periodic task - Interrupt generated.

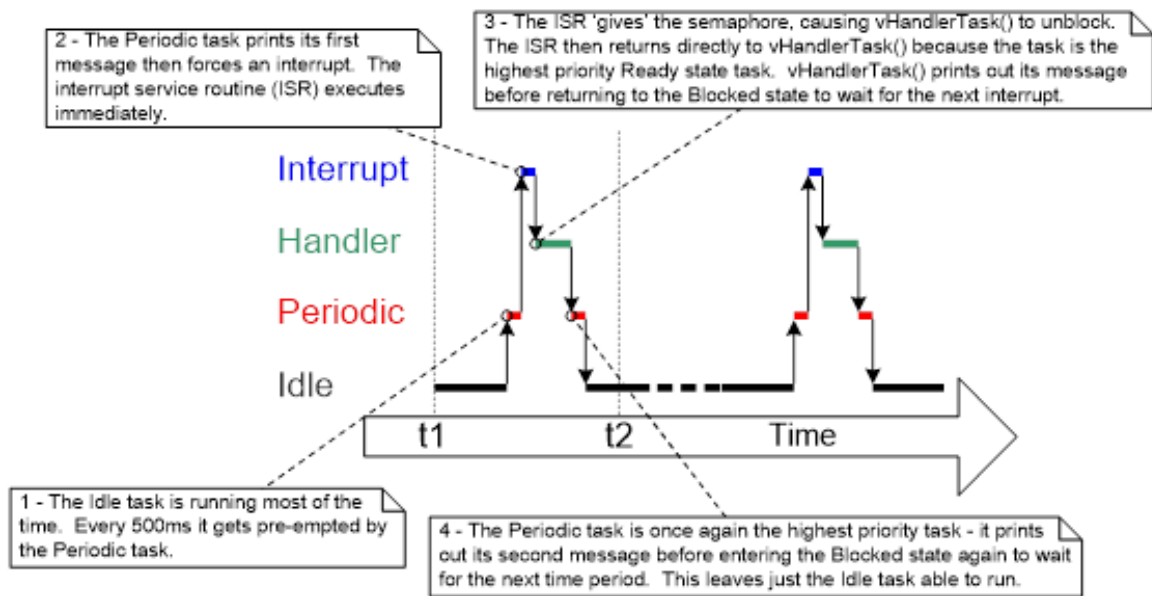
Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Periodic task - Interrupt generated.
```

Esta es la secuencia de ejecución.



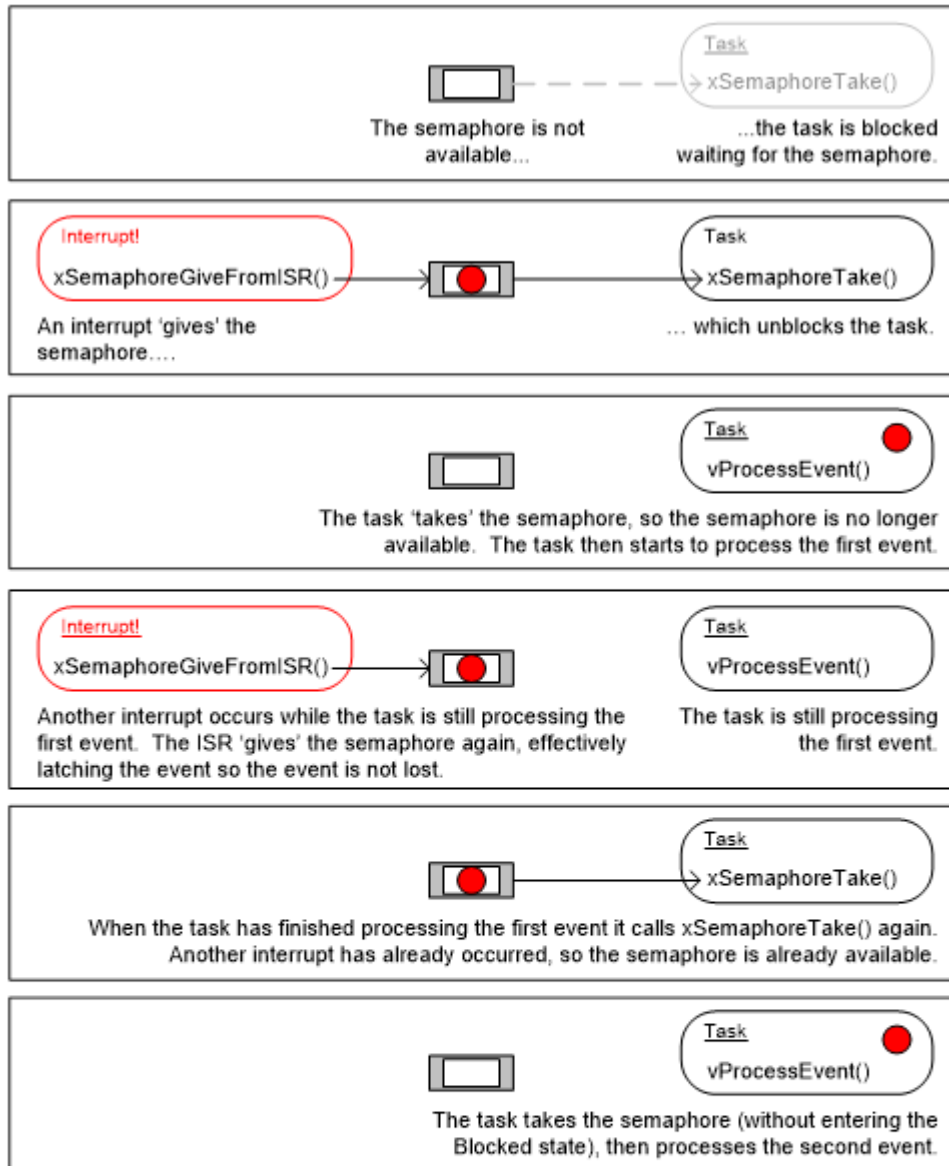
Mejora de la implementación de la tarea utilizada en el Ejemplo 16

En el Ejemplo 16, se utiliza un semáforo binario para sincronizar una tarea con una interrupción. Esta es la secuencia de ejecución:

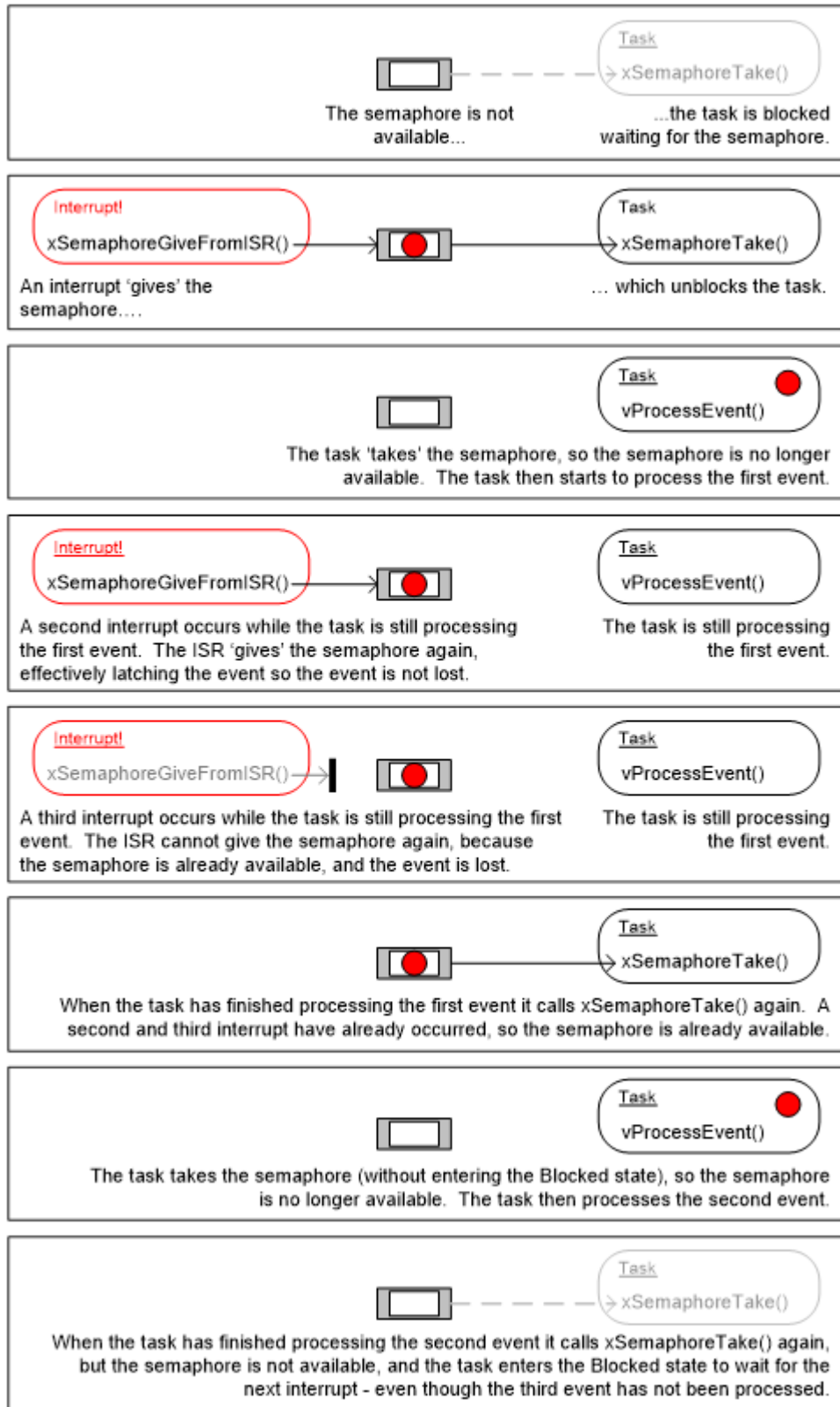
1. Se ha produce la interrupción.
2. la ISR se ejecuta y da el semáforo para desbloquear la tarea.
3. La tarea se ejecuta inmediatamente después de la ISR y toma el semáforo.
4. La tarea procesa el evento y, a continuación, intenta tomar el semáforo de nuevo y pasa al estado Bloqueado porque el semáforo todavía no está disponible (aún no se ha producido otra interrupción).

La estructura de la tarea que se utiliza en el Ejemplo 16 solo es adecuada si se producen interrupciones a una frecuencia relativamente baja. Piense qué pasaría si se hubiera producido una segunda y, a continuación, una tercera interrupción antes de que la tarea hubiera terminado de procesar la primera interrupción.

Cuando se ejecutara el segundo ISR, el semáforo estaría vacío, por lo que la ISR daría el semáforo y la tarea procesaría el segundo evento inmediatamente después de que se hubiera terminado de procesar el primer evento. Aquí se muestra esa situación.



Cuando se ejecutase el tercer ISR, el semáforo ya estaría disponible, lo que impediría que la ISR diera de nuevo el semáforo, por lo que la tarea no sabría que se ha producido el tercer evento. Aquí se muestra esa situación.



La tarea de control de interrupciones diferidas, `static void vHandlerTask(void *pvParameters)`, se estructura de forma que solo procesa un evento entre cada llamada a `xSemaphoreTake()`. Eso era suficiente para el Ejemplo 16, porque las interrupciones que generaron los eventos se activaron mediante software y se produjeron en un momento predecible. En aplicaciones reales, el hardware es el que genera interrupciones y estas se producen en momentos impredecibles. Por lo tanto, para minimizar la probabilidad de que se pierda una interrupción, la tarea de control de interrupciones diferidas debe estar estructurada para procesar todos los eventos que ya están disponibles entre cada llamada a `xSemaphoreTake()`. El código siguiente muestra cómo se podría estructurar un controlador de interrupciones diferidas para un UART. Se supone que el UART genera un interruptor de recepción cada vez que se recibe un carácter y que el UART coloca los caracteres que recibe en un FIFO de hardware (búfer de hardware).

La tarea de control de interrupciones diferidas que se utiliza en el Ejemplo 16 tenía otro problema. No utilizó un tiempo de espera al llamar a `xSemaphoreTake()`. En lugar de ello, la tarea pasó `portMAX_DELAY` como parámetro `xTicksToWait` de `xSemaphoreTake()`, lo que se traduce en que la tarea debe esperar de forma indefinida (sin un tiempo de espera) a que el semáforo esté disponible. En el código de ejemplo, suelen utilizarse tiempos de espera indefinidos porque se simplifica la estructura del ejemplo y hace que sea más fácil de entender. Sin embargo, en aplicaciones reales, no es conveniente utilizar tiempos de espera indefinidos, ya que dificultan la recuperación de un error. Supongamos que una tarea está esperando a que una interrupción dé un semáforo, pero un estado de error en el hardware impide que se genere la interrupción.

- Si la tarea está esperando sin un tiempo de espera, no conocerá la existencia del estado de error y esperará para siempre.
- Si la tarea está esperando con un tiempo de espera, `xSemaphoreTake()` devolverá `pdFAIL` cuando ese tiempo de espera se agote y la tarea puede detectar y borrar el error la próxima vez que se ejecute. Esta situación también se demuestra aquí.

```
static void vUARTReceiveHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime holds the maximum time expected between two interrupts. */
    const TickType_t xMaxExpectedBlockTime = pdMS_TO_TICKS( 500 );

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* The semaphore is given by the UART's receive (Rx) interrupt. Wait a maximum of
        xMaxExpectedBlockTime ticks for the next interrupt. */

        if( xSemaphoreTake( xBinarySemaphore, xMaxExpectedBlockTime ) == pdPASS )
        {
            /* The semaphore was obtained. Process ALL pending Rx events before calling
            xSemaphoreTake() again. Each Rx event will have placed a character in the UART's receive
            FIFO, and UART_RxCount() is assumed to return the number of characters in the FIFO. */

            while( UART_RxCount() > 0 )
            {
                /* UART_ProcessNextRxEvent() is assumed to process one Rx character,
                reducing the number of characters in the FIFO by 1. */
            }
        }
    }
}
```

```
        UART_ProcessNextRxEvent();

    }

    /* No more Rx events are pending (there are no more characters in the FIFO),
    so loop back and call xSemaphoreTake() to wait for the next interrupt. Any interrupts
    occurring between this point in the code and the call to xSemaphoreTake() will be latched
    in the semaphore, so will not be lost. */

    }

    else

    {

        /* An event was not received within the expected time. Check for and, if
        necessary, clear any error conditions in the UART that might be preventing the UART from
        generating any more interrupts. */

        UART_ClearErrors();

    }

}

}
```

Semáforos de recuento

De la misma forma que los semáforos binarios son como colas con una longitud de uno, los semáforos de recuento son como colas que tienen una longitud de más de uno. A las tareas no les interesan los datos que se almacenan en la cola, solo el número de elementos que hay en la cola. Para que los semáforos de recuento estén disponibles, en FreeRTOSConfig.h, establezca configUSE_COUNTING_SEMAPHORES en 1.

Cada vez que se dé un semáforo de recuento, se utiliza otro espacio en su cola. El número de elementos de la cola es el valor de "recuento" del semáforo.

Los semáforos de recuento suelen usarse para:

1. Eventos de recuento

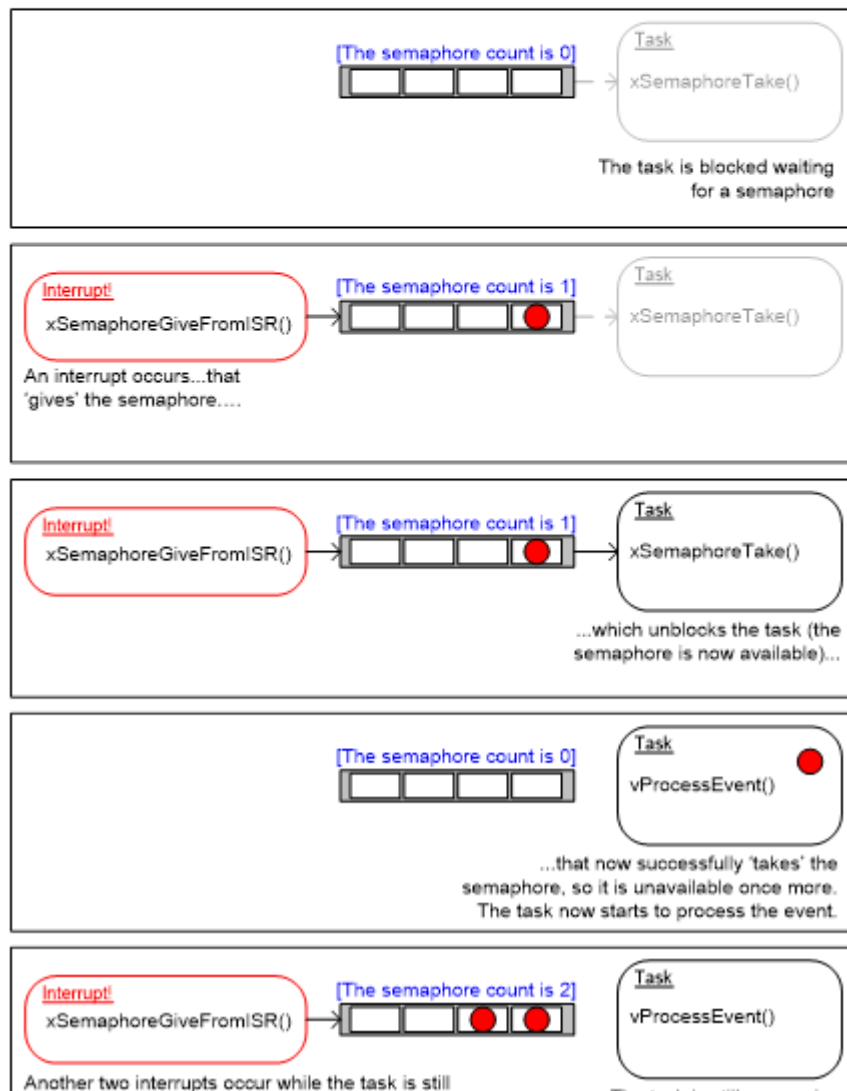
En este caso, un controlador de eventos dará un semáforo cada vez que se produzca un evento, lo que hace que el valor de recuento del semáforo se incremente en cada ocasión. Una tarea tomará un semáforo cada vez que procese un evento, lo que hace que el valor de recuento del semáforo se reduzca en cada ocasión. El valor del recuento es la diferencia entre el número de eventos que se han producido y el número de eventos que se han procesado. En la siguiente figura se muestra este mecanismo.

Los semáforos de recuento que se utilizan para contar eventos se crean con un valor de recuento inicial de cero.

2. Administración de recursos

En esta situación, el valor de recuento indica el número de recursos disponibles. Para obtener el control de un recurso, una tarea debe obtener primero un semáforo para reducir el valor de recuento de semáforos. Cuando el valor de recuento alcanza cero, eso significa que no hay recursos libres. Cuando una tarea termina con el recurso, devuelve el semáforo y se incrementa el valor de recuento de semáforos.

Los semáforos de recuento que se utilizan para administrar los recursos se crean de forma que su valor de recuento inicial sea igual al número de recursos que están disponibles.



Función de API xSemaphoreCreateCounting()

FreeRTOS V9.0.0 también incluye la función xSemaphoreCreateCountingStatic(), que asigna la memoria necesaria para crear un semáforo de recuento de forma estática durante la compilación. Los controladores de todos los distintos tipos de semáforos de FreeRTOS se almacenan en una variable de tipo SemaphoreHandle_t.

Para poder utilizar un semáforo, antes hay que crearlo. Para crear un semáforo de recuento, utilice la función de API xSemaphoreCreateCounting().

En la siguiente tabla, se muestran los parámetros de xSemaphoreCreateCounting() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
uxMaxCount	<p>El valor máximo hasta el que contará el semáforo. Para continuar con la analogía de la cola, el valor de uxMaxCount es, efectivamente, la longitud de la cola.</p> <p>Cuando el semáforo se utiliza para contar o bloquear eventos, uxMaxCount es el número máximo de eventos que se pueden bloquear.</p> <p>Cuando el semáforo se utiliza para administrar el acceso a una recopilación de recursos, uxMaxCount debe establecerse en el número total de recursos que están disponibles.</p>
uxInitialCount	<p>El valor de recuento inicial del semáforo después de crearlo.</p> <p>Cuando el semáforo se utiliza para contar o bloquear eventos, uxInitialCount debe establecerse en cero, ya que, cuando se crea el semáforo, se supone que aún no se ha producido ningún evento.</p> <p>Cuando el semáforo se utiliza para administrar el acceso a una recopilación de recursos, uxInitialCount debe establecerse en el mismo valor que uxMaxCount porque, cuando se crea el semáforo, se supone que todos los recursos están disponibles.</p>
Valor devuelto	<p>Si se devuelve NULL, el semáforo no se puede crear porque no hay suficiente memoria disponible en el montón para que FreeRTOS asigne las estructuras de datos del semáforo. Para obtener más información, consulte Administración de memoria en montón (p. 14).</p> <p>Si se devuelve valor NULL, eso significa que el semáforo se ha creado correctamente. El valor devuelto debe almacenarse como el controlador en el semáforo creado.</p>

Uso de un semáforo de recuento para sincronizar una tarea con una interrupción (Ejemplo 17)

El Ejemplo 17 mejora la implementación del Ejemplo 16 utilizando un semáforo de recuento en lugar del semáforo binario. La función main() se ha cambiado para incluir una llamada a xSemaphoreCreateCounting() en lugar de la llamada a xSemaphoreCreateBinary().

En el siguiente código, se muestra la nueva llamada a la API.

```
/* Before a semaphore is used it must be explicitly created. In this example, a counting semaphore is created. The semaphore is created to have a maximum count value of 10, and an initial count value of 0. */  
  
xCountingSemaphore = xSemaphoreCreateCounting( 10, 0 );
```

Para simular que se producen varios eventos a una frecuencia alta, se cambia la rutina del servicio de interrupciones para dar el semáforo más de una vez por interrupción. Cada evento se bloquea en el valor de recuento del semáforo.

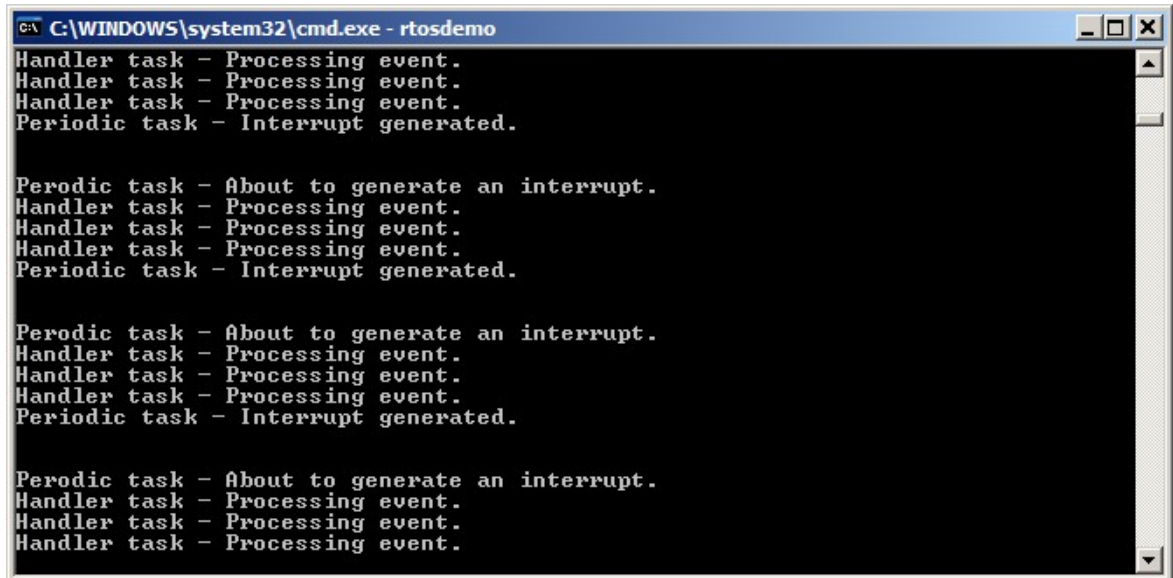
El código siguiente muestra la rutina del servicio de interrupciones modificada.

```
static uint32_t ulExampleInterruptHandler( void )  
{  
  
    BaseType_t xHigherPriorityTaskWoken;  
  
    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it will get set to pdTRUE inside the interrupt-safe API function if a context switch is required. */  
  
    xHigherPriorityTaskWoken = pdFALSE;  
  
    /* Give the semaphore multiple times. The first will unblock the deferred interrupt handling task. The following gives are to demonstrate that the semaphore latches the events to allow the task to which interrupts are deferred to process them in turn, without events getting lost. This simulates multiple interrupts being received by the processor, even though in this case the events are simulated within a single interrupt occurrence. */  
  
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );  
  
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );  
  
    xSemaphoreGiveFromISR( xCountingSemaphore, &xHigherPriorityTaskWoken );  
  
    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken was set to pdTRUE inside xSemaphoreGiveFromISR(), then calling portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS ports, the Windows port requires the ISR to return a value. The return statement is inside the Windows version of portYIELD_FROM_ISR(). */  
  
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}
```

Todas las demás funciones son las mismas que las que se utilizan en el Ejemplo 16.

Esta es la salida que se genera cuando se ejecuta el código. Como verá, la tarea a la que se refiere el control de interrupciones procesa los tres eventos (simulados) cada vez que se genera una interrupción.

Los eventos se bloquean en el valor de recuento del semáforo, lo que permite que la tarea los procese por turnos.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

Diferir el trabajo en la tarea de demonio de RTOS

Los ejemplos de control de interrupciones diferidas que hemos visto hasta ahora requerían que el programador de la aplicación creara una tarea para cada interrupción que utilizara la técnica de procesamiento diferido. También es posible utilizar la función de API `xTimerPendFunctionCallFromISR()` para diferir el procesamiento de interrupciones a la tarea de demonio de RTOS, lo que elimina la necesidad de crear una tarea independiente para cada interrupción. Diferir el procesamiento de interrupciones a la tarea de demonio se denomina procesamiento de interrupciones diferido centralizado.

La tarea de demonio se denominaba originalmente tarea de servicio del temporizador, porque solo se usaba para ejecutar funciones de devolución de llamadas del temporizador de software. Por este motivo, se implementa `xTimerPendFunctionCall()` en `timers.c` y, de acuerdo con la convención de usar como prefijo en el nombre de la función el nombre del archivo en la que se ha implementado la función, el nombre de la función tiene el prefijo "Timer".

En la sección `software_tier_management`, se describe cómo las funciones de API de FreeRTOS relacionadas con temporizadores de software envían comandos a la tarea de demonio en la cola de comandos del temporizador. Las funciones de API `xTimerPendFunctionCall()` y `xTimerPendFunctionCallFromISR()` utilizan la misma cola de comandos del temporizador para enviar un comando "execute function" a la tarea de demonio. Luego, la función que se envía a la tarea de demonio se ejecuta en el contexto de la tarea de demonio.

Ventajas del procesamiento de interrupciones diferido centralizado:

- Menor uso de recursos

No es necesario crear una tarea independiente para cada interrupción diferida.

- Modelo de usuario simplificado

La función de control de interrupciones diferidas es una función estándar de C.

Desventajas del procesamiento de interrupciones diferido centralizado:

- Menos flexibilidad

No se puede establecer la prioridad de cada tarea de control de interrupciones diferidas por separado. Cada función de control de interrupciones diferidas se ejecuta con la prioridad de la tarea de demonio. La prioridad de la tarea de demonio se establece por medio de la constante de configuración en tiempo de compilación configTIMER_TASK_PRIORITY en FreeRTOSConfig.h.

- Menos determinismo

xTimerPendFunctionCallFromISR() envía un comando a la parte posterior de la cola de comandos del temporizador. La tarea de demonio procesa los comandos que ya estaban en la cola del temporizador antes que el comando "execute function" que xTimerPendFunctionCallFromISR() ha enviado a la cola.

Cada interrupción tiene restricciones de tiempo diferentes, por lo que es común utilizar ambos métodos en la misma aplicación.

Función de API xTimerPendFunctionCallFromISR()

xTimerPendFunctionCallFromISR() es la versión a prueba de interrupciones de xTimerPendFunctionCall(). Ambas funciones de API permiten que la tarea de demonio de RTOS ejecute una función proporcionada por el programador de la aplicación en el contexto de esa tarea de demonio. La función que se va a ejecutar y el valor de los parámetros de entrada se envían a la tarea de demonio en la cola de comandos del temporizador. El momento en que se ejecuta la función depende de la prioridad de la tarea de demonio con respecto a otras tareas de la aplicación.

Aquí se muestra el prototipo de la función de API xTimerPendFunctionCallFromISR().

```
BaseType_t xTimerPendFunctionCallFromISR( PendedFunction_t xFunctionToPend, void
*pvParameter1, uint32_t ulParameter2, BaseType_t *pxHigherPriorityTaskWoken );
```

Aquí se muestra el prototipo al que debe ajustarse una función que se pasa en el parámetro xFunctionToPend de xTimerPendFunctionCallFromISR().

```
void vPendableFunction( void *pvParameter1, uint32_t ulParameter2 );
```

En la siguiente tabla, se muestran los parámetros de xTimerPendFunctionCallFromISR() y el valor de retorno.

xFunctionToPend

Puntero a la función que se ejecutará en la tarea de demonio (en la práctica, solo el nombre de la función). El prototipo de la función debe ser el mismo que se muestra en el código.

pvParameter1

Valor que se pasará a la función que ejecuta la tarea de demonio como parámetro pvParameter1 de la función. El parámetro tiene un tipo * nulo para que pueda utilizarse para pasar cualquier tipo de datos. Por ejemplo, los tipos enteros se pueden convertir directamente en un * nulo. De forma alternativa, el * nulo puede utilizarse para apuntar a una estructura.

ulParameter2

El valor que se pasará a la función que ejecuta la tarea de demonio como parámetro ulParameter2 de la función.

pxHigherPriorityTaskWoken

`xTimerPendFunctionCallFromISR()` escribe en la cola de comandos del temporizador. Si la tarea de demonio de RTOS estaba en estado Bloqueado a la espera de que los datos estén disponibles en la cola de comandos del temporizador, al escribir en la cola de comandos del temporizador, la tarea de demonio saldrá del estado Bloqueado. Si la prioridad de la tarea de demonio es mayor que la prioridad de la tarea que se está ejecutando actualmente (la tarea que se ha interrumpido), `xTimerPendFunctionCallFromISR()` establecerá internamente `*pxHigherPriorityTaskWoken` en `pdTRUE`. Si `xTimerPendFunctionCallFromISR()` establece este valor en `pdTRUE`, debe efectuarse un cambio de contexto antes de salir de la interrupción. De este modo, se garantiza que la interrupción se devuelve directamente a la tarea de demonio, porque esta será la tarea con el estado Listo de mayor prioridad.

Hay dos valores de retorno posibles:

- `pdPASS`

Se devuelve si el comando "execute function" se ha escrito en el comando del temporizador.

- `pdFAIL`

Se devuelve si el comando "execute function" no se ha podido escribir en la cola de comandos del temporizador porque ya estaba llena. Para obtener más información acerca de cómo establecer la duración del comando del temporizador, consulte `software_tier_management`.

Procesamiento de interrupciones diferido centralizado (Ejemplo 18)

En el Ejemplo 18, se proporciona una funcionalidad similar a la del Ejemplo 16, pero sin usar un semáforo y creando una tarea para realizar el procesamiento que requiere la interrupción. En su lugar, el procesamiento lo realiza la tarea de demonio de RTOS.

La rutina del servicio de interrupciones que se utiliza en el Ejemplo 18 se muestra en el siguiente código. Llama a `xTimerPendFunctionCallFromISR()` para pasar un puntero a una función denominada `vDeferredHandlingFunction()` a la tarea de demonio. El procesamiento de interrupciones diferido lo realiza la función `vDeferredHandlingFunction()`.

La rutina del servicio de interrupciones incrementa una variable denominada `ulParameterValue` cada vez que se ejecuta. `ulParameterValue` se utiliza como el valor de `ulParameter2` en la llamada a `xTimerPendFunctionCallFromISR()`, por lo que también se utiliza como el valor de `ulParameter2` en la llamada a `vDeferredHandlingFunction()` cuando la tarea de demonio ejecuta `vDeferredHandlingFunction()`. El otro parámetro de la función, `pvParameter1`, no se utiliza en este ejemplo.

```
static uint32_t ulExampleInterruptHandler( void )
{
    static uint32_t ulParameterValue = 0;

    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is
    required. */

    xHigherPriorityTaskWoken = pdFALSE;
```

```
/* Send a pointer to the interrupt's deferred handling function to the daemon task.
The deferred handling function's pvParameter1 parameter is not used, so just set to NULL.
The deferred handling function's ulParameter2 parameter is used to pass a number that is
incremented by one each time this interrupt handler executes. */

xTimerPendFunctionCallFromISR( vDeferredHandlingFunction, /* Function to
execute. */ NULL, /* Not used. */ ulParameterValue, /* Incrementing value. */
&xHigherPriorityTaskWoken );

ulParameterValue++;

/* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
xHigherPriorityTaskWoken was set to pdTRUE inside xTimerPendFunctionCallFromISR() then
calling portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is
still pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. Unlike most FreeRTOS
ports, the Windows port requires the ISR to return a value. The return statement is inside
the Windows version of portYIELD_FROM_ISR(). */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

Aquí se muestra la implementación de `vDeferredHandlingFunction()`. Imprime una cadena fija y el valor de su parámetro `ulParameter2`.

```
static void vDeferredHandlingFunction( void *pvParameter1, uint32_t ulParameter2 )
{
    /* Process the event - in this case, just print out a message and the value of
    ulParameter2. pvParameter1 is not used in this example. */

    vPrintStringAndNumber( "Handler function - Processing event ", ulParameter2 );
}
```

Aquí se muestra la función `main()`. `vPeriodicTask()` es la tarea que genera periódicamente interrupciones de software. Se crea con una prioridad por debajo de la prioridad de la tarea de demonio para garantizar que la tarea de demonio tenga prioridad en cuanto salga del estado Bloqueado.

```
int main( void )
{
    /* The task that generates the software interrupt is created at a priority below
    the priority of the daemon task. The priority of the daemon task is set by the
    configTIMER_TASK_PRIORITY compile time configuration constant in FreeRTOSConfig.h. */

    const UBaseType_t ulPeriodicTaskPriority = configTIMER_TASK_PRIORITY - 1;

    /* Create the task that will periodically generate a software interrupt. */

    xTaskCreate( vPeriodicTask, "Periodic", 1000, NULL, ulPeriodicTaskPriority, NULL );

    /* Install the handler for the software interrupt. The syntax necessary to do this is
    dependent on the FreeRTOS port being used. The syntax shown here can only be used with the
    FreeRTOS windows port, where such interrupts are only simulated. */

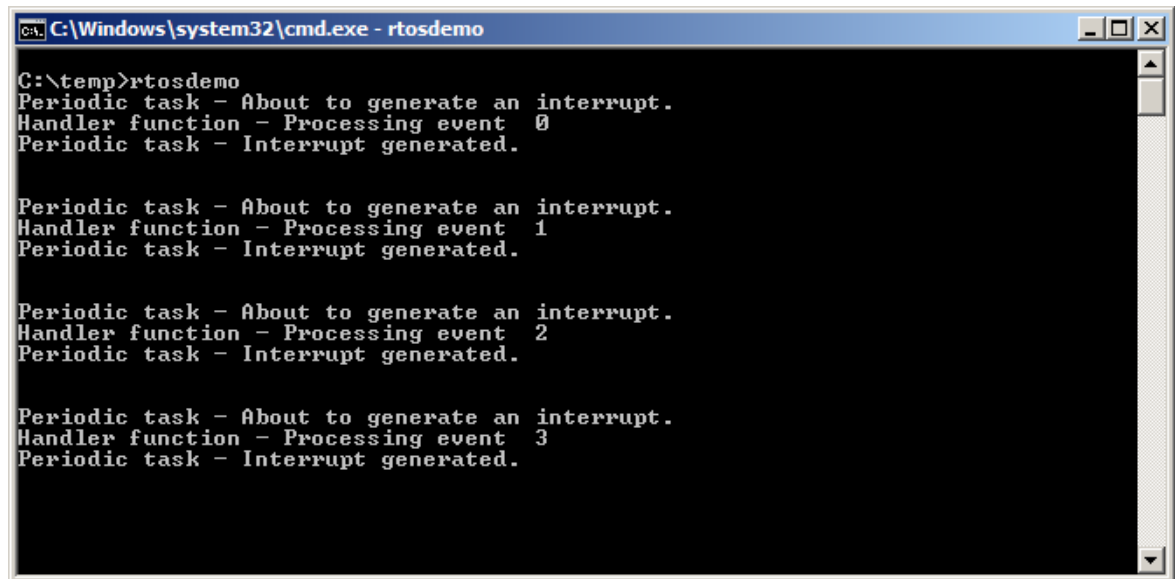
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

    /* Start the scheduler so the created task starts executing. */

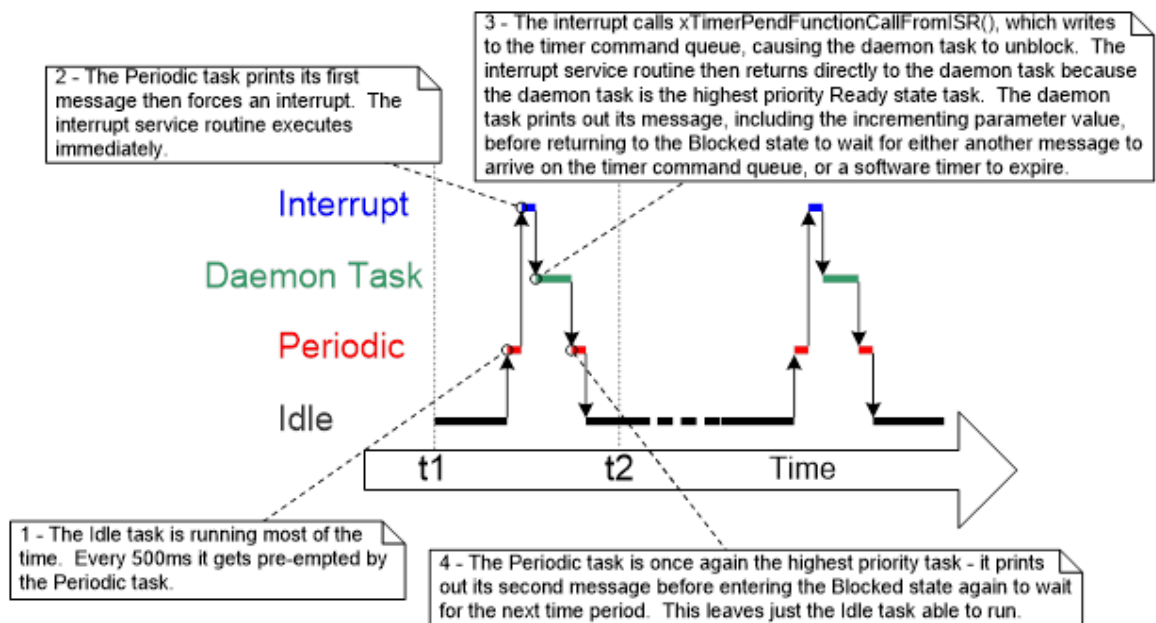
    vTaskStartScheduler();
}
```

```
/* As normal, the following line should never be reached. */  
  
for( ;; );  
  
}
```

El código produce la salida que se muestra aquí. La prioridad de la tarea de demonio es mayor que la prioridad de la tarea que genera la interrupción de software, por lo que la tarea de demonio ejecuta `vDeferredHandlingFunction()` en cuanto se genera la interrupción. Como resultado, la salida del mensaje de `vDeferredHandlingFunction()` aparece entre las dos salidas de mensajes de la tarea periódica, igual que cuando se utilizó un semáforo para desbloquear una tarea de procesamiento de interrupciones diferido especial.



Esta es la secuencia de ejecución.



Uso de colas en una rutina del servicio de interrupciones

Los semáforos de recuento y binarios se utilizan para comunicar eventos. Las colas se utilizan para comunicar eventos y transferir datos.

`xQueueSendToFrontFromISR()` es la versión de `xQueueSendToFront()` más segura para utilizar en una rutina del servicio de interrupciones. `xQueueSendToBackFromISR()` es la versión de `xQueueSendToBack()` más segura para utilizar en una rutina del servicio de interrupciones. `xQueueReceiveFromISR()` es la versión de `xQueueReceive()` más segura para utilizar en una rutina del servicio de interrupciones.

Funciones de API `xQueueSendToFrontFromISR()` y `xQueueSendToBackFromISR()`

Aquí se muestra el prototipo de la función de API `xQueueSendToFrontFromISR()`.

```
BaseType_t xQueueSendToFrontFromISR( QueueHandle_t xQueue, void *pvItemToQueue BaseType_t  
*pxHigherPriorityTaskWoken );
```

Aquí se muestra el prototipo de la función de API `xQueueSendToBackFromISR()`.

```
BaseType_t xQueueSendToBackFromISR( QueueHandle_t xQueue, void *pvItemToQueue BaseType_t  
*pxHigherPriorityTaskWoken );
```

`xQueueSendFromISR()` y `xQueueSendToBackFromISR()` son equivalentes desde el punto de vista funcional.

A continuación, se enumeran los parámetros de `xQueueSendToFrontFromISR()` y `xQueueSendToBackFromISR()` y los valores de retorno.

xQueue

El controlador de la cola a la que se envían los datos (escritos). El controlador de cola se habrá devuelto desde la llamada a `xQueueCreate()` que se utiliza para crear la cola.

pvItemToQueue

Un puntero a los datos que se copiarán en la cola. El tamaño de cada elemento que puede contener la cola se establece al crear la cola, por lo que esta cantidad de bytes se copiará desde `pvItemToQueue` al área de almacenamiento de la cola.

pxHigherPriorityTaskWoken

Es posible que una única cola tenga una o varias tareas bloqueadas a la espera de que los datos estén disponibles. Al llamar a `xQueueSendToBackFromISR()` o `xQueueSendToFrontFromISR()`, los datos pueden estar disponibles y, por tanto, eso hace que una tarea salga del estado Bloqueado. Si al llamar a la función de API, una tarea sale del estado Bloqueado y la prioridad de la tarea desbloqueada es mayor que la de la tarea que se está ejecutando en ese momento (la tarea que se interrumpió), la función de API establecerá internamente `*pxHigherPriorityTaskWoken` en `pdTRUE`. Si `xQueueSendToFrontFromISR()` o `xQueueSendToBackFromISR()` establecen este valor en `pdTRUE`, debe efectuarse un cambio de contexto antes de salir de la interrupción. De este modo, se asegurará de que la interrupción vuelve directamente a la tarea con el estado Listo de máxima prioridad.

Hay dos valores de retorno posibles:

- pdPASS

Solo se devuelve si los datos se han enviado correctamente a la cola.

- errQUEUE_FULL

Se devuelve si los datos no se pueden enviar a la cola, porque la cola ya está llena.

Aspectos a tener en cuenta al usar una cola desde una ISR

Las colas son una forma cómoda y sencilla de pasar datos de una interrupción a una tarea. No es eficaz utilizar una cola si los datos llegan con mucha frecuencia.

Muchas de las aplicaciones de demostración de la descarga de FreeRTOS incluyen un controlador de UART sencillo que utiliza una cola para pasar caracteres de la ISR de recepción de UART. En las demostraciones, una cola sirve para demostrar el uso de colas desde una ISR y para cargar deliberadamente el sistema con el fin de probar el puerto de FreeRTOS. Las ISR que utilizan una cola de esta manera no tienen un diseño eficiente y, a menos que los datos lleguen lentamente, no recomendamos utilizar esta técnica en el código de producción. Las técnicas más eficaces incluyen:

- Uso de hardware de acceso directo a memoria (DMA) para recibir y almacenar caracteres en el búfer. Este método no tiene prácticamente ninguna sobrecarga para el software. Luego, se puede usar una notificación directa a tareas para desbloquear la tarea que procesará el búfer solo después de que se haya detectado un corte en la transmisión. Las notificaciones directas a tareas son el método más eficiente para desbloquear una tarea de una ISR. Para obtener más información, consulte [Notificaciones de tareas \(p. 204\)](#).
- Copia de cada carácter recibido en un búfer de RAM seguro para subprocessos. Para este fin, se puede utilizar el "búfer de transmisión" que se proporciona como parte de FreeRTOS+TCP. Una vez más, se puede usar una notificación directa a tareas para desbloquear la tarea que procesará el búfer después de que se haya recibido un mensaje completo o después de que se haya detectado un corte en la transmisión.
- Procesamiento de los caracteres recibidos directamente en la ISR y, a continuación, uso de una cola para enviar solo el resultado del procesamiento de los datos (en lugar de los datos sin procesar) a una tarea.

Envío y recepción en una cola desde una interrupción (Ejemplo 19)

En este ejemplo, se demuestra el uso de `xQueueSendToBackFromISR()` y `xQueueReceiveFromISR()` dentro de la misma interrupción. Para mayor comodidad, la interrupción se genera por software.

Se crea una tarea periódica que envía cinco números a una cola cada 200 milisegundos. Se genera una interrupción de software solo después de haber enviado los cinco valores. La implementación de la tarea se muestra aquí.

```
static void vIntegerGenerator( void *pvParameters )
```



```
{

    TickType_t xLastExecutionTime;

    uint32_t ulValueToSend = 0;

    int i;

    /* Initialize the variable used by the call to vTaskDelayUntil(). */
    xLastExecutionTime = xTaskGetTickCount();

    for( ;; )

    {

        /* This is a periodic task. Block until it is time to run again. The task will
        execute every 200 ms. */

        vTaskDelayUntil( &xLastExecutionTime, pdMS_TO_TICKS( 200 ) );

        /* Send five numbers to the queue, each value one higher than the previous value.
        The numbers are read from the queue by the interrupt service routine. The interrupt
        service routine always empties the queue, so this task is guaranteed to be able to write
        all five values without needing to specify a block time. */

        for( i = 0; i < 5; i++ )

        {

            xQueueSendToBack( xIntegerQueue, &ulValueToSend, 0 );

            ulValueToSend++;

        }

        /* Generate the interrupt so the interrupt service routine can read the values from
        the queue. The syntax used to generate a software interrupt depends on the FreeRTOS port
        being used. The syntax used below can only be used with the FreeRTOS Windows port, in
        which such interrupts are only simulated.*/

        vPrintString( "Generator task - About to generate an interrupt.\r\n" );

        vPortGenerateSimulatedInterrupt( mainINTERRUPT_NUMBER );

        vPrintString( "Generator task - Interrupt generated.\r\n\r\n\r\n" );

    }

}
```

La rutina del servicio de interrupciones llama a `xQueueReceiveFromISR()` repetidamente hasta que se han leído todos los valores que la tarea periódica ha escrito en la cola y la cola se queda vacía. Los dos últimos bits de cada valor recibido se utilizan como un índice en una matriz de cadenas. Luego, se envía un puntero a la cadena en la posición de índice correspondiente a una cola diferente mediante una llamada a `xQueueSendFromISR()`. Aquí se muestra la implementación de la rutina del servicio de interrupciones.

```
static uint32_t ulExampleInterruptHandler( void )

{

    BaseType_t xHigherPriorityTaskWoken;

    uint32_t ulReceivedNumber;
```

```
/* The strings are declared static const to ensure they are not allocated on the
interrupt service routine's stack, and so exist even when the interrupt service routine is
not executing. */

static const char *pcStrings[] =

{

    "String 0\r\n",

    "String 1\r\n",

    "String 2\r\n",

    "String 3\r\n"

};

/* As always, xHigherPriorityTaskWoken is initialized to pdFALSE to be able to detect
it getting set to pdTRUE inside an interrupt-safe API function. Because an interrupt-safe
API function can only set xHigherPriorityTaskWoken to pdTRUE, it is safe to use the same
xHigherPriorityTaskWoken variable in both the call to xQueueReceiveFromISR() and the call
to xQueueSendToBackFromISR(). */

xHigherPriorityTaskWoken = pdFALSE;

/* Read from the queue until the queue is empty. */

while( xQueueReceiveFromISR( xIntegerQueue, &ulReceivedNumber,
&xHigherPriorityTaskWoken ) != errQUEUE_EMPTY )

{

    /* Truncate the received value to the last two bits (values 0 to 3 inclusive), and
then use the truncated value as an index into the pcStrings[] array to select a string
(char *) to send on the other queue. */

    ulReceivedNumber &= 0x03;

    xQueueSendToBackFromISR( xStringQueue, &pcStrings[ ulReceivedNumber ],
&xHigherPriorityTaskWoken );

}

/* If receiving from xIntegerQueue caused a task to leave the Blocked state, and if the
priority of the task that left the Blocked state is higher than the priority of the task
in the Running state, then xHigherPriorityTaskWoken will have been set to pdTRUE inside
xQueueReceiveFromISR(). If sending to xStringQueue caused a task to leave the Blocked
state, and if the priority of the task that left the Blocked state is higher than the
priority of the task in the Running state, then xHigherPriorityTaskWoken will have been
set to pdTRUE inside xQueueSendToBackFromISR(). xHigherPriorityTaskWoken is used as the
parameter to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
this function does not explicitly return a value. */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

La tarea que recibe los punteros de caracteres de la rutina del servicio de interrupciones se bloquea en la cola hasta que llega un mensaje e imprime cada cadena a medida que la recibe. La implementación se muestra aquí.

```
static void vStringPrinter( void *pvParameters )
{
    char *pcString;

    for( ;; )
    {
        /* Block on the queue to wait for data to arrive. */
        xQueueReceive( xStringQueue, &pcString, portMAX_DELAY );

        /* Print out the string received. */
        vPrintString( pcString );
    }
}
```

Como es normal, la función main() crea las colas y las tareas requeridas antes de iniciar el programador. La implementación se muestra aquí.

```
int main( void )
{
    /* Before a queue can be used, it must first be created. Create both queues used by
    this example. One queue can hold variables of type uint32_t. The other queue can hold
    variables of type char*. Both queues can hold a maximum of 10 items. A real application
    should check the return values to ensure the queues have been successfully created. */

    xIntegerQueue = xQueueCreate( 10, sizeof( uint32_t ) );
    xStringQueue = xQueueCreate( 10, sizeof( char * ) );

    /* Create the task that uses a queue to pass integers to the interrupt service routine.
    The task is created at priority 1. */
    xTaskCreate( vIntegerGenerator, "IntGen", 1000, NULL, 1, NULL );

    /* Create the task that prints out the strings sent to it from the interrupt service
    routine. This task is created at the higher priority of 2. */
    xTaskCreate( vStringPrinter, "String", 1000, NULL, 2, NULL );

    /* Install the handler for the software interrupt. The syntax required to do this is
    depends on the FreeRTOS port being used. The syntax shown here can only be used with the
    FreeRTOS Windows port, where such interrupts are only simulated. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulExampleInterruptHandler );

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

    /* If all is well, then main() will never reach here because the scheduler will
    now be running the tasks. If main() does reach here, then it is likely that there was
    insufficient heap memory available for the idle task to be created. */

    for( ;; );
}
```

```
}
```

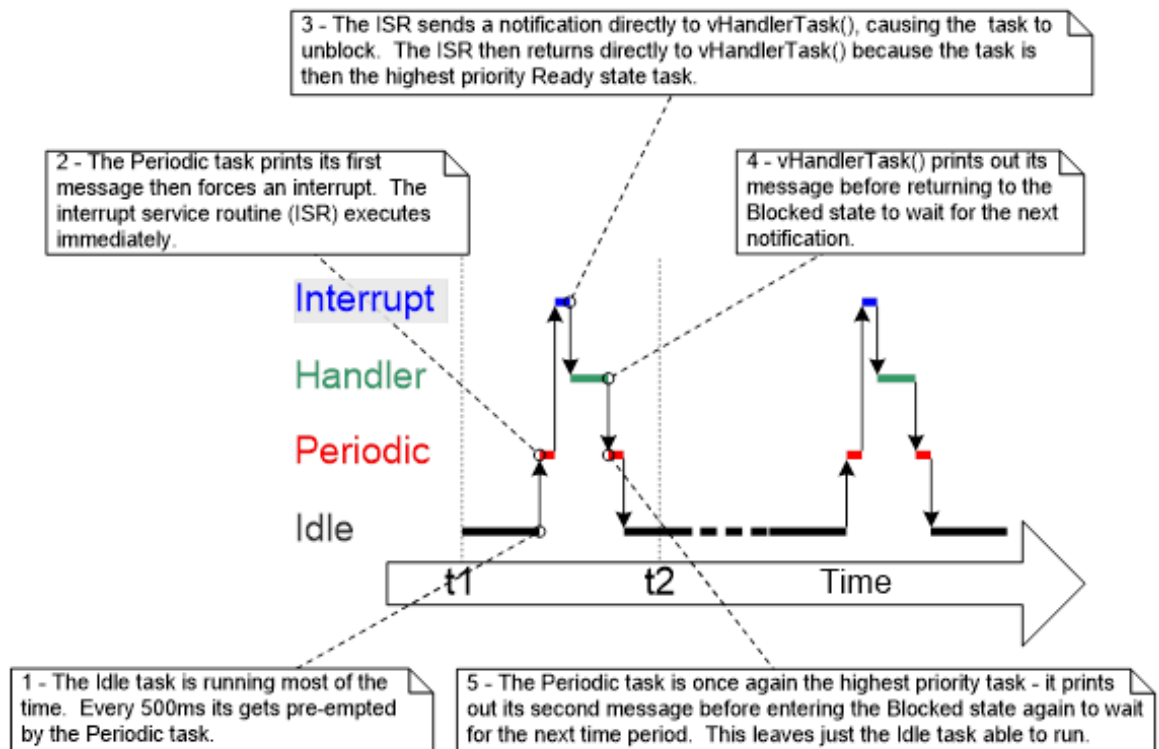
El resultado se muestra aquí. Como verá, la interrupción recibe los cinco números enteros y produce como respuesta cinco cadenas.

```
C:\WINDOWS\system32\cmd.exe - rtosdemo
String 3
String 0
String 1
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 2
String 3
String 0
String 1
String 2
Generator task - Interrupt generated.

Generator task - About to generate an interrupt.
String 3
String 0
String 1
String 2
String 3
Generator task - Interrupt generated.
```

Esta es la secuencia de ejecución.



Anidamiento de interrupciones

Es frecuente confundir las prioridades de las tareas y las prioridades de las interrupciones. En esta sección, se describen las prioridades de las interrupciones, que son las prioridades con las que se ejecutan las ISR en relación entre sí. La prioridad asignada a una tarea no está relacionada de ningún modo con la prioridad asignada a una interrupción. El hardware decide cuándo se ejecutará una ISR. El software decide cuándo se ejecutará una tarea. Una ISR que se ejecuta en respuesta a una interrupción de hardware interrumpirá una tarea, pero una tarea no puede tener preferencia sobre una ISR.

Los puertos que admiten el anidamiento de interrupciones requieren que se defina una o ambas de las constantes que se muestran en la siguiente tabla en FreeRTOSConfig.h. Tanto `configMAX_SYSCALL_INTERRUPT_PRIORITY` como `configMAX_API_CALL_INTERRUPT_PRIORITY` definen la misma propiedad. Los puertos de FreeRTOS más antiguos utilizan `configMAX_SYSCALL_INTERRUPT_PRIORITY`. Los puertos de FreeRTOS más recientes utilizan `configMAX_API_CALL_INTERRUPT_PRIORITY`.

En la siguiente tabla, se muestran constantes que controlan el anidamiento de interrupciones.

Constant	Descripción
<code>configMAX_SYSCALL_INTERRUPT_PRIORITY</code> o <code>configMAX_API_CALL_INTERRUPT_PRIORITY</code>	Establece la mayor prioridad de interrupción desde la que se pueden llamar funciones de API de FreeRTOS a prueba de interrupciones.
<code>configKERNEL_INTERRUPT_PRIORITY</code>	<p>Establece la prioridad de interrupción que utiliza la interrupción de ciclos. Siempre debe establecerse en la prioridad de interrupción mínima posible.</p> <p>Si el puerto de FreeRTOS en uso tampoco utiliza la constante <code>configMAX_SYSCALL_INTERRUPT_PRIORITY</code>, cualquier interrupción que utilice funciones de API de FreeRTOS a prueba de interrupciones también debe ejecutarse con la prioridad definida por <code>configKERNEL_INTERRUPT_PRIORITY</code>.</p>

Cada origen de interrupción posee una prioridad numérica y lógica.

- Numérica

El número asignado a la prioridad de la interrupción. Por ejemplo, si a una interrupción se le asigna una prioridad de 7, entonces su prioridad numérica es 7. Igualmente, si a una interrupción se le asigna una prioridad de 200, su prioridad numérica es 200.

- Lógica

Describe la precedencia de la interrupción con respecto a otras interrupciones. Si se producen a la vez dos interrupciones con una prioridad diferente, el procesador ejecutará la ISR para la interrupción que tenga la mayor prioridad lógica.

Una interrupción puede interrumpir (anidar en) cualquier interrupción que tenga una prioridad lógica menor, pero no puede interrumpir una que tenga una prioridad lógica igual o mayor.

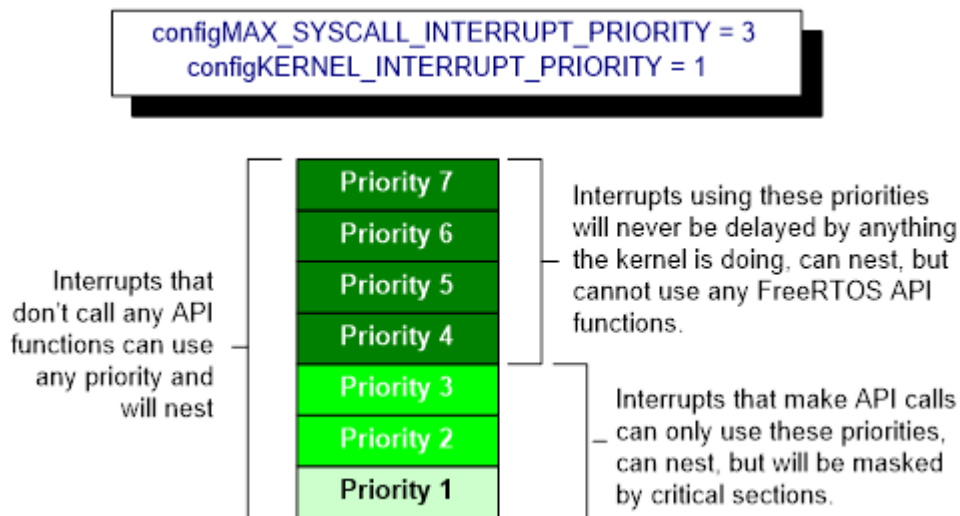
La relación entre la prioridad numérica y la prioridad lógica de una interrupción depende de la arquitectura del procesador. En algunos procesadores, cuanto mayor sea la prioridad numérica asignada a una interrupción, mayor será la prioridad lógica de esa interrupción. En las arquitecturas de otros procesadores,

cuanto mayor sea la prioridad numérica asignada a una interrupción, menor será la prioridad lógica de esa interrupción.

Puede crear un modelo de anidamiento de interrupciones completo estableciendo `configMAX_SYSCALL_INTERRUPT_PRIORITY` en una prioridad lógica de las interrupciones mayor que `configKERNEL_INTERRUPT_PRIORITY`. En la siguiente figura se muestra una situación en la que:

- El procesador tiene 7 prioridades de interrupciones únicas.
- Las interrupciones que tienen asignada una prioridad numérica de 7 poseen una prioridad lógica mayor que las interrupciones que tienen asignada una prioridad numérica de 1.
- `configKERNEL_INTERRUPT_PRIORITY` está establecido en 1.
- `configMAX_SYSCALL_INTERRUPT_PRIORITY` está establecido en 3.

Esta figura muestra las constantes que afectan al comportamiento de anidamiento de las interrupciones.



- Las interrupciones que utilizan las prioridades 1 a 3, inclusive, no se ejecutan mientras el kernel o la aplicación estén dentro de una sección crítica. Las ISR que se ejecutan con estas prioridades pueden utilizar funciones de API de FreeRTOS a prueba de interrupciones. Para obtener más información acerca de las secciones críticas, consulte [Administración de recursos \(p. 161\)](#).
- Las interrupciones que utilizan una prioridad de 4 o superior no se ven afectadas por las secciones críticas, por lo que nada de lo que haga el programador impide que estas interrupciones se ejecuten de forma inmediata dentro de las limitaciones del hardware. Las ISR que se ejecutan con estas prioridades no pueden utilizar ninguna de las funciones de API de FreeRTOS.
- Normalmente, una funcionalidad que requiera una precisión de tiempo muy estricta (el control de un motor, por ejemplo) utilizaría una prioridad por encima de `configMAX_SYSCALL_INTERRUPT_PRIORITY` para garantizar que el programador no introdujera fluctuaciones en el tiempo de respuesta de las interrupciones.

Usuarios de Cortex-M y GIC de ARM

Esta sección solo se aplica parcialmente a los núcleos de Cortex-M0 y Cortex-M0+. La configuración de interrupciones en los procesadores de Cortex-M es confusa y propensa a errores. Para ayudarle en su desarrollo, los puertos de Cortex-M de FreeRTOS comprueban automáticamente la configuración de interrupciones, pero solo si se define `configASSERT()`. Para obtener más información acerca de `configASSERT`, consulte [Developer Support \(p. 231\)](#).

Los núcleos de Cortex de ARM y los controladores de interrupciones genéricos (GIC) de ARM utilizan números de baja prioridad numérica para representar interrupciones de alta prioridad lógica. Esto parece contradictorio. Si desea asignar a una interrupción una prioridad baja lógica, se le debe asignar un valor numérico alto. Si desea asignar a una interrupción una prioridad alta lógica, se le debe asignar un valor numérico bajo.

El controlador de interrupciones de Cortex-M permite utilizar un máximo de ocho bits para especificar la prioridad de cada interrupción y 255 es la prioridad más baja posible. Cero es la prioridad más alta. Sin embargo, normalmente los microcontroladores de Cortex-M solo implementan un subconjunto de los ocho bits posibles. El número de bits implementados depende de la familia de microcontroladores.

Cuando únicamente se implementa un subconjunto, solo se pueden utilizar los bits más significativos del byte. Los bits menos significativos se quedan sin implementar. Los bits no implementados pueden tomar cualquier valor, pero lo normal es que establezcan en 1. Esta figura muestra cómo una prioridad de 101 binaria se almacena en un microcontrolador de Cortex-M que implementa cuatro bits de prioridad.



Como verá, el valor binario 101 se ha cambiado por los cuatro bits más significativos porque los cuatro bits menos significativos no se han implementado. Los bits no implementados se han establecido en 1.

Algunas funciones de biblioteca esperan que se especifiquen valores de prioridad después de que se hayan cambiado a los bits implementados (más significativos). Cuando se utiliza esta función, la prioridad que se muestra en esta figura se puede especificar como 95 decimal. 95 decimal es 101 binario con un desvío de cuatro para obtener 101nnnn binario (donde n es un bit no implementado). Los bits no implementados se establecen en 1 para obtener 1011111 binario.

Algunas funciones de biblioteca esperan que se especifiquen valores de prioridad antes de que se hayan desviado a los bits implementados (más significativos). Cuando se utiliza esta función, la prioridad que se muestra en esta figura se debe especificar como 5 decimal. El 5 decimal 5 es el 101 binario sin ningún desvío.

Es necesario especificar `configMAX_SYSCALL_INTERRUPT_PRIORITY` y `configKERNEL_INTERRUPT_PRIORITY` de forma que se puedan escribir directamente en los registros de Cortex-M (es decir, después de que los valores de prioridad se hayan desviado a los bits implementados).

`configKERNEL_INTERRUPT_PRIORITY` debe establecerse siempre en la prioridad de interrupción más baja posible. Los bits de prioridad no implementados se pueden establecer en 1, por lo que la constante siempre pueden establecerse en 255, con independencia del número de bits de prioridad que se implementen.

Las interrupciones de Cortex-M utilizan una prioridad de cero de forma predeterminada, que es la máxima prioridad posible. La implementación del hardware de Cortex-M no permite establecer `configMAX_SYSCALL_INTERRUPT_PRIORITY` en 0, por lo que la prioridad de una interrupción que utilice la API de FreeRTOS nunca debe dejarse en su valor predeterminado.

Administración de recursos

En esta sección se explica lo siguiente:

- Cuándo y por qué es importante a la administración y el control de los recursos.
- Qué es una sección crítica.
- Qué significa la exclusión mutua.
- Qué significa suspender el programador.
- Cómo utilizar un mutex.
- Cómo crear y utilizar una tarea de guardián.
- Qué es la inversión de prioridades y cómo puede reducir (pero no eliminar) su impacto la herencia de prioridades.

En un sistema multitarea, existe la posibilidad de que se produzca un error si una tarea comienza a acceder a un recurso, pero no completa su acceso antes de que salga del estado En ejecución. Si la tarea deja el recurso en un estado incoherente, cuando cualquier otra tarea o interrupción acceden a ese mismo recurso, los datos podrían dañarse o producirse otro problema similar.

A continuación se muestran algunos ejemplos:

1. Acceso a periféricos

Pongamos como ejemplo la siguiente situación en la que dos tareas intentan escribir en una pantalla de cristal líquido (LCD).

- a. La Tarea A se ejecuta y comienza a escribir la cadena "Hola mundo" en la pantalla LCD.
- b. La Tarea B asume la preferencia sobre la Tarea A después de producir solo el principio de la cadena "Hola m".
- c. La Tarea B escribe "¿Anular, Reintentar, Error?" en la pantalla LCD antes de pasar al estado Bloqueado.
- d. La Tarea A continúa desde el punto en que perdió la preferencia y produce el resto de caracteres de su cadena ("undo").

Ahora, en la pantalla LCD aparece la cadena dañada "Hola m¿Anular, Reintentar, Error?undo".

2. Operaciones de lectura, modificación y escritura

A continuación, se muestra una línea de código en C y un ejemplo de cómo se suele traducirse en código de ensamblado. Como verá, el valor de PORTA se lee primero de la memoria en un registro, se modifica en el registro y, a continuación, se vuelve a escribir en la memoria. Esto es lo que se conoce como una operación de lectura, modificación y escritura.

```
/* The C code being compiled. */  
  
PORTA |= 0x01;  
  
/* The assembly code produced when the C code is compiled. */  
  
LOAD R1,[#PORTA] ; Read a value from PORTA into R1  
  
MOVE R2,#0x01 ; Move the absolute constant 1 into R2
```



```
OR R1,R2 ; Bitwise OR R1 (PORTA) with R2 (constant 1)

STORE R1,[#PORTA] ; Store the new value back to PORTA
```

Se trata de una operación no atómica porque tarda más que una instrucción en completarse y se puede interrumpir. Considere la siguiente situación en la que dos tareas intentan actualizar un registro asignado a la memoria denominado PORTA.

1. La Tarea A carga el valor de PORTA en un registro, que es la parte de lectura de la operación.
2. La Tarea B asume la preferencia sobre la Tarea A antes de que se completen las partes de modificación y escritura de la misma operación.
3. La Tarea B actualiza el valor de PORTA y, a continuación, pasa al estado Bloqueado.
4. La Tarea A continúa desde el punto en que se perdió la preferencia. Modifica la copia del valor de PORTA valor que ya guarda en un registro antes de escribir el valor actualizado de nuevo en PORTA.

En este caso, la Tarea A actualiza y escribe de nuevo un valor desactualizado para PORTA. La Tarea B modifica PORTA después de que la Tarea A tome una copia del valor de PORTA y antes de que la Tarea A escriba su valor modificado en el registro de PORTA. Cuando la Tarea A escribe en PORTA, sobrescribe la modificación que ya ha realizado la Tarea B, por lo que daña el valor de registro de PORTA.

En este ejemplo, se utiliza un registro de periférico, pero este mismo principio se aplica al realizar operaciones de lectura, modificación y escritura en variables.

1. Acceso no atómico a las variables

La actualización de varios miembros de una estructura o la actualización de una variable que es mayor que el tamaño real de la arquitectura (por ejemplo, actualizar una variable de 32 bits en una máquina de 16 bits) son ejemplos de operaciones no atómicas. Si se interrumpen, los datos podrían perderse o dañarse.

2. Reentrada de funciones

Una función es reentrante si es seguro llamar a la función desde más de una tarea o desde tareas e interrupciones. Se dice que las funciones reentrantes son seguras para subprocesos porque se puede acceder a ellas desde más de un subproceso de ejecución sin el riesgo de que se dañen las operaciones de datos o lógicas. Cada tarea mantiene su propia pila y su propio conjunto de valores de registro de procesadores (hardware). Si una función no tiene acceso a datos que no sean los datos almacenados en la pila o que se mantienen en un registro, la función es reentrante y es segura para subprocesos. A continuación se muestra un ejemplo de una función reentrante.

```
/* A parameter is passed into the function. This will either be passed on the stack, or
   in a processor register. Either way is safe because each task or interrupt that calls
   the function maintains its own stack and its own set of register values, so each task or
   interrupt that calls the function will have its own copy of lVar1. */
long lAddOneHundred( long lVar1 )

{

    /* This function scope variable will also be allocated to the stack or a register,
       depending on the compiler and optimization level. Each task or interrupt that calls this
       function will have its own copy of lVar2. */

    long lVar2;

    lVar2 = lVar1 + 100;

    return lVar2;

}
```

A continuación se muestra un ejemplo de una función que no es reentrante.

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

long lVar1;

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

    static long lState = 0;

    long lReturn;

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

                lState = 1;

                break;

        case 1 : lReturn = lVar1 + 20;

                lState = 0;

                break;

    }
}
```

Exclusión mutua

Para garantizar la coherencia de los datos en todo momento, utilice la técnica de exclusión mutua para administrar el acceso a un recurso que se comparte entre tareas o entre tareas e interrupciones. El objetivo es garantizar que, una vez que la tarea comience a acceder a un recurso compartido que no sea reentrante y seguro para subprocesos, la misma tarea tenga acceso exclusivo al recurso hasta que el recurso se haya devuelto a un estado coherente.

FreeRTOS dispone de varias características que se pueden utilizar para implementar la exclusión mutua, pero el mejor método de exclusión mutua es, siempre que resulte práctico, diseñar la aplicación de tal forma que los recursos no se compartan y que solo se acceda a cada recurso desde una sola tarea.

Secciones críticas y suspensión del programador

Secciones críticas básicas

Las secciones críticas básicas son regiones de código que están rodeadas de las llamadas a las macros `taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()`, respectivamente. Las secciones críticas también se conocen como regiones críticas.

`taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()` no toman ningún parámetro ni devuelven un valor. (En realidad, una macro de tipo función no devuelve un valor de la misma forma que una función real, pero es más fácil pensar en la macro como si fuera una función).

Aquí se muestra su uso, donde se utiliza una sección crítica para proteger el acceso a un registro.

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

taskENTER_CRITICAL();

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

PORTA |= 0x01;

/* Access to PORTA has finished, so it is safe to exit the critical section. */

taskEXIT_CRITICAL();
```

Varios de los ejemplos utilizan una función denominada `vPrintString()` para escribir cadenas en una salida estándar, que es la ventana de terminal cuando se utiliza el puerto de Windows de FreeRTOS. `vPrintString()` se llama desde muchas tareas diferentes, por lo que, en teoría, su implementación podría proteger el acceso a la salida estándar mediante una sección crítica, tal y como se muestra aquí.

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, using a critical section as a crude method of mutual
exclusion. */

    taskENTER_CRITICAL();

    {
        printf( "%s", pcString );

        fflush( stdout );
    }

    taskEXIT_CRITICAL();
}
```

Las secciones críticas que se implementan de esta forma son un método muy rudimentario de proporcionar exclusión mutua. Funcionan desactivando las interrupciones, ya sea por completo o hasta la prioridad de interrupción que establece `configMAX_SYSCALL_INTERRUPT_PRIORITY`, en función del puerto de FreeRTOS que se esté utilizando. Los cambios de contexto preferentes solo pueden realizarse desde una interrupción, por lo que, mientras las interrupciones estén deshabilitadas, se garantiza que la tarea que ha llamado a `taskENTER_CRITICAL()` permanecerá en el estado En ejecución hasta que la sección crítica salga.

Las secciones críticas básicas deben ser muy cortas. De lo contrario, afectarían negativamente a los tiempos de respuesta de las interrupciones. Cada llamada a `taskENTER_CRITICAL()` debe emparejarse estrechamente con una llamada a `taskEXIT_CRITICAL()`. Por este motivo, la salida estándar (stdout o la

secuencia donde un equipo escribe sus datos de salida) no se debe proteger mediante una sección crítica (tal y como se muestra en el código anterior), ya que la escritura en el terminal puede ser una operación relativamente larga. En los ejemplos de esta sección se exploran soluciones alternativas.

No hay problema en que las secciones críticas se aniden porque el kernel mantiene el recuento de la profundidad de anidamiento. Solo se saldrá de la sección crítica cuando la profundidad de anidamiento vuelva a cero, es decir, cuando se haya ejecutado una llamada a `taskEXIT_CRITICAL()` por cada llamada anterior a `taskENTER_CRITICAL()`.

Llamar a `taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()` es la única forma legítima de que una tarea modifique el estado de activación de la interrupción del procesador en el que se está ejecutando FreeRTOS. La modificación del estado de la interrupción por otros medios invalidará el recuento de anidamiento de la macro.

`taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()` no terminan en "FromISR", por lo que no deben llamarse desde una rutina del servicio de interrupciones. `taskENTER_CRITICAL_FROM_ISR()` es una versión a prueba de interrupciones de `taskENTER_CRITICAL()`. `taskEXIT_CRITICAL_FROM_ISR()` es una versión a prueba de interrupciones de `taskEXIT_CRITICAL()`. Las versiones a prueba de interrupciones solo se proporcionan solo para los puertos de FreeRTOS que permiten el anidamiento de interrupciones. Estarían obsoletas en puertos que no permiten el anidamiento de interrupciones.

`taskENTER_CRITICAL_FROM_ISR()` devuelve un valor que debe pasarse en la llamada a correspondiente a `taskEXIT_CRITICAL_FROM_ISR()`, tal y como se muestra aquí.

```
void vAnInterruptServiceRoutine( void )
{
    /* Declare a variable in which the return value from taskENTER_CRITICAL_FROM_ISR() will
    be saved. */

    UBaseType_t uxSavedInterruptStatus;

    /* This part of the ISR can be interrupted by any higher priority interrupt. */

    /* Use taskENTER_CRITICAL_FROM_ISR() to protect a region of this ISR. Save the value
    returned from taskENTER_CRITICAL_FROM_ISR() so it can be passed into the matching call to
    taskEXIT_CRITICAL_FROM_ISR(). */

    uxSavedInterruptStatus = taskENTER_CRITICAL_FROM_ISR();

    /* This part of the ISR is between the call to taskENTER_CRITICAL_FROM_ISR() and
    taskEXIT_CRITICAL_FROM_ISR(), so can only be interrupted by interrupts that have a
    priority above that set by the configMAX_SYSCALL_INTERRUPT_PRIORITY constant. */

    /* Exit the critical section again by calling taskEXIT_CRITICAL_FROM_ISR(), passing in
    the value returned by the matching call to taskENTER_CRITICAL_FROM_ISR(). */

    taskEXIT_CRITICAL_FROM_ISR( uxSavedInterruptStatus );

    /* This part of the ISR can be interrupted by any higher priority interrupt. */
}
```

Es un desperdicio utilizar más tiempo de procesamiento para ejecutar el código que entra y sale posteriormente de una sección crítica que el código que está protegido por la sección crítica. Las secciones críticas entran muy rápido y salen muy rápido y siempre son deterministas, por lo que su uso es ideal cuando la región de código que se protege es muy corta.

Suspensión (o bloqueo) del programador

Las secciones críticas también se pueden crear suspendiendo el programador. Suspende el programador también se conoce en ocasiones como bloqueo del programador.

Las secciones críticas básicas protegen una región de código del acceso de otras tareas e interrupciones. Una sección crítica implementada mediante la suspensión del programador solo protege una región de código del acceso de otras tareas, porque las interrupciones permanecen habilitadas.

Una sección crítica que es demasiado larga para implementarla simplemente desactivando las interrupciones se puede implementar suspendiendo el programador. Sin embargo, la actividad de las interrupciones mientras el programador está suspendido puede hacer que el programador tarde en reanudarse (o en cancelar la suspensión) un tiempo relativamente largo. Piense en el mejor método para cada caso.

Función de API vTaskSuspendAll()

Aquí se muestra el prototipo de la función de API vTaskSuspendAll().

```
void vTaskSuspendAll( void );
```

El programador se suspende llamando a vTaskSuspendAll(). Suspende el programador evita que se produzca un cambio de contexto, pero deja habilitadas las interrupciones. Si una interrupción solicita un cambio de contexto mientras el programador está suspendido, la solicitud queda pendiente y solo se realiza cuando el programador se reanuda (se cancela la suspensión).

No hay que llamar a las funciones de API de FreeRTOS mientras el programador está suspendido.

Función de API xTaskResumeAll()

Aquí se muestra el prototipo de la función de API xTaskResumeAll().

```
BaseType_t xTaskResumeAll( void );
```

El programador se reanuda (deja de estar suspendido) llamando a xTaskResumeAll().

En la siguiente tabla, se muestra el valor de retorno de xTaskResumeAll().

Valor devuelto	Descripción
Valor devuelto	Los cambios de contexto que se solicitan mientras el programador está suspendido se quedan pendientes y solo se realizan cuando se reanuda el programador. Si se realiza un cambio de contexto antes de que se devuelva xTaskResumeAll(), se devuelve pdTRUE. De lo contrario, se devuelve pdFALSE.

No hay ningún problema en que las llamadas a vTaskSuspendAll() y xTaskResumeAll() se aniden porque el kernel mantiene el recuento de la profundidad de anidamiento. El programador solo se reanudará

cuando la profundidad de anidamiento vuelva a cero, es decir, cuando una llamada a `xTaskResumeAll()` se haya ejecutado por cada llamada anterior a `vTaskSuspendAll()`.

El código siguiente muestra la implementación de `vPrintString()`, que suspende el programador para proteger el acceso a la salida de terminal.

```
void vPrintString( const char *pcString )
{
    /* Write the string to stdout, suspending the scheduler as a method of mutual
    exclusion. */

    vTaskSuspendScheduler();

    {
        printf( "%s", pcString );

        fflush( stdout );
    }

    xTaskResumeScheduler();
}
```

Mutex (y semáforos binarios)

Un mutex (MUTual EXclusion) es un tipo especial de semáforo binario que se utiliza para controlar el acceso a un recurso que se comparte entre dos o más tareas. Para que los mutex estén disponibles, en `FreeRTOSConfig.h`, establezca `configUSE_MUTEXES` en 1.

Cuando se utiliza en una situación de exclusión mutua, el mutex puede considerarse un token que se asocia al recurso que se está compartiendo. Para que una tarea acceda al recurso de forma legítima, primero debe tomar correctamente el token (ser el titular del token). Cuando el titular del token ha terminado con el recurso, debe devolver el token. Solo cuando el token se haya devuelto otra tarea puede tomar correctamente el token y, a continuación, acceder de forma segura al mismo recurso compartido. Una tarea no tiene permitido el acceso al recurso compartido a menos que tenga el token.

Aunque los mutex y los semáforos binarios comparten muchas características, la situación en la que se usa un mutex es totalmente diferente a otra en la que se utiliza un semáforo binario para la sincronización. La principal diferencia es lo que le ocurre al semáforo después de que se haya obtenido:

- Un semáforo que se utiliza para la exclusión mutua debe devolverse siempre.
- Un semáforo que se utiliza para la sincronización normalmente se descarta y no se devuelve.

El mecanismo funciona únicamente a través de la disciplina del programador de la aplicación. No hay ninguna razón por la que una tarea no pueda tener acceso al recurso en cualquier momento, pero cada tarea acuerda no hacer hacerlo a menos que pueda convertirse en el titular del mutex.

Función de API `xSemaphoreCreateMutex()`

FreeRTOS V9.0.0 también incluye la función `xSemaphoreCreateMutexStatic()`, que asigna la memoria necesaria para crear un mutex estáticamente durante la compilación. Un mutex es un tipo de semáforo.

Los controladores de todos los distintos tipos de semáforos de FreeRTOS se almacenan en una variable de tipo SemaphoreHandle_t.

Para poder utilizar un mutex, antes hay que crearlo. Para crear un semáforo de tipo mutex, utilice la función de API xSemaphoreCreateMutex().

Aquí se muestra el prototipo de la función de API xSemaphoreCreateMutex().

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

En la siguiente tabla se muestra el valor de retorno de xSemaphoreCreateMutex().

Reescritura de vPrintString() para utilizar un semáforo (Ejemplo 20)

En este ejemplo, se crea una nueva versión de vPrintString() llamada prvNewPrintString() y, a continuación, se llama a la nueva función desde varias tareas. prvNewPrintString() es funcionalmente idéntica a vPrintString(), pero controla el acceso a una salida estándar mediante un mutex, en lugar de hacerlo bloqueando el programador.

Aquí se muestra la implementación de prvNewPrintString().

```
static void prvNewPrintString( const char *pcString )
{
    /* The mutex is created before the scheduler is started, so already exists by the time
    this task executes. Attempt to take the mutex, blocking indefinitely to wait for the mutex
    if it is not available right away. The call to xSemaphoreTake() will only return when the
    mutex has been successfully obtained, so there is no need to check the function return
    value. If any other delay period was used, then the code must check that xSemaphoreTake()
    returns pdTRUE before accessing the shared resource (which in this case is standard out).
    Indefinite timeouts are not recommended for production code. */

    xSemaphoreTake( xMutex, portMAX_DELAY );

    {
        /* The following line will only execute after the mutex has been successfully
        obtained. Standard out can be accessed freely now because only one task can have the mutex
        at any one time. */

        printf( "%s", pcString );

        fflush( stdout );

        /* The mutex MUST be given back! */
    }

    xSemaphoreGive( xMutex );
}
```

Dos instancias de una tarea implementada por prvPrintTask() llaman de forma repetida a prvNewPrintString(). Se utiliza un tiempo de retardo aleatorio entre cada llamada. El parámetro de la tarea

se utiliza para pasar una cadena única a cada instancia de la tarea. La implementación de prvPrintTask() se muestra aquí.

```
static void prvPrintTask( void *pvParameters )
{
    char *pcStringToPrint;

    const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created. The string printed by the task is passed
    into the task using the task's parameter. The parameter is cast to the required type. */

    pcStringToPrint = ( char * ) pvParameters;

    for( ;; )
    {
        /* Print out the string using the newly defined function. */

        prvNewPrintString( pcStringToPrint );

        /* Wait a pseudo random time. Note that rand() is not necessarily reentrant,
        but in this case it does not really matter because the code does not care what value is
        returned. In a more secure application, a version of rand() that is known to be reentrant
        should be used or calls to rand() should be protected using a critical section. */

        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );
    }
}
```

Como es normal, la función main() simplemente crea el mutex y las tareas y luego inicia el programador.

Las dos instancias de prvPrintTask() se crean con diferentes prioridades, por lo que la tarea de mayor prioridad asumirá la preferencia sobre la tarea de menor prioridad. Dado que se utiliza un mutex para garantizar que cada tarea obtiene acceso mutuamente excluyente al terminal, incluso cuando se aplica la preferencia, las cadenas que se muestran serán correctas y no estarán dañadas. Puede aumentar la frecuencia de la aplicación de la prioridad reduciendo el tiempo máximo que las tareas están en el estado Bloqueado, que establece la constante xMaxBlockTimeTicks.

Notes para utilizar el Ejemplo 20 con el puerto de Windows de FreeRTOS:

- La llamada a printf() genera una llamada del sistema Windows. Las llamadas del sistema Windows quedan fuera del control de FreeRTOS y pueden producir inestabilidad.
- La forma en que se ejecutan las llamadas del sistema Windows hace que no sea frecuente ver una cadena dañada, incluso cuando no se utiliza el mutex.

```
int main( void )
{
    /* Before a semaphore is used it must be explicitly created. In this example, a mutex
    type semaphore is created. */

    xMutex = xSemaphoreCreateMutex();

    /* Check that the semaphore was created successfully before creating the tasks. */
```



```
if( xMutex != NULL )
{
    /* Create two instances of the tasks that write to stdout. The string they write
    is passed in to the task as the task's parameter. The tasks are created at different
    priorities so some preemption will occur. */

    xTaskCreate( prvPrintTask, "Print1", 1000, "Task 1
    *****\r\n", 1, NULL );

    xTaskCreate( prvPrintTask, "Print2", 1000, "Task 2
    -----\r\n", 2, NULL );

    /* Start the scheduler so the created tasks start executing. */

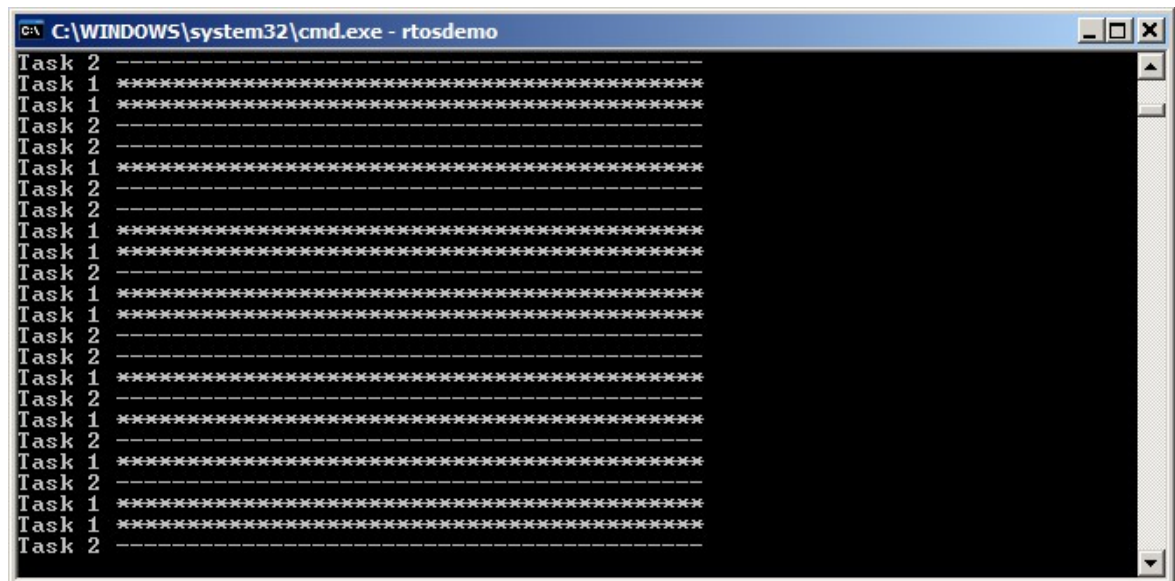
    vTaskStartScheduler();

}

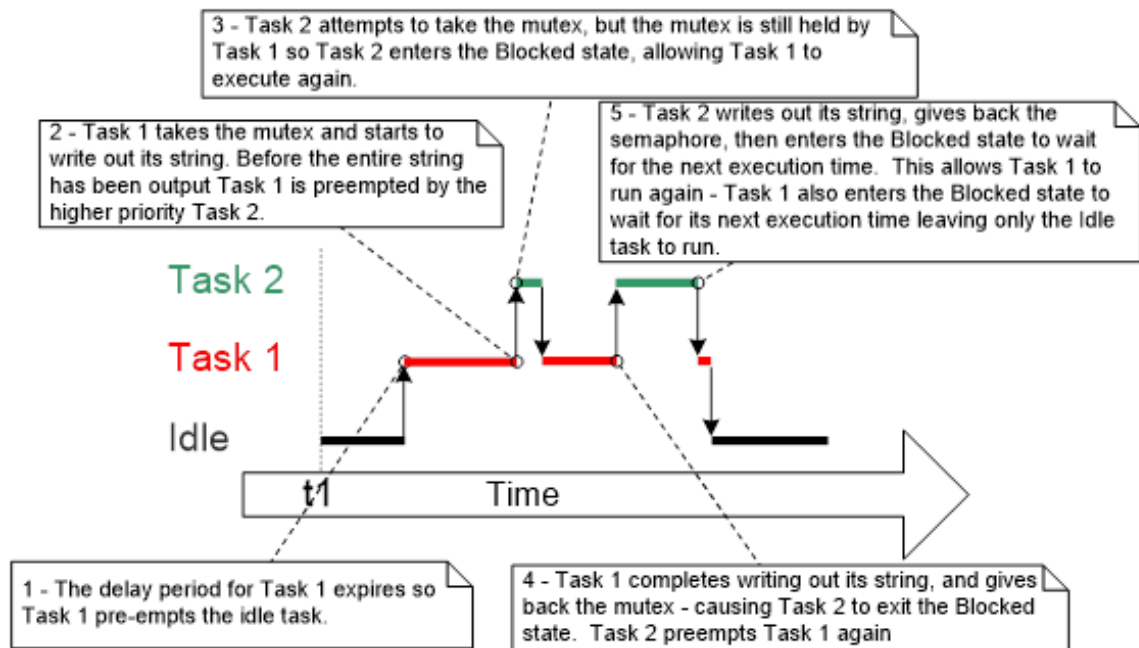
/* If all is well, then main() will never reach here because the scheduler will
now be running the tasks. If main() does reach here, then it is likely that there was
insufficient heap memory available for the idle task to be created. */

for( ;; );
}
```

Esta es la salida.



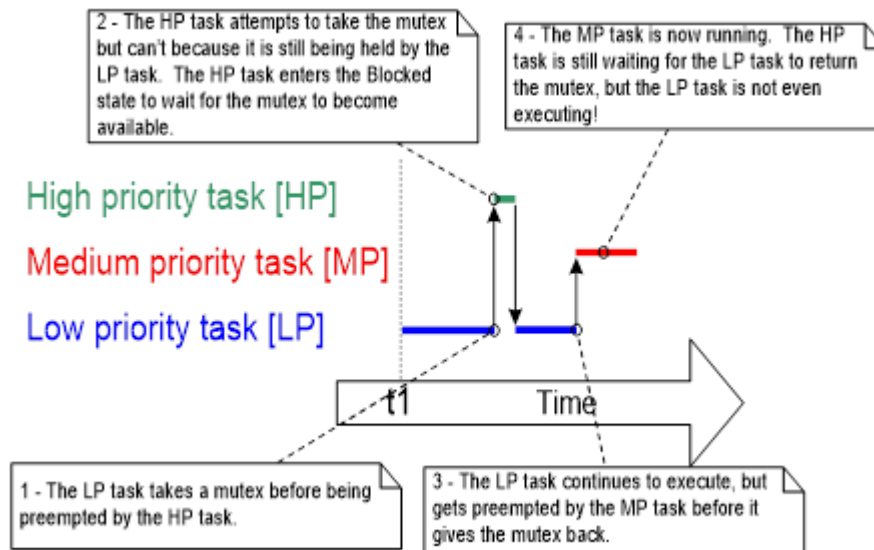
Esta figura muestra una posible secuencia de ejecución.



Como era de esperar, las cadenas que se muestran en el terminal no están dañadas. El orden aleatorio se debe a los periodos de retardo aleatorios que utilizan las tareas.

Inversión de prioridades

La figura anterior ilustra uno de los posibles inconvenientes de utilizar un mutex para proporcionar la exclusión mutua. La secuencia de ejecución que se ilustra muestra que la tarea 2 de mayor prioridad tiene que esperar a que la tarea 1 de menor prioridad deje el control del mutex. El hecho de que una tarea de menor prioridad retrase a una de mayor prioridad de esta manera se denomina inversión de prioridades. Este comportamiento inapropiado se agrava si una tarea de prioridad media comienza a ejecutarse mientras la tarea de mayor prioridad está esperando al semáforo. Como resultado, una tarea de alta prioridad tendría que esperar a una tarea de baja prioridad y la tarea de baja prioridad ni siquiera podría ejecutarse. En esta figura, se muestra el peor de los casos.

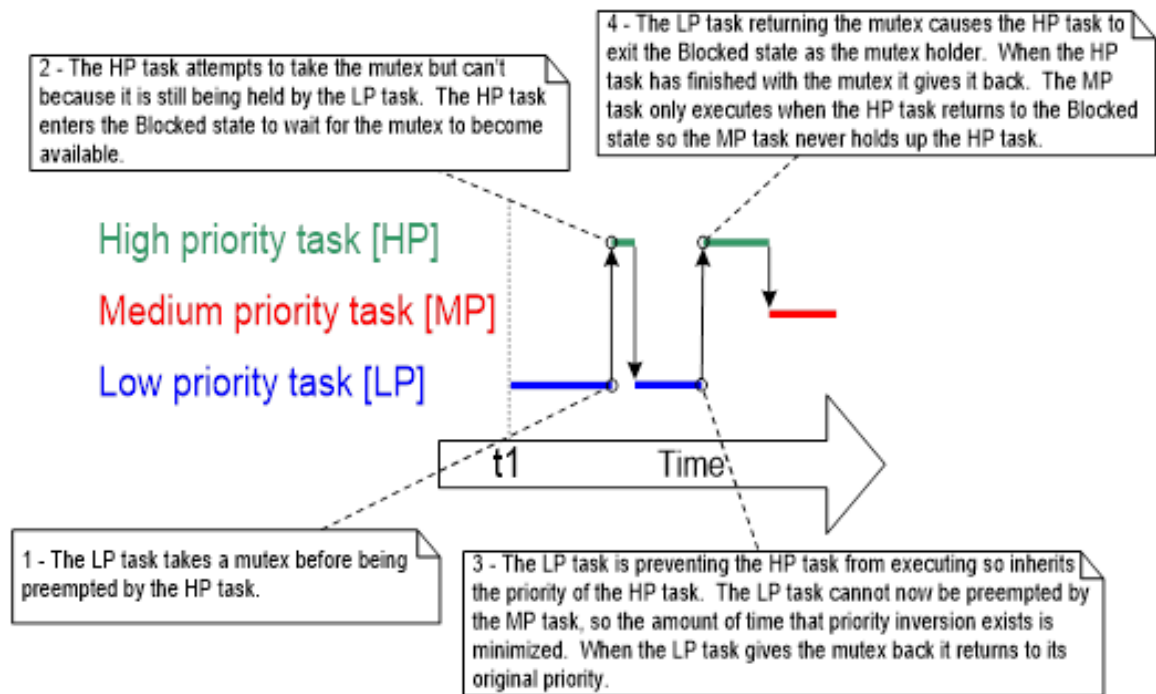


La inversión de prioridades puede ser un problema importante, pero en sistemas integrados pequeños, se puede evitar teniendo en cuenta cómo se accede a los recursos al diseñar el sistema.

Herencia de prioridades

Los mutex de FreeRTOS y los semáforos binarios son muy similares. La diferencia es que los mutex incluyen un mecanismo de herencia de prioridades básico y los semáforos binarios no. La herencia de prioridades es un sistema que minimiza los efectos negativos de la inversión de prioridades. No soluciona la inversión de prioridades, pero reduce su impacto asegurándose de que la inversión siempre esté limitada en el tiempo. Sin embargo, la herencia de prioridades complica el análisis del tiempo del sistema. No es conveniente basarse en ella para que el sistema funcione correctamente.

La herencia de prioridades aumenta de forma temporal la prioridad del titular del mutex hasta la prioridad de la tarea de máxima prioridad que está intentando obtener el mismo mutex. La tarea de baja prioridad que contiene el mutex hereda la prioridad de la tarea que está a la espera del mutex. En la siguiente figura, se muestra que la prioridad del titular del mutex se restablece automáticamente a su valor original cuando devuelve el mutex.



La funcionalidad de herencia de prioridades afecta a la prioridad de las tareas que utilizan el mutex. Por este motivo, no deben utilizarse mutex desde rutinas del servicio de interrupciones.

Interbloqueo (o abrazo mortal)

Un interbloqueo o abrazo mortal es otro de los posibles inconvenientes del uso de mutex para la exclusión mutua.

El interbloqueo se produce cuando dos tareas no pueden continuar porque ambas están a la espera de un recurso que tiene el otro. Considere la siguiente situación en la que la Tarea A y la Tarea B tienen que adquirir el mutex X y el mutex Y para realizar una acción.

1. La Tarea A se ejecuta y toma correctamente el mutex X.
2. La Tarea B asume la preferencia sobre la Tarea A.
3. La Tarea B toma correctamente el mutex Y antes de intentar tomar también el mutex X, pero la Tarea A tiene ese mutex, por lo que no está disponible para la Tarea B. La Tarea B opta por pasar al estado Bloqueado a la espera de que se libere el mutex X.
4. La Tarea A se continúa ejecutando. Intenta tomar el mutex Y, pero la Tarea B tiene ese mutex, por lo que no está disponible para la Tarea A. La Tarea A opta por pasar al estado Bloqueado a la espera de que se libere el mutex Y.

La Tarea A está esperando un mutex que tiene la Tarea B y la Tarea B está esperando un mutex que tiene la Tarea A. Se ha producido un interbloqueo porque ninguna de las tareas puede continuar.

Al igual que ocurre con la inversión de prioridades, el mejor método para evitar el interbloqueo consiste en diseñar el sistema de forma que no pueda producirse ese interbloqueo. Normalmente, no es conveniente que una tarea espere de forma indefinida (sin un tiempo de espera) para obtener un mutex. En su lugar, utilice un tiempo de espera que sea un poco más largo que el tiempo máximo que es normal que tenga que

esperar al mutex. Si no se obtiene el mutex en ese tiempo, eso significa que hay un error de diseño, que podría ser un interbloqueo.

El interbloqueo no es un problema importante en pequeños sistemas integrados, porque si usted, como diseñador del sistema, conoce bien toda la aplicación, puede identificar y eliminar las áreas en las que podría producirse.

Mutex recursivos

También es posible que una tarea se interbloquee consigo misma. Esto ocurre si una tarea intenta tomar el mismo mutex más de una vez, sin antes devolverlo. Veamos la siguiente situación:

1. Una tarea obtiene un mutex correctamente.
2. Mientras tiene el mutex, la tarea llama a una función de biblioteca.
3. La implementación de la función de biblioteca intenta tomar el mismo mutex y pasa al estado Bloqueado a la espera de que el mutex esté disponible.

La tarea se encuentra en el estado Bloqueado a la espera de que se devuelva el mutex, pero la tarea ya tiene el mutex. Se ha producido un interbloqueo porque la tarea se encuentra en el estado Bloqueado esperándose a sí misma.

Puede evitar este tipo de interbloqueo mediante un mutex recursivo en lugar de un mutex estándar. Una misma tarea puede tomar un mutex recursivo más de una vez. Solo se devolverá después de que se haya ejecutado una llamada para dar el mutex recursivo por cada llamada anterior para tomar el mutex recursivo.

Los mutex estándar y los mutex recursivos se crean y se utilizan de forma similar:

- Los mutex estándar se crean con `xSemaphoreCreateMutex()`. Los mutex recursivos se crean con `xSemaphoreCreateRecursiveMutex()`. Las dos funciones de API tienen el mismo prototipo.
- Los mutex estándar se toman con `xSemaphoreTake()`. Los mutex recursivos se toman con `xSemaphoreTakeRecursive()`. Las dos funciones de API tienen el mismo prototipo.
- Los mutex estándar se dan con `xSemaphoreGive()`. Los mutex recursivos se dan con `xSemaphoreGiveRecursive()`. Las dos funciones de API tienen el mismo prototipo.

El siguiente código demuestra cómo crear y utilizar un mutex recursivo.

```
/* Recursive mutexes are variables of type SemaphoreHandle_t. */  
  
SemaphoreHandle_t xRecursiveMutex;  
  
/* The implementation of a task that creates and uses a recursive mutex. */  
  
void vTaskFunction( void *pvParameters )  
{  
  
    const TickType_t xMaxBlock20ms = pdMS_TO_TICKS( 20 );  
  
    /* Before a recursive mutex is used it must be explicitly created. */  
  
    xRecursiveMutex = xSemaphoreCreateRecursiveMutex();  
  
    /* Check the semaphore was created successfully. configASSERT() is described in section  
    11.2. */  

```

```
configASSERT( xRecursiveMutex );

/* As per most tasks, this task is implemented as an infinite loop. */

for( ;; )

{

    /* ... */

    /* Take the recursive mutex. */

    if( xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms ) == pdPASS)

    {

        /* The recursive mutex was successfully obtained. The task can now access the
        resource the mutex is protecting. At this point the recursive call count (which is the
        number of nested calls to xSemaphoreTakeRecursive()) is 1 because the recursive mutex has
        only been taken once. */

        /* While it already holds the recursive mutex, the task takes the mutex
        again. In a real application, this is only likely to occur inside a subfunction called
        by this task because there is no practical reason to knowingly take the same mutex
        more than once. The calling task is already the mutex holder, so the second call to
        xSemaphoreTakeRecursive() does nothing more than increment the recursive call count to 2.
        */

        xSemaphoreTakeRecursive( xRecursiveMutex, xMaxBlock20ms );

        /* ... */

        /* The task returns the mutex after it has finished accessing the resource the
        mutex is protecting. At this point the recursive call count is 2, so the first call to
        xSemaphoreGiveRecursive() does not return the mutex. Instead, it simply decrements the
        recursive call count back to 1. */

        xSemaphoreGiveRecursive( xRecursiveMutex );

        /* The next call to xSemaphoreGiveRecursive() decrements the recursive call
        count to 0, so this time the recursive mutex is returned.*/

        xSemaphoreGiveRecursive( xRecursiveMutex );

        /* Now one call to xSemaphoreGiveRecursive() has been executed for every
        proceeding call to xSemaphoreTakeRecursive(), so the task is no longer the mutex holder.
        */

    }

}

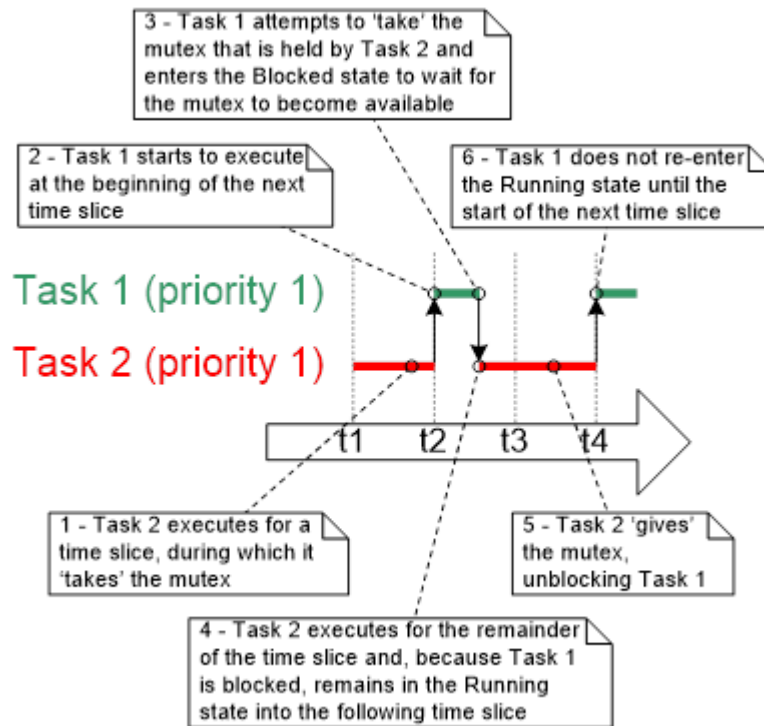
}
```

Mutex y programación de tareas

Si dos tareas de diferentes prioridades utilizan el mismo mutex, la política de programación de FreeRTOS deja claro el orden en que se ejecutan las tareas. Se seleccionará la tarea de máxima prioridad que pueda ejecutarse como la tarea que entra en el estado En ejecución. Por ejemplo, si una tarea de alta prioridad se encuentra en el estado Bloqueado esperando a un mutex que tiene una tarea de baja prioridad, la tarea de alta prioridad tendrá preferencia sobre la tarea de baja prioridad en cuanto la tarea de baja prioridad devuelva el mutex. La tarea de alta prioridad pasará a convertirse en el titular del mutex. Esta situación ya se ha explicado en la sección [Herencia de prioridades \(p. 172\)](#).

Es habitual presuponer de forma incorrecta el orden en que se ejecutarán las tareas cuando tienen la misma prioridad. Si las Tareas 1 y 2 tienen la misma prioridad y la tarea 1 está en el estado Bloqueado a la espera de un mutex que tiene la tarea 2, la tarea 1 no tendrá preferencia sobre la tarea 2 cuando la tarea 2 dé el mutex. En su lugar, la tarea 2 permanecerá en el estado En ejecución. La tarea 1 simplemente pasará del estado Bloqueado al estado Listo.

En esta figura, las líneas verticales marcan en qué momento se produce una interrupción de ciclo.



Como verá, el programador de FreeRTOS no pasa la tarea 1 al estado En ejecución en cuanto el mutex está disponible porque:

1. La tarea 1 y 2 tienen la misma prioridad, por lo que, a menos que la tarea 2 pase al estado Bloqueado, no se debería producir un cambio a la tarea 1 hasta la siguiente interrupción de ciclo (suponiendo que configUSE_TIME_SLICING esté establecido en 1 en FreeRTOSConfig.h).
2. Si una tarea utiliza un mutex en un bucle cerrado y se produce un cambio de contexto cada vez que la tarea da el mutex, la tarea permanecerá en el estado En ejecución durante un breve periodo de tiempo. Si dos o más tareas utilizan el mismo mutex en un bucle cerrado, se desperdiciará rápidamente tiempo de procesamiento al cambiar entre las tareas.

Si más de una tarea utiliza un mutex en un bucle cerrado y las tareas que utilizan el mutex tienen la misma prioridad, asegúrese de que las tareas reciben aproximadamente la misma cantidad de tiempo de procesamiento. La figura anterior muestra una secuencia de ejecución que podría producirse si se crean dos instancias de la tarea que se muestra en el siguiente código con la misma prioridad.

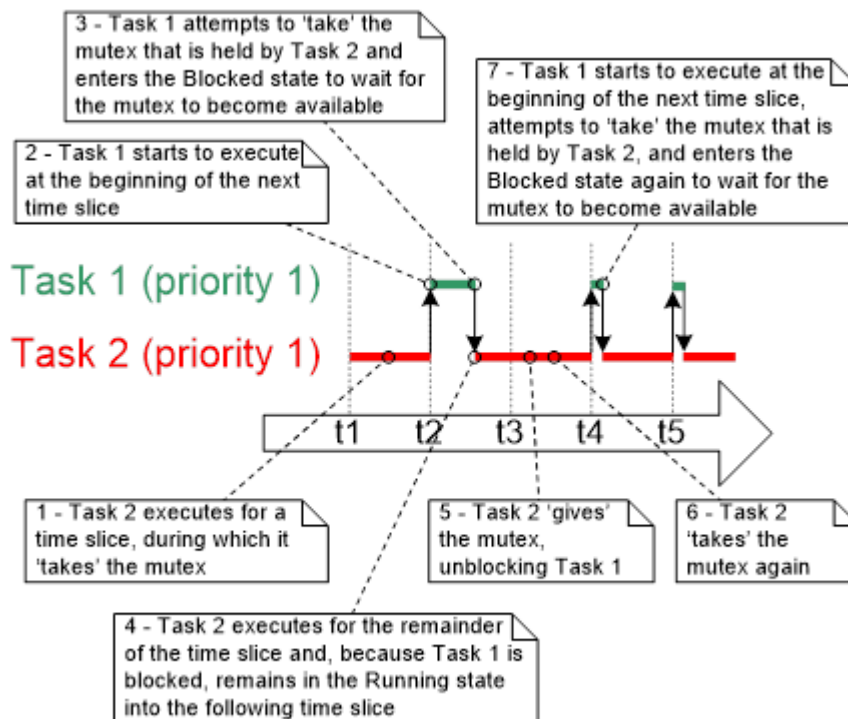
```
/* The implementation of a task that uses a mutex in a tight loop. The task creates a text
string in a local buffer, and then writes the string to a display. Access to the display
is protected by a mutex. */
```

```
void vATask( void *pvParameter )
```

```
{  
  
    extern SemaphoreHandle_t xMutex;  
  
    char cTextBuffer[ 128 ];  
  
    for( ;; )  
    {  
  
        /* Generate the text string. This is a fast operation. */  
        vGenerateTextInALocalBuffer( cTextBuffer );  
  
        /* Obtain the mutex that is protecting access to the display. */  
        xSemaphoreTake( xMutex, portMAX_DELAY );  
  
        /* Write the generated text to the display. This is a slow operation. */  
        vCopyTextToFrameBuffer( cTextBuffer );  
  
        /* The text has been written to the display, so return the mutex. */  
        xSemaphoreGive( xMutex );  
  
    }  
}
```

En los comentarios del código se advierte que la creación de la cadena es una operación rápida y la actualización de la pantalla es una operación lenta. Por lo tanto, dado que el mutex se mantiene mientras se está actualizando la pantalla, la tarea tendrá el mutex durante la mayor parte de su tiempo de ejecución.

En la siguiente figura, las líneas verticales marcan en qué momento se produce una interrupción de ciclo.



El paso 7 de esta figura muestra la tarea 1 volviendo a pasar al estado Bloqueado. Todo esto sucede en el interior la función de API `xSemaphoreTake()`.

La tarea 1 no podrá obtener el mutex hasta que el inicio de un intervalo de tiempo coincida con uno de los breves periodos de tiempo durante los cuales la tarea 2 no es el titular del mutex.

Para evitar esta situación, añada una llamada a `taskYIELD()` después de la llamada a `xSemaphoreGive()`. Esto se demuestra en el código que se muestra a continuación, donde se llama a `taskYIELD()` si el recuento de ciclos cambia mientras la tarea tiene el mutex. El código garantiza que las tareas que utilizan un mutex en un bucle reciban una cantidad más igualitaria de tiempo de procesamiento, además de asegurarse de que no se pierda tiempo de procesamiento al cambiar entre tareas con demasiada rapidez.

```
void vFunction( void *pvParameter )
{
    extern SemaphoreHandle_t xMutex;

    char cTextBuffer[ 128 ];

    TickType_t xTimeAtWhichMutexWasTaken;

    for( ;; )
    {
        /* Generate the text string. This is a fast operation. */
        vGenerateTextInALocalBuffer( cTextBuffer );

        /* Obtain the mutex that is protecting access to the display. */
        xSemaphoreTake( xMutex, portMAX_DELAY );

        /* Record the time at which the mutex was taken. */
        xTimeAtWhichMutexWasTaken = xTaskGetTickCount();

        /* Write the generated text to the display. This is a slow operation. */
        vCopyTextToFrameBuffer( cTextBuffer );

        /* The text has been written to the display, so return the mutex. */
        xSemaphoreGive( xMutex );

        /* If taskYIELD() was called on each iteration, then this task would only ever
        remain in the Running state for a short period of time, and processing time would be
        wasted by rapidly switching between tasks. Therefore, only call taskYIELD() if the tick
        count changed while the mutex was held. */

        if( xTaskGetTickCount() != xTimeAtWhichMutexWasTaken )
        {
            taskYIELD();
        }
    }
}
```

Tareas de guardián

Las tareas de guardián son un método claro para implementar la exclusión mutua sin el riesgo de que se produzca una inversión de prioridades o un interbloqueo.

Una tarea de guardián es una tarea que tiene la propiedad exclusiva de un recurso. La tarea de guardián es la única que tiene permitido el acceso al recurso directamente. Cualquier otra tarea que tenga que acceder al recurso solo puede hacerlo de forma indirecta mediante los servicios del guardián.

Reescritura de vPrintString() para utilizar una tarea de guardián (Ejemplo 21)

En este ejemplo, se proporciona otra implementación alternativa de vPrintString(). Esta vez, se utiliza una tarea de guardián para administrar el acceso a una salida estándar. Cuando una tarea quiere escribir un mensaje en una salida estándar, no llama directamente a una función de impresión. En su lugar, envía el mensaje al guardián.

La tarea de guardián utiliza una cola de FreeRTOS para serializar el acceso a una salida estándar. La implementación interna de la tarea no tiene que tener en cuenta la exclusión mutua porque es la única tarea a la que se le permite el acceso directo a la salida estándar.

La tarea de guardián pasa la mayor parte de su tiempo en el estado Bloqueado esperando que lleguen mensajes a la cola. Cuando llega un mensaje, el guardián simplemente escribe el mensaje en una salida estándar antes de volver al estado Bloqueado a esperar al siguiente mensaje.

Se pueden enviar interrupciones a las colas, por lo que las rutinas del servicio de interrupciones también pueden utilizar de forma segura los servicios de guardián para escribir mensajes en el terminal. En este ejemplo, se utiliza una función de enlace de ciclos para escribir un mensaje cada 200 ciclos.

Un enlace de ciclos (o devolución de llamada de ciclos) es una función que llama el kernel durante cada interrupción de ciclo. Para utilizar una función de enlace de ciclos:

1. En FreeRTOSConfig.h, establezca configUSE_TICK_HOOK en 1.
2. Proporcione la implementación de la función de enlace, utilizando el nombre de función exacto y el prototipo que se muestran aquí.

```
void vApplicationTickHook( void );
```

Las funciones de enlace de ciclos se ejecutan en el contexto de la interrupción de ciclo y, por tanto, deben ser muy cortas. Solo deben usar una cantidad moderada de espacio de pila y no deben llamar a ninguna función de API de FreeRTOS que no termine con "FromISR()".

Aquí se muestra la implementación de la tarea de guardián. El programador siempre se ejecuta inmediatamente después de la función de enlace de ciclos, por lo que las funciones de API de FreeRTOS a prueba de interrupciones que se llaman desde el enlace de ciclos no necesitan utilizar su parámetro pxHigherPriorityTaskWoken y el parámetro puede establecerse en NULL.

```
static void prvStdioGatekeeperTask( void *pvParameters )
{
    char *pcMessageToPrint;
```

```
/* This is the only task that is allowed to write to standard out. Any other task
wanting to write a string to the output does not access standard out directly, but
instead sends the string to this task. Because this is the only task that accesses
standard out, there are no mutual exclusion or serialization issues to consider within the
implementation of the task itself. */

for( ;; )

{

    /* Wait for a message to arrive. An indefinite block time is specified so there is
    no need to check the return value. The function will return only when a message has been
    successfully received. */

    xQueueReceive( xPrintQueue, &pcMessageToPrint, portMAX_DELAY );

    /* Output the received string. */

    printf( "%s", pcMessageToPrint );

    fflush( stdout );

    /* Loop back to wait for the next message. */

}

}
```

Aquí se muestra la tarea que escribe en la cola. Como antes, se crean dos instancias independientes de la tarea y la cadena que la tarea escribe en la cola se pasa a la tarea mediante el parámetro de tareas.

```
static void prvPrintTask( void *pvParameters )

{

    int iIndexToString;

    const TickType_t xMaxBlockTimeTicks = 0x20;

    /* Two instances of this task are created. The task parameter is used to pass an index
    into an array of strings into the task. Cast this to the required type. */

    iIndexToString = ( int ) pvParameters;

    for( ;; )

    {

        /* Print out the string, not directly, but by passing a pointer to the string to
        the gatekeeper task through a queue. The queue is created before the scheduler is started
        so will already exist by the time this task executes for the first time. A block time is
        not specified because there should always be space in the queue. */

        xQueueSendToBack( xPrintQueue, &(amp; pcStringsToPrint[ iIndexToString ] ), 0 );

        /* Wait a pseudo random time. Note that rand() is not necessarily reentrant, but in
        this case it does not really matter because the code does not care what value is returned.
        In a more secure application, a version of rand() that is known to be reentrant should be
        used or calls to rand() should be protected using a critical section. */

        vTaskDelay( ( rand() % xMaxBlockTimeTicks ) );

    }

}
```

```
}
```

La función de enlace de ciclos cuenta la cantidad de veces que se llama y envía su mensaje a la tarea de guardián cada vez que el recuento llega a 200. Solo para fines de demostración, el enlace de ciclos escribe en la parte delantera de la cola y las tareas escriben en la parte posterior de la cola. La implementación del enlace de ciclos se muestra aquí.

```
void vApplicationTickHook( void )
{
    static int iCount = 0;

    /* Print out a message every 200 ticks. The message is not written out directly, but
    sent to the gatekeeper task. */

    iCount++;

    if( iCount >= 200 )
    {
        /* Because xQueueSendToFrontFromISR() is being called from the tick hook, it is not
        necessary to use the xHigherPriorityTaskWoken parameter (the third parameter), and the
        parameter is set to NULL. */

        xQueueSendToFrontFromISR( xPrintQueue, &(amp; pcStringsToPrint[ 2 ] ), NULL );

        /* Reset the count ready to print out the string again in 200 ticks time. */

        iCount = 0;
    }
}
```

Como es habitual, la función main() crea las colas y las tareas necesarias para ejecutar el ejemplo y, a continuación, inicia el programador. Aquí se muestra la implementación de main().

```
/* Define the strings that the tasks and interrupt will print out via the gatekeeper. */
static char *pcStringsToPrint[] =
{
    "Task 1 *****\r\n", "Task 2
    -----\r\n", "Message printed from the tick
    hook interrupt #####\r\n"
};

/*-----*/

/* Declare a variable of type QueueHandle_t. The queue is used to send messages from the
print tasks and the tick interrupt to the gatekeeper task. */
QueueHandle_t xPrintQueue;

/*-----*/

int main( void )
{
```

```
/* Before a queue is used it must be explicitly created. The queue is created to hold a
maximum of 5 character pointers. */

xPrintQueue = xQueueCreate( 5, sizeof( char * ) );

/* Check the queue was created successfully. */

if( xPrintQueue != NULL )
{
    /* Create two instances of the tasks that send messages to the gatekeeper. The
    index to the string the task uses is passed to the task through the task parameter (the
    4th parameter to xTaskCreate()). The tasks are created at different priorities, so the
    higher priority task will occasionally preempt the lower priority task. */

    xTaskCreate( prvPrintTask, "Print1", 1000, ( void * ) 0, 1, NULL );

    xTaskCreate( prvPrintTask, "Print2", 1000, ( void * ) 1, 2, NULL );

    /* Create the gatekeeper task. This is the only task that is permitted to directly
    access standard out. */

    xTaskCreate( prvStdioGatekeeperTask, "Gatekeeper", 1000, NULL, 0, NULL
);

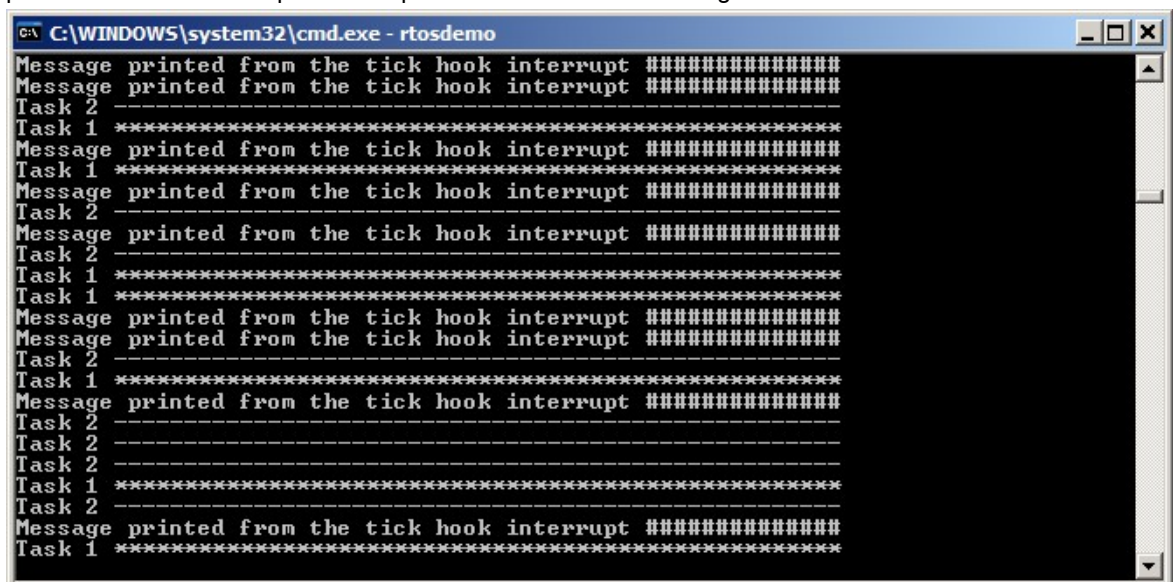
    /* Start the scheduler so the created tasks start executing. */

    vTaskStartScheduler();

}

/* If all is well, then main() will never reach here because the scheduler will
now be running the tasks. If main() does reach here, then it is likely that there was
insufficient heap memory available for the idle task to be created.*/
for( ;; );
}
```

El resultado se muestra aquí. Como verá, las cadenas procedentes de las tareas y las cadenas procedentes de la interrupción se imprimen correctamente sin ningún daño.



A la tarea de guardián se le asigna una prioridad menor que a las tareas de impresión, por lo que los mensajes que se envían al guardián permanecen en la cola hasta que ambas tareas de impresión se encuentran en el estado Bloqueado. En algunas situaciones, no es adecuado asignar al guardián una prioridad mayor para que los mensajes se procesen de inmediato. Si lo hace, el guardián retrasaría las tareas de menor prioridad hasta que haya terminado de acceder al recurso protegido.

Grupos de eventos

En esta sección se explica lo siguiente:

- Usos prácticos de los grupos de eventos
- Ventajas y desventajas de los grupos de eventos en relación con otras características de FreeRTOS
- Cómo configurar bits en un grupo de eventos
- Cómo esperar en el estado Bloqueado a que se configuren los bits en un grupo de eventos
- Cómo utilizar un grupo de eventos para sincronizar un conjunto de tareas

Los grupos de eventos son otra característica de FreeRTOS que permite comunicar eventos a las tareas. A diferencia de las colas y los semáforos, los grupos de eventos:

- Permiten que una tarea espere en el estado Bloqueado a que se produzca una combinación de uno o más eventos.
- Desbloquean todas las tareas que estaban a la espera del mismo evento, o combinación de eventos, cuando se produce el evento.

Estas propiedades únicas de los grupos de eventos hacen que sean útiles para sincronizar varias tareas y transmitir eventos a más de una tarea, lo que permite que una tarea espere en el estado Bloqueado a que se produzca un evento de un conjunto y que una tarea espere en el estado Bloqueado a que se completen varias acciones.

Asimismo, los grupos de eventos ofrecen la oportunidad de reducir la RAM que utiliza una aplicación porque, con frecuencia, es posible reemplazar numerosos semáforos binarios por un único grupo de eventos.

La funcionalidad de los grupos de eventos es opcional. Para incluir la funcionalidad de los grupos de eventos, incluya el archivo de origen de FreeRTOS `event_groups.c` como parte de su proyecto.

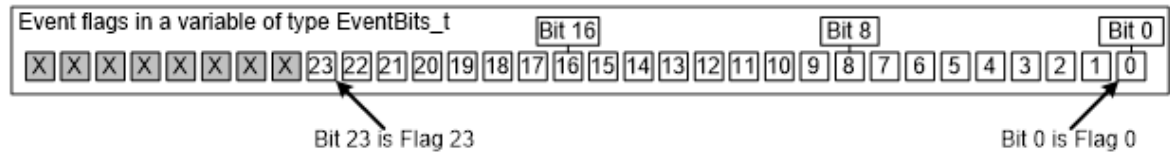
Características de un grupo de eventos

Grupos de eventos, marcas de eventos y bits de eventos

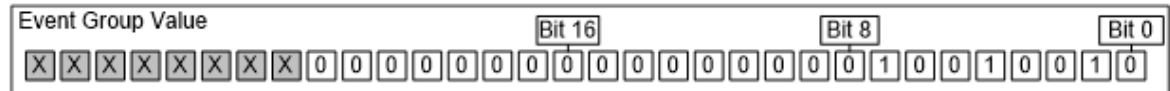
Una marca de eventos es un valor booleano (1 o 0) que se utiliza para indicar si se ha producido un evento. Un grupo de eventos es un conjunto de marcas de eventos.

Una marca de eventos solo puede ser 1 o 0, lo que permite almacenar su estado en un único bit y almacenar el estado de todas las marcas de eventos de un grupo de eventos en una única variable. El estado de cada marca de eventos de un grupo de eventos está representado por un solo bit en una variable de tipo `EventBits_t`. Por este motivo, las marcas de eventos también se conocen como bits de eventos. Si un bit está establecido en 1 en la variable `EventBits_t`, eso significa que el evento representado por ese bit se ha producido. Si un bit está establecido en 0 en la variable `EventBits_t`, eso significa que el evento representado por ese bit no se ha producido.

Esta figura muestra cómo se asignan marcas de eventos individuales a bits individuales en una variable de tipo `EventBits_t`.



Por ejemplo, si el valor de un grupo de eventos es 0x92 (1001 0010 binario), solo se establecen los bits de eventos 1, 4 y 7, por lo que únicamente se han producido los eventos representados por los bits 1, 4 y 7. Esta figura muestra una variable de tipo EventBits_t que tiene establecidos los bits de eventos 1, 4 y 7 y todos los demás bits de eventos se han borrado, por lo que el grupo de eventos tiene un valor de 0x92.



Corresponde al programador de la aplicación asignar un significado a los bits individuales de un grupo de eventos. Por ejemplo, el programador de la aplicación podría crear un grupo de eventos y, a continuación:

- Definir que el bit 0 del grupo de eventos signifique que se ha recibido un mensaje de la red.
- Definir que el bit 1 del grupo de eventos signifique que hay un mensaje listo para ser enviado a la red.
- Definir que el bit 2 del grupo de eventos signifique que se anula la conexión de red actual.

Más acerca del tipo de datos EventBits_t

El número de bits de eventos de un grupo de eventos depende de la constante de configuración en tiempo de compilación configUSE_16_BIT_TICKS en FreeRTOSConfig.h. Esta constante configura el tipo que se utiliza para almacenar el recuento de ciclos de RTOS, por lo que no parece estar relacionada con la característica de los grupos de eventos. Su efecto en el tipo EventBits_t es una consecuencia de la implementación interna y deseable de FreeRTOS, porque configUSE_16_BIT_TICKS solo debe establecerse en 1 cuando FreeRTOS se esté ejecutando en una arquitectura que pueda administrar tipos de 16 bits de forma más eficaz que los tipos de 32 bits.

- Si configUSE_16_BIT_TICKS es 1, eso significa que cada grupo de eventos contiene 8 bits de eventos que se pueden utilizar.
- Si configUSE_16_BIT_TICKS es 0, eso significa que cada grupo de eventos contiene 24 bits de eventos que se pueden utilizar.

Acceso de varias tareas

Los grupos de eventos son objetos por derecho propio a los que puede acceder cualquier tarea o ISR que conozca su existencia. Cualquier número de tareas puede establecer bits en el mismo grupo de eventos y cualquier número de tareas puede leer bits desde el mismo grupo de eventos.

Ejemplo práctico del uso de un grupo de eventos

La implementación de la pila TCP/IP de FreeRTOS+TCP es un buen ejemplo práctico de cómo se puede utilizar un grupo de eventos para simplificar simultáneamente un diseño y minimizar el uso de los recursos.

Un conector TCP debe responder a muchos eventos diferentes, incluidos los eventos de aceptación, de vinculación, de lectura y de cierre. Los eventos que puede esperar recibir un socket en cualquier momento dado dependen del estado del socket. Por ejemplo, si se ha creado un socket, pero aún no se ha vinculado a una dirección, puede esperar recibir un evento de enlace, pero no un evento de lectura. (No puede leer datos si no tiene una dirección).

El estado de un socket de FreeRTOS+TCP se guarda en una estructura denominada FreeRTOS_Socket_t. La estructura contiene un grupo de eventos que tiene definido un bit de eventos por cada evento que debe procesar el socket. Las llamadas a la API de FreeRTOS+TCP que se bloquean a la espera de que se produzca un evento o un grupo de eventos simplemente se bloquean en el grupo de eventos.

El grupo de eventos también contiene un bit de "anulación", que permite anular una conexión TCP independientemente del evento que esté esperando el socket en ese momento.

Administración de eventos mediante grupos de eventos

Función de API xEventGroupCreate()

FreeRTOS V9.0.0 también incluye la función xEventGroupCreateStatic(), que asigna la memoria necesaria para crear un grupo de eventos de forma estática durante la compilación. El grupo de eventos debe crearse de forma explícita para poder utilizarlo.

Se hace referencia a los grupos de eventos con variables de tipo EventGroupHandle_t. La función de API xEventGroupCreate() se utiliza para crear un grupo de eventos. Devuelve un EventGroupHandle_t para hacer referencia al grupo de eventos que crea.

Aquí se muestra el prototipo de la función de API xEventGroupCreate().

```
EventGroupHandle_t xEventGroupCreate( void );
```

En la siguiente tabla se muestra el valor de retorno de xEventGroupCreate().

Nombre del parámetro	Descripción
Valor de retorno	<p>Si se devuelve NULL, el grupo de eventos no se puede crear porque no hay suficiente memoria disponible en el montón para que FreeRTOS asigne las estructuras de datos del grupo de eventos. Para obtener más información, consulte Administración de memoria en montón (p. 14).</p> <p>Si se devuelve un valor NULL, eso significa que el grupo de eventos se ha creado correctamente. El valor devuelto debe almacenarse como el controlador del grupo de eventos creado.</p>

Función de API xEventGroupSetBits()

La función de API xEventGroupSetBits() establece uno o varios bits en un grupo de eventos. Suele utilizarse para notificar a una tarea que se han producido los eventos que representa el bit o los bits que se han establecido.

Nota: No llame a xEventGroupSetBits() desde una rutina del servicio de interrupciones. En su lugar, utilice la versión a prueba de interrupciones, xEventGroupSetBitsFromISR().

Aquí se muestra el prototipo de la función de API xEventGroupSetBits().

```
EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup, const EventBits_t  
uxBitsToSet );
```

En la siguiente tabla se muestran los parámetros de xEventGroupSetBits() y el valor de retorno.

Nombre del parámetro	Descripción
xEventGroup	El controlador del grupo de eventos en el que se están estableciendo los bits. El controlador del grupo de eventos se ha devuelto desde la llamada a xEventGroupCreate(), que se ha utilizado para crear el grupo de eventos.
uxBitsToSet	<p>Una máscara de bits que especifica el bit o los bits del evento que se va a establecer en 1 en el grupo de eventos. El valor del grupo de eventos se actualiza mediante ORing bit a bit para cambiar el valor existente del grupo de eventos por el valor que se pasa en uxBitsToSet.</p> <p>Por ejemplo, al establecer uxBitsToSet en 0x04 (0100 binario), dará como resultado que se establezca el bit de eventos 3 en el grupo de eventos (si aún no se ha establecido), mientras que todos los demás bits de eventos del grupo de eventos permanecen sin cambios.</p>
Valor devuelto	El valor del grupo de eventos en el momento en que se devuelve la llamada a xEventGroupSetBits(). El valor devuelto no tendrá establecidos necesariamente los bits especificados por uxBitsToSet, ya que es posible que otra tarea haya vuelto a borrar los bits.

Función de API xEventGroupSetBitsFromISR()

xEventGroupSetBitsFromISR() es la versión a prueba de interrupciones de xEventGroupSetBits().

Dar un semáforo es una operación determinista, ya que se sabe de antemano que, al dar un semáforo, lo máximo que puede ocurrir es que una tarea salga del estado Bloqueado. El establecimiento de bits en un grupo de eventos no es una operación determinista, porque cuando se establecen bits en un grupo de eventos, no se sabe de antemano cuántas tareas saldrán del estado Bloqueado.

El estándar de diseño e implementación de FreeRTOS no permite realizar operaciones no deterministas dentro de una rutina del servicio de interrupciones o cuando las interrupciones están deshabilitadas. Por este motivo, xEventGroupSetBitsFromISR() no establece bits de eventos directamente dentro de la rutina del servicio de interrupciones. En lugar de ello, difiere la acción a la tarea de demonio de RTOS.

Aquí se muestra el prototipo de la función de API xEventGroupSetBitsFromISR().

```
BaseType_t xEventGroupSetBitsFromISR( EventGroupHandle_t xEventGroup, const EventBits_t  
uxBitsToSet, BaseType_t *pxHigherPriorityTaskWoken );
```

En la siguiente tabla se muestran los parámetros de xEventGroupSetBitsFromISR() y el valor de retorno.

xEventGroup

El controlador del grupo de eventos en el que se están estableciendo los bits. El controlador del grupo de eventos se ha devuelto desde la llamada a xEventGroupCreate(), que se ha utilizado para crear el grupo de eventos.

uxBitsToSet

Una máscara de bits que especifica el bit o los bits de eventos que se van a establecer en 1 en el grupo de eventos. El valor del grupo de eventos se actualiza mediante ORing bit a bit para cambiar el valor existente del grupo de eventos por el valor que se pasa a uxBitsToSet. Por ejemplo, al establecer uxBitsToSet en 0x05 (0101 binario), dará como resultado que se establezca el bit de eventos 3 y el bit de eventos 0 (si aún no se han establecido), mientras que todos los demás bits de eventos del grupo de eventos permanecen sin cambios.

pxHigherPriorityTaskWoken

xEventGroupSetBitsFromISR() no establece los bits de eventos directamente dentro de la rutina del servicio de interrupciones. En lugar de ello, difiere la acción a la tarea de demonio de RTOS enviando un comando en la cola de comandos del temporizador. Si a tarea de demonio estaba en estado Bloqueado a la espera de que los datos estén disponibles en la cola de comandos del temporizador, al escribir en la cola de comandos del temporizador, la tarea de demonio saldrá del estado Bloqueado. Si la prioridad de la tarea de demonio es superior a la prioridad de la tarea que se está ejecutando actualmente (la tarea que se ha interrumpido), xEventGroupSetBitsFromISR() establecerá internamente pxHigherPriorityTaskWoken en pdTRUE.

Si xEventGroupSetBitsFromISR() establece este valor en pdTRUE, debe efectuarse un cambio de contexto antes de salir de la interrupción. De este modo, se garantiza que la interrupción se devuelve directamente a la tarea de demonio, porque esta será la tarea con el estado Listo de mayor prioridad.

Hay dos valores de retorno posibles:

pdPASS solo se devuelve si los datos se han enviado correctamente a la cola de comandos del temporizador.

pdFALSE se devuelve si el comando "set bits" no se ha podido escribir en la cola de comandos del temporizador porque ya estaba llena.

Función de API xEventGroupWaitBits()

La función de API xEventGroupWaitBits() permite que una tarea lea el valor de un grupo de eventos y, de forma opcional, que espere en estado Bloqueado a que se establezcan uno o varios bits de eventos en el grupo de eventos en caso de que aún no se hayan establecido.

Aquí se muestra el prototipo de la función de API xEventGroupWaitBits().

```
EventBits_t xEventGroupWaitBits( const EventGroupHandle_t xEventGroup, const EventBits_t
    uxBitsToWaitFor, const BaseType_t xClearOnExit, const BaseType_t xWaitForAllBits,
    TickType_t xTicksToWait );
```

La condición de desbloqueo es la condición que utiliza el programador para determinar si una tarea se pondrá en estado Bloqueado y cuándo saldrá de ese estado. La condición de desbloqueo se especifica mediante una combinación de los valores de los parámetros uxBitsToWaitFor y xWaitForAllBits:

- uxBitsToWaitFor, que especifica los bits de eventos en el grupo de eventos que se va a probar.
- xWaitForAllBits, que especifica si se debe usar una prueba O una operación bit a bit o una prueba Y una operación bit a bit.

Una tarea no se pondrá en el estado Bloqueado si se cumple su condición de desbloqueo en el momento de llamar a xEventGroupWaitBits().

En la siguiente tabla, se proporcionan ejemplos de las condiciones que producen que una tarea entre o salga del estado Bloqueado. Solo muestra los cuatro bits binarios menos significativos del grupo de eventos y los valores de uxBitsToWaitFor. Se presupone que los demás bits de estos dos valores son cero.

Valor existente del grupo de eventos	Valor de uxBitsToWaitFor	Valor de xWaitForAllBits	Comportamiento resultante
0000	0101	pdFALSE	La tarea que realiza la llamada entra en el estado Bloqueado porque no se ha establecido el bit 0 ni el bit 2 en el grupo de eventos y sale del estado Bloqueado cuando el bit 0 O el bit 2 se establecen en el grupo de eventos.
0100	0101	pdTRUE	La tarea que realiza la llamada entra en el estado Bloqueado porque no se han establecido los dos bits, el bit 0 y el bit 2, en el grupo de eventos y sale del estado Bloqueado cuando los dos bits, el bit 0 Y el bit 2, se establecen en el grupo de eventos.
0100	0110	pdFALSE	La tarea que realiza la llamada no se pondrá en el estado Bloqueado porque xWaitForAllBits es pdFALSE y uno de los dos bits que especifica uxBitsToWaitFor ya está establecido en el grupo de eventos.
0100	0110	pdTRUE	La tarea que realiza la llamada se pondrá en el estado Bloqueado porque xWaitForAllBits es pdTRUE y solo uno de los dos bits que especifica uxBitsToWaitFor ya está establecido en el grupo de eventos. La tarea saldrá del estado Bloqueado cuando

		ambos bits, el 2 y el 3, estén establecidos en el grupo de eventos.
--	--	---

La tarea que realiza la llamada especifica los bits que se van a probar con el parámetro `uxBitsToWaitFor`. Lo más probable que es la tarea que realiza la llamada tenga que borrar estos bits para volver a cero después de que se haya cumplido su condición de desbloqueo. Los bits de eventos pueden borrarse con la función de API `xEventGroupClearBits()`, pero si se usa esa función para borrar manualmente los bits de eventos, se producirán condiciones de carrera en el código de la aplicación si:

- Existe más de una tarea que utiliza el mismo grupo de eventos.
- Se establecen bits en el grupo de eventos por medio de otra tarea o una ISR.

El parámetro `xClearOnExit` tiene como finalidad evitar estas condiciones de carrera potenciales. Si `xClearOnExit` se establece en `pdTRUE`, las pruebas y el borrado de bits de eventos aparecen a la tarea que realiza la llamada como una operación atómica (que no pueden interrumpir otras tareas o interrupciones).

En la siguiente tabla se muestran los parámetros de `xEventGroupWaitBits()` y el valor de retorno.

Nombre del parámetro	Descripción
<code>xEventGroup</code>	El controlador del grupo de eventos que contiene los bits de eventos que se leen. El controlador del grupo de eventos se ha devuelto desde la llamada a <code>xEventGroupCreate()</code> , que se ha utilizado para crear el grupo de eventos.
<code>uxBitsToWaitFor</code>	Una máscara de bits que especifica el bit o los bits de eventos que se van a probar en el grupo de eventos. Por ejemplo, si la tarea que realiza la llamada desea esperar a que el bit de eventos 0 y/o el bit de eventos 2 se establezcan en el grupo de eventos, establezca <code>uxBitsToWaitFor</code> en <code>0x05</code> (<code>0101</code> binario). Consulte la Tabla 45 para ver más ejemplos.
<code>xClearOnExit</code>	Si se ha cumplido la condición de desbloqueo de la tarea que realiza la llamada y <code>xClearOnExit</code> está establecido en <code>pdTRUE</code> , los bits de eventos especificados por <code>uxBitsToWaitFor</code> se borrarán para volver a ser 0 en el grupo de eventos antes de que la tarea que realiza la llamada salga de la función de API <code>xEventGroupWaitBits()</code> . Si <code>xClearOnExit</code> está establecido en <code>pdFALSE</code> , la función de API <code>xEventGroupWaitBits()</code> no modifica el estado de los bits de eventos en el grupo de eventos.
<code>xWaitForAllBits</code>	El parámetro <code>uxBitsToWaitFor</code> especifica los bits de eventos que se van a probar en el grupo de eventos. <code>xWaitForAllBits</code> especifica si la tarea que realiza la llamada debe salir del estado

	<p>Bloqueado cuando se establecen uno o más de los bits de eventos especificados por el parámetro <code>uxBitsToWaitFor</code> o solo cuando se establezcan todos los bits de eventos especificados por el parámetro <code>uxBitsToWaitFor</code>.</p> <p>Si <code>xWaitForAllBits</code> está establecido en <code>pdFALSE</code>, una tarea que haya entrado en el estado Bloqueado a la espera de que se cumpla la condición de desbloqueo saldrá de este estado cuando se establezcan cualquiera de los bits que especifica <code>uxBitsToWaitFor</code> (o se agote el tiempo de espera especificado por el parámetro <code>xTicksToWait</code>).</p> <p>Si <code>xWaitForAllBits</code> está establecido en <code>pdTRUE</code>, una tarea que haya entrado en el estado Bloqueado a la espera de que se cumpla la condición de desbloqueo solo saldrá de ese estado cuando se establezcan todos los bits que especifica <code>uxBitsToWaitFor</code> (o se agote el tiempo de espera especificado por el parámetro <code>xTicksToWait</code>).</p> <p>Para ver ejemplos, consulte la tabla anterior.</p>
<code>xTicksToWait</code>	<p>El tiempo máximo durante el que la tarea debe permanecer en el estado Bloqueado a la espera de que se cumpla la condición de desbloqueo.</p> <p><code>xEventGroupWaitBits()</code> se devolverá inmediatamente si <code>xTicksToWait</code> es cero o se cumple la condición de desbloqueo en el momento en que se llama a <code>xEventGroupWaitBits()</code>.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. Se puede usar la macro <code>pdMS_TO_TICKS()</code> para convertir un tiempo especificado en milisegundos en un tiempo especificado en ciclos.</p> <p>Al establecer <code>xTicksToWait</code> en <code>portMAX_DELAY</code>, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando <code>INCLUDE_vTaskSuspend</code> esté establecido en 1 en <code>FreeRTOSConfig.h</code>.</p>

Valor devuelto

Si se ha devuelto `xEventGroupWaitBits()` porque se ha cumplido la condición de desbloqueo de la tarea que realiza la llamada, el valor devuelto es el valor del grupo de eventos en el momento en que se cumple la condición de desbloqueo de la tarea que realiza la llamada (antes de que se haya borrado automáticamente algún bit si `xClearOnExit` era `pdTRUE`). En este caso, el valor devuelto también cumplirá la condición de desbloqueo.

Si se devuelve `xEventGroupWaitBits()` porque el tiempo de bloqueo especificado por el parámetro `xTicksToWait` ha vencido, el valor devuelto es el valor del grupo de eventos en el momento en que ha vencido el tiempo del bloqueo. En este caso, el valor devuelto no cumplirá la condición de desbloqueo.

Experimentación con grupos de eventos (Ejemplo 22)

En este ejemplo, puede ver cómo:

- Crear un grupo de eventos.
- Establecer bits en un grupo de eventos desde una ISR.
- Establecer bits en un grupo de eventos desde una tarea.
- Bloquear un grupo de eventos.

El efecto del parámetro `xWaitForAllBits` de `xEventGroupWaitBits()` se demuestra ejecutando primero el ejemplo con `xWaitForAllBits` establecido en `pdFALSE` y, a continuación, ejecutando el ejemplo con `xWaitForAllBits` establecido en `pdTRUE`.

El bit de eventos 0 y el bit de eventos 1 se establecen desde una tarea. El bit de eventos 2 se establece desde una ISR. Estos tres bits reciben nombres descriptivos utilizando las instrucciones `#define` que se muestran aquí.

```
/* Definitions for the event bits in the event group. */  
  
#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, which is set by a task. */  
  
#define mainSECOND_TASK_BIT ( 1UL << 1UL ) /* Event bit 1, which is set by a task. */  
  
#define mainISR_BIT ( 1UL << 2UL ) /* Event bit 2, which is set by an ISR. */
```

El código siguiente muestra la implementación de la tarea que establece el bit de eventos 0 y el bit de eventos 1. Se queda en un bucle, estableciendo de manera repetida un bit y luego otro, con un retardo de 200 milisegundos entre cada llamada a `xEventGroupSetBits()`. Se imprime una cadena antes de establecer cada bit para permitir que la secuencia de ejecución se vea en la consola.

```
static void vEventBitSettingTask( void *pvParameters )  
{  
  
    const TickType_t xDelay200ms = pdMS_TO_TICKS( 200UL ), xDontBlock = 0;
```

```
for( ;; )
{
    /* Delay for a short while before starting the next loop. */
    vTaskDelay( xDelay200ms );

    /* Print out a message to say event bit 0 is about to be set by the task, and then
    set event bit 0. */
    vPrintString( "Bit setting task -\t about to set bit 0.\r\n" );
    xEventGroupSetBits( xEventGroup, mainFIRST_TASK_BIT );

    /* Delay for a short while before setting the other bit. */
    vTaskDelay( xDelay200ms );

    /* Print out a message to say event bit 1 is about to be set by the task, and then
    set event bit 1. */
    vPrintString( "Bit setting task -\t about to set bit 1.\r\n" );
    xEventGroupSetBits( xEventGroup, mainSECOND_TASK_BIT );
}
}
```

El código siguiente muestra la implementación de la rutina del servicio de interrupciones que establece el bit 2 en el grupo de eventos. Una vez más, se imprime una cadena antes de establecer el bit para permitir que la secuencia de ejecución se vea en la consola. En este caso, dado que la salida de la consola no debe realizarse directamente en una rutina del servicio de interrupciones, `xTimerPendFunctionCallFromISR()` se utiliza para realizar la salida en el contexto de la tarea de demonio de RTOS.

Como en ejemplos anteriores, la rutina del servicio de interrupciones se activa por medio de una sencilla tarea periódica que fuerza una interrupción del software. En este ejemplo, la interrupción se genera cada 500 milisegundos.

```
static uint32_t ulEventBitSettingISR( void )
{
    /* The string is not printed within the interrupt service routine, but is instead
    sent to the RTOS daemon task for printing. It is therefore declared static to ensure the
    compiler does not allocate the string on the stack of the ISR because the ISR's stack
    frame will not exist when the string is printed from the daemon task. */

    static const char *pcString = "Bit setting ISR -\t about to set bit 2.\r\n";

    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Print out a message to say bit 2 is about to be set. Messages cannot be printed from
    an ISR, so defer the actual output to the RTOS daemon task by pending a function call to
    run in the context of the RTOS daemon task. */

    xTimerPendFunctionCallFromISR( vPrintStringFromDaemonTask, ( void * ) pcString, 0,
    &xHigherPriorityTaskWoken );

    /* Set bit 2 in the event group. */
}
```



```
xEventGroupSetBitsFromISR( xEventGroup, mainISR_BIT, &xHigherPriorityTaskWoken );

/* xTimerPendFunctionCallFromISR() and xEventGroupSetBitsFromISR() both write to the
timer command queue, and both used the same xHigherPriorityTaskWoken variable. If writing
to the timer command queue resulted in the RTOS daemon task leaving the Blocked state,
and if the priority of the RTOS daemon task is higher than the priority of the currently
executing task (the task this interrupt interrupted), then xHigherPriorityTaskWoken
will have been set to pdTRUE. xHigherPriorityTaskWoken is used as the parameter
to portYIELD_FROM_ISR(). If xHigherPriorityTaskWoken equals pdTRUE, then calling
portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
this function does not explicitly return a value. */

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

El código siguiente muestra la implementación de la tarea que llama a `xEventGroupWaitBits()` para realizar el bloqueo en el grupo de eventos. La tarea imprime una cadena para cada bit que se establece en el grupo de eventos.

El parámetro `xClearOnExit` de `xEventGroupWaitBits()` está establecido en `pdTRUE`, de modo que el bit o los bits de eventos que hicieron que se devolviera la llamada a `xEventGroupWaitBits()` se borrarán automáticamente antes de que se devuelva `xEventGroupWaitBits()`.

```
static void vEventBitReadingTask( void *pvParameters )
{
    EventBits_t xEventGroupValue;

    const EventBits_t xBitsToWaitFor = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
mainISR_BIT );

    for( ;; )
    {
        /* Block to wait for event bits to become set within the event group.*/

        xEventGroupValue = xEventGroupWaitBits( /* The event group to read. */
xEventGroup, /* Bits to test. */ xBitsToWaitFor, /* Clear bits on exit if the unblock
condition is met. */ pdTRUE, /* Don't wait for all bits. This parameter is set to pdTRUE
for the second execution. */ pdFALSE, /* Don't time out. */ portMAX_DELAY );

        /* Print a message for each bit that was set. */

        if( ( xEventGroupValue & mainFIRST_TASK_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 0 was set\r\n" );
        }

        if( ( xEventGroupValue & mainSECOND_TASK_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 1 was set\r\n" );
        }
    }
}
```

```
        if( ( xEventGroupValue & mainISR_BIT ) != 0 )
        {
            vPrintString( "Bit reading task -\t Event bit 2 was set\r\n" );
        }
    }
}
```

La función `main()` crea el grupo de eventos y las tareas antes de iniciar el programador. En el siguiente código, se muestra su implementación. La prioridad de la tarea que lee el grupo de eventos es mayor que la prioridad de la tarea que escribe en el grupo de eventos, lo que garantiza que la tarea de lectura tendrá preferencia sobre la tarea de escritura cada vez que se cumpla la condición de desbloqueo de la tarea de lectura.

```
int main( void )
{
    /* Before an event group can be used it must first be created. */
    xEventGroup = xEventGroupCreate();

    /* Create the task that sets event bits in the event group. */
    xTaskCreate( vEventBitSettingTask, "Bit Setter", 1000, NULL, 1, NULL );

    /* Create the task that waits for event bits to get set in the event group. */
    xTaskCreate( vEventBitReadingTask, "Bit Reader", 1000, NULL, 2, NULL );

    /* Create the task that is used to periodically generate a software interrupt. */
    xTaskCreate( vInterruptGenerator, "Int Gen", 1000, NULL, 3, NULL );

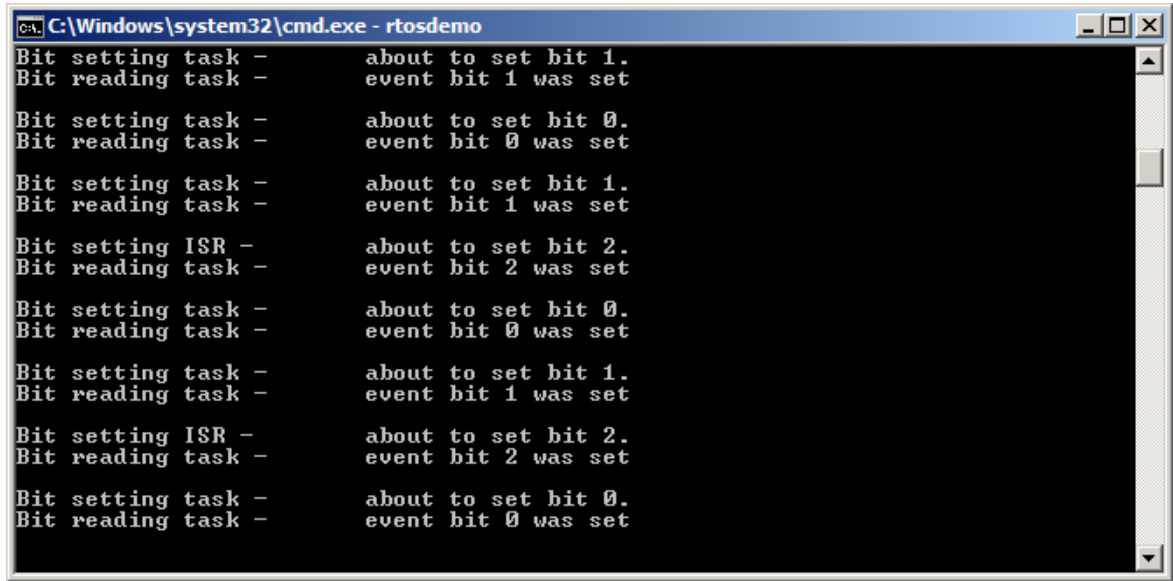
    /* Install the handler for the software interrupt. The syntax required to do this is
    depends on the FreeRTOS port being used. The syntax shown here can only be used with the
    FreeRTOS Windows port, where such interrupts are only simulated. */
    vPortSetInterruptHandler( mainINTERRUPT_NUMBER, ulEventBitSettingISR );

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

    /* The following line should never be reached. */
    for( ;; );

    return 0;
}
```

Aquí se muestra la salida que se obtiene cuando se ejecuta el Ejemplo 22 con el parámetro `xWaitForAllBits` de `xEventGroupWaitBits()` establecido en `pdFALSE`. Como verá, dado que el parámetro `xWaitForAllBits` en la llamada a `xEventGroupWaitBits()` está establecido en `pdFALSE`, la tarea que lee del grupo de eventos sale del estado Bloqueado y se ejecuta inmediatamente cada vez que se establece cualquiera de los bits de eventos.



```
C:\Windows\system32\cmd.exe - rtdemo
Bit setting task -      about to set bit 1.
Bit reading task -      event bit 1 was set

Bit setting task -      about to set bit 0.
Bit reading task -      event bit 0 was set

Bit setting task -      about to set bit 1.
Bit reading task -      event bit 1 was set

Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 2 was set

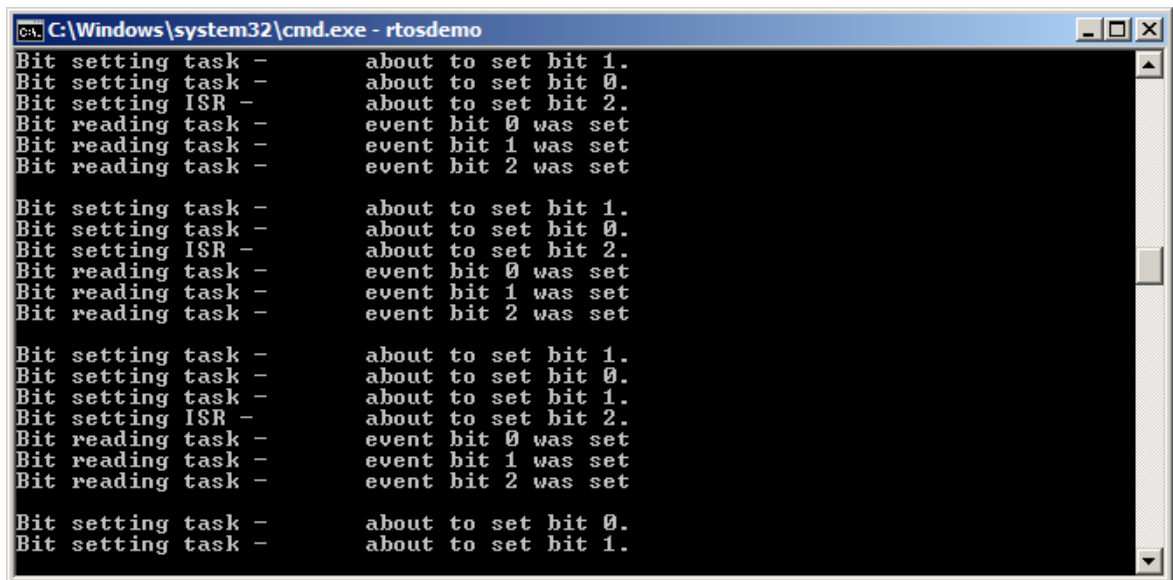
Bit setting task -      about to set bit 0.
Bit reading task -      event bit 0 was set

Bit setting task -      about to set bit 1.
Bit reading task -      event bit 1 was set

Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 0.
Bit reading task -      event bit 0 was set
```

Aquí se muestra la salida que se obtiene cuando se ejecuta el código con el parámetro `xWaitForAllBits` de `xEventGroupWaitBits()` establecido en `pdTRUE`. Como verá, dado que el parámetro `xWaitForAllBits` se ha establecido en `pdTRUE`, la tarea que lee desde el grupo de eventos solo sale del estado Bloqueado cuando se han establecido los tres bits de eventos.



```
C:\Windows\system32\cmd.exe - rtdemo
Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 0 was set
Bit reading task -      event bit 1 was set
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 0 was set
Bit reading task -      event bit 1 was set
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 1.
Bit setting task -      about to set bit 0.
Bit setting task -      about to set bit 1.
Bit setting ISR -       about to set bit 2.
Bit reading task -      event bit 0 was set
Bit reading task -      event bit 1 was set
Bit reading task -      event bit 2 was set

Bit setting task -      about to set bit 0.
Bit setting task -      about to set bit 1.
```

Sincronización de tareas mediante un grupo de eventos

A veces, para el diseño de una aplicación, es necesario sincronizar entre sí dos o más tareas. Por ejemplo, vamos a imaginarnos un diseño donde la Tarea A recibe un evento y, a continuación, delega parte del procesamiento que exige el evento a otras tres tareas: la Tarea B, la Tarea C y la Tarea D. Si la Tarea A no puede recibir otro evento hasta que las Tareas B, C y D hayan completado el procesamiento del evento anterior y, a continuación, las cuatro tareas tendrán que sincronizarse entre sí. El punto de sincronización

de cada tarea se produce después de que dicha tarea haya completado su procesamiento y no puede continuar hasta que cada una de las otras tareas hayan hecho lo mismo. La Tarea A solo puede recibir otro evento después de que las cuatro tareas hayan alcanzado su punto de sincronización.

Encontrará un ejemplo menos abstracto de la necesidad de utilizar este tipo de sincronización de tareas en uno de los proyectos de demostración de FreeRTOS+TCP. La demostración comparte un socket TCP entre dos tareas. Una tarea envía datos al socket y la otra recibe datos de ese mismo socket. (Actualmente, esta es la única forma de que un solo socket de FreeRTOS+TCP se pueda compartir entre tareas). No es seguro que ninguna tarea cierre el socket TCP hasta que esté garantizado que la otra tarea no intentará obtener acceso de nuevo al socket. Si una de las dos tareas quiere cerrar el socket, debe informar a la otra tarea de su intención y esperar a que la otra tarea deje de utilizar el socket antes de continuar.

La situación en la que se encuentra la tarea que envía datos al socket y que desea cerrarlo es trivial, ya que solo hay dos tareas que deben sincronizarse entre sí. Es fácil ver cómo se complicaría la situación y sería necesario que más tareas se unieran a la sincronización si otras tareas realizaran el procesamiento que depende de que el socket esté abierto.

```
void SocketTxTask( void *pvParameters )
{
    xSocket_t xSocket;

    uint32_t ulTxCount = 0UL;

    for( ;; )
    {
        /* Create a new socket. This task will send to this socket, and another task
        will receive from this socket. */

        xSocket = FreeRTOS_socket( ... );

        /* Connect the socket. */

        FreeRTOS_connect( xSocket, ... );

        /* Use a queue to send the socket to the task that receives data. */

        xQueueSend( xSocketPassingQueue, &xSocket, portMAX_DELAY );

        /* Send 1000 messages to the socket before closing the socket. */

        for( ulTxCount = 0; ulTxCount < 1000; ulTxCount++ )
        {
            if( FreeRTOS_send( xSocket, ... ) < 0 )
            {
                /* Unexpected error - exit the loop, after which the socket
                will be closed. */

                break;
            }
        }

        /* Let the Rx task know the Tx task wants to close the socket. */
    }
}
```

```
TxTaskWantsToCloseSocket();

/* This is the Tx task's synchronization point. The Tx task waits here
for the Rx task to reach its synchronization point. The Rx task will only reach its
synchronization point when it is no longer using the socket, and the socket can be closed
safely. */

xEventGroupSync( ... );

/* Neither task is using the socket. Shut down the connection, and then close
the socket. */

FreeRTOS_shutdown( xSocket, ... );

WaitForSocketToDisconnect();

FreeRTOS_closesocket( xSocket );

}

}

/*-----*/

void SocketRxTask( void *pvParameters )
{
    xSocket_t xSocket;

    for( ;; )
    {
        /* Wait to receive a socket that was created and connected by the Tx task. */
        xQueueReceive( xSocketPassingQueue, &xSocket, portMAX_DELAY );

        /* Keep receiving from the socket until the Tx task wants to close the socket.
*/

        while( TxTaskWantsToCloseSocket() == pdFALSE )
        {
            /* Receive then process data. */

            FreeRTOS_recv( xSocket, ... );

            ProcessReceivedData();

        }

        /* This is the Rx task's synchronization point. It reaches here only when it is
no longer using the socket, and it is therefore safe for the Tx task to close the socket.
*/

        xEventGroupSync( ... );

    }

}
```

El pseudocódigo anterior muestra dos tareas que se sincronizan entre sí para garantizar que un socket TCP compartido ya no esté siendo utilizado por ninguna de las tareas antes de que el socket se cierre.

Se puede usar un grupo de eventos para crear un punto de sincronización:

- A cada tarea que debe participar en la sincronización se le asigna un bit de eventos único en el grupo de eventos.
- Cada tarea establece su propio bit de eventos cuando alcanza el punto de sincronización.
- Al establecer su propio bit de eventos, cada tarea se bloquea en el grupo de eventos a la espera de que se establezcan también los bits de eventos que representan a todas las demás tareas de sincronización.

En esta situación, no se pueden utilizar las funciones de API xEventGroupSetBits() y xEventGroupWaitBits(). Si se utilizaran, el establecimiento de un bit (para indicar que una tarea ha alcanzado su punto de sincronización) y la realización de pruebas de bits (para determinar si las demás tareas de sincronización han llegado a su punto de sincronización) serían dos operaciones independientes. Para ver por qué esto sería un problema, vamos a imaginarnos que la Tarea A, la Tarea B y la Tarea C intentan sincronizarse utilizando un grupo de eventos:

1. La Tarea A y la Tarea B ya han alcanzado el punto de sincronización, por lo que sus bits de eventos se establecen en el grupo de eventos. Están en estado Bloqueado a la espera de que se establezca el bit de eventos de la Tarea C.
2. La Tarea C alcanza el punto de sincronización y utiliza xEventGroupSetBits() para establecer su bit en el grupo de eventos. En cuanto se establece el bit de la Tarea C, la Tarea A y la Tarea B salen del estado Bloqueado y borran los tres bits de eventos.
3. Luego, la Tarea C llama a xEventGroupWaitBits() a la espera de que se establezcan los tres bits de eventos, pero en ese tiempo, los tres bits de eventos se han borrado, la Tarea A y la Tarea B han salido de sus respectivos puntos de sincronización y, por tanto, la sincronización ha fallado.

Para utilizar correctamente un grupo de eventos para crear un punto de sincronización, el establecimiento de un bit de evento y la prueba subsiguiente de los bits de eventos deben realizarse como una única operación ininterrumpida. La función de API xEventGroupSync() sirve para ese fin.

Función de API xEventGroupSync()

xEventGroupSync() sirve para permitir que dos o más tareas utilicen un grupo de eventos para sincronizarse entre sí. La función permite que una tarea establezca uno o varios bits de eventos en un grupo de eventos y, a continuación, espere a que se establezca una combinación de bits de eventos en el mismo grupo de eventos como una única operación ininterrumpida.

El parámetro uxBitsToWaitFor de xEventGroupSync() especifica la condición de desbloqueo de la tarea que realiza la llamada. Los bits de eventos que especifica uxBitsToWaitFor se borrarán para volver a cero antes de que se devuelva xEventGroupSync(), si se ha devuelto xEventGroupSync() porque se ha cumplido la condición de desbloqueo.

Aquí se muestra el prototipo de la función de API xEventGroupSync().

```
EventBits_t xEventGroupSync( EventGroupHandle_t xEventGroup, const EventBits_t uxBitsToSet,
                             const EventBits_t uxBitsToWaitFor, TickType_t xTicksToWait );
```

En la siguiente tabla, se muestran los parámetros xEventGroupSync() y el valor de retorno.

Nombre del parámetro	Descripción
xEventGroup	El controlador del grupo de eventos en el que se van a establecer, y luego probar, los bits de

	<p>eventos. El controlador del grupo de eventos se ha devuelto desde la llamada a xEventGroupCreate(), que se ha utilizado para crear el grupo de eventos.</p>
uxBitsToSet	<p>Una máscara de bits que especifica el bit o los bits de eventos que se van a establecer en 1 en el grupo de eventos. El valor del grupo de eventos se actualiza mediante ORing bit a bit para cambiar el valor existente del grupo de eventos por el valor que se pasa en uxBitsToSet.</p> <p>Por ejemplo, al establecer uxBitsToSet en 0x04 (0100 binario), se establecerá el bit de eventos 3 (si aún no se ha establecido), mientras que todos los demás bits de eventos del grupo de eventos permanecerán sin cambios.</p>
uxBitsToWaitFor	<p>Una máscara de bits que especifica el bit o los bits de eventos que se van a probar en el grupo de eventos.</p> <p>Por ejemplo, si la tarea que realiza la llamada desea esperar a que los bits de eventos 0, 1 y 2 se establezcan en el grupo de eventos, establezca uxBitsToWaitFor en 0x07 (111 binario).</p>
xTicksToWait	<p>El tiempo máximo durante el que la tarea debe permanecer en el estado Bloqueado a la espera de que se cumpla la condición de desbloqueo.</p> <p>xEventGroupSync() se devolverá inmediatamente si xTicksToWait es cero o se cumple la condición de desbloqueo en el momento en que se llama a xEventGroupSync().</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. Se puede usar la macro pdMS_TO_TICKS() para convertir un tiempo especificado en milisegundos en un tiempo especificado en ciclos.</p> <p>Al establecer xTicksToWait en portMAX_DELAY, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando INCLUDE_vTaskSuspend esté establecido en 1 en FreeRTOSConfig.h.</p>

Valor devuelto

Si se ha devuelto `xEventGroupSync()` porque se ha cumplido la condición de desbloqueo de la tarea que realiza la llamada, el valor devuelto es el valor del grupo de eventos en el momento en que se cumple la condición de desbloqueo de la tarea que realiza la llamada (antes de que se haya borrado automáticamente algún bit para volver a cero). En este caso, el valor devuelto también cumplirá la condición de desbloqueo de la tarea que realiza la llamada.

Si se ha devuelto `xEventGroupSync()` porque el tiempo de bloqueo especificado por el parámetro `xTicksToWait` ha vencido, el valor devuelto es el valor del grupo de eventos en el momento en que el tiempo del bloqueo ha vencido. En este caso, el valor devuelto no cumplirá la condición de desbloqueo de la tarea que realiza la llamada.

Sincronización de tareas (Ejemplo 23)

El código siguiente muestra cómo sincronizar tres instancias de la implementación de una única tarea. El parámetro de la tarea se utiliza para pasar a cada instancia el bit de eventos que establecerá la tarea cuando llame a `xEventGroupSync()`.

La tarea imprime un mensaje antes de llamar a `xEventGroupSync()` y de nuevo después de que se ha devuelto la llamada a `xEventGroupSync()`. Cada mensaje incluye una marca temporal. Esto permite que la secuencia de ejecución se respete en la salida obtenida. Se utiliza un retraso pseudoaleatorio para evitar que todas las tareas alcancen el punto de sincronización al mismo tiempo.

```
static void vSyncingTask( void *pvParameters )
{
    const TickType_t xMaxDelay = pdMS_TO_TICKS( 4000UL );
    const TickType_t xMinDelay = pdMS_TO_TICKS( 200UL );
    TickType_t xDelayTime;
    EventBits_t uxThisTasksSyncBit;

    const EventBits_t uxAllSyncBits = ( mainFIRST_TASK_BIT | mainSECOND_TASK_BIT |
    mainTHIRD_TASK_BIT );

    /* Three instances of this task are created - each task uses a different event bit in
    the synchronization. The event bit to use is passed into each task instance using the task
    parameter. Store it in the uxThisTasksSyncBit variable. */

    uxThisTasksSyncBit = ( EventBits_t ) pvParameters;

    for( ;; )
    {
        /* Simulate this task taking some time to perform an action by delaying for a
        pseudo-random time. This prevents all three instances of this task from reaching the
        synchronization point at the same time, and so allows the example's behavior to be
        observed more easily. */
    }
}
```



```
xDelayTime = ( rand() % xMaxDelay ) + xMinDelay;

vTaskDelay( xDelayTime );

/* Print out a message to show this task has reached its synchronization point.
pcTaskGetTaskName() is an API function that returns the name assigned to the task when the
task was created. */

vPrintTwoStrings( pcTaskGetTaskName( NULL ), "reached sync point" );

/* Wait for all the tasks to have reached their respective synchronization points.
*/

xEventGroupSync( /* The event group used to synchronize. */ xEventGroup, /*
The bit set by this task to indicate it has reached the synchronization point. */
uxThisTasksSyncBit, /* The bits to wait for, one bit for each task taking part in the
synchronization. */ uxAllSyncBits, /* Wait indefinitely for all three tasks to reach the
synchronization point. */ portMAX_DELAY );

/* Print out a message to show this task has passed its synchronization point. As
an indefinite delay was used the following line will only be executed after all the tasks
reached their respective synchronization points. */

vPrintTwoStrings( pcTaskGetTaskName( NULL ), "exited sync point" );

}

}
```

La función main() crea el grupo de eventos, crea las tres tareas y luego inicia el programador. En el siguiente código, se muestra su implementación.

```
/* Definitions for the event bits in the event group. */

#define mainFIRST_TASK_BIT ( 1UL << 0UL ) /* Event bit 0, set by the first task. */
#define mainSECOND_TASK_BIT( 1UL << 1UL ) /* Event bit 1, set by the second task. */
#define mainTHIRD_TASK_BIT ( 1UL << 2UL ) /* Event bit 2, set by the third task. */

/* Declare the event group used to synchronize the three tasks. */
EventGroupHandle_t xEventGroup;

int main( void )
{
    /* Before an event group can be used it must first be created. */
    xEventGroup = xEventGroupCreate();

    /* Create three instances of the task. Each task is given a different name, which
    is later printed out to give a visual indication of which task is executing. The event bit
    to use when the task reaches its synchronization point is passed into the task using the
    task parameter. */

    xTaskCreate( vSyncingTask, "Task 1", 1000, mainFIRST_TASK_BIT, 1, NULL);
    xTaskCreate( vSyncingTask, "Task 2", 1000, mainSECOND_TASK_BIT, 1, NULL );
    xTaskCreate( vSyncingTask, "Task 3", 1000, mainTHIRD_TASK_BIT, 1, NULL );

    /* Start the scheduler so the created tasks start executing. */
}
```

```
vTaskStartScheduler();

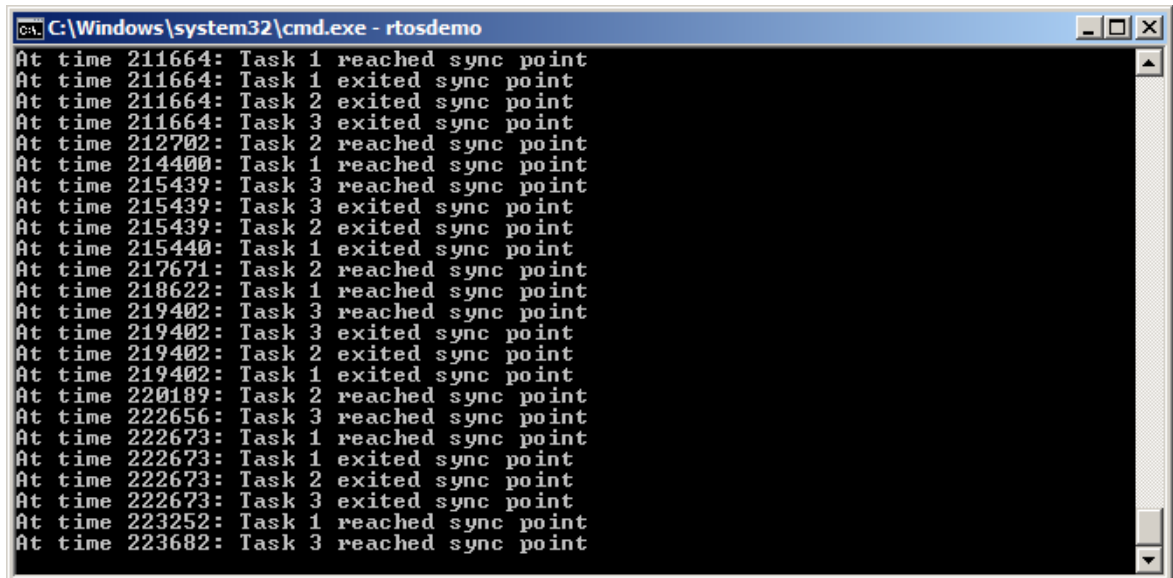
/* As always, the following line should never be reached. */

for( ;; );

return 0;

}
```

Aquí se muestra la salida que se genera cuando se ejecuta el Ejemplo 23. Como verá, aunque cada tarea alcanza el punto de sincronización en un momento diferente (pseudoaleatorio), cada tarea sale del punto de sincronización al mismo tiempo (que es el momento en el que la última tarea alcanza el punto de sincronización). En la figura, se muestra el ejemplo que se ejecuta en el puerto de Windows de FreeRTOS, que no representa el comportamiento auténtico en tiempo real (especialmente si se utilizan llamadas del sistema Windows para imprimir en la consola). Por lo tanto, habrá alguna variación en lo que se refiere al tiempo.



```
C:\Windows\system32\cmd.exe - rtosdemo
At time 211664: Task 1 reached sync point
At time 211664: Task 1 exited sync point
At time 211664: Task 2 exited sync point
At time 211664: Task 3 exited sync point
At time 212702: Task 2 reached sync point
At time 214400: Task 1 reached sync point
At time 215439: Task 3 reached sync point
At time 215439: Task 3 exited sync point
At time 215439: Task 2 exited sync point
At time 215440: Task 1 exited sync point
At time 217671: Task 2 reached sync point
At time 218622: Task 1 reached sync point
At time 219402: Task 3 reached sync point
At time 219402: Task 3 exited sync point
At time 219402: Task 2 exited sync point
At time 219402: Task 1 exited sync point
At time 220189: Task 2 reached sync point
At time 222656: Task 3 reached sync point
At time 222673: Task 1 reached sync point
At time 222673: Task 1 exited sync point
At time 222673: Task 2 exited sync point
At time 222673: Task 3 exited sync point
At time 223252: Task 1 reached sync point
At time 223682: Task 3 reached sync point
```

Notificaciones de tareas

En esta sección se explica lo siguiente:

- El estado de notificación de una tarea y el valor de notificación.
- Cómo y cuándo se puede usar una notificación de tarea en lugar de un objeto de comunicación, como, por ejemplo, un semáforo.
- Las ventajas de utilizar una tarea notificación en lugar de un objeto de comunicación.

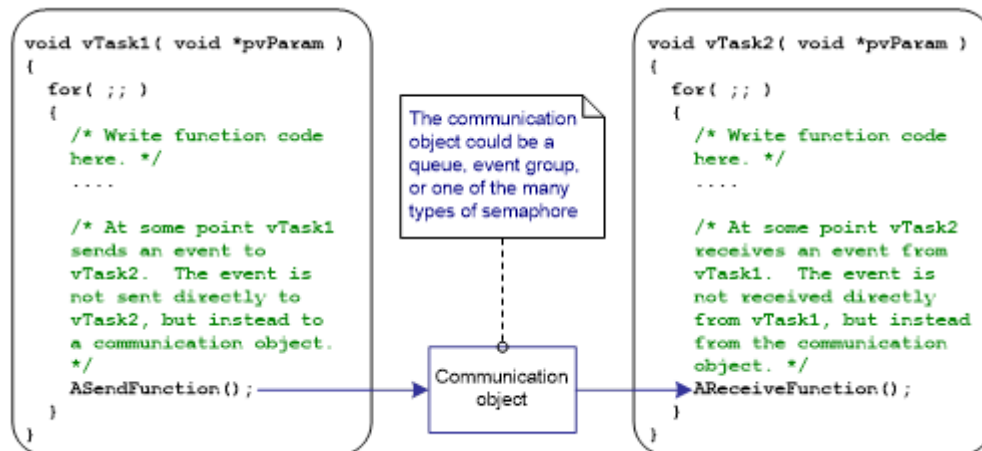
Las aplicaciones que utilizan FreeRTOS se estructuran como un conjunto de tareas independientes y estas tareas autónomas tienen que comunicarse entre sí de manera que, de forma colectiva, puedan ofrecer una funcionalidad de sistema útil.

Comunicación a través de objetos intermediarios

Hasta ahora los métodos en que las tareas se pueden comunicar entre sí han requerido la creación de un objeto de comunicación como colas, grupos de eventos y distintos tipos de semáforos.

Cuando se utiliza un objeto de comunicación, los eventos y los datos no se envían directamente a una tarea que los recibe o una ISR de recepción, sino al objeto de comunicación. Del mismo modo, las tareas y las ISR reciben eventos y datos desde el objeto de comunicación, en vez de recibirlos directamente desde la tarea o la ISR que envió el evento o los datos.

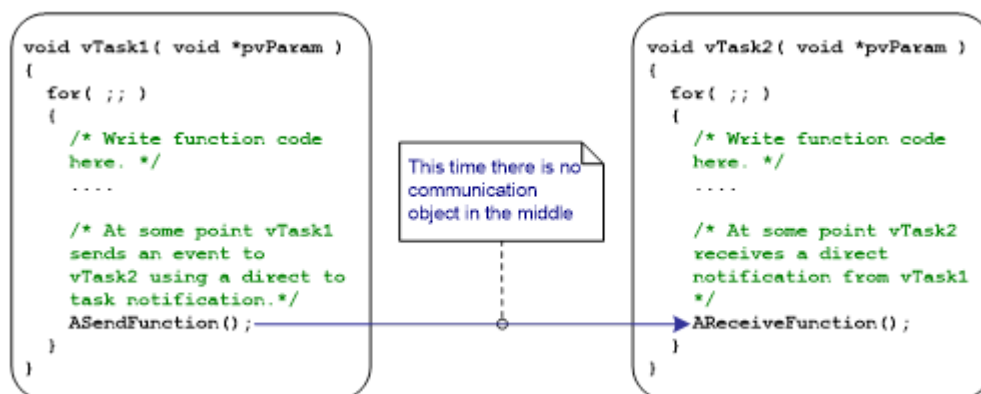
Este gráfico muestra un objeto de comunicación que se usa para enviar un evento de una tarea a otra.



Notificaciones de tareas: comunicación directa a la tarea

Las notificaciones de tareas permiten a las tareas interactuar con otras tareas así como sincronizarse con ISR, sin necesidad de un objeto de comunicación independiente. Con una notificación de tarea, una tarea o una ISR puede enviar un evento directamente a la tarea que lo recibe.

Esta figura muestra una notificación de tarea que se usa para enviar un evento directamente de una tarea a otra.



La funcionalidad de notificación de tarea es opcional. Para incluirla, en `FreeRTOSConfig.h` establezca `configUSE_TASK_NOTIFICATIONS` en 1.

Cuando `configUSE_TASK_NOTIFICATIONS` se establece en 1, cada tarea tiene un estado de notificación, que puede ser pendiente o no pendiente, y un valor de notificación, que es un número entero sin firmar de 32 bits. Cuando una tarea recibe una notificación, su estado de notificación se establece en pendiente. Cuando una tarea lee su valor de notificación, el estado de notificación se establece en no pendiente.

Una tarea puede esperar en el estado Bloqueado, con un tiempo de espera opcional, a que su estado de notificación pase a convertirse en pendiente.

Beneficios y limitaciones de las notificaciones de tareas

Usar una notificación de tarea para enviar un evento o datos a una tarea es significativamente más rápido que usar una cola, un semáforo o un grupo de eventos.

Asimismo, usar una notificación de tarea para enviar un evento o datos a una tarea requiere una cantidad claramente inferior de RAM que usar una cola, un semáforo o un grupo de eventos. Esto se debe a que cada objeto de comunicación (cola, semáforo o grupo de eventos) tiene que crearse para poderlo utilizar, mientras que habilitar la funcionalidad de notificación de tareas tiene una sobrecarga fija de tan solo ocho bytes de RAM por tarea.

Limitaciones de las notificaciones de tareas

Las notificaciones de tareas no se pueden utilizar en las situaciones siguientes:

- Enviar un evento o datos a una ISR.

Los objetos de comunicación se pueden usar para enviar eventos y datos desde una ISR a una tarea o desde una tarea a una ISR.

Las notificaciones de tareas se pueden utilizar para enviar eventos y datos desde una ISR a una tarea, pero no se pueden utilizar para enviar eventos o datos desde una tarea a una ISR.

- Habilitar más de una tarea de recepción.

Cualquier tarea o ISR puede obtener acceso a un objeto de comunicación cuyo controlador conozca (puede ser el controlador de una cola, un semáforo o un grupo de eventos). Las tareas y las ISR, sea cual sea su cantidad, pueden procesar eventos o datos enviados a cualquier objeto de comunicación.

Las notificaciones de tareas se envían directamente a la tarea que las recibe, por lo que solo las puede procesar la tarea a la que se envía la notificación. En la mayoría de los casos esto es rara vez una limitación puesto que, si bien es normal que varias tareas e ISR realicen envíos al mismo objeto de comunicación, no suele producirse lo contrario, es decir que varias tareas e ISR reciban notificaciones del mismo objeto de comunicación.

- Almacenar en búfer varios elementos de datos.

Una cola es un objeto de comunicación que puede contener más de un elemento de datos a la vez. Los datos que se envían a la cola pero que esta todavía no ha recibido se almacenan en búfer dentro del objeto de cola.

Las notificaciones de tareas envían datos a una tarea actualizando el valor de notificación de la tarea que los recibe. El valor de notificación de una tarea solo puede contener un valor a la vez.

- Retransmitir a más de una tarea.

Un grupo de eventos es un objeto de comunicación que se puede utilizar para enviar un evento a más de una tarea a la vez.

Las notificaciones de tareas se envían directamente a la tarea de recepción, por lo que solo las puede procesar la tarea que las recibe.

- Esperar en el estado Bloqueado para que se complete un envío.

Si un objeto de comunicación se encuentra temporalmente en un estado que impide que se escriban en él más datos o eventos (por ejemplo, cuando una cola está llena ya no se pueden enviar más datos a la cola), las tareas que intentan escribir en el objeto pueden también entrar en el estado Bloqueado para esperar a que la operación de escritura se complete.

Si una tarea intenta enviar una notificación de tarea a una tarea que ya tiene una notificación pendiente, la tarea que realiza el envío no puede esperar en el estado Bloqueado a que la tarea que recibe el envío restablezca su estado de notificación. Esto es rara vez una limitación en la mayoría de los casos en los que se utiliza una tarea de notificación.

Uso de notificaciones de tareas

Las notificaciones de tareas constituyen una característica muy importante que a menudo puede utilizarse en lugar de un semáforo binario, un semáforo de recuentos, un grupo de eventos y, en ocasiones, incluso una cola.

Opciones de API de notificación de tareas

Puede utilizar la función de API `xTaskNotify()` para enviar una notificación de tarea y la función de API `xTaskNotifyWait()` para recibir una notificación de tarea.

Sin embargo, en la mayoría de los casos, la flexibilidad que proporcionan las funciones de API `xTaskNotify()` y `xTaskNotifyWait()` no es necesaria. Basta con funciones más sencillas. La función de API `xTaskNotifyGive()` es una alternativa más sencilla pero menos flexible que `xTaskNotify()`. La función de API `ulTaskNotifyTake()` es una alternativa más sencilla pero menos flexible que `xTaskNotifyWait()`.

La función de API xTaskNotifyGive()

xTaskNotifyGive() envía una notificación directamente a una tarea e incrementa (añade uno) el valor de notificación de la tarea que lo recibe. Si se llama a xTaskNotifyGive() se establecerá el estado de notificación de la tarea que recibe la llamada en pendiente si no se encuentra ya en dicho estado.

La función de API xTaskNotifyGive() se proporciona para permitir usar una notificación de tarea como alternativa más ligera y rápida a un semáforo binario o de recuento. Aquí se muestra el prototipo de la función de API xTaskNotifyGive().

```
BaseType_t xTaskNotifyGive( TaskHandle_t xTaskToNotify );
```

En la siguiente tabla se muestran los parámetros xTaskNotifyGive() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTaskToNotify	El controlador de la tarea a la que se envía la notificación. Para obtener información acerca de cómo obtener controladores para la tarea, consulte el parámetro pxCreatedTask de la función de API xTaskCreate().
Valor devuelto	xTaskNotifyGive() es una macro que llama a xTaskNotify(). Los parámetros que la macro pasa a xTaskNotify() se establecen de manera que pdPASS sea el único valor de retorno posible. xTaskNotify() se describe más adelante en esta sección.

La función de API vTaskNotifyGiveFromISR()

vTaskNotifyGiveFromISR() es una versión de xTaskNotifyGive() que se puede utilizar en una rutina de servicio de interrupción. Aquí se muestra el prototipo de la función de la API vTaskNotifyGiveFromISR().

```
void vTaskNotifyGiveFromISR( TaskHandle_t xTaskToNotify, BaseType_t  
*pxHigherPriorityTaskWoken );
```

En la siguiente tabla se muestran los parámetros vTaskNotifyGiveFromISR() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTaskToNotify	El controlador de la tarea a la que se envía la notificación. Para obtener información acerca de cómo obtener controladores para las tareas, consulte el parámetro pxCreatedTask de la función de API xTaskCreate().
pxHigherPriorityTaskWoken	<p>Si la tarea a la que se envía la notificación está esperando en el estado Bloqueado a recibir una notificación, el envío de la notificación hará que la tarea abandone el estado Bloqueado.</p> <p>Si llamar a vTaskNotifyGiveFromISR() provoca que una tarea abandone el estado Bloqueado</p>

y la prioridad de la tarea desbloqueada es superior a la de la tarea que se está ejecutando en ese momento (la tarea que se interrumpió), vTaskNotifyGiveFromISR() establecerá internamente **<problematic>*</problematic>** pxHigherPriorityTaskWoken en pdTRUE.

Si vTaskNotifyGiveFromISR() establece este valor en pdTRUE, debe efectuarse un cambio de contexto antes de salir de la interrupción. De este modo, se asegurará de que la interrupción vuelve directamente a la tarea con el estado Listo de máxima prioridad.

Al igual que ocurre con todas las funciones de API a prueba de interrupciones, el parámetro pxHigherPriorityTaskWoken debe establecerse en pdFALSE para poderlo utilizar.

La función de API ulTaskNotifyTake()

ulTaskNotifyTake() permite que una tarea espere en el estado Bloqueado a que su valor de notificación sea superior a cero y disminuye (resta uno) o borra el valor de notificación de la tarea antes de que regrese.

La función de API ulTaskNotifyTake() se proporciona para permitir usar una notificación de tarea como alternativa más ligera y rápida a un semáforo binario o de recuento. Aquí se muestra el prototipo de la función de API ulTaskNotifyTake().

```
uint32_t ulTaskNotifyTake( BaseType_t xClearCountOnExit, TickType_t xTicksToWait );
```

En la siguiente tabla se muestran los parámetros ulTaskNotifyTake() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xClearCountOnExit	<p>Si se establece xClearCountOnExit en pdTRUE, el valor de notificación de la tarea que llama se borrará y se pondrá en cero antes de que la llamada a ulTaskNotifyTake() regrese.</p> <p>Si se establece xClearCountOnExit en pdFALSE y el valor de notificación de la tarea que llama es superior a cero, el valor de notificación de dicha tarea disminuirá antes de que la llamada a ulTaskNotifyTake() regrese.</p>
xTicksToWait	<p>La duración máxima de tiempo que la tarea que realiza la llamada debe permanecer en el estado Bloqueado para esperar a que su valor de notificación sea mayor que cero.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. La macro pdMS_TO_TICKS() se puede usar</p>

	<p>para convertir a ciclos un tiempo especificado en milisegundos.</p> <p>Al establecer xTicksToWait en portMAX_DELAY, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando INCLUDE_vTaskSuspend esté establecido en 1 en FreeRTOSConfig.h.</p>
Valor devuelto	<p>El valor devuelto es el valor de notificación de la tarea que realiza la llamada antes de que se borra y se pusiera en cero o se redujera, tal y como especifica el valor del parámetro xClearCountOnExit.</p> <p>Si se ha especificado un tiempo de bloqueo (xTicksToWait no es cero) y el valor de retorno no es cero, es posible que la tarea de llamada se hubiera puesto en el estado Bloqueado a esperar a que su valor de notificación fuese superior a cero, y su valor de notificación se actualizó antes de que el tiempo de bloqueo caducara.</p> <p>Si se especifica un tiempo de bloqueo (xTicksToWait no es cero) y el valor devuelto es cero, la tarea que realiza la llamada se pone en el estado Bloqueado a esperar a que su valor de notificación sea superior a cero, pero el tiempo de bloqueo especificado caduca antes de que se produzca este evento.</p>

Método 1 para usar una tarea notificación en lugar de un semáforo (ejemplo 24)

En el ejemplo 16 se ha utilizado un semáforo binario para desbloquear una tarea desde una rutina de servicio de interrupción, lo que en la práctica sincroniza la tarea con la interrupción. En este ejemplo se replica la funcionalidad del ejemplo 16, aunque se utiliza una notificación directa a la tarea en lugar de un semáforo binario.

En el código siguiente se muestra la implementación de la tarea que se sincroniza con la interrupción. La llamada a xSemaphoreTake() que se ha utilizado en el ejemplo 16 se ha sustituido por una llamada a ulTaskNotifyTake().

El parámetro xClearCountOnExit de ulTaskNotifyTake() se establece en pdTRUE, lo que hace que el valor de notificación de la tarea que recibe la llamada se ponga en cero antes de que la llamada regrese a ulTaskNotifyTake(). Por lo tanto es necesario procesar todos los eventos que ya están disponibles entre cada llamada a ulTaskNotifyTake(). En el ejemplo 16, como se ha usado un semáforo binario, se ha tenido que determinar el número de eventos pendientes a partir del hardware, método que no siempre es práctico. En este ejemplo el número de eventos pendientes se obtiene desde ulTaskNotifyTake().

Los eventos de interrupción que se producen entre llamadas a ulTaskNotifyTake se integran en el valor de notificación de la tarea y las llamadas a ulTaskNotifyTake() regresarán de inmediato si la tarea que realiza la llamada ya tiene notificaciones pendientes.

```
/* The rate at which the periodic task generates software interrupts.*/
```



```
const TickType_t xInterruptFrequency = pdMS_TO_TICKS( 500UL );

static void vHandlerTask( void *pvParameters )
{
    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time
    between events. */

    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );
    uint32_t ulEventsToProcess;

    /* As per most tasks, this task is implemented within an infinite loop. */
    for( ;; )
    {
        /* Wait to receive a notification sent directly to this task from the interrupt
        service routine. */

        ulEventsToProcess = ulTaskNotifyTake( pdTRUE, xMaxExpectedBlockTime );

        if( ulEventsToProcess != 0 )
        {
            /* To get here at least one event must have occurred. Loop here until all
            the pending events have been processed (in this case, just print out a message for each
            event). */

            while( ulEventsToProcess > 0 )
            {
                vPrintString( "Handler task - Processing event.\r\n" );

                ulEventsToProcess--;
            }
        }
        else
        {
            /* If this part of the function is reached, then an interrupt did not arrive
            within the expected time. In a real application, it might be necessary to perform some
            error recovery operations. */

        }
    }
}
```

La tarea periódica que se utiliza para generar interrupciones de software imprime un mensaje antes y después de que se genere la interrupción. Esto permite que la secuencia de ejecución se respete en la salida.

En el código siguiente se muestra el controlador de interrupciones. Esto simplemente significa que se envía una notificación directamente a la tarea a la que se cede el controlador de interrupciones.

```
static uint32_t ulExampleInterruptHandler( void )
{
    BaseType_t xHigherPriorityTaskWoken;

    /* The xHigherPriorityTaskWoken parameter must be initialized to pdFALSE because it
    will get set to pdTRUE inside the interrupt-safe API function if a context switch is
    required. */

    xHigherPriorityTaskWoken = pdFALSE;

    /* Send a notification directly to the task to which interrupt processing is being
    deferred. */

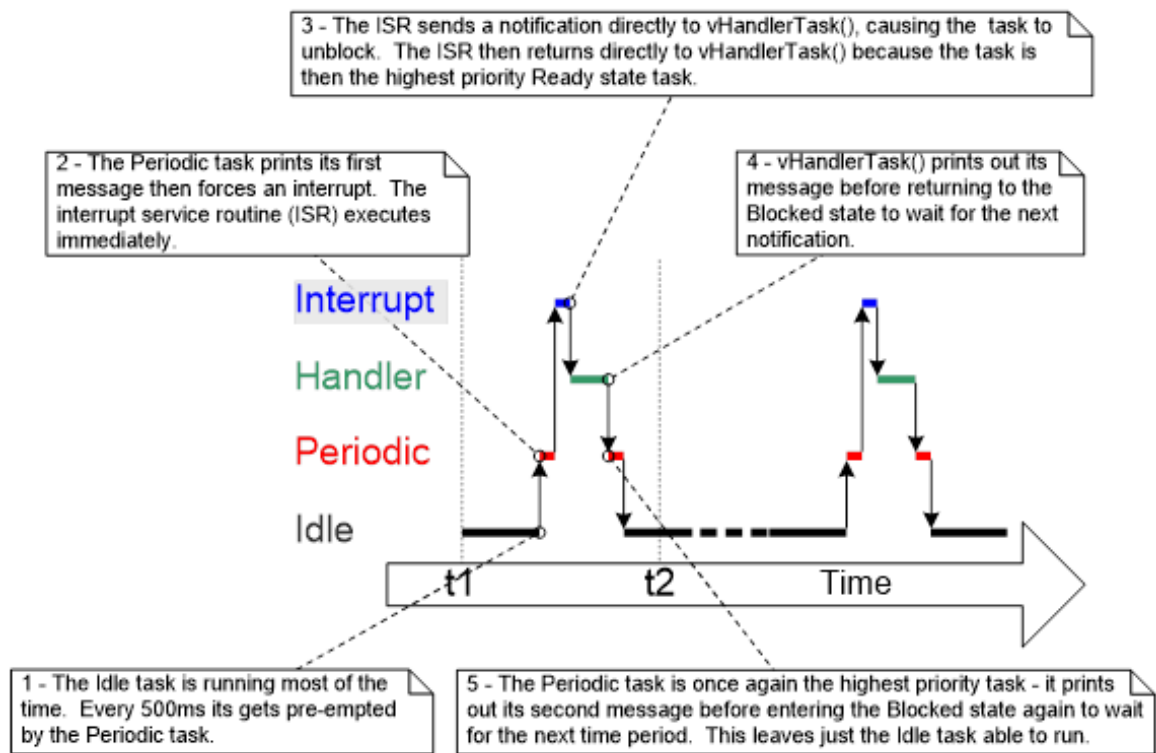
    vTaskNotifyGiveFromISR( /* The handle of the task to which the notification is
    being sent. The handle was saved when the task was created. */ xHandlerTask, /*
    xHigherPriorityTaskWoken is used in the usual way. */ &xHigherPriorityTaskWoken );

    /* Pass the xHigherPriorityTaskWoken value into portYIELD_FROM_ISR(). If
    xHigherPriorityTaskWoken was set to pdTRUE inside vTaskNotifyGiveFromISR(), then calling
    portYIELD_FROM_ISR() will request a context switch. If xHigherPriorityTaskWoken is still
    pdFALSE, then calling portYIELD_FROM_ISR() will have no effect. The implementation of
    portYIELD_FROM_ISR() used by the Windows port includes a return statement, which is why
    this function does not explicitly return a value. */

    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );
}
```

La salida que se genera cuando se ejecuta el código se muestra aquí.

Como era de esperar, es idéntica a la salida que se genera cuando se ejecuta el ejemplo 16. `vHandlerTask()` entra en el estado En ejecución tan pronto como se genera la interrupción, por lo que la salida de la tarea divide la salida que la tarea periódica produce. La secuencia de ejecución se muestra aquí.



Método 2 para usar una tarea notificación en lugar de un semáforo (ejemplo 25)

En el ejemplo 24, el parámetro xClearOnExit de ulTaskNotifyTake() se ha establecido en pdTRUE. El ejemplo 25 es una pequeña modificación del ejemplo 24 y sirve para demostrar el comportamiento cuando el parámetro xClearOnExit de ulTaskNotifyTake() se establece en pdFALSE.

Cuando xClearOnExit es pdFALSE, llamar a ulTaskNotifyTake() solo reducirá (en uno) el valor de notificación de la tarea que realiza la llamada, en lugar de borrarla y ponerla en cero. El recuento de notificaciones es, por lo tanto, la diferencia entre el número de eventos que se han producido y el número de eventos que se han procesado. Esto permite simplificar la estructura de vHandlerTask() de dos formas:

1. El número de eventos a la espera de ser procesados está contenido en el valor de notificación, por lo que no es necesario integrarlo en una variable local.
2. Solo es necesario procesar un evento entre cada llamada a ulTaskNotifyTake().

La implementación de vHandlerTask() se muestra aquí.

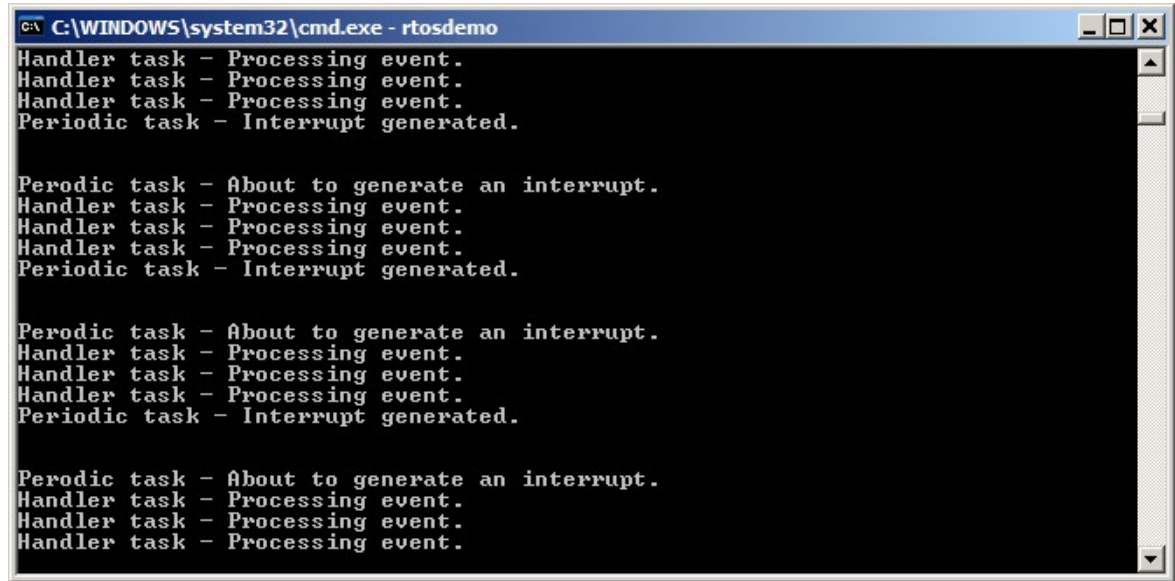
```
static void vHandlerTask( void *pvParameters )  
{  
    /* xMaxExpectedBlockTime is set to be a little longer than the maximum expected time  
    between events. */  
    const TickType_t xMaxExpectedBlockTime = xInterruptFrequency + pdMS_TO_TICKS( 10 );
```

```
/* As per most tasks, this task is implemented within an infinite loop. */  
  
for( ;; )  
{  
  
    /* Wait to receive a notification sent directly to this task from the interrupt  
    service routine. The xClearCountOnExit parameter is now pdFALSE, so the task's  
    notification value will be decremented by ulTaskNotifyTake() and not cleared to zero. */  
  
    if( ulTaskNotifyTake( pdFALSE, xMaxExpectedBlockTime ) != 0 )  
    {  
  
        /* To get here, an event must have occurred. Process the event (in this case,  
        just print out a message). */  
  
        vPrintString( "Handler task - Processing event.\r\n" );  
  
    }  
  
    else  
    {  
  
        /* If this part of the function is reached, then an interrupt did not arrive  
        within the expected time. In a real application, it might be necessary to perform some  
        error recovery operations. */  
  
    }  
  
}  
}
```

Para facilitar la demostración, también se ha modificado la rutina de servicio a fin de enviar más de una notificación de tarea por interrupción y, al hacerlo, simular varias interrupciones que se producen a alta frecuencia. Aquí se muestra la implementación de la rutina del servicio de interrupciones.

```
static uint32_t ulExampleInterruptHandler(void)  
{  
  
    BaseType_t xHigherPriorityTaskWoken;  
  
    xHigherPriorityTaskWoken = pdFALSE;  
  
    /* Send a notification to the handler task multiple times. The first give will unblock  
    the task. The following gives are to demonstrate that the receiving task's notification  
    value is being used to count (latch) events, allowing the task to process each event in  
    turn. */  
  
    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);  
  
    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);  
  
    vTaskNotifyGiveFromISR(xHandlerTask, &xHigherPriorityTaskWoken);  
  
    portYIELD_FROM_ISR(xHigherPriorityTaskWoken);  
  
}
```

El resultado se muestra aquí. Verá que vHandlerTask() procesa los tres eventos cada vez que se genera una interrupción.



```
C:\WINDOWS\system32\cmd.exe - rtosdemo
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
Periodic task - Interrupt generated.

Periodic task - About to generate an interrupt.
Handler task - Processing event.
Handler task - Processing event.
Handler task - Processing event.
```

Funciones de API xTaskNotify() y xTaskNotifyFromISR()

xTaskNotify() es una versión más capaz de xTaskNotifyGive() que se puede utilizar para actualizar el valor de notificación de la tarea que recibe la llamada de cualquiera de las siguientes formas:

- Incrementando o añadiendo uno al valor de notificación de la tarea que recibe la llamada, en cuyo caso xTaskNotify() es equivalente a xTaskNotifyGive().
- Estableciendo uno o varios bits en el valor de notificación de la tarea que recibe la llamada. Esto permite usar el valor de notificación de una tarea como alternativa ligera y más rápida a un grupo de eventos.
- Escribiendo un número totalmente nuevo en el valor de notificación de la tarea que recibe la llamada, pero solo si esta ha leído su valor de notificación desde que se actualizó por última vez. Esto permite que el valor de notificación de una tarea proporcione una funcionalidad similar a la que proporciona una cola que tiene una longitud de uno.
- Escribiendo un número totalmente nuevo en el valor de notificación de la tarea que recibe la llamada, incluso si esta no ha leído su valor de notificación desde que se actualizó por última vez. Esto permite que el valor de notificación de una tarea proporcione una funcionalidad similar a la que proporciona la función de API xQueueOverwrite(). El comportamiento obtenido se denomina en ocasiones buzón de correo.

La función xTaskNotify() es más flexible y potente que xTaskNotifyGive(). También es algo más compleja de utilizar.

xTaskNotifyFromISR() es una versión de xTaskNotify() que se puede utilizar en una rutina de servicio de interrupción. Por lo tanto, tiene un parámetro pxHigherPriorityTaskWoken adicional.

Llamar a xTaskNotify() siempre establecerá el estado de notificación de la tarea que recibe la llamada en pendiente si esta todavía no estaba en este estado.

```
BaseType_t xTaskNotify( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction
eAction );
```

```
BaseType_t xTaskNotifyFromISR( TaskHandle_t xTaskToNotify, uint32_t ulValue, eNotifyAction  
eAction, BaseType_t *pxHigherPriorityTaskWoken );
```

En la siguiente tabla se muestran los parámetros xTaskNotify() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
xTaskToNotify	El controlador de la tarea a la que se envía la notificación. Para obtener información acerca de cómo obtener controladores para las tareas, consulte el parámetro pxCreatedTask de la función de API xTaskCreate().
ulValue	El uso de ulValue depende del valor de eNotifyAction. Véase la tabla 52.
eNotifyAction	Tipo enumerado que especifica cómo actualizar el valor de notificación de la tarea que recibe la llamada. Consulte cont.
Valor devuelto	xTaskNotify() devolverá pdPASS, salvo en el caso indicado a continuación.

En la siguiente tabla se muestran los valores de los parámetros eNotifyAction de xTaskNotify() válidos y el efecto que producen en el valor de notificación de la tarea que recibe la llamada.

Valor eNotifyAction	Efecto que produce en la tarea que recibe la llamada
eNoAction	El estado de notificación de la tarea que recibe la llamada se establece en pendiente sin que su valor de notificación se actualice. No se usa el parámetro ulValue de xTaskNotify(). La acción eNoAction permite usar una notificación de tarea como alternativa más rápida y ligera a un semáforo binario.
eSetBits	<p>El valor de notificación de la tarea que recibe la llamada se ejecuta usando O BIEN bit a bit con el valor que se pasa en el parámetro ulValue de xTaskNotify(). Por ejemplo, si ulValue se establece en 0x01, el bit 0 se establecerá en el valor de notificación de la tarea que recibe la llamada. Otro ejemplo: si ulValue se establece en 0x06 (binario 0110), el bit 1 y el bit 2 se establecerán en el valor de notificación de la tarea que recibe la llamada.</p> <p>La acción eSetBits permite usar una notificación de tarea como alternativa más rápida y ligera a un grupo de eventos.</p>
eIncrement	Se incrementa el valor de notificación de la tarea que recibe la llamada. No se usa el parámetro ulValue de xTaskNotify().

	La acción eIncrement permite usar una notificación de tarea como alternativa más rápida y ligera a un semáforo binario o de recuento. Es equivalente a la función de API xTaskNotifyGive() más sencilla.
eSetValueWithoutOverwrite	<p>Si la tarea que recibe la llamada tenía una notificación pendiente antes de que se llamara a xTaskNotify(), no se realiza ninguna acción y xTaskNotify() devolverá pdFAIL.</p> <p>Si la tarea que recibe la llamada no tenía una notificación pendiente antes de que se llamara a xTaskNotify(), el valor de notificación de la tarea que recibe la llamada se establece en el valor que se transfiere en el parámetro ulValue de xTaskNotify().</p>
eSetValueWithOverwrite	El valor de notificación de la tarea que recibe la llamada se establece en el valor que se transfiere en el parámetro ulValue de xTaskNotify(), sin tener en cuenta si la tarea que recibe la llamada ya tenía una notificación pendiente antes de que se llamara a xTaskNotify().

La función de API xTaskNotifyWait()

xTaskNotifyWait() es una versión más capaz de ulTaskNotifyTake(). Permite que una tarea espere, con un tiempo de espera opcional, a que el estado de notificación de la tarea que realiza la llamada pase a estar pendiente, si todavía no lo está. xTaskNotifyWait() proporciona opciones para que se borren bits en el valor de notificación de la tarea que realiza la llamada tanto en la entrada de la función como en la salida de esta.

```
BaseType_t xTaskNotifyWait( uint32_t ulBitsToClearOnEntry, uint32_t ulBitsToClearOnExit,
    uint32_t *pulNotificationValue, TickType_t xTicksToWait );
```

En la siguiente tabla se muestran los parámetros xTaskNotifyWait() y el valor de retorno.

Nombre de parámetro / Valor devuelto	Descripción
ulBitsToClearOnEntry	<p>Si la tarea que realiza la llamada no tenía una notificación pendiente antes de llamar a xTaskNotifyWait(), todos los bits de ulBitsToClearOnEntry se borrarán en el valor de notificación de la tarea al entrar en la función.</p> <p>Por ejemplo, si ulBitsToClearOnEntry es 0x01, el bit 0 del valor de notificación de la tarea se borrará. Otro ejemplo: si se establece ulBitsToClearOnEntry en 0xffffffff (ULONG_MAX), se borrarán todos los bits del valor de notificación de la tarea, lo que borrará el valor y lo pondrá en 0.</p>
ulBitsToClearOnExit	Si la tarea que realiza la llamada sale de xTaskNotifyWait() porque ha recibido una notificación o porque ya tenía una notificación

	<p>pendiente cuando se llamó a xTaskNotifyWait(), todos los bits establecidos en ulBitsToClearOnExit se borrarán del valor de notificación antes de que la tarea salga de la función xTaskNotifyWait().</p> <p>Los bits se borran después de que el valor de notificación de la tarea se guarde en <problematic>*</problematic> pulNotificationValue (véase abajo la descripción de pulNotificationValue).</p> <p>Por ejemplo, si ulBitsToClearOnExit es 0x03, el bit 0 y el bit 1 del valor de notificación de la tarea se borrarán antes de que la función salga.</p> <p>Si se establece ulBitsToClearOnExit en 0xffffffff (ULONG_MAX), se borrarán todos los bits del valor de notificación de la tarea, lo que borrará el valor y lo pondrá en 0.</p>
pulNotificationValue	<p>Se utiliza para pasar el valor de notificación de la tarea. El valor que se copia en <problematic>*</problematic> pulNotificationValue es el valor de notificación de la tarea tal como era antes de que se borrarán los bits debido al valor de ulBitsToClearOnExit.</p> <p>pulNotificationValue es un parámetro opcional y se puede configurar en NULL si no es obligatorio.</p>
xTicksToWait	<p>La duración máxima de tiempo que la tarea que realiza la llamada debe permanecer en el estado Bloqueado para esperar a que su estado de notificación pase a estar pendiente.</p> <p>El tiempo de bloqueo se especifica en periodos de ciclos, por lo que el tiempo absoluto que representa depende de la frecuencia de ciclos. La macro pdMS_TO_TICKS() se puede usar para convertir una duración especificada en milisegundos en una duración especificada en ciclos.</p> <p>Al establecer xTicksToWait en portMAX_DELAY, la tarea tendrá que esperar indefinidamente (sin agotar el tiempo de espera), siempre y cuando INCLUDE_vTaskSuspend esté establecido en 1 en FreeRTOSConfig.h.</p>

Valor devuelto	<p>Hay dos valores de retorno posibles:</p> <p>1. pdTRUE</p> <p>Esto indica que xTaskNotifyWait() ha regresado porque se ha recibido una notificación o porque la tarea que ha realizado la llamada ya tenía una notificación pendiente cuando se llamó a xTaskNotifyWait().</p> <p>Si se ha especificado un tiempo de bloqueo (xTicksToWait no es cero), es posible que la tarea que realiza la llamada se hubiera puesto en el estado Bloqueado para esperar a que su valor de notificación se volviese Pendiente, pero su estado de notificación se estableció en Pendiente antes de que el tiempo de bloqueo caducara.</p> <p>1. pdFALSE</p> <p>Esto indica que xTaskNotifyWait() ha vuelto sin que la tarea que ha realizado la llamada haya recibido una notificación de tarea.</p> <p>Si xTicksToWait no era cero, la tarea que ha realizado la llamada se habrá contenido en el estado Bloqueado a la espera de que su estado de notificación pase a pendiente, pero la duración de bloqueo especificada caducó antes de que esto se produjera.</p>
----------------	---

Notificaciones de tarea usadas en controladores de dispositivos periféricos: ejemplo de UART

Las bibliotecas de controladores periféricos proporcionan funciones que realizan operaciones habituales en las interfaces de hardware. Entre los periféricos de dichas bibliotecas están los receptores y transmisores asíncronos universales (UART), los puertos de SPI (Serial Peripheral Interface), los ADC (convertidores analógicos digitales) y los puertos Ethernet. Las funciones que estas bibliotecas suelen proporcionar incluyen funciones para inicializar un periférico, enviar datos a un periférico o recibir datos desde un periférico.

Algunas operaciones realizadas en periféricos tardan un tiempo relativamente largo en completarse. Esto se debe a que incluyen una conversión ADC de alta precisión y la transmisión de un paquete de datos de gran tamaño en un UART. En estos casos, la función de biblioteca de controladores se puede implementar para sondear (leer repetidamente) los registros de estado del periférico para determinar cuándo se ha completado la operación. Sin embargo, el sondeo realizado de esta forma es casi siempre una pérdida de tiempo, ya que utiliza el 100% del tiempo del procesador y no se puede llevar a cabo ningún procesamiento productivo. Este derroche es especialmente costoso en un sistema de multitarea, donde una tarea que está realizando un sondeo en un periférico podría estar impidiendo que se ejecutara una tarea de menor prioridad que tuviese que ejecutar un procesamiento productivo.

Para evitar la posible pérdida de tiempo de procesamiento, un controlador de dispositivos compatible con RTOS eficiente tendría que estar basado en interrupciones y dar a las tareas que inician una larga

operación la opción de esperar en el estado Bloqueado a que se complete la operación. De esta forma las tareas de menor prioridad pueden ejecutarse mientras que la tarea que realiza la operación más larga se encuentra en estado Bloqueado y ninguna tarea usa tiempo de procesamiento a menos que pueda utilizarlo de manera productiva.

Es habitual que las bibliotecas de controladores compatibles con RTOS usen un semáforo binario para poner tareas en el estado Bloqueado. La técnica se demuestra con el siguiente pseudocódigo, que proporciona el esquema de una función de biblioteca compatible con RTOS que transmite datos en un puerto UART. En la siguiente lista de códigos:

- xUART es una estructura que describe el periférico UART y contiene información de estado. El miembro xTxSemaphore de la estructura es una variable de tipo SemaphoreHandle_t. Se presupone que el semáforo ya se ha creado.
- La función xUART_Send() no incluye ninguna lógica de exclusión mutua. Si más de una tarea va a utilizar la función xUART_Send(), el programador de la aplicación deberá administrar la exclusión mutua dentro mismo de la aplicación. Por ejemplo, es posible que se necesite una tarea para obtener una exclusión mutua antes de llamar a xUART_Send().
- La función de API xSemaphoreTake() se utiliza para poner la tarea de llamada en el estado Bloqueado después de que se haya iniciado la transmisión UART.
- La función de API xSemaphoreGiveFromISR() se utiliza para sacar la tarea del estado Bloqueado después de que la transmisión se haya completado, que es cuando se ejecuta la rutina de servicio de interrupciones de fin de transmisión del periférico UART.

```
/* Driver library function to send data to a UART. */

BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )
{
    BaseType_t xReturn;

    /* Ensure the UART's transmit semaphore is not already available by attempting to take
    the semaphore without a timeout. */

    xSemaphoreTake( pxUARTInstance->xTxSemaphore, 0 );

    /* Start the transmission. */

    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* Block on the semaphore to wait for the transmission to complete. If the semaphore is
    obtained, then xReturn will get set to pdPASS. If the semaphore take operation times out,
    then xReturn will get set to pdFAIL. If the interrupt occurs between UART_low_level_send()
    being called and xSemaphoreTake() being called, then the event will be latched in the
    binary semaphore, and the call to xSemaphoreTake() will return immediately. */

    xReturn = xSemaphoreTake( pxUARTInstance->xTxSemaphore, pxUARTInstance->xTxTimeout );

    return xReturn;
}

/*-----*/

/* The service routine for the UART's transmit end interrupt, which executes after the last
byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )
{

```

```
BaseType_t xHigherPriorityTaskWoken = pdFALSE;

/* Clear the interrupt. */

UART_low_level_interrupt_clear( pxUARTInstance );

/* Give the Tx semaphore to signal the end of the transmission. If a task is Blocked
waiting for the semaphore, then the task will be removed from the Blocked state. */

xSemaphoreGiveFromISR( pxUARTInstance->TxSemaphore, &xHigherPriorityTaskWoken );

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

La técnica demostrada en este código es viable y se usa con frecuencia, pero tiene algunos inconvenientes:

- La biblioteca utiliza varias semáforos, lo que aumenta su huella de RAM.
- Los semáforos no se pueden utilizar hasta que se han creado, por lo que una biblioteca que utilice semáforos no se puede utilizar hasta que se haya inicializado de forma explícita.
- Los semáforos son objetos genéricos que se pueden aplicar a una amplia gama de casos de uso. Incluyen lógica para permitir que un número cualquiera de tareas espere en el estado Bloqueado a que el semáforo pase a estar disponible y para seleccionar (de un manera determinista) qué tarea debe sacarse del estado Bloqueado cuando el semáforo pasa a estar disponible. La ejecución de esta lógica precisa una duración finita. Esa sobrecarga de procesamiento no es necesaria en el escenario que se muestra en este código, donde no puede haber más de una tarea esperando el semáforo en cualquier momento.

El código siguiente muestra cómo evitar estos inconvenientes gracias al uso de una notificación de tarea en lugar de un semáforo binario.

Nota: Si una biblioteca utiliza notificaciones de tareas, la documentación de la biblioteca debe indicar claramente que llamar a una función de biblioteca puede cambiar el estado y el valor de la notificación de la tarea de llamada.

En el siguiente código:

- El miembro `xTxSemaphore` de la estructura `xUART` se ha reemplazado por el miembro `xTaskToNotify`. `xTaskToNotify` es una variable de tipo `TaskHandle_t`, y se utiliza para almacenar el identificador de la tarea que está a la espera de que la operación UART se complete.
- La función de API de FreeRTOS `xTaskGetCurrentTaskHandle()` se utiliza para obtener el controlador de la tarea que se encuentra en el estado En ejecución.
- La biblioteca no crea ningún objeto FreeRTOS, por lo que no se produce una sobrecarga de la RAM y no tiene inicializarse de forma explícita.
- La notificación de tarea se envía directamente a la tarea que está esperando que la operación UART se complete, por lo que no se ejecuta una lógica innecesaria.

Se accede al miembro `xTaskToNotify` de la estructura `xUART` desde una tarea y una rutina de servicio de interrupción, que requiere tener en cuenta cómo el procesador actualizará su valor:

- Si `xTaskToNotify` se actualiza con una única operación de escritura de memoria, se puede actualizar fuera de una sección crítica, exactamente tal y como se muestra en el siguiente código. Este sería el caso si `xTaskToNotify` fuera una variable de 32 bits (`TaskHandle_t` fuera un tipo de 32 bits) y el procesador en el que se ejecuta FreeRTOS fuera un procesador de 32 bits.

- Si se necesitase más de una operación de escritura de memoria para actualizar xTaskToNotify, xTaskToNotify solo tiene que actualizarse desde una sección crítica. De lo contrario, podría pasar que la rutina de interrupción de servicio accediera a xTaskToNotify mientras este se encontrase en un estado incoherente. Este sería el caso si xTaskToNotify fuese una variable de 32 bits y el procesador en el que se ejecuta FreeRTOS fuese un procesador de 16 bits, ya que se necesitarían dos operaciones de escritura de memoria de 16 bits para actualizar todos los 32 bits.

Internamente, dentro de la implementación de FreeRTOS, TaskHandle_t es un puntero, por lo que sizeof (TaskHandle_t) siempre equivale a sizeof (void *).

```
/* Driver library function to send data to a UART. */

BaseType_t xUART_Send( xUART *pxUARTInstance, uint8_t *pucDataSource, size_t uxLength )
{
    BaseType_t xReturn;

    /* Save the handle of the task that called this function. The book text contains notes
    as to whether the following line needs to be protected by a critical section or not. */

    pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

    /* Ensure the calling task does not already have a notification pending by calling
    ulTaskNotifyTake() with the xClearCountOnExit parameter set to pdTRUE, and a block time of
    0 (don't block). */

    ulTaskNotifyTake( pdTRUE, 0 );

    /* Start the transmission. */

    UART_low_level_send( pxUARTInstance, pucDataSource, uxLength );

    /* Block until notified that the transmission is complete. If the notification is
    received, then xReturn will be set to 1 because the ISR will have incremented this task's
    notification value to 1 (pdTRUE). If the operation times out, then xReturn will be 0
    (pdFALSE) because this task's notification value will not have been changed since it was
    cleared to 0 above. If the ISR executes between the calls to UART_low_level_send() and
    the call to ulTaskNotifyTake(), then the event will be latched in the task's notification
    value, and the call to ulTaskNotifyTake() will return immediately.*/

    xReturn = ( BaseType_t ) ulTaskNotifyTake( pdTRUE, pxUARTInstance->xTxTimeout );

    return xReturn;
}

/*-----*/

/* The ISR that executes after the last byte has been sent to the UART. */

void xUART_TransmitEndISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* This function should not execute unless there is a task waiting to be notified.
    Test this condition with an assert. This step is not strictly necessary, but will aid
    debugging. configASSERT() is described in Developer Support.*/

    configASSERT( pxUARTInstance->xTaskToNotify != NULL );
```

```
/* Clear the interrupt. */

UART_low_level_interrupt_clear( pxUARTInstance );

/* Send a notification directly to the task that called xUART_Send(). If the task
is Blocked waiting for the notification, then the task will be removed from the Blocked
state. */

vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify, &xHigherPriorityTaskWoken );

/* Now there are no tasks waiting to be notified. Set the xTaskToNotify member of the
xUART structure back to NULL. This step is not strictly necessary, but will aid debugging.
*/

pxUARTInstance->xTaskToNotify = NULL;

portYIELD_FROM_ISR( xHigherPriorityTaskWoken );

}
```

Las notificaciones de tareas también pueden sustituir semáforos en la recepción de funciones, tal y como se demuestra en el siguiente pseudocódigo, que proporciona el esquema de una función de biblioteca compatible con RTOS que recibe datos en un puerto UART.

- La función `xUART_Receive()` no incluye ninguna lógica de exclusión mutua. Si más de una tarea va a utilizar la función `xUART_Receive()`, el programador de la aplicación deberá administrar la exclusión mutua dentro mismo de la aplicación. Por ejemplo, es posible que se necesite una tarea para obtener una exclusión mutua antes de llamar a `xUART_Receive()`.
- La rutina del servicio de interrupción de recepciones de UART pone los caracteres que UART recibe en un búfer de RAM. La función `xUART_Receive()` devuelve caracteres desde el búfer de RAM.
- El parámetro `uxWantedBytes` de `xUART_Receive()` se utiliza para especificar el número de caracteres que se deben recibir. Si el búfer de RAM todavía no contiene el número de caracteres solicitado, la tarea de llamada se pone en el estado Bloqueado a la espera de recibir una notificación de que el número de caracteres que hay en el búfer ha aumentado. El bucle `while()` se usa para repetir esta secuencia hasta que el búfer de recepción contenga el número de caracteres solicitado o bien se agote el tiempo de espera.
- La tarea de llamada puede entrar en el estado Bloqueado más de una vez. El tiempo de bloqueo, por lo tanto, se ajusta para que tenga en cuenta la cantidad de tiempo que ya ha transcurrido desde que se llamara a `xUART_Receive()`. Los ajustes sirven para garantizar que el tiempo total que se pasa en `xUART_Receive()` no supere el tiempo de bloqueo especificado por el miembro `xRxTimeout` de la estructura `xUART`. El tiempo de bloqueo se ajusta con las funciones auxiliares `vTaskSetTimeoutState()` y `xTaskCheckForTimeout()` de FreeRTOS.

```
/* Driver library function to receive data from a UART. */

size_t xUART_Receive( xUART *pxUARTInstance, uint8_t *pucBuffer, size_t uxWantedBytes )
{
    size_t uxReceived = 0;

    TickType_t xTicksToWait;

    TimeOut_t xTimeout;

    /* Record the time at which this function was entered. */

    vTaskSetTimeoutState( &xTimeout );
```

```
/* xTicksToWait is the timeout value. It is initially set to the maximum receive
timeout for this UART instance. */

xTicksToWait = pxUARTInstance->RxTimeout;

/* Save the handle of the task that called this function. */
pxUARTInstance->xTaskToNotify = xTaskGetCurrentTaskHandle();

/* Loop until the buffer contains the wanted number of bytes or a timeout occurs. */
while( UART_bytes_in_rx_buffer( pxUARTInstance ) < uxWantedBytes )
{
    /* Look for a timeout, adjusting xTicksToWait to account for the time spent in this
    function so far. */

    if( xTaskCheckForTimeOut( &xTimeOut, &xTicksToWait ) != pdFALSE )
    {
        /* Timed out before the wanted number of bytes were available, exit the loop.
        */

        break;
    }

    /* The receive buffer does not yet contain the required amount of bytes. Wait for
    a maximum of xTicksToWait ticks to be notified that the receive interrupt service routine
    has placed more data into the buffer. It does not matter if the calling task already had a
    notification pending when it called this function. If it did, it would just iterate around
    this while loop one extra time. */

    ulTaskNotifyTake( pdTRUE, xTicksToWait );
}

/* No tasks are waiting for receive notifications, so set xTaskToNotify back to NULL.
*/

pxUARTInstance->xTaskToNotify = NULL;

/* Attempt to read uxWantedBytes from the receive buffer into pucBuffer. The actual
number of bytes read (which might be less than uxWantedBytes) is returned. */

uxReceived = UART_read_from_receive_buffer( pxUARTInstance, pucBuffer, uxWantedBytes );

return uxReceived;
}

/*-----*/

/* The interrupt service routine for the UART's receive interrupt */
void xUART_ReceiveISR( xUART *pxUARTInstance )
{
    BaseType_t xHigherPriorityTaskWoken = pdFALSE;

    /* Copy received data into this UART's receive buffer and clear the interrupt. */
    UART_low_level_receive( pxUARTInstance );
}
```

```
/* If a task is waiting to be notified of the new data, then notify it now. */  
if( pxUARTInstance->xTaskToNotify != NULL )  
{  
    vTaskNotifyGiveFromISR( pxUARTInstance->xTaskToNotify,  
&xHigherPriorityTaskWoken );  
    portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}  
}
```

Notificaciones de tarea usadas en controladores de dispositivos periféricos: ejemplo de ADC

En la sección anterior se ha mostrado cómo utilizar `vTaskNotifyGiveFromISR()` para enviar una notificación de tarea desde una interrupción a una tarea. `vTaskNotifyGiveFromISR()` es una función de uso sencillo, pero sus capacidades están limitadas. Solamente puede enviar una notificación de tarea como un evento sin valor. No puede enviar datos. En esta sección se muestra cómo utilizar `xTaskNotifyFromISR()` para enviar datos con un evento de notificación de tareas. La técnica se demuestra con el siguiente pseudocódigo, que proporciona el esquema de una rutina de servicio de interrupciones compatible con RTOS interrumpir para un convertidor de análogo a digital (ADC).

- Se presupone que comienza una conversión ADC como mínimo cada 50 milisegundos.
- `ADC_ConversionEndISR()` es la rutina del servicio de interrupciones de interrupción final de la conversión de ADC, que es la interrupción que se ejecuta cada vez que hay un nuevo valor de ADC disponible.
- La tarea que `vADCTask()` implementa procesa cada uno de los valores generados por ADC. Se presupone que el controlador de la tarea se ha almacenado en `xADCTaskToNotify` cuando se creó la tarea.
- `ADC_ConversionEndISR()` utiliza `xTaskNotifyFromISR()` con el parámetro `eAction` establecido en `eSetValueWithoutOverwrite` para enviar una notificación de tarea a la tarea `vADCTask()` y escribir el resultado de la conversión ADC en el valor de notificación de la tarea.
- La tarea `vADCTask()` utiliza `xTaskNotifyWait()` para esperar a recibir una notificación de que hay un nuevo valor ADC disponible y para recuperar el resultado de la conversión ADC desde su valor de notificación.

```
/* A task that uses an ADC. */  
void vADCTask( void *pvParameters )  
{  
    uint32_t ulADCValue;  
    BaseType_t xResult;  
  
    /* The rate at which ADC conversions are triggered. */  
    const TickType_t xADCConversionFrequency = pdMS_TO_TICKS( 50 );
```

```
for( ;; )
{
    /* Wait for the next ADC conversion result. */

    xResult = xTaskNotifyWait( /* The new ADC value will overwrite the old value, so
there is no need to clear any bits before waiting for the new notification value. */ 0, /*
Future ADC values will overwrite the existing value, so there is no need to clear any bits
before exiting xTaskNotifyWait(). */ 0, /* The address of the variable into which the
task's notification value (which holds the latest ADC conversion result) will be copied.
*/ &ulADCValue, /* A new ADC value should be received every xADCConversionFrequency ticks.
*/ xADCConversionFrequency * 2 );

    if( xResult == pdPASS )
    {
        /* A new ADC value was received. Process it now. */

        ProcessADCResult( ulADCValue );
    }
    else
    {
        /* The call to xTaskNotifyWait() did not return within the expected time.
Something must be wrong with the input that triggers the ADC conversion or with the ADC
itself. Handle the error here. */

    }
}

/*-----*/

/* The interrupt service routine that executes each time an ADC conversion completes. */
void ADC_ConversionEndISR( xADC *pxADCInstance )
{
    uint32_t ulConversionResult;

    BaseType_t xHigherPriorityTaskWoken = pdFALSE, xResult;

    /* Read the new ADC value and clear the interrupt. */

    ulConversionResult = ADC_low_level_read( pxADCInstance );

    /* Send a notification, and the ADC conversion result, directly to vADCTask(). */

    xResult = xTaskNotifyFromISR( xADCTaskToNotify, /* xTaskToNotify parameter. */
ulConversionResult, /* ulValue parameter. */ eSetValueWithoutOverwrite, /* eAction
parameter. */ &xHigherPriorityTaskWoken );

    /* If the call to xTaskNotifyFromISR() returns pdFAIL then the task is not keeping
up with the rate at which ADC values are being generated. configASSERT() is described in
section 11.2.*/

    configASSERT( xResult == pdPASS );
}
```



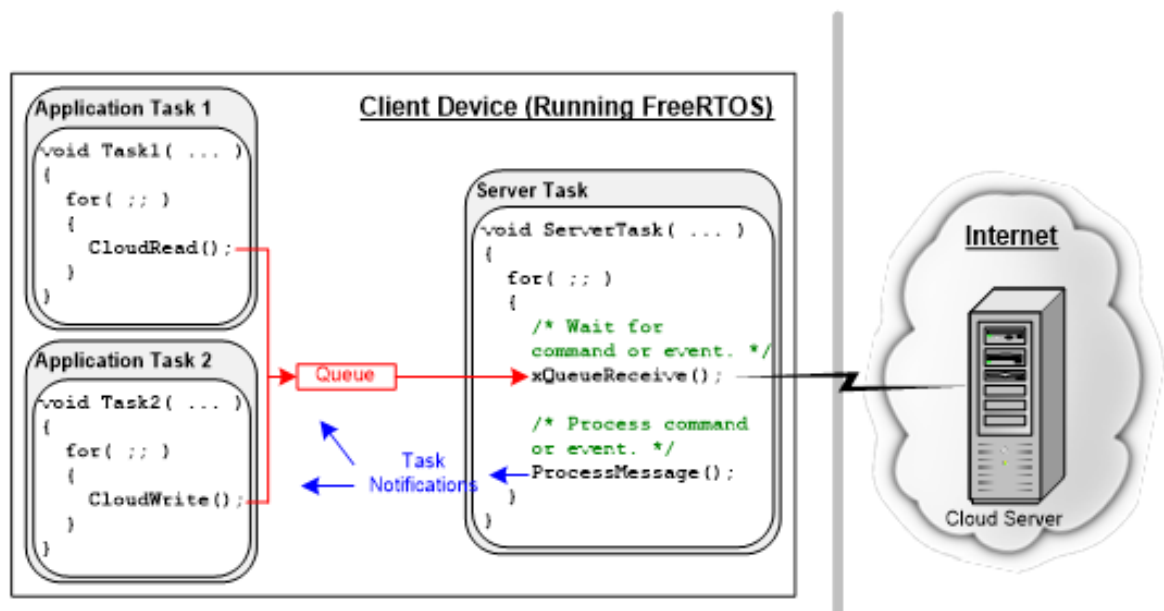
```
portYIELD_FROM_ISR( xHigherPriorityTaskWoken );  
}
```

Notificaciones de tarea usadas directamente dentro de una aplicación

En esta sección se muestra el uso de las notificaciones de tareas en una aplicación hipotética que incluye la siguiente funcionalidad:

1. La aplicación se comunica a través de una conexión a Internet lenta para enviar y solicitar datos a un servidor de datos remoto (el servidor de la nube).
2. Después de solicitar datos al servidor de la nube, la tarea que realiza la solicitud debe esperar en el estado Bloqueado a recibir los datos solicitados.
3. Después de enviar datos al servidor de la nube, la tarea que realiza el envío tiene que esperar en el estado Bloqueado a recibir un reconocimiento de que el servidor de la nube ha recibido los datos correctamente.

El diseño de software se muestra aquí.



- La complejidad de gestionar varias conexiones de Internet al servidor de la nube se encapsula en una única tarea FreeRTOS. La tarea actúa como servidor proxy dentro de la aplicación de FreeRTOS y se conoce como tarea de servidor.
- Las tareas de aplicación leen datos del servidor de la nube llamando a `CloudRead()`. `CloudRead()` no se comunica con el servidor de la nube directamente. En su lugar, envía la solicitud de lectura a la tarea de servidor en una cola y recibe los datos solicitados desde la tarea de servidor como notificación de tarea.
- Fecha de escritura de tareas de aplicación en el servidor de la nube llamando a `CloudWrite()`. `CloudWrite()` no se comunica con el servidor de la nube directamente. En su lugar, envía la solicitud de escritura a la tarea de servidor de una cola y recibe el resultado de la operación de escritura desde la tarea de servidor como notificación de tarea.

El código siguiente muestra la estructura que las funciones CloudRead() y CloudWrite() envían a la tarea de servidor.

```
typedef enum CloudOperations
{
    eRead, /* Send data to the cloud server. */
    eWrite /* Receive data from the cloud server. */
} Operation_t;

typedef struct CloudCommand
{
    Operation_t eOperation; /* The operation to perform (read or write). */
    uint32_t ulDataID; /* Identifies the data being read or written. */
    uint32_t ulDataValue; /* Only used when writing data to the cloud server. */
    TaskHandle_t xTaskToNotify; /* The handle of the task performing the operation. */
} CloudCommand_t;
```

El pseudocódigo para CloudRead() se muestra aquí. La función envía su solicitud a la tarea de servidor y, a continuación, llama a xTaskNotifyWait() para que espere en el estado Bloqueado hasta que reciba la notificación de que los datos solicitados están disponibles.

```
/* ulDataID identifies the data to read. pulValue holds the address of the variable into
which the data received from the cloud server is to be written. */

BaseType_t CloudRead( uint32_t ulDataID, uint32_t *pulValue )
{
    CloudCommand_t xRequest;

    BaseType_t xReturn;

    /* Set the CloudCommand_t structure members to be correct for this read request. */
    xRequest.eOperation = eRead; /* This is a request to read data. */

    xRequest.ulDataID = ulDataID; /* A code that identifies the data to read. */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

    /* Ensure there are no notifications already pending by reading the notification value
    with a block time of 0, and then send the structure to the server task. */

    xTaskNotifyWait( 0, 0, NULL, 0 );

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

    /* Wait for a notification from the server task. The server task writes the value
    received from the cloud server directly into this task's notification value, so there
    is no need to clear any bits in the notification value on entry to or exit from the
    xTaskNotifyWait() function. The received value is written to *pulValue, so pulValue is
    passed as the address to which the notification value is written. */
```

```
xReturn = xTaskNotifyWait( 0, /* No bits cleared on entry. */ 0, /* No bits to clear
on exit. */ pulValue, /* Notification value into *pulValue. */ pdMS_TO_TICKS( 250 ) );
    /* Wait a maximum of 250ms. */
    /* If xReturn is pdPASS, then the value was obtained. If xReturn is pdFAIL,
then the request timed out. */

    return xReturn;
}
```

El pseudocódigo que muestra cómo la tarea de servidor administra una solicitud de lectura se muestra aquí. Una vez que se han recibido los datos del servidor de la nube, la tarea de servidor desbloquea la tarea de aplicación y envía los datos recibidos a la tarea de aplicación llamando a `xTaskNotify()` con el parámetro `eAction` establecido en `eSetValueWithOverwrite`.

Se trata de una situación simplificada, ya que presupone que `GetCloudData()` no tiene que esperar a obtener un valor del servidor de la nube.

```
void ServerTask( void *pvParameters )
{
    CloudCommand_t xCommand;

    uint32_t ulReceivedValue;

    for( ;; )
    {
        /* Wait for the next CloudCommand_t structure to be received from a task. */
        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );

        switch( xCommand.eOperation ) /* Was it a read or write request? */
        {
            case eRead:

                /* Obtain the requested data item from the remote cloud server. */
                ulReceivedValue = GetCloudData( xCommand.ulDataID );

                /* Call xTaskNotify() to send both a notification and the value received from
the cloud server to the task that made the request. The handle of the task is obtained
from the CloudCommand_t structure. */
                xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.
*/ ulReceivedValue, /* Cloud data sent as notification value. */ eSetValueWithOverwrite );

                break;

            /* Other switch cases go here. */

        }
    }
}
```

El pseudocódigo para CloudWrite() se muestra aquí. A efectos de esta demostración, CloudWrite() devuelve un código de estado bit a bit, donde a cada bit del código de estado se le asigna un significado único. Las instrucciones #define muestran cuatro ejemplos de bits de estado en la parte superior.

La tarea borra los cuatro bits de estado, envía su solicitud a la tarea de servidor y, a continuación, llama a xTaskNotifyWait() para que espere en el estado Bloqueado a la notificación de estado.

```
/* Status bits used by the cloud write operation. */

#define SEND_SUCCESSFUL_BIT ( 0x01 << 0 )

#define OPERATION_TIMED_OUT_BIT ( 0x01 << 1 )

#define NO_INTERNET_CONNECTION_BIT ( 0x01 << 2 )

#define CANNOT_LOCATE_CLOUD_SERVER_BIT ( 0x01 << 3 )

/* A mask that has the four status bits set. */

#define CLOUD_WRITE_STATUS_BIT_MASK ( SEND_SUCCESSFUL_BIT \
| OPERATION_TIMED_OUT_BIT \
| NO_INTERNET_CONNECTION_BIT \
| CANNOT_LOCATE_CLOUD_SERVER_BIT )

uint32_t CloudWrite( uint32_t ulDataID, uint32_t ulDataValue )
{
    CloudCommand_t xRequest;

    uint32_t ulNotificationValue;

    /* Set the CloudCommand_t structure members to be correct for this write request. */

    xRequest.eOperation = eWrite; /* This is a request to write data. */

    xRequest.ulDataID = ulDataID; /* A code that identifies the data being written. */

    xRequest.ulDataValue = ulDataValue; /* Value of the data written to the cloud server.
    */

    xRequest.xTaskToNotify = xTaskGetCurrentTaskHandle(); /* Handle of the calling task. */

    /* Clear the three status bits relevant to the write operation by
    calling xTaskNotifyWait() with the ulBitsToClearOnExit parameter set to
    CLOUD_WRITE_STATUS_BIT_MASK, and a block time of 0. The current notification value is not
    required, so the pulNotificationValue parameter is set to NULL. */

    xTaskNotifyWait( 0, CLOUD_WRITE_STATUS_BIT_MASK, NULL, 0 );

    /* Send the request to the server task. */

    xQueueSend( xServerTaskQueue, &xRequest, portMAX_DELAY );

    /* Wait for a notification from the server task. The server task writes a bitwise
    status code into this task's notification value, which is written to ulNotificationValue.
    */

    xTaskNotifyWait( 0, /* No bits cleared on entry. */ CLOUD_WRITE_STATUS_BIT_MASK, /
    * Clear relevant bits to 0 on exit. */ &ulNotificationValue, /* Notified value. */
```

```
pdMS_TO_TICKS( 250 ) ); /* Wait a maximum of 250ms. */ /* Return the status code to the  
calling task. */ return ( ulNotificationValue & CLOUD_WRITE_STATUS_BIT_MASK );
```

El pseudocódigo que demuestra cómo la tarea de servidor administra una solicitud de escritura se muestra aquí. Una vez que se han enviado los datos al servidor de la nube, la tarea de servidor desbloquea la tarea de aplicación y envía el código de estado bit a bit a la tarea de aplicación llamando a `xTaskNotify()` con el parámetro `eAction` establecido en `eSetBits`. Solo los bits definidos por la constante `CLOUD_WRITE_STATUS_BIT_MASK` pueden resultar alterados en el valor de notificación de la tarea que recibe la llamada, por lo que esta tarea puede utilizar otros bits en su valor de notificación para otros fines.

Esto es una situación simplificada, ya que presupone que `SetCloudData()` no tiene que esperar a obtener un reconocimiento del servidor de la nube remoto.

```
void ServerTask( void *pvParameters )  
{  
    CloudCommand_t xCommand;  
    uint32_t ulBitwiseStatusCode;  
    for( ;; )  
    {  
        /* Wait for the next message. */  
        xQueueReceive( xServerTaskQueue, &xCommand, portMAX_DELAY );  
        /* Was it a read or write request? */  
        switch( xCommand.eOperation )  
        {  
            case eWrite:  
                /* Send the data to the remote cloud server. SetCloudData() returns a bitwise  
                status code that only uses the bits defined by the CLOUD_WRITE_STATUS_BIT_MASK definition  
                (shown in the preceding code). */  
                ulBitwiseStatusCode = SetCloudData( xCommand.ulDataID, xCommand.ulDataValue );  
                /* Send a notification to the task that made the write request. The eSetBits  
                action is used so any status bits set in ulBitwiseStatusCode will be set in the  
                notification value of the task being notified. All the other bits remain unchanged. The  
                handle of the task is obtained from the CloudCommand_t structure. */  
                xTaskNotify( xCommand.xTaskToNotify, /* The task's handle is in the structure.  
                */ ulBitwiseStatusCode, /* Cloud data sent as notification value. */ eSetBits );  
                break;  
                /* Other switch cases go here. */  
            }  
        }  
    }  
}
```

Developer Support

En esta sección, se explican características que maximizan la productividad de la siguiente manera:

- Proporcionando información sobre el comportamiento de una aplicación.
- Destacando las oportunidades de optimización.
- Capturando errores en el punto en el que se producen.

configASSERT()

En C, la macro `assert()` se utiliza para verificar una aserción (suposición) que realiza el programa. La aserción se escribe como una expresión de C. Si la expresión se evalúa como false (0), se considera que la aserción ha fallado. Por ejemplo, este código prueba la aserción que indica que el puntero `pxMyPointer` no es NULL.

```
/* Test the assertion that pxMyPointer is not NULL */  
  
assert( pxMyPointer != NULL );
```

El programador de la aplicación especifica la acción que debe realizarse si una aserción falla proporcionando una implementación de la macro `assert()`.

El código fuente de FreeRTOS no llama a `assert()`, porque `assert()` no está disponible en todos los compiladores en los que se compila FreeRTOS. En su lugar, el código fuente de FreeRTOS contiene una gran cantidad de llamadas a una macro denominada `configASSERT()`, que el programador de la aplicación puede definir en `FreeRTOSConfig.h` y que se comporta exactamente como el `assert()` estándar de C.

El error de una aserción debe tratarse como un error fatal. No intente realizar la ejecución después de una línea que tiene un error de aserción.

Con `configASSERT()`, se mejora la productividad mediante la captura y la identificación inmediatas de muchos de los orígenes de errores más comunes. Le recomendamos encarecidamente que defina `configASSERT()` mientras desarrolla o depura una aplicación FreeRTOS.

La definición de `configASSERT()` le ayudará a realizar la depuración en tiempo de ejecución, pero también aumentará el tamaño del código de la aplicación y, por lo tanto, ralentizará su ejecución. Si no se proporciona una definición de `configASSERT()`, se utilizará la definición vacía predeterminada y el preprocesador de C eliminará por completo todas las llamadas a `configASSERT()`.

Definiciones de ejemplo de configASSERT()

La definición de `configASSERT()` que se muestra en el siguiente código es útil cuando se está ejecutando una aplicación bajo el control de un depurador. Detiene la ejecución en cualquier línea en la que falla una aserción, para que el depurador muestre la línea en la que ha fallado la aserción cuando la sesión de depuración se pone en pausa.

```
/* Disable interrupts so the tick interrupt stops executing, and then sit in a loop so  
execution does not move past the line that failed the assertion. If the hardware supports
```

```
a debug break instruction, then the debug break instruction can be used in place of the
for() loop. */

#define configASSERT( x ) if( ( x ) == 0 ) { taskDISABLE_INTERRUPTS();

for(;;); }
```

La definición de configASSERT() es útil cuando no se está ejecutando una aplicación bajo el control de un depurador. Imprime o registra de alguna forma la línea de código fuente en la que ha fallado una aserción. Puede identificar la línea en la que ha fallado la aserción mediante la macro estándar de C `__FILE__` para obtener el nombre del archivo de origen y la macro estándar de C `__LINE__` para obtener el número de línea en el archivo de origen.

Este código muestra una definición configASSERT() que registra la línea de código fuente en la que ha fallado una aserción.

Tracealyzer

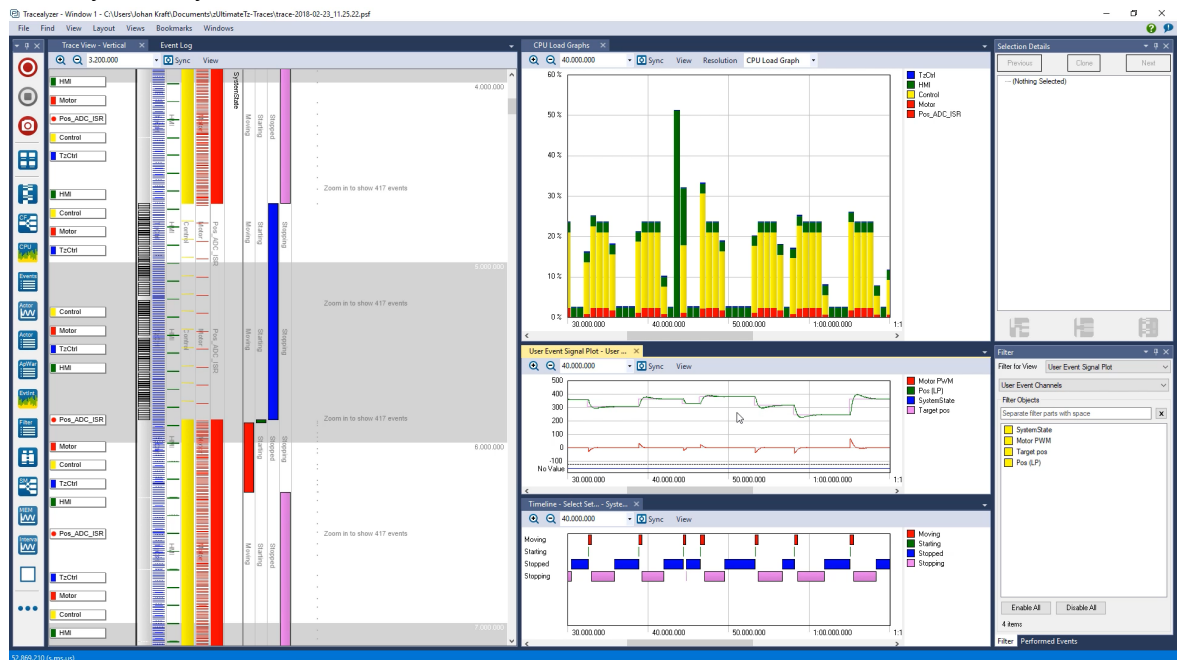
Tracealyzer es una herramienta de diagnóstico y optimización en tiempo de ejecución de nuestro socio Percepio.

Tracealyzer captura valiosa información sobre el comportamiento dinámico y la presenta en vistas gráficas interconectadas. La herramienta también puede mostrar varias vistas sincronizadas.

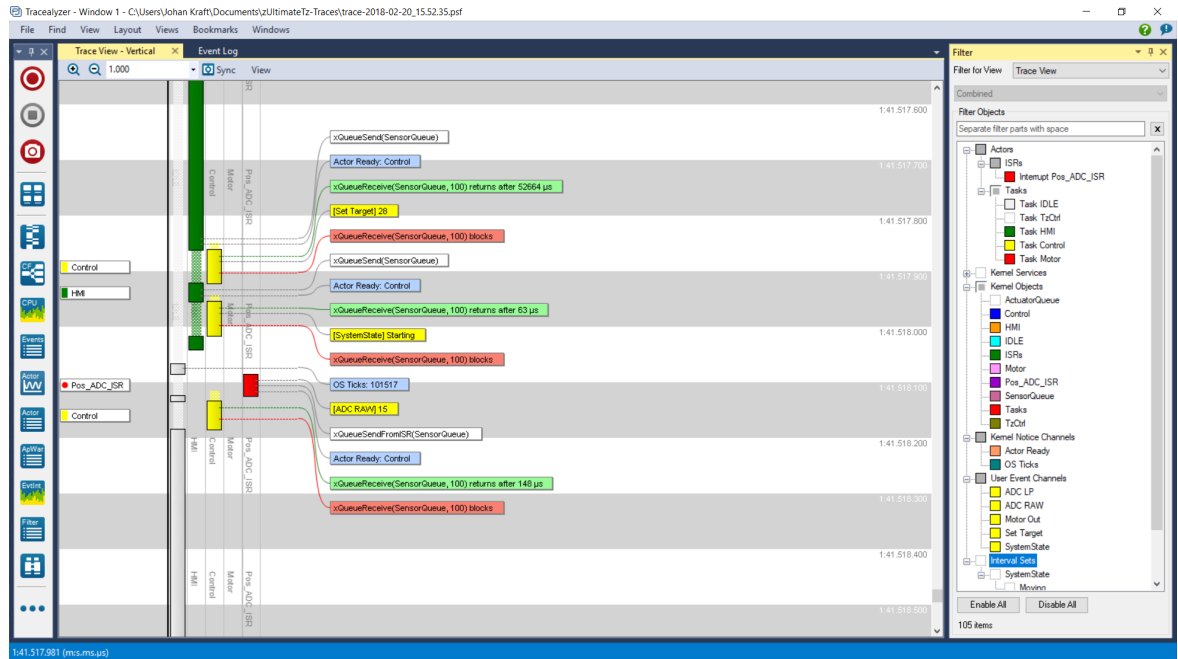
La información que obtiene es útil para realizar análisis, resolver problemas o simplemente optimizar una aplicación de FreeRTOS.

Tracealyzer se puede utilizar en paralelo con un depurador tradicional. Complementa la vista del depurador con una perspectiva basada en el tiempo de nivel superior.

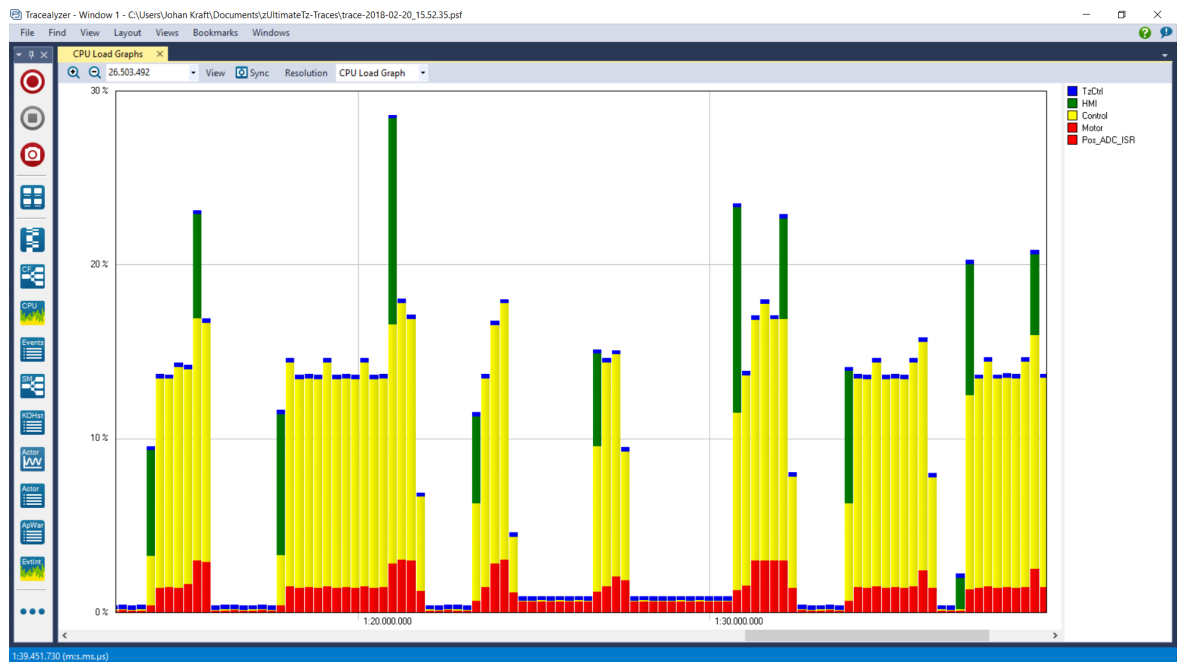
Tracealyzer incluye más de 20 vistas interconectadas:



Vista de rastreo vertical:

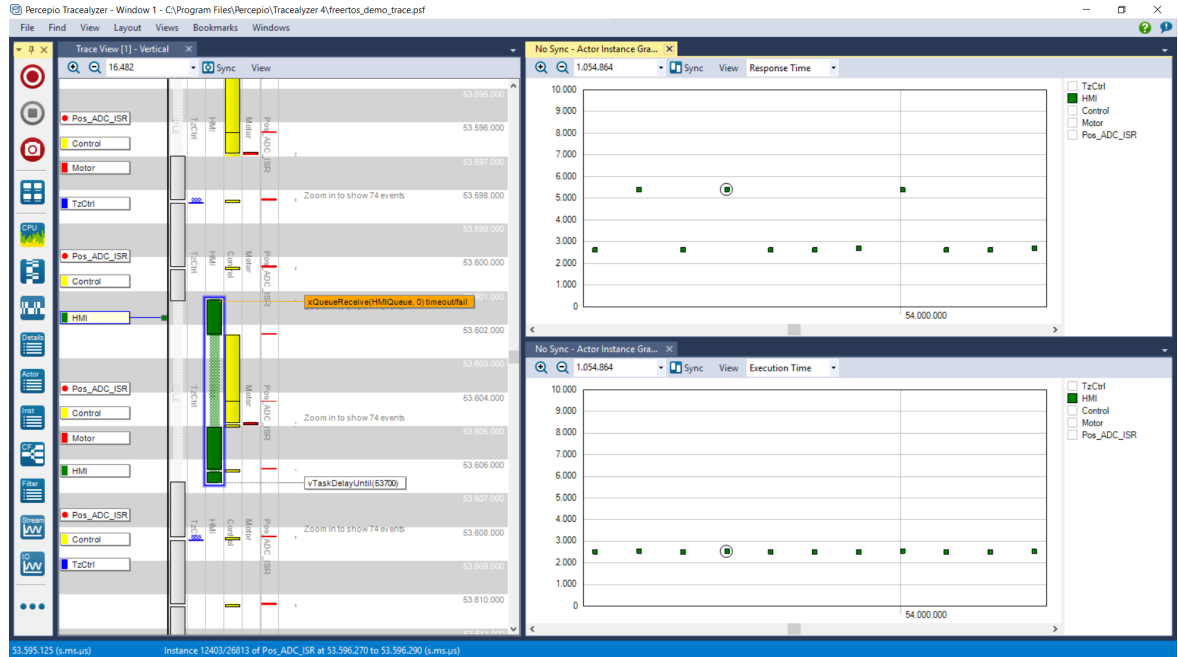


Vista de carga de CPU:

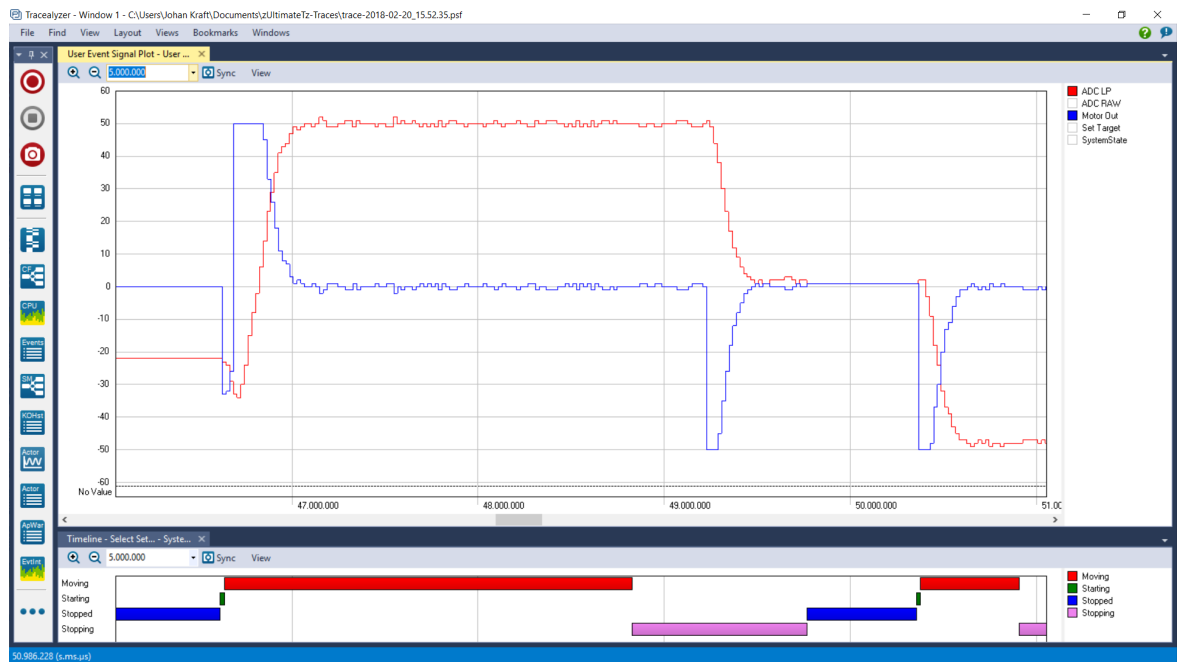


Vista de tiempo de respuesta:

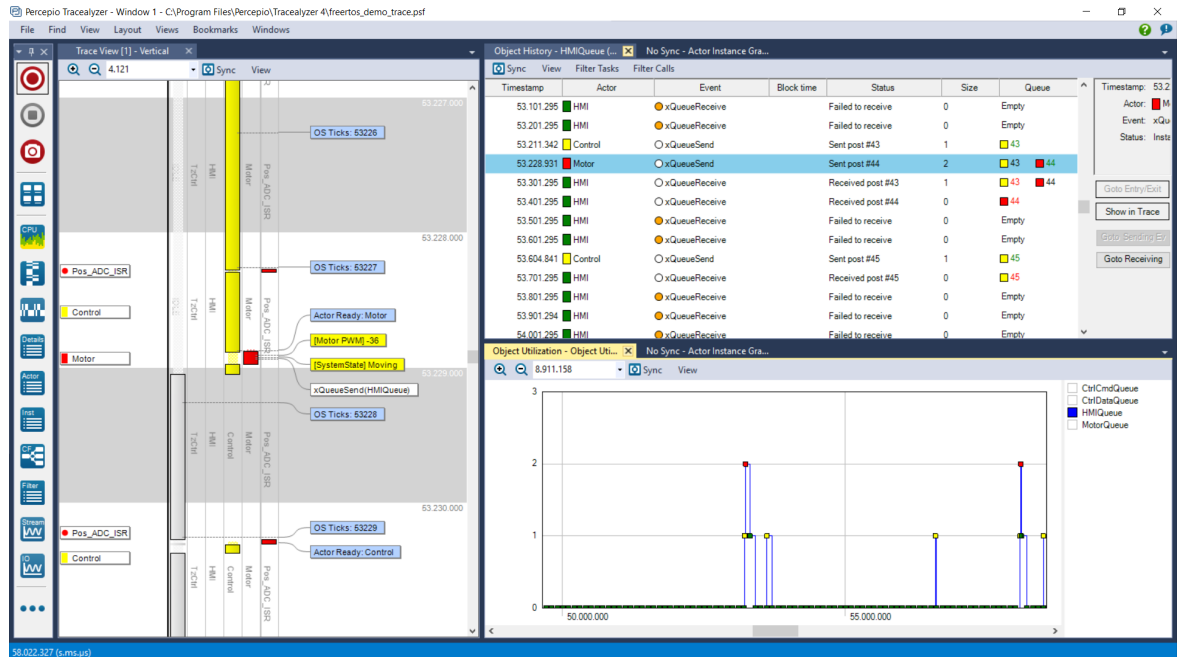
Kernel FreeRTOS Guía para desarrolladores Tracealyzer



Vista de gráfico de dispersión de eventos de usuario:



Vista del historial de objetos:



Funciones de enlace relacionadas con la depuración (devolución de llamadas)

La definición de un enlace a una llamada Malloc fallida garantiza que se notifique inmediatamente al desarrollador de la aplicación si se produce un error al intentar crear una tarea, una cola, un semáforo o un grupo de eventos. Para obtener más información sobre el enlace a la llamada Malloc fallida (o devolución de llamada), consulte [Administración de la memoria en montón \(p. 14\)](#).

La definición de un enlace de desbordamiento de pila garantiza que se notifique al desarrollador de la aplicación si la cantidad de pila que utiliza una tarea supera el espacio de pila asignado a la tarea. Para obtener más información sobre el enlace de desbordamiento de pila, consulte la sección sobre el [desbordamiento de pilas \(p. 249\)](#) del apartado de resolución de problemas.

Visualización de la información de estado de tareas y tiempo de ejecución

Estadísticas del tiempo de ejecución de tareas

Las estadísticas del tiempo de ejecución de tareas proporcionan información sobre la cantidad de tiempo de procesamiento que ha recibido cada tarea. El tiempo de ejecución de una tarea es el tiempo total que la tarea lleva en el estado En ejecución desde que se inició la aplicación.

Las estadísticas del tiempo de ejecución están pensadas como ayuda para crear perfiles y realizar la depuración durante la fase de desarrollo de un proyecto. La información que proporcionan solo es válida hasta que el contador que se utiliza como reloj para las estadísticas de tiempo de ejecución se desborda. La recopilación de las estadísticas de tiempo de ejecución aumenta el tiempo de cambio de contexto de las tareas.

Para obtener la información de las estadísticas de tiempo de ejecución binario, llame a la función de API `uxTaskGetSystemState()`. Para obtener la información de las estadísticas de tiempo de ejecución en una tabla ASCII legible, llame a la función auxiliar `vTaskGetRunTimeStats()`.

Reloj de estadísticas de tiempo de ejecución

Las estadísticas de tiempo de ejecución deben medir fracciones de un periodo de ciclo. Por este motivo, el recuento de ciclos de RTOS no se utiliza como reloj de estadísticas de tiempo de ejecución. Este reloj lo proporciona el código de la aplicación. Le recomendamos que aumente la frecuencia del reloj de estadísticas de tiempo de ejecución para que sea entre 10 y 100 veces más rápido que la frecuencia de la interrupción de ciclos. Cuanto más rápido sea el reloj de estadísticas de tiempo de ejecución, más precisas serán las estadísticas, pero también se desbordará antes el valor de tiempo.

Lo ideal es que el valor de tiempo lo genere un temporizador/contador periférico de 32 bits de libre ejecución para que este valor se pueda leer sin ninguna otra sobrecarga de procesamiento. Si los periféricos disponibles y las velocidades de reloj hacen que no sea posible usar esa técnica, hay técnicas alternativas, pero son menos eficientes:

1. Configurar un periférico para generar una interrupción periódica en la frecuencia de reloj de las estadísticas de tiempo de ejecución deseada y, a continuación, usar un recuento del número de interrupciones generadas como reloj de estadísticas de tiempo de ejecución.

Este método es muy poco eficaz si la interrupción periódica solo se utiliza con el fin de proporcionar un reloj de estadísticas de tiempo de ejecución. Sin embargo, si la aplicación ya utiliza una interrupción periódica con una frecuencia adecuada, resulta sencillo y eficaz añadir un recuento del número de interrupciones generadas en la rutina del servicio de interrupciones existente.

2. Generar un valor de 32 bits utilizando el valor actual de un temporizador periférico de 16 bits de libre ejecución como los 16 bits menos importantes del valor de 32 bits y el número de veces que el temporizador se ha desbordado como los 16 bits más importantes del valor de 32 bits.

Con una manipulación un poco compleja, puede generar un reloj de estadísticas de tiempo de ejecución combinando el recuento de ciclos de RTOS con el valor actual de un temporizador Cortex-M SysTick de ARM. Este procedimiento se explica en algunos de los proyectos de demostración que se incluyen en la descarga de FreeRTOS.

Configuración de una aplicación para recopilar estadísticas de tiempo de ejecución

En la siguiente tabla, se muestran las macros necesarias para recopilar estadísticas de tiempo de ejecución de las tareas. Las macros tienen el prefijo "port", ya que su objetivo original era ser incluidas en la capa del puerto de RTOS. Es más práctico definir las macros en `FreeRTOSConfig.h`.

Macro	Descripción
<code>configGENERATE_RUN_TIME_STATS</code>	Esta macro se debe establecer en 1 en <code>FreeRTOSConfig.h</code> . Cuando esta macro se establece en 1, el programador llama al resto de macros que se detallan en esta tabla en los momentos apropiados.
<code>portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()</code>	Esta macro debe proporcionarse para inicializar el periférico que se utiliza para proporcionar el reloj de estadísticas de tiempo de ejecución.

portGET_RUN_TIME_COUNTER_VALUE() o	Una de estas dos macros se debe proporcionar para devolver el valor del reloj de las estadísticas de tiempo de ejecución actual. Es el tiempo total que se ha estado ejecutando la aplicación, en unidades del reloj de estadísticas de tiempo de ejecución, desde que la aplicación se inició por primera vez.
portALT_GET_RUN_TIME_COUNTER_VALUE(Time)	Si se ha utilizado la primera macro, debe definirse para evaluar el valor de reloj actual. Si se ha utilizado la segunda macro, debe definirse para establecer su parámetro "Time" en el valor de reloj actual.

Función de API uxTaskGetSystemState()

uxTaskGetSystemState() proporciona una instantánea de la información de estado de cada tarea que está bajo el control del programador de FreeRTOS. La información se proporciona como una matriz de estructuras TaskStatus_t, con un índice en la matriz para cada tarea. Aquí se muestra el prototipo de la función de API uxTaskGetSystemState().

En la siguiente tabla se muestran los parámetros de uxTaskGetSystemState() y el valor de retorno.

Nombre del parámetro	Descripción
pxTaskStatusArray	<p>Un puntero a una matriz de estructuras TaskStatus_t.</p> <p>La matriz debe contener al menos una estructura TaskStatus_t para cada tarea. El número de tareas puede determinarse mediante la función de API uxTaskGetNumberOfTasks().</p> <p>La estructura TaskStatus_t se muestra en el siguiente código. Los miembros de la estructura TaskStatus_t se describen en la siguiente tabla.</p>
uxArraySize	<p>El tamaño de la matriz a la que apunta el parámetro pxTaskStatusArray. El tamaño se especifica como el número de índices de la matriz (el número de estructuras TaskStatus_t incluidas en la matriz) y no por el número de bytes de la matriz.</p>
pulTotalRunTime	<p>Si configGENERATE_RUN_TIME_STATS se establece en 1 en FreeRTOSConfig.h, <problematic>*</problematic> uxTaskGetSystemState() se establece por medio de pulTotalRunTime() en el tiempo de ejecución total (tal y como define el reloj de estadísticas de tiempo de ejecución que proporciona la aplicación) desde que se inició el destino.</p> <p>pulTotalRunTime es opcional y se puede configurar en NULL si el tiempo de ejecución total no es necesario.</p>

Valor devuelto	<p>Se devuelve el número de estructuras TaskStatus_t que ha rellenado uxTaskGetSystemState().</p> <p>El valor devuelto debe coincidir con el número devuelto por la función de API uxTaskGetNumberOfTasks(), pero será cero si el valor que se ha pasado en el parámetro uxArraySize era demasiado pequeño.</p>
----------------	---

A continuación, se muestra la estructura TaskStatus_t.

```
typedef struct xTASK_STATUS
{
    TaskHandle_t xHandle;

    const char *pcTaskName;

    UBaseType_t xTaskNumber;

    eTaskState eCurrentState;

    UBaseType_t uxCurrentPriority;

    UBaseType_t uxBasePriority;

    uint32_t ulRunTimeCounter;

    uint16_t usStackHighWaterMark;
} TaskStatus_t;
```

En la siguiente tabla, se enumeran los miembros de la estructura TaskStatus_t.

Nombre de parámetro / Valor devuelto	Descripción
xHandle	El controlador de la tarea con la que está relacionada la información en la estructura.
pcTaskName	El nombre en texto legible de la tarea.
xTaskNumber	<p>Cada tarea tiene un valor de xTaskNumber único.</p> <p>Si una aplicación crea y elimina tareas en tiempo de ejecución, es posible que una tarea tenga el mismo controlador que una tarea que se ha eliminado. xTaskNumber se proporciona para que el código de la aplicación y los depuradores compatibles con el kernel distingan entre una tarea que sigue siendo válida y una tarea eliminada que tenía el mismo controlador que la tarea válida.</p>
eCurrentState	Un tipo enumerado que contiene el estado de la tarea. eCurrentState puede tener uno de los siguientes valores: eRunning, eReady, eBlocked, eSuspended, eDeleted.

	<p>Solo se informará de que una tarea se encuentra en el estado eDeleted durante el breve periodo, entre el momento en que se ha eliminado por medio de una llamada a vTaskDelete() y el momento en que la tarea de inactividad libera la memoria que se había asignado a las estructuras de datos internas y la pila de la tarea eliminada. Después de ese momento, la tarea dejará de existir por completo y no es válido intentar utilizar su controlador.</p>
uxCurrentPriority	<p>La prioridad a la que se estaba ejecutando la tarea en el momento en que se llamó a uxTaskGetSystemState(). uxCurrentPriority solo tendrá una prioridad mayor que la asignada a la tarea por el programador de la aplicación si a la tarea se le ha asignado temporalmente una prioridad mayor de acuerdo con el mecanismo de herencia de prioridades que se describe en Administración de recursos (p. 161).</p>
uxBasePriority	<p>La prioridad que asigna a la tarea el programador de la aplicación. uxBasePriority solo es válido si Config.h. configUSE_MUTEXES está establecido en 1 en FreeRTOSConfig.h.</p>
ulRunTimeCounter	<p>El tiempo de ejecución total que ha empleado la tarea desde que se creó. El tiempo de ejecución total se proporciona como un tiempo absoluto que utiliza el reloj que proporciona el programador de la aplicación para recopilar estadísticas de tiempo de ejecución. ulRunTimeCounter solo es válido si configGENERATE_RUN_TIME_STATS está establecido en 1 en FreeRTOSConfig.h.</p>
usStackHighWaterMark	<p>La marca de agua máxima de la pila de la tarea. Es la cantidad mínima de espacio de pila que ha quedado para la tarea desde que se creó. Es una indicación de lo cerca que ha estado la tarea de desbordar su pila. Cuanto más cerca esté este valor de cero, más cerca habrá estado la tarea de desbordar su pila. usStackHighWaterMark se especifica en bytes.</p>

Función auxiliar vTaskList()

vTaskList() proporciona información sobre el estado de la tarea muy parecida a la que proporciona uxTaskGetSystemState(), pero la presenta como una tabla ASCII legible en lugar de en una matriz de valores binarios.

vTaskList() es una función que utiliza mucho procesador. Deja al programador suspendido durante un largo periodo de tiempo. Por lo tanto, le recomendamos que utilice la función solo para la depuración y no en un sistema de producción en tiempo real.

vTaskList() está disponible si configUSE_TRACE_FACILITY y configUSE_STATS_FORMATTING_FUNCTIONS están establecidas en 1 en FreeRTOSConfig.h.

```
void vTaskList( signed char *pcWriteBuffer );
```

En la tabla siguiente se enumera el parámetro de vTaskList().

Nombre del parámetro	Descripción
pcWriteBuffer	Un puntero a un búfer de caracteres en el que se escribe la tabla formateada y en formato legible. El búfer debe ser lo suficientemente grande como para almacenar toda la tabla. No se realiza ninguna comprobación de límite.

Aquí se muestra la salida generada por vTaskList().

tcpip	R	3	393	0
Tmr Svc	R	3	111	48
QConsB1	R	1	143	3
QProdB5	R	0	144	7
QConsB6	R	0	143	8
PolSEM1	R	0	145	11
PolSEM2	R	0	145	12
GenQ	R	0	155	17
MuLow	R	0	147	18
Rec3	R	0	141	30
SUSP_RX	R	0	148	36
Math1	R	0	167	38
Math2	R	0	167	39

En la salida:

- Cada fila proporciona información sobre una sola tarea.
- La primera columna es el nombre de la tarea.
- La segunda columna es el estado de la tarea, donde "R" significa Ready (Listo) "B" significa Blocked (Bloqueado), "S" significa Suspended (Suspendido) y "D" significa que la tarea se ha eliminado. Solo se informará de que una tarea se encuentra en el estado eliminado durante el breve periodo entre el momento en que se ha eliminado por medio de una llamada a vTaskDelete() y el momento en que la tarea de inactividad libera la memoria que se ha asignado a las estructuras de datos internas y la pila de la tarea eliminada. Después de ese momento, la tarea dejará de existir por completo y no es válido intentar utilizar su controlador.
- La tercera columna es la prioridad de la tarea.
- La cuarta columna es marca de agua máxima de la pila de la tarea. Consulte la descripción de usStackHighWaterMark en la tabla para ver los miembros de la estructura TaskStatus_t.
- La quinta columna es el número único asignado a la tarea. Consulte la descripción de xTaskNumber en la tabla para ver los miembros de la estructura TaskStatus_t.

Función auxiliar vTaskGetRunTimeStats()

vTaskGetRunTimeStats() formatea las estadísticas de tiempo de ejecución recopiladas en una tabla ASCII legible.

vTaskGetRunTimeStats() es una función que utiliza mucho procesador. Deja al programador suspendido durante un largo periodo de tiempo. Por este motivo, le recomendamos que utilice la función solo para la depuración y no en un sistema de producción en tiempo real.

vTaskGetRunTimeStats() está disponible cuando configGENERATE_RUN_TIME_STATS y configUSE_STATS_FORMATTING_FUNCTIONS están establecidos en 1 en FreeRTOSConfig.h.

Aquí se muestra el prototipo de la función de API vTaskGetRunTimeStats().

```
void vTaskGetRunTimeStats( signed char *pcWriteBuffer );
```

En la tabla siguiente se enumera el parámetro de vTaskGetRunTimeStats().

Nombre del parámetro	Descripción
pcWriteBuffer	Un puntero a un búfer de caracteres en el que se escribe la tabla formateada y en formato legible. El búfer debe ser lo suficientemente grande como para almacenar toda la tabla. No se realiza ninguna comprobación de límite.

Aquí se muestra la salida generada por vTaskGetRunTimeStats().

PolSEM1	994	<1%
PolSEM2	23248	1%
GenQ	194479	16%
MuLow	3690	<1%
Rec3	229450	18%
CNT1	242720	19%
PeekL	94	<1%
CNT_INC	165	<1%
CNT2	243166	20%
SUSP_RX	243192	20%
IDLE	55	<1%

En la salida:

- Cada fila proporciona información sobre una sola tarea.
- La primera columna es el nombre de la tarea.
- La segunda columna es la cantidad de tiempo que ha estado la tarea en el estado En ejecución como un valor absoluto. Consulte la descripción de ulRunTimeCounter en la tabla para ver los miembros de la estructura TaskStatus_t.
- La tercera columna es la cantidad de tiempo que ha estado la tarea en el estado En ejecución como porcentaje del tiempo total desde que se inició el destino. El total de los tiempos porcentuales mostrados suele ser inferior al 100 % esperado, ya que las estadísticas se recopilan y calculan utilizando cálculos de enteros que se redondean hacia abajo al valor del entero más próximo.

Generación y visualización de estadísticas en tiempo de ejecución: ejemplo de trabajo

En este ejemplo, se utiliza un hipotético temporizador de 16 bits para generar un reloj de estadísticas de tiempo de ejecución de 32 bits. El contador está configurado para generar una interrupción cada vez que el valor de 16 bits alcanza su valor máximo, lo que crea una interrupción de desbordamiento. La rutina del servicio de interrupciones determina la cantidad de casos de desbordamiento.

El valor de 32 bits se crea utilizando el recuento de casos de desbordamiento como los dos bytes más significativos del valor de 32 bits y el valor actual del contador de 16 bits como los dos bytes menos significativos del valor de 32 bits. A continuación, se muestra el pseudocódigo de la rutina del servicio de interrupciones.

```
void TimerOverflowInterruptHandler( void )
{
    /* Just count the number of interrupts. */

    ulOverflowCount++;

    /* Clear the interrupt. */

    ClearTimerInterrupt();
}
```

Este es el código que permite la recopilación de las estadísticas de tiempo de ejecución.

```
/* Set configGENERATE_RUN_TIME_STATS to 1 to enable collection of runtime
statistics. When this is done, both portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() and
portGET_RUN_TIME_COUNTER_VALUE() or portALT_GET_RUN_TIME_COUNTER_VALUE(x) must also be
defined. */

#define configGENERATE_RUN_TIME_STATS 1

/* portCONFIGURE_TIMER_FOR_RUN_TIME_STATS() is defined to call the function that sets up
the hypothetical 16-bit timer. (The function's implementation is not shown.) */

void vSetupTimerForRunTimeStats( void );

#define portCONFIGURE_TIMER_FOR_RUN_TIME_STATS()

vSetupTimerForRunTimeStats()

/* portALT_GET_RUN_TIME_COUNTER_VALUE() is defined to set its parameter to the current
runtime counter/time value. The returned time value is 32-bits long, and is formed by
shifting the count of 16-bit timer overflows into the top two bytes of a 32-bit number,
and then bitwise ORing the result with the current 16-bit counter value. */

#define portALT_GET_RUN_TIME_COUNTER_VALUE( ulCountValue ) \\\

{

    extern volatile unsigned long ulOverflowCount;

    /* Disconnect the clock from the counter so it does not change while its value is being
used. */
```

```
    PauseTimer();

    /* The number of overflows is shifted into the most significant two bytes of the
    returned 32-bit value. */

    ulCountValue = ( ulOverflowCount << 16UL );

    /* The current counter value is used as the least significant two bytes of the returned
    32-bit value. */

    ulCountValue |= ( unsigned long ) ReadTimerCount();

    /* Reconnect the clock to the counter. */

    ResumeTimer(); \\
}
```

La tarea que se muestra aquí imprime las estadísticas de tiempo de ejecución recopiladas cada 5 segundos.

```
/* For clarity, calls to fflush() have been omitted from this codelist. */
static void prvStatsTask( void *pvParameters )
{
    TickType_t xLastExecutionTime;

    /* The buffer used to hold the formatted runtime statistics text needs to be quite
    large. It is therefore declared static to ensure it is not allocated on the task stack.
    This makes this function non-reentrant. */

    static signed char cStringBuffer[ 512 ];

    /* The task will run every 5 seconds. */

    const TickType_t xBlockPeriod = pdMS_TO_TICKS( 5000 );

    /* Initialize xLastExecutionTime to the current time. This is the only time this
    variable needs to be written to explicitly. Afterward, it is updated internally within the
    vTaskDelayUntil() API function. */

    xLastExecutionTime = xTaskGetTickCount();

    /* As per most tasks, this task is implemented in an infinite loop. */

    for( ;; )
    {
        /* Wait until it is time to run this task again. */

        vTaskDelayUntil( &xLastExecutionTime, xBlockPeriod );

        /* Generate a text table from the runtime stats. This must fit into the
        cStringBuffer array. */

        vTaskGetRunTimeStats( cStringBuffer );

        /* Print out column headings for the runtime stats table. */
    }
}
```

```
printf( "\nTask\t\tAbs\t\t\t\t%" );

printf("-----\n" );

/* Print out the runtime stats themselves. The table of data contains multiple
lines, so the vPrintMultipleLines() function is called instead of calling printf()
directly. vPrintMultipleLines() simply calls printf() on each line individually, to ensure
the line buffering works as expected. */

vPrintMultipleLines( cStringBuffer );

}

}
```

Macros de enlace de seguimiento

Se han colocado macros de seguimiento en puntos clave del código fuente de FreeRTOS. De forma predeterminada, las macros están vacías, por lo que no generan ningún código y no tienen sobrecarga de tiempo de ejecución. Al anular las implementaciones vacías predeterminadas, el programador de la aplicación puede:

- Insertar código en FreeRTOS sin modificar los archivos de origen de FreeRTOS.
- Obtener la información detallada de la secuencia de la ejecución por cualquier medio disponible en el hardware de destino. Las macros de seguimiento aparecen en suficientes lugares del código fuente de FreeRTOS para poder utilizarlas para crear un registro detallado y completo de los perfiles y el seguimiento de las actividades del programador.

Macros de enlace de seguimiento disponibles

En la siguiente tabla, se muestran las macros más útiles para un programador de aplicaciones.

Muchas de las descripciones de esta tabla se refieren a una variable llamada `pxCurrentTCB`. `pxCurrentTCB` es una variable privada de FreeRTOS que contiene el controlador de la tarea en estado En ejecución. Está disponible para cualquier macro que se llame desde el archivo de origen FreeRTOS/Source/tasks.c.

Macro	Descripción
<code>traceTASK_INCREMENT_TICK(xTickCount)</code>	Se llama durante la interrupción de ciclos, después de que esta se incremente. El parámetro <code>xTickCount</code> pasa el nuevo valor de recuento de ciclos a la macro.
<code>traceTASK_SWITCHED_OUT()</code>	Se llama antes de que se seleccione una nueva tarea para ejecutarla. En este punto, <code>pxCurrentTCB</code> contiene el controlador de la tarea que está a punto de salir del estado En ejecución.
<code>traceTASK_SWITCHED_IN()</code>	Se llama después de que seleccionar una tarea para ejecutarla. En este punto, <code>pxCurrentTCB</code> contiene el controlador de la tarea que está a punto de entrar en el estado En ejecución.

<code>traceBLOCKING_ON_QUEUE_RECEIVE(pxQueue)</code>	Se llama inmediatamente antes de que la tarea que se está ejecutando actualmente entre en el estado Bloqueado después de un intento de leer una cola vacía o un intento de "tomar" un semáforo o un mutex vacíos. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino o el semáforo a la macro.
<code>traceBLOCKING_ON_QUEUE_SEND(pxQueue)</code>	Se llama inmediatamente antes de que la tarea que se está ejecutando actualmente entre en el estado Bloqueado después de un intento de escribir en una cola que está llena. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino a la macro.
<code>traceQUEUE_SEND(pxQueue)</code>	Se llama desde <code>xQueueSend()</code> , <code>xQueueSendToFront()</code> , <code>xQueueSendToBack()</code> o cualquiera de las funciones para "dar" del semáforo, cuando la operación de envío de la cola o para "dar" del semáforo se realizan correctamente. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino o el semáforo a la macro.
<code>traceQUEUE_SEND_FAILED(pxQueue)</code>	Se llama desde <code>xQueueSend()</code> , <code>xQueueSendToFront()</code> , <code>xQueueSendToBack()</code> o cualquiera de las funciones para "dar" del semáforo, cuando la operación de envío de la cola o para "dar" del semáforo fallan. Una operación de envío de la cola o para "dar" del semáforo fallan si la cola está llena y permanece llena durante el tiempo que dure cualquier tiempo de bloqueo especificado. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino o el semáforo a la macro.
<code>traceQUEUE_RECEIVE(pxQueue)</code>	Se llama desde <code>xQueueReceive()</code> o cualquiera de las funciones para "tomar" del semáforo cuando la operación de recepción de la cola o para "tomar" del semáforo se realizan correctamente. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino o el semáforo a la macro.
<code>traceQUEUE_RECEIVE_FAILED(pxQueue)</code>	Se llama desde <code>xQueueReceive()</code> o cualquiera de las funciones para "tomar" del semáforo cuando la operación de recepción de la cola o del semáforo fallan. Una operación de recepción de la cola o para "dar" del semáforo fallan si la cola o el semáforo están vacíos y permanecen vacíos durante el tiempo que dure cualquier tiempo de bloqueo especificado. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino o el semáforo a la macro.
<code>traceQUEUE_SEND_FROM_ISR(pxQueue)</code>	Se llama desde <code>xQueueSendFromISR()</code> cuando la operación de envío se realiza correctamente. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino a la macro.

<code>traceQUEUE_SEND_FROM_ISR_FAILED(pxQueue)</code>	Se llama desde <code>xQueueSendFromISR()</code> cuando la operación de envío falla. Una operación de envío falla si la cola ya está llena. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino a la macro.
<code>traceQUEUE_RECEIVE_FROM_ISR(pxQueue)</code>	Se llama desde <code>xQueueReceiveFromISR()</code> cuando la operación de recepción se realiza correctamente. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino a la macro.
<code>traceQUEUE_RECEIVE_FROM_ISR_FAILED(pxQueue)</code>	Se llama desde <code>xQueueReceiveFromISR()</code> cuando la operación de recepción falla debido a que la cola ya está vacía. El parámetro <code>pxQueue</code> pasa el controlador de la cola de destino a la macro.
<code>traceTASK_DELAY_UNTIL()</code>	Se llama desde <code>vTaskDelayUntil()</code> inmediatamente antes de que la tarea que realiza la llamada entre en el estado Bloqueado.
<code>traceTASK_DELAY()</code>	Se llama desde <code>vTaskDelay()</code> inmediatamente antes de que la tarea que realiza la llamada entre en el estado Bloqueado.

Definición de macros de enlace de seguimiento

Cada macro de seguimiento tiene una definición vacía de manera predeterminada. Puede anular la definición predeterminada proporcionando una nueva definición de macro en `FreeRTOSConfig.h`. Si las definiciones de macros de seguimiento se hacen muy largas o complejas, se pueden implementar en un nuevo archivo de encabezado que se incluye desde `FreeRTOSConfig.h`.

Según la práctica recomendada en ingeniería de software, FreeRTOS mantiene una política estricta de ocultación de datos. Las macros de seguimiento permiten añadir el código de usuario a los archivos fuente de FreeRTOS para que los tipos de datos visibles para las macros de seguimiento sean diferentes de los visibles para el código de la aplicación:

- En el archivo de origen de FreeRTOS/Source/tasks.c, un controlador de tarea es un puntero a la estructura de datos que describe una tarea. Este es el bloque de control de tareas (TCB) de la tarea. Fuera del archivo de origen FreeRTOS/Source/tasks.c, un controlador de tarea es un puntero a nulo.
- En el archivo de origen de FreeRTOS/Source/queue.c, un controlador de cola es un puntero a la estructura de datos que describe una cola. Fuera del archivo de origen FreeRTOS/Source/queue.c, un controlador de cola es un puntero a nulo.

Tenga mucho cuidado si una macro de seguimiento accede directamente a una estructura de datos de FreeRTOS que normalmente es privada. Las estructuras de datos privadas pueden cambiar de una versión de FreeRTOS a otra.

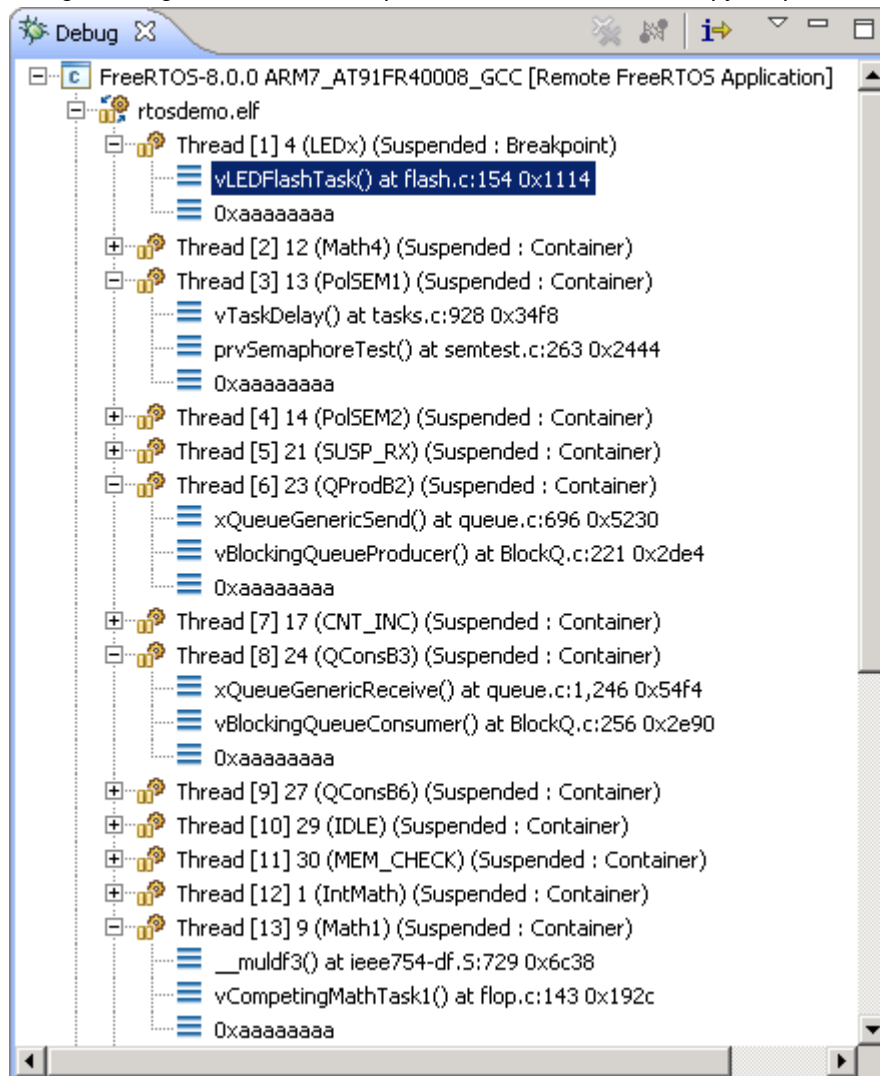
Complementos de depurador compatibles con FreeRTOS

Hay disponibles complementos que ofrecen alguna compatibilidad con FreeRTOS para los siguientes IDE. Esta lista no es exhaustiva.

- Eclipse (StateViewer)

- Eclipse (ThreadSpy)
- IAR
- ARM DS-5
- Atollic TrueStudio
- Microchip MPLAB
- iSYSTEM WinIDEA

La siguiente figura muestra el complemento FreeRTOS ThreadSpy Eclipse de Code Confidence Ltd.



Solución de problemas

Introducción y ámbito del capítulo

En esta sección se describen los problemas más frecuentes con que se encuentran los usuarios que no están familiarizados con FreeRTOS, en concreto:

- Asignaciones de prioridad de interrupción incorrectas;
- Desbordamiento de pila;
- Uso incorrecto de `printf()`.

Con `configASSERT()`, se mejora la productividad mediante la captura y la identificación inmediatas de muchos de los orígenes de errores más comunes. Le recomendamos encarecidamente que defina `configASSERT()` mientras desarrolla o depura una aplicación FreeRTOS. Para obtener más información acerca de `configASSERT()`, consulte [Developer Support \(p. 231\)](#).

Prioridades de interrupción

Nota: Esta es la principal causa de solicitud de soporte. En la mayoría de los puertos, definir `configASSERT()` permitirá capturar el error inmediatamente.

Si el puerto FreeRTOS que se está usando admite el anidamiento de interrupciones y la rutina de servicio de una interrupción usa la API de FreeRTOS, tiene que establecer la prioridad de interrupción en el valor de `configMAX_SYSCALL_INTERRUPT_PRIORITY` o por debajo, tal y como se describe en [Administración de interrupciones \(p. 125\)](#). Si no configura la prioridad correctamente, sus secciones críticas no serán efectivas, lo que, a su vez, dará lugar a errores intermitentes.

Sea precavido si ejecuta FreeRTOS en un procesador donde:

- Las prioridades de interrupción tengan de manera predeterminada la prioridad más alta posible, lo que ocurre con algunos procesadores ARM Cortex. En dichos procesadores la prioridad de una interrupción que utilice la API de FreeRTOS no se puede dejar sin inicializar.
- Los números de alta prioridad numérica representen lógicamente prioridades de baja interrupción, lo que a primera vista puede parecer contradictorio. En este caso, de nuevo, se trata de los procesadores ARM Cortex y posiblemente también de otros procesadores.
- Por ejemplo, en este tipo de procesador una interrupción que se está ejecutando con una prioridad de 5 puede interrumpirse con una prioridad de 4. Por lo tanto, si se establece `configMAX_SYSCALL_INTERRUPT_PRIORITY` en 5, solo se puede asignar a las interrupciones que usen la API de FreeRTOS una prioridad numérica que sea igual o superior a 5. En dicho caso, las prioridades de interrupción de 5 o 6 son válidas, pero una prioridad de interrupción que sea de 3 es definitivamente no válida.
- Haya diferentes implementaciones de biblioteca que esperen que la prioridad de una interrupción se especifique de otra manera. Esto es especialmente pertinente en el caso de las bibliotecas dirigidas a procesadores ARM Cortex, donde las prioridades de interrupción se desplazan con bits antes de escribirlas en los registros de hardware. Algunas bibliotecas realizan ellas mismas los desplazamientos de bits, mientras que otras esperan a que se ejecute el desplazamiento de bits antes de que la prioridad se transfiera a la función de biblioteca.

- Diferentes implementaciones de la misma arquitectura implementen un número diferente de bits de prioridad de interrupción. Por ejemplo, un procesador Cortex-M de un fabricante podría implementar 3 bits de prioridad, mientras que un procesador Cortex-M de otros fabricantes podría implementar 4 bits de prioridad.
- Los bits que definen la prioridad de una interrupción se pueden dividir entre bits que definen una prioridad de preferencia y bits que definen una subprioridad. Asegúrese de que todos los bits se asignan a especificar una prioridad de reemplazo, por lo que no se utilizarán subprioridades.

En algunos puertos FreeRTOS, `configMAX_SYSCALL_INTERRUPT_PRIORITY` tiene el nombre alternativo `configMAX_API_CALL_INTERRUPT_PRIORITY`.

Desbordamiento de pila

El desbordamiento de pila es la segunda causa más frecuente de solicitudes de soporte. FreeRTOS dispone de varias características útiles para capturar y depurar problemas relacionados con las pilas. (Estas características no están disponibles en el puerto Windows de FreeRTOS).

La función de API `uxTaskGetStackHighWaterMark()`

Cada tarea mantiene su propia pila, cuyo tamaño total se especifica cuando se crea la tarea. `uxTaskGetStackHighWaterMark()` se utiliza para consultar lo cerca que ha estado una tarea de desbordar el espacio de pila que tiene asignado. Este valor se denomina nivel máximo de la pila.

Aquí se muestra el prototipo de la función de API `uxTaskGetStackHighWaterMark()`.

```
UBaseType_t uxTaskGetStackHighWaterMark( TaskHandle_t xTask );
```

En la siguiente tabla se muestran los parámetros `uxTaskGetStackHighWaterMark()` y el valor de retorno.

Nombre de parámetro/Valor devuelto	Descripción
<code>xTask</code>	El controlador de la tarea cuyo nivel máximo de pila se está consultando (la tarea del asunto). Para obtener información acerca de cómo obtener controladores para las tareas, consulte el parámetro <code>pxCreatedTask</code> de la función de API <code>xTaskCreate()</code> . Una tarea puede consultar su propio nivel máximo de pila pasando <code>NULL</code> en lugar de un controlador de tarea válido.
Valor devuelto	La cantidad de pila que la tarea utiliza aumenta y se reduce a medida que se procesan las ejecuciones e interrupciones de la tarea. <code>uxTaskGetStackHighWaterMark()</code> devuelve la cantidad mínima de espacio de pila restante disponible desde que la tarea comenzó a ejecutarse. Esta es la cantidad de pila que permanece sin usar cuando el uso de la pila está en su valor máximo (o más profundo). Cuanto más cerca esté el nivel máximo a cero, más cerca habrá estado la tarea de desbordar su pila.

Información general de la comprobación de pila en tiempo de ejecución

FreeRTOS incluye dos mecanismos de comprobación de pila en tiempo de ejecución opcional. Estos mecanismos se controlan mediante la constante de configuración de tiempo de compilación `configCHECK_FOR_STACK_OVERFLOW` en `FreeRTOSConfig.h`. Ambos métodos aumentan el tiempo que se tarda en realizar un cambio de contexto.

El enlace de desbordamiento de pila (o devolución de llamada de desbordamiento de pila) es una función a la que llama el kernel cuando detecta un desbordamiento de pila. Para utilizar una función de enlace de desbordamiento de pila:

1. Establezca `configCHECK_FOR_STACK_OVERFLOW` en 1 o 2, en `FreeRTOSConfig.h`.
2. Proporcione la implementación de la función de enlace, utilizando el nombre y el prototipo de función que se muestran aquí.

```
void vApplicationStackOverflowHook( TaskHandle_t *pxTask, signed char*pcTaskName );
```

El enlace de desbordamiento de pila se proporciona para facilitar la captura y la depuración de los errores de pila, pero no hay una verdadera manera de recuperarse de un desbordamiento de pila cuando este se produce. Los parámetros de la función pasan a la función de enlace el controlador y el nombre de la tarea que ha desbordado su pila.

El enlace de desbordamiento de pila recibe una llamada desde el contexto de una interrupción.

Algunos microcontroladores generan una excepción de error cuando detectan que se ha producido un acceso a la memoria incorrecto. Es posible que se desencadene un error antes de que el kernel tenga la posibilidad de llamar a una función de enlace de desbordamiento de pila.

Método 1 para comprobar la pila en tiempo de ejecución

El método 1 es rápido de ejecutar, pero puede perder desbordamientos de pila que se producen entre cambios de contexto. Este método se usa cuando `configCHECK_FOR_STACK_OVERFLOW` está establecido en 1.

El contexto completo de ejecución de una tarea se guarda en su pila cada vez que se intercambia. Es probable que sea cuando el uso de la pila llegue a su máximo. Cuando `configCHECK_FOR_STACK_OVERFLOW` se establece en 1, el kernel comprueba que el puntero de pila permanezca dentro del espacio de pila válido después de que se haya guardado el contexto. Se llama al enlace de desbordamiento de pila si el puntero de pila está fuera de su rango válido.

Método 2 para comprobar la pila en tiempo de ejecución

El método 2 realiza comprobaciones adicionales. Este método se usa cuando `configCHECK_FOR_STACK_OVERFLOW` está establecido en 2.

Cuando se crea una tarea, su pila se llena con un patrón conocido. El método 2 comprueba los últimos 20 bytes válidos del espacio de pila de la tarea para verificar que no se haya sobrescrito este patrón. La función de enlace de desbordamiento de pila se llama si el valor de alguno de los 20 bytes ha cambiado en relación con los valores previstos.

El método 2 no se ejecuta tan rápido como el método 1, aunque sigue siendo relativamente rápido ya que solo se prueban 20 bytes. Es muy probable que capture todos los desbordamientos de pila.

Uso incorrecto de printf() y sprintf().

El uso incorrecto de printf() suele ser una fuente de errores. Los desarrolladores de aplicaciones que no son conscientes de este problema, en ocasiones añaden más llamadas a printf() para ayudar a la depuración y, al hacerlo, empeoran el problema.

Muchos proveedores de compiladores cruzados proveen una implementación de printf() adecuada para usarla en pequeños sistemas integrados. Incluso cuando ese es el caso, la implementación podría no ser adecuada para subprocesos, probablemente no sea adecuada para usarla dentro de una rutina de servicio de interrupción y en función de adónde se dirija la salida, podría tardar un tiempo relativamente largo en ejecutarse.

Si no dispone de una implementación de printf() diseñada específicamente para sistemas integrados pequeños, asegúrese de utilizar una implementación de printf() genérica porque:

- Solo con incluir una llamada a printf() o sprintf() puede incrementar masivamente el tamaño del archivo ejecutable de la aplicación.
- printf() y sprintf() podrían llamar a malloc(), lo que podría ser una operación no válida si se usa un método de asignación de memoria que no sea heap_3. Para obtener más información, consulte [Métodos de asignación de memoria de ejemplo \(p. 15\)](#).
- printf() y sprintf() podrían necesitar una pila muchas veces más grande de lo que se necesitase.

Printf-stdarg.c

Muchos de los proyectos de demostración de FreeRTOS utilizan un archivo llamado printf-stdarg.c que proporciona una implementación mínima y eficiente para pilas de sprintf() que se puede utilizar en lugar de la versión de biblioteca estándar. En la mayoría de los casos, esto permitirá asignar una pila mucho más pequeña a cada tarea que llame a sprintf() y sus funciones relacionadas.

printf-stdarg.c también proporciona un mecanismo para dirigir la salida de printf() a un puerto, carácter a carácter. Aunque es un proceso lento, esto permite reducir aún más el uso de la pila.

No todas las copias de printf-stdarg.c implementan snprintf(). Las copias que no implementan snprintf() simplemente no tienen en cuenta el parámetro de tamaño del búfer que asignan directamente a sprintf().

Printf-stdarg.c es código abierto, pero es propiedad de un tercero y, por lo tanto, tiene una licencia independiente de FreeRTOS. Los términos de licencia se encuentran en la parte superior del archivo de origen.

Otros errores habituales

En esta sección se incluyen otros errores habituales, sus posibles causas y sus soluciones.

Síntoma: si se añade una tarea sencilla a una demostración esta se bloquea.

Para crear una tarea es preciso obtener memoria del montón. Muchos de los proyectos de aplicaciones de demostración dimensionan el montón para que sea exactamente lo suficientemente grande para crear

las tareas de demostración, por lo que después de crear las tareas, el montón que quede será insuficiente para añadir más tareas, colas, grupos de eventos o semáforos.

La tarea de inactividad y posiblemente la tarea de demonio de RTOS se crean automáticamente cuando se llama a `vTaskStartScheduler()`. `vTaskStartScheduler()` solo volverá si no queda suficiente memoria en montón para crear estas tareas. Si se incluye un bucle nulo [`for(;;);`] después de la llamada a `vTaskStartScheduler()` será más fácil depurar este error.

Para poder añadir más tareas, aumente el tamaño del montón o elimine algunas de las tareas de demostración existentes. Para obtener más información, consulte [Métodos de asignación de memoria de ejemplo](#) (p. 15).

Síntoma: si se usa una función de la API en una interrupción la aplicación se bloquea.

No utilice funciones de API en rutinas en servicio de interrupción a menos que el nombre de la función de API termine por "...FromISR()". En concreto, no cree una sección crítica en una interrupción a menos que utilice macros a prueba de seguridad. Para obtener más información, consulte [Administración de interrupciones](#) (p. 125).

En los puertos FreeRTOS que admiten el anidado de interrupciones, no utilice funciones de API en una interrupción a la que se ha asignado una prioridad de interrupción superior a `configMAX_SYSCALL_INTERRUPT_PRIORITY`. Para obtener más información, consulte [Anidamiento de interrupciones](#) (p. 158).

Síntoma: en ocasiones la aplicación se bloquea en una rutina de servicio de interrupción.

Primero hay que comprobar que la interrupción no provoque un desbordamiento de pila. Algunos puertos comprueban si hay un desbordamiento de pila solo en tareas, no en interrupciones.

La forma en que se definen y se usan las interrupciones varía según los puertos y los compiladores. Por lo tanto, en segundo lugar hay que comprobar que la sintaxis, las macros y las convenciones de llamada que se usan en la rutina del servicio de interrupción sean exactamente como las que se describen en la página de documentación proporcionada para el puerto que se usa y sean exactamente tal y como se demuestra en la aplicación de demostración proporcionada con el puerto.

Si la aplicación se ejecuta en un procesador que utiliza números de baja prioridad numéricamente para representar altas prioridades, asegúrese de que la prioridad asignada a cada interrupción lo tenga en cuenta, ya que puede parecer contrario a la lógica. Si la aplicación se ejecuta en un procesador que aplica de forma predeterminada a la prioridad de cada interrupción la máxima prioridad posible, asegúrese de que no se deje la prioridad de cada interrupción en su valor predeterminado. Para obtener más información, consulte [Anidamiento de interrupciones](#) (p. 158).

Síntoma: el programador se bloquea al intentar comenzar la primera tarea.

Asegúrese de que los controladores de interrupción de FreeRTOS se hayan instalado. Para ver un ejemplo, consulte la aplicación de demostración que se proporciona para el puerto.

Algunos procesadores tienen que estar en modo privilegiado para poder iniciar el programador. La forma más sencilla de hacerlo es poner el procesador en un modo privilegiado en el código de inicio de C, antes de que se llame a `main()`.

Síntoma: las interrupciones se quedan desactivadas de forma inesperada o bien hay secciones críticas que no se anidan correctamente

Si se llama a una función de API de FreeRTOS antes de que se inicie el programador, las interrupciones se dejarán deshabilitadas a propósito. No se volverán a habilitar hasta que se comience a ejecutar la primera tarea. De esta forma se protege al sistema contra bloqueos causados por interrupciones que intentan utilizar funciones de API de FreeRTOS durante la inicialización del sistema, antes de que se haya iniciado el programador y mientras este pueda estar en un estado incoherente.

Utilice llamadas a `taskENTER_CRITICAL()` y `taskEXIT_CRITICAL()` para modificar los bits de habilitación de interrupción del microcontrolador o las marcas de prioridad. No utilice ningún otro método. Estas macros llevan un recuento de la profundidad de anidación de las llamadas para asegurarse de que las interrupciones se vuelvan a habilitar solo cuando el anidamiento de llamadas se haya deshecho totalmente y esté en cero. Tenga en cuenta que algunas funciones de biblioteca pueden habilitar y deshabilitar interrupciones.

Síntoma: la aplicación se bloquea incluso antes de que se inicie el programador.

No permita que una rutina de servicio de interrupción que podría provocar potencialmente un cambio de contexto se ejecute antes de que se inicie el programador. Lo mismo se aplica a todas las rutinas de servicio de interrupción que intentan enviar o recibir desde un objeto de FreeRTOS, como una cola o un semáforo. Un cambio de contexto solo se puede producir después de que el programador se haya iniciado.

Muchas funciones de API solo se pueden llamar después de que se haya iniciado el programador. Es mejor restringir el uso de la API a la creación de objetos, como tareas, colas y semáforos, en lugar de usar estos objetos hasta después llamar a `vTaskStartScheduler()`.

Síntoma: si se llama a funciones de API mientras el programador está suspendido o desde dentro de una sección crítica, la aplicación se bloquea.

El programador se suspende llamando a `vTaskSuspendAll()` y se reanuda (anula la suspensión) llamando a `xTaskResumeAll()`. Se entra en una sección crítica llamando a `taskENTER_CRITICAL()` y se sale llamando a `taskEXIT_CRITICAL()`.

No llame a funciones de API desde dentro de una sección crítica o mientras el programador esté suspendido.