FreeRTOS

Brief

Overview

Task creation

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

Listing 13. The xTaskCreate() API function prototype

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

Listing 14. Implementation of the first task used in Example 1

```
int main( void )
{
    /* Create one of the two tasks.  Note that a real application should check
    the return value of the xTaskCreate() call to ensure the task was created
    successfully. */
    xTaskCreate(    vTask1,  /* Pointer to the function that implements the task. */
                    "Task 1",/* Text name for the task.  This is to facilitate
                             debugging only. */
                    1000,    /* Stack depth - small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* This example does not use the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* This example does not use the task handle. */

    /* Create the other task in exactly the same way and at the same priority. */
    xTaskCreate( vTask2, "Task 2", 1000, NULL, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

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

Listing 16. Starting the Example 1 tasks

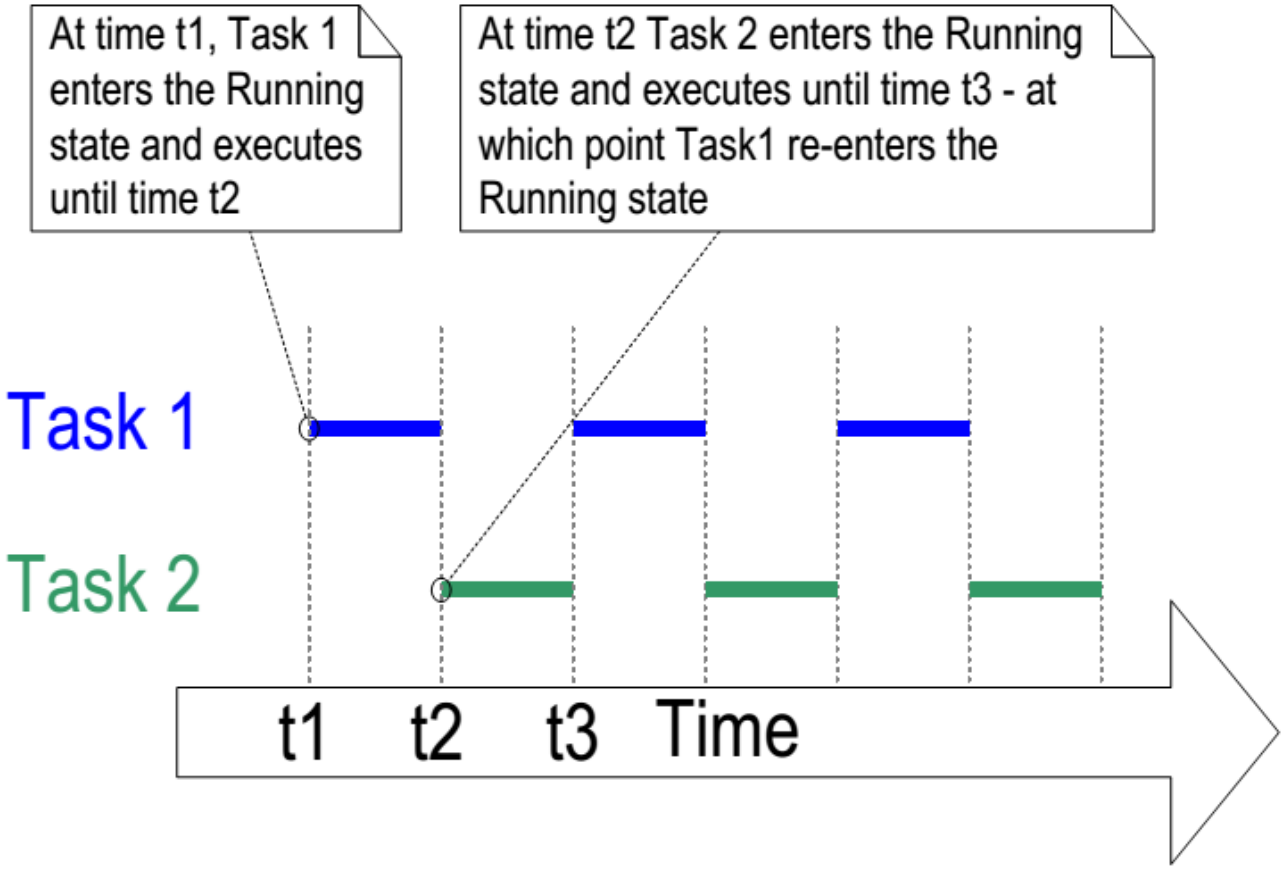


Figure 11. The actual execution pattern of the two Example 1 tasks

Create a task within a task

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

Listing 17. Creating a task from within another task after the scheduler has started

Using the task parameter

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
volatile uint32_t ul; /* volatile to ensure ul is not optimized away. */

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

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

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

Listing 18. The single task function used to create two tasks in Example 2

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

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

    /* Create the other task in exactly the same way.  Note this time that multiple
    tasks are being created from the SAME task implementation (vTaskFunction).  Only
    the value passed in the parameter is different.  Two instances of the same
    task are being created. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 1, NULL );

    /* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();

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

Task life cycle

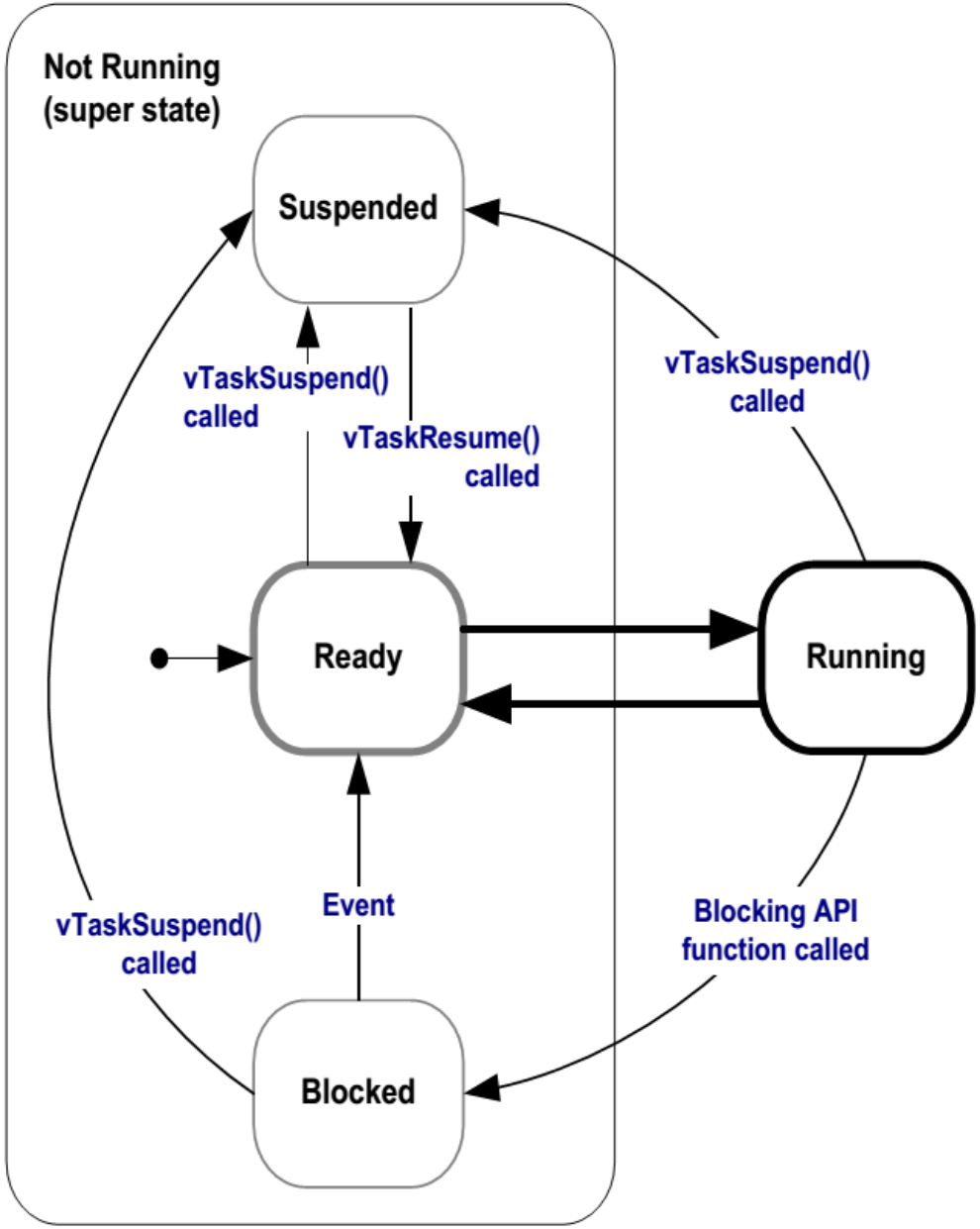


Figure 15. Full task state machine

vTaskDelay

```
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 );
    }
}
```

Listing 23. The source code for the example task after the null loop delay has been replaced by a call to vTaskDelay()

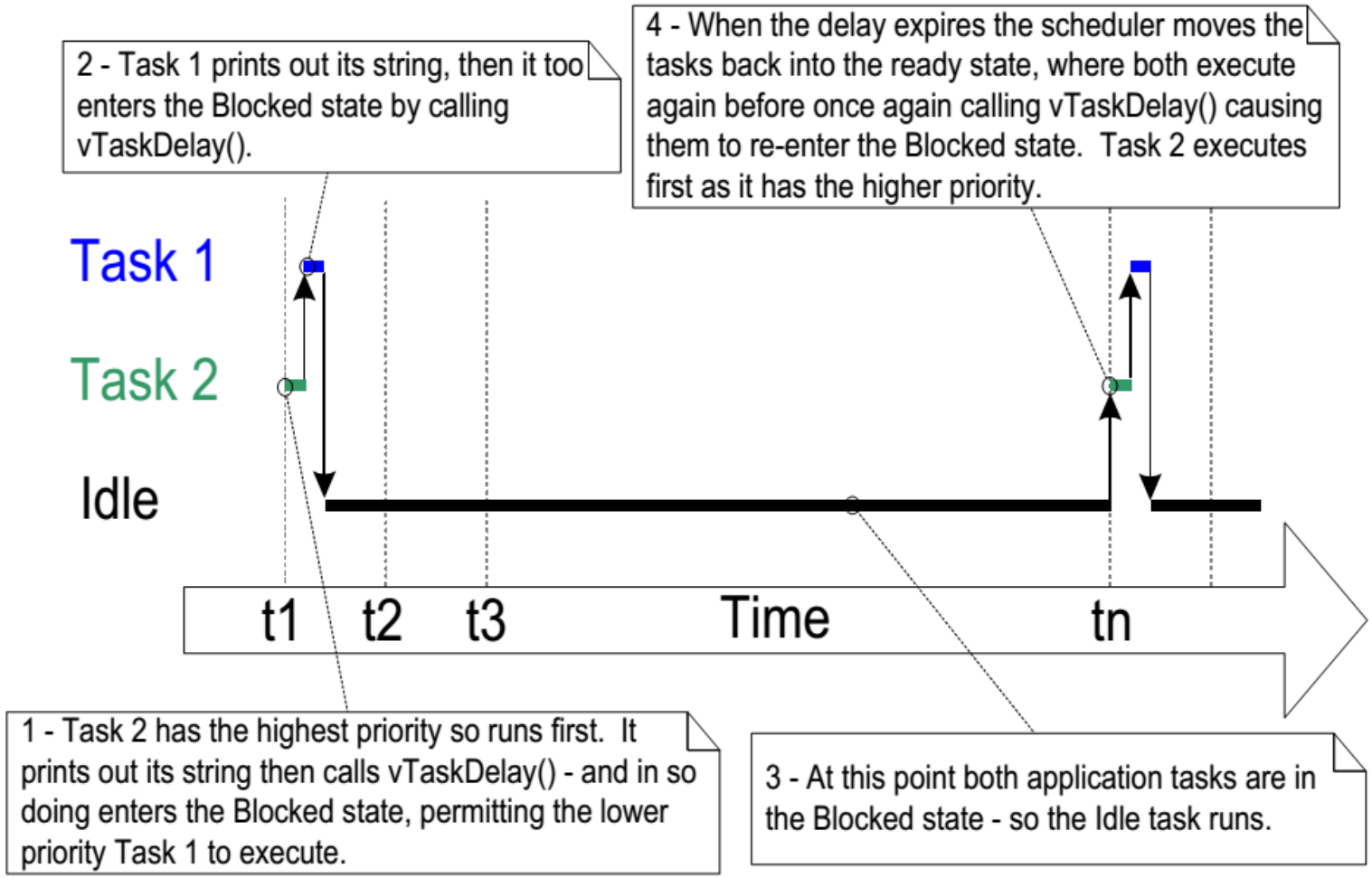


Figure 17. The execution sequence when the tasks use vTaskDelay() in place of the NULL loop

Periodic tasks

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

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

    /* The xLastWakeTime variable needs to be initialized with the current tick
    count.  Note that 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 ) );
    }
}
```

Listing 25. The implementation of the example task using vTaskDelayUntil()

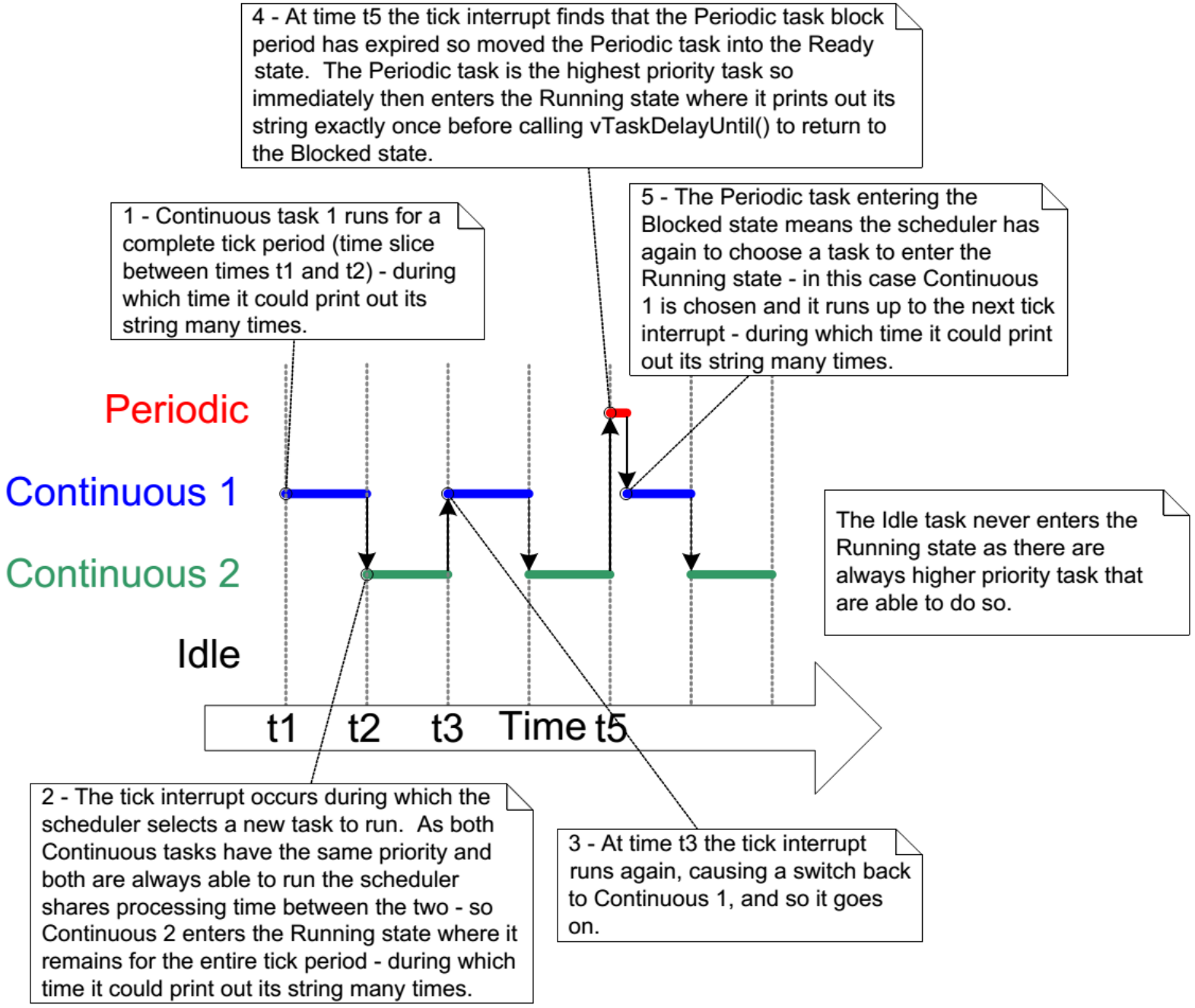


Figure 20. The execution pattern of Example 6

Changing priority

```
void vTask1( void *pvParameters )
{
UBaseType_t uxPriority;

    /* This task will always run before Task 2 as it is created with the higher
    priority.  Neither Task 1 nor Task 2 ever 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 (as then 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().  Listing 35 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. */
    }
}
```

Task deletions

```
void vTask2( void *pvParameters )
{
    /* Task 2 does nothing but delete itself.  To do this it could call vTaskDelete()
    using NULL as the parameter, but instead, and purely 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 );
}
```

Listing 39. The implementation of Task 2 for Example 9

Table 15. An explanation of the terms used to describe the scheduling policy

Term	Definition
Fixed Priority	Scheduling algorithms described as 'Fixed Priority' do not change the priority assigned to the tasks being scheduled, but also do not prevent the tasks themselves from changing their own priority, or that of other tasks.
Pre-emptive	Pre-emptive scheduling algorithms will immediately 'pre-empt' the Running state task if a task that has a priority higher than the Running state task enters the Ready state. Being pre-empted means being involuntarily (without explicitly yielding or blocking) moved out of the Running state and into the Ready state to allow a different task to enter the Running state.
Time Slicing	Time slicing is used to share processing time between tasks of equal priority, even when the tasks do not explicitly yield or enter the Blocked state. Scheduling algorithms described as using 'Time Slicing' will select a new task to enter the Running state at the end of each time slice if there are other Ready state tasks that have the same priority as the Running task. A time slice is equal to the time between two RTOS tick interrupts.

Preemptive scheduling and round-Robin

Table 14. The FreeRTOSConfig.h settings that configure the kernel to use Prioritized Pre-emptive Scheduling with Time Slicing

Constant	Value
configUSE_PREEMPTION	1
configUSE_TIME_SLICING	1

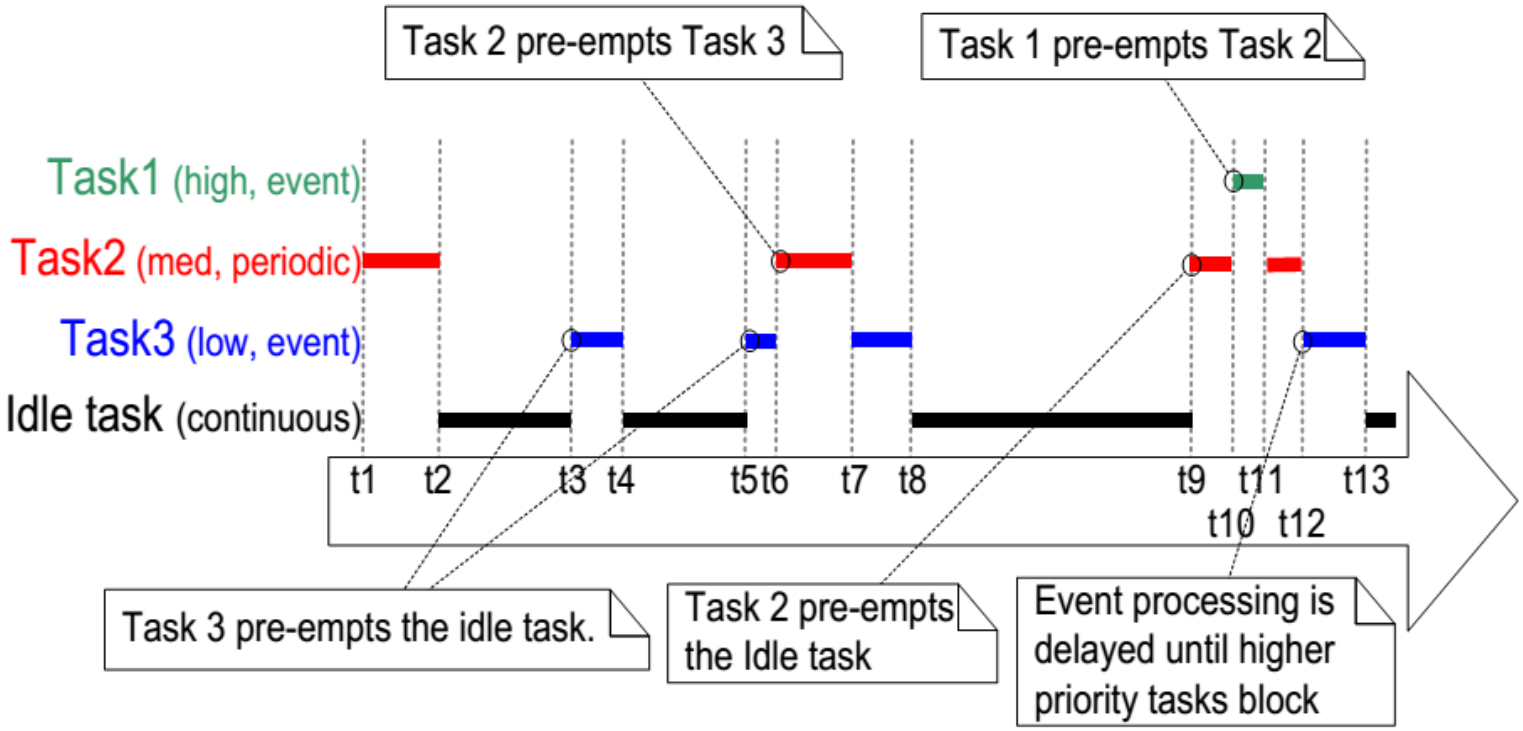
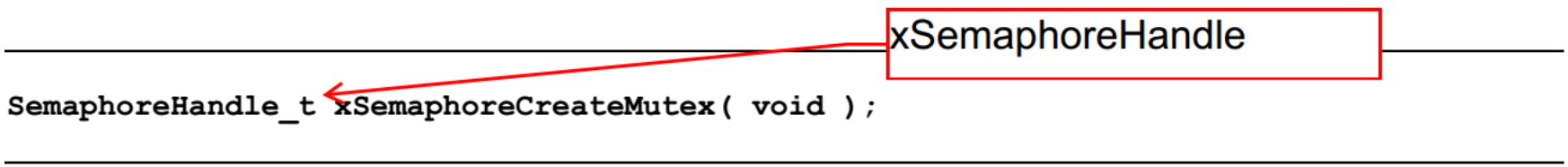


Figure 26. Execution pattern highlighting task prioritization and pre-emption in a hypothetical application in which each task has been assigned a unique priority

Mutexes



Listing 120. The xSemaphoreCreateMutex() API function prototype

- FreeRTOS uses theirqueues to implement semaphores
- FreeRTOS uses binary semaphores to implement mutexes

```
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 straight 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).  As noted earlier in this book, indefinite
    time outs are not recommended for production code. */
    xSemaphoreTake( xMutex, portMAX_DELAY );
    {
        /* The following line will only execute once the mutex has been successfully
        obtained.  Standard out can be accessed freely now as only one task can have
        the mutex at any one time. */
        printf( "%s", pcString );
        fflush( stdout );

        /* The mutex MUST be given back! */
    }
    xSemaphoreGive( xMutex );
}
```

Listing 121. The implementation of prvNewPrintString()

```
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 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 pre-emption 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 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( ;; );
}
```

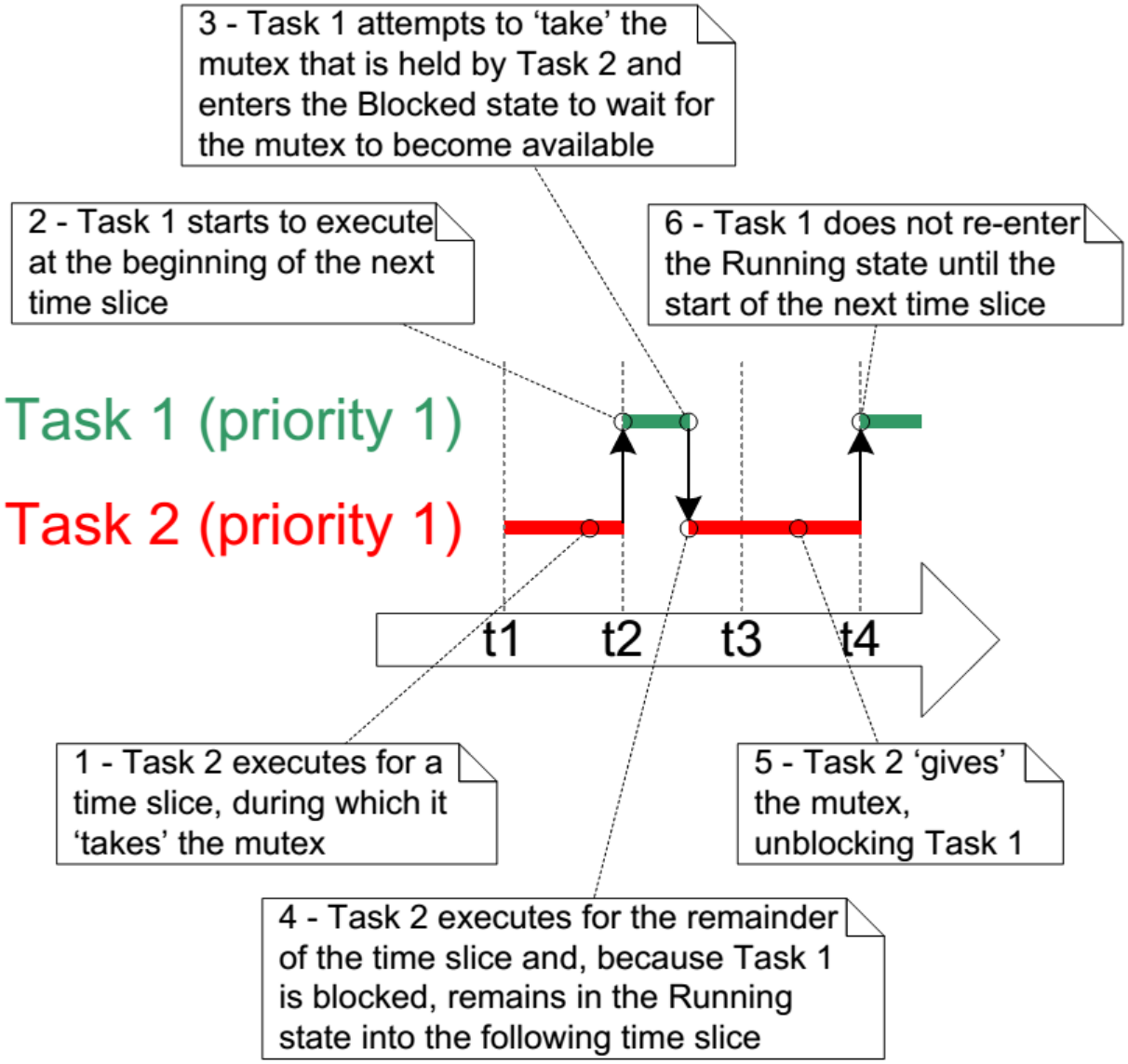


Figure 68 A possible sequence of execution when tasks that have the same priority use the same mutex

Priority inversion

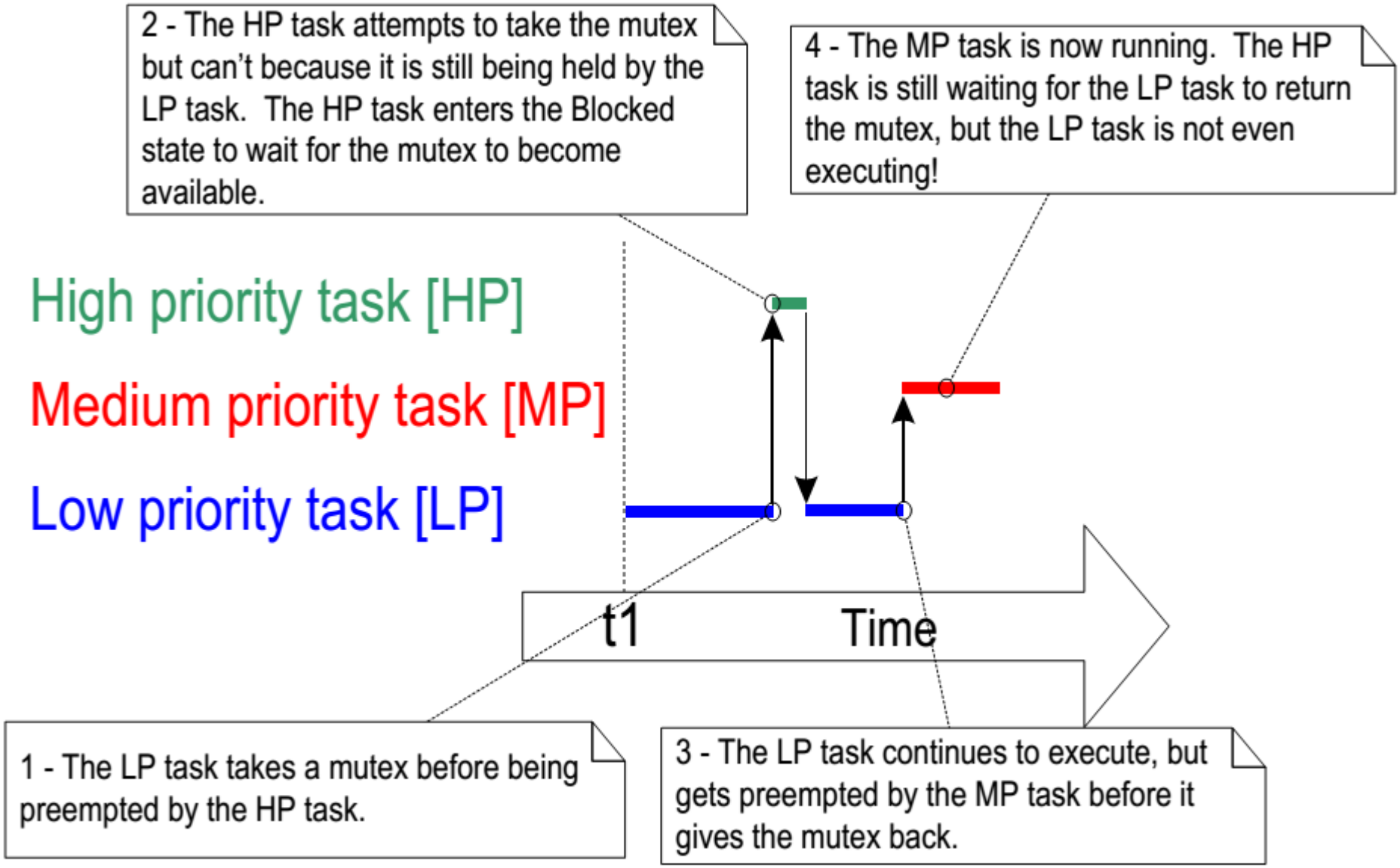


Figure 66. A worst case priority inversion scenario

Priority inheritance solves the problem

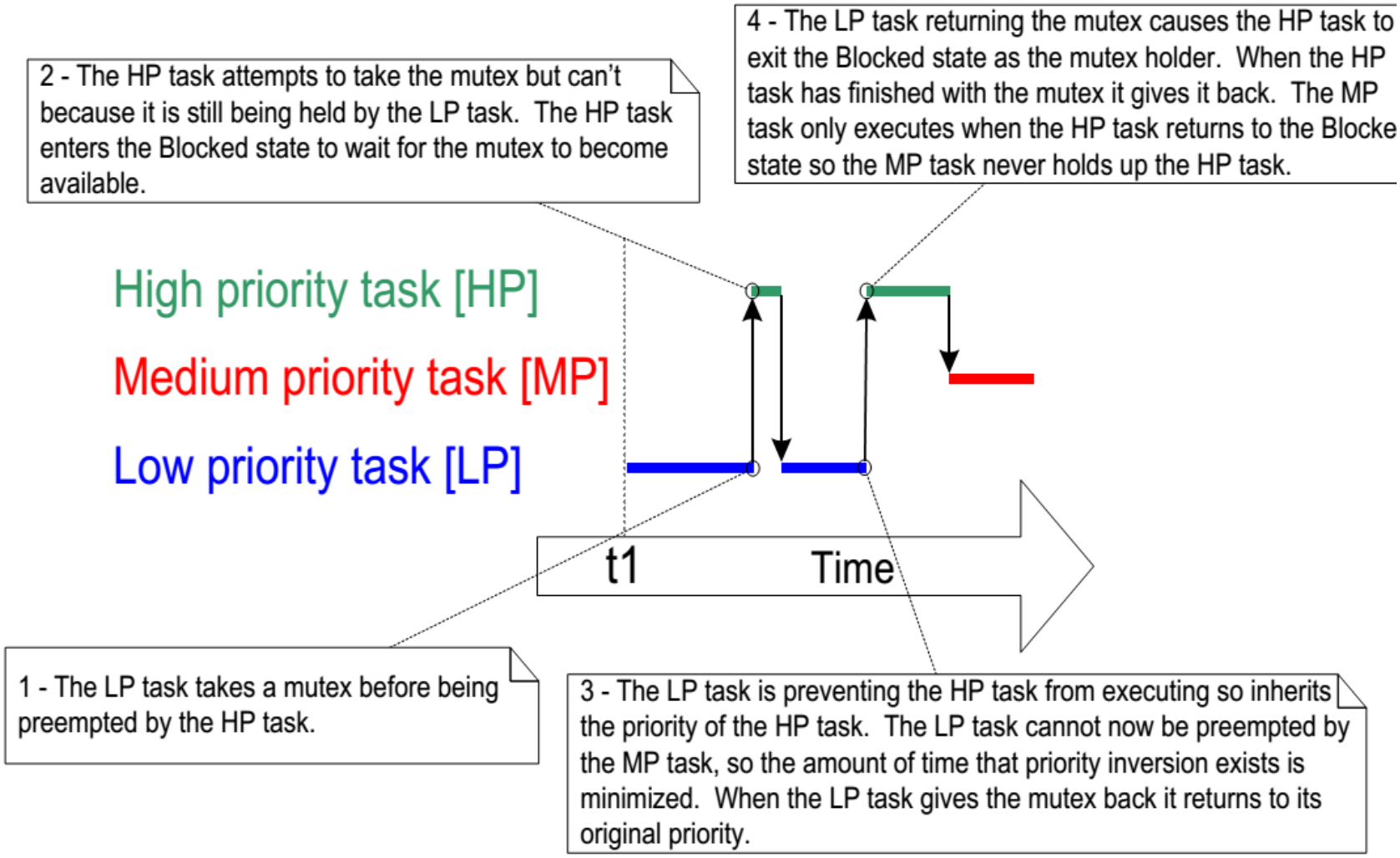


Figure 67. Priority inheritance minimizing the effect of priority inversion

Recursive mutex

```
/* 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 );
```

- Continues on next slide

```
/* 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, as 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 sub-
       function called by this task, as 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 fairness

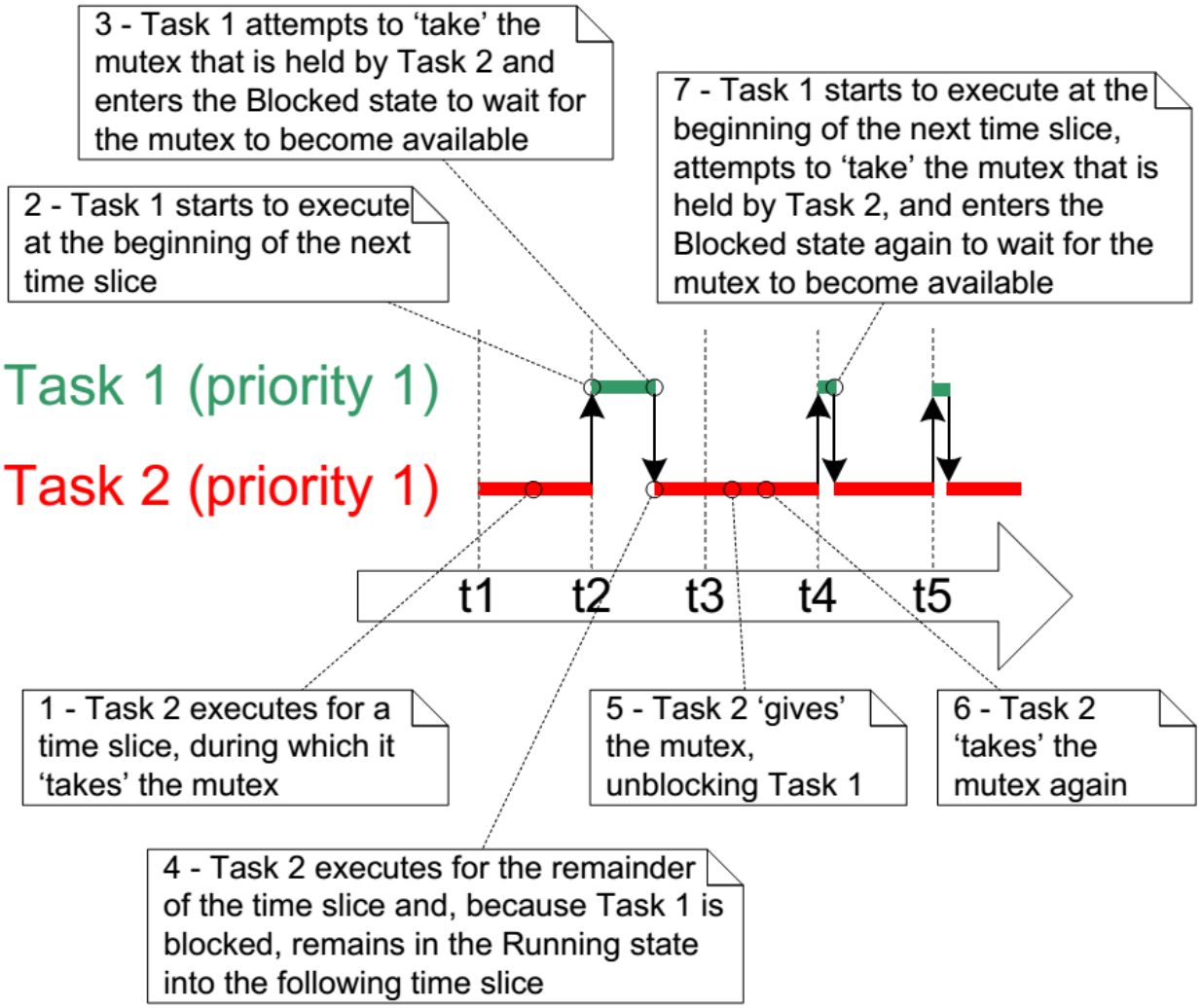


Figure 69 A sequence of execution that could occur if two instances of the task shown by Listing 125 are created at the same priority

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

Listing 126. Ensuring tasks that use a mutex in a loop receive a more equal amount

Queues in freeRTOS

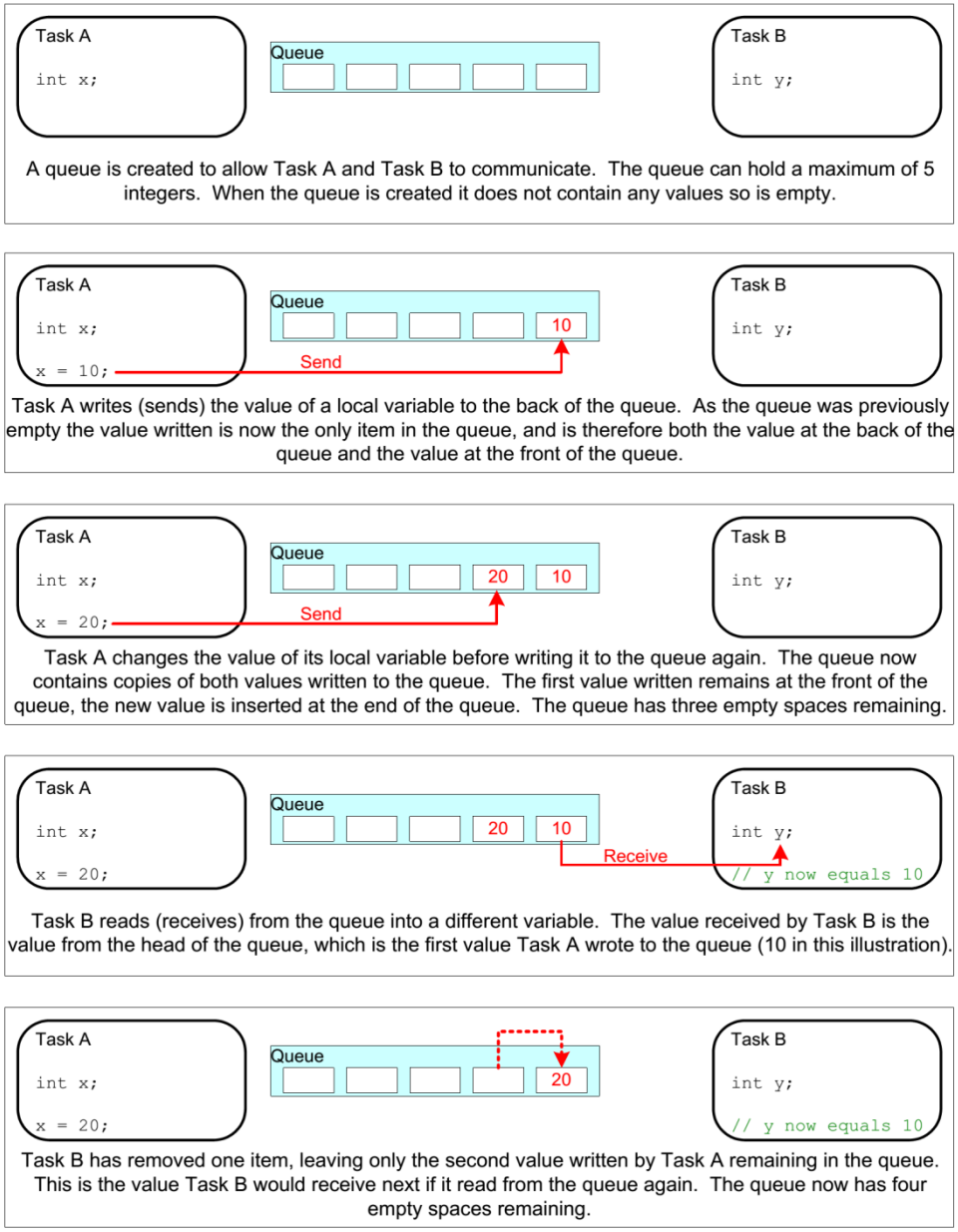


Figure 31. An example sequence of writes to, and reads from a queue

- Notice that copies are sent

3.3 xQueueCreate()

```
#include "FreeRTOS.h"
#include "queue.h"

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

Listing 108 xQueueCreate() function prototype

```
/* Define the data type that will be queued. */
typedef struct A_Message
{
    char ucMessageID;
    char ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )
{
QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variab
    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if( xQueue == NULL )
    {
        /* The queue could not be created. */
    }

    /* Rest of code goes here. */
}
```

Listing 109 Example use of xQueueCreate()

3.22 xQueueSend(), xQueueSendToFront(), xQueueSendToBack()

```
#include "FreeRTOS.h"
#include "queue.h"

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

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

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

Listing 140 xQueueSend(), xQueueSendToFront() and xQueueSendToBack() function prototypes

3.16 xQueueReceive()

```
#include "FreeRTOS.h"
#include "queue.h"

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

Listing 130 xQueueReceive() function prototype

```
/* Define the data type that will be queued. */
typedef struct A_Message
{
    char ucMessageID;
    char ucData[ 20 ];
} AMessage;

/* Define the queue parameters. */
#define QUEUE_LENGTH 5
#define QUEUE_ITEM_SIZE sizeof( AMessage )

int main( void )
{
QueueHandle_t xQueue;

    /* Create the queue, storing the returned handle in the xQueue variable. */
    xQueue = xQueueCreate( QUEUE_LENGTH, QUEUE_ITEM_SIZE );
    if( xQueue == NULL )
    {
        /* The queue could not be created - do something. */
    }

    /* Create a task, passing in the queue handle as the task parameter. */
    xTaskCreate( vAnotherTask,
                 "Task",
                 STACK_SIZE,
                 ( void * ) xQueue, /* The queue handle is used as the task parame
                 TASK_PRIORITY,
                 NULL );

    /* Start the task executing. */
```

```
void vAnotherTask( void *pvParameters )
{
QueueHandle_t xQueue;
AMessage xMessage;

    /* The queue handle is passed into this task as the task parameter.  Cast the
    void * parameter back to a queue handle. */
    xQueue = ( QueueHandle_t ) pvParameters;

    for( ;; )
    {
        /* Wait for the maximum period for data to become available on the queue.
        The period will be indefinite if INCLUDE_vTaskSuspend is set to 1 in
        FreeRTOSConfig.h. */
        if( xQueueReceive( xQueue, &xMessage, portMAX_DELAY ) != pdPASS )
        {
            /* Nothing was received from the queue - even after blocking to wait
            for data to arrive. */
        }
        else
        {
            /* xMessage now contains the received data. */
        }
    }
}
```

Listing 131 Example use of xQueueReceive()

```
void vATask( void *pvParameters )
{
QueueHandle_t xQueue;
AMessage xMessage;

    /* The queue handle is passed into this task as the task parameter.  Cast
    the parameter back to a queue handle. */
    xQueue = ( QueueHandle_t ) pvParameters;

    for( ;; )
    {
        /* Create a message to send on the queue. */
        xMessage.ucMessageID = SEND_EXAMPLE;

        /* Send the message to the queue, waiting for 10 ticks for space to become
        available if the queue is already full. */
        if( xQueueSendToBack( xQueue, &xMessage, 10 ) != pdPASS )
        {
            /* Data could not be sent to the queue even after waiting 10 ticks. */
        }
    }
}
```

Listing 141 Example use of xQueueSendToBack()

3.14 xQueuePeek()

```
#include "FreeRTOS.h"
#include "queue.h"

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

Listing 127 xQueuePeek() function prototype

Summary

Reads an item from a queue, but without removing the item from the queue. The same item will be returned the next time xQueueReceive() or xQueuePeek() is used to obtain an item from the same queue.

```
/* Task to peek the data from the queue. */
void vADifferentTask( void *pvParameters )
{
struct AMessage *pxRxedMessage;

    if( xQueue != 0 )
    {
        /* Peek a message on the created queue.  Block for 10 ticks if a message is
        not available immediately. */
        if( xQueuePeek( xQueue, &( pxRxedMessage ), 10 ) == pdPASS )
        {
            /* pxRxedMessage now points to the struct AMessage variable posted by
            vATask, but the item still remains on the queue. */
        }
    }
    else
    {
        /* The queue could not or has not been created. */
    }

    /* ... Rest of the task code. */
    for( ;; )
    {
    }
}
```

Listing 128 Example use of xQueuePeek()

3.4 xQueueCreateSet()

```
#include "FreeRTOS.h"
#include "queue.h"

QueueSetHandle_t xQueueCreateSet( const UBaseType_t uxEventQueueLength );
```

Listing 110 xQueueCreateSet() function prototype

Summary

Queue sets provide a mechanism to allow an RTOS task to block (pend) on a read operation from multiple RTOS queues or semaphores simultaneously. Note that there are simpler alternatives to using queue sets. See the Blocking on Multiple Objects page of the FreeRTOS.org website for more information.

3.20 xQueueSelectFromSet()

```
#include "FreeRTOS.h"
#include "queue.h"

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

Listing 137 xQueueSelectFromSet() function prototype

```
/* Define the lengths of the queues that will be added to the queue set. */
#define QUEUE_LENGTH_1            10
#define QUEUE_LENGTH_2            10

/* Binary semaphores have an effective length of 1. */
#define BINARY_SEMAPHORE_LENGTH   1

/* Define the size of the item to be held by queue 1 and queue 2 respectively. The
values used here are just for demonstration purposes. */
#define ITEM_SIZE_QUEUE_1  sizeof( uint32_t )
#define ITEM_SIZE_QUEUE_2  sizeof( something_else_t )

/* The combined length of the two queues and binary semaphore that will be added to
the queue set. */
#define COMBINED_LENGTH ( QUEUE_LENGTH_1 + QUEUE_LENGTH_2 + BINARY_SEMAPHORE_LENGTH

void vAFunction( void )
{
static QueueSetHandle_t xQueueSet;
QueueHandle_t xQueue1, xQueue2, xSemaphore;
QueueSetMemberHandle_t xActivatedMember;
uint32_t xReceivedFromQueue1;
something_else_t xReceivedFromQueue2;

    /* Create a queue set large enough to hold an event for every space in every
    queue and semaphore that is to be added to the set. */
    xQueueSet = xQueueCreateSet( COMBINED_LENGTH );

    /* Create the queues and semaphores that will be contained in the set. */
    xQueue1 = xQueueCreate( QUEUE_LENGTH_1, ITEM_SIZE_QUEUE_1 );
    xQueue2 = xQueueCreate( QUEUE_LENGTH_2, ITEM_SIZE_QUEUE_2 );

    /* Create the semaphore that is being added to the set. */
    xSemaphore = xSemaphoreCreateBinary();

    /* Take the semaphore, so it starts empty.  A block time of zero can be used
    as the semaphore is guaranteed to be available - it has just been created. */
    xSemaphoreTake( xSemaphore, 0 );

    /* Add the queues and semaphores to the set.  Reading from these queues and
    semaphore can only be performed after a call to xQueueSelectFromSet() has
    returned the queue or semaphore handle from this point on. */
    xQueueAddToSet( xQueue1, xQueueSet );
    xQueueAddToSet( xQueue2, xQueueSet );
    xQueueAddToSet( xSemaphore, xQueueSet );

    /* CONTINUED ON NEXT PAGE */
```

```
    for( ;; )
    {
        /* Block to wait for something to be available from the queues or semaphor
        that have been added to the set.  Don't block longer than 200ms. */
        xActivatedMember = xQueueSelectFromSet( xQueueSet, pdMS_TO_TICKS( 200 ) );

        /* Which set member was selected?  Receives/takes can use a block time of
        zero as they are guaranteed to pass because xQueueSelectFromSet() would no
        have returned the handle unless something was available. */
        if( xActivatedMember == xQueue1 )
        {
            xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 );
            vProcessValueFromQueue1( xReceivedFromQueue1 );
        }
        else if( xActivatedQueue == xQueue2 )
        {
            xQueueReceive( xActivatedMember, &xReceivedFromQueue2, 0 );
            vProcessValueFromQueue2( &xReceivedFromQueue2 );
        }
        else if( xActivatedQueue == xSemaphore )
        {
            /* Take the semaphore to make sure it can be "given" again. */
            xSemaphoreTake( xActivatedMember, 0 );
            vProcessEventNotifiedBySemaphore();
            break;
        }
        else
        {
            /* The 200ms block time expired without an RTOS queue or semaphore
            being ready to process. */
        }
    }
}
```

3.12 xQueueOverwrite()

```
#include "FreeRTOS.h"
#include "queue.h"

BaseType_t xQueueOverwrite( QueueHandle_t xQueue, const void *pvItemToQueue );
```

Listing 123 xQueueOverwrite() function prototype

Summary

A version of xQueueSendToBack() that will write to the queue even if the queue is full, overwriting data that is already held in the queue.

xQueueOverwrite() is intended for use with queues that have a length of one, meaning the queue is either empty or full.

This function must not be called from an interrupt service routine. See xQueueOverwriteFromISR() for an alternative which may be used in an interrupt service routine.

```
void vFunction( void *pvParameters )
{
QueueHandle_t xQueue;
unsigned long ulVarToSend, ulValReceived;

   /* Create a queue to hold one unsigned long value.  It is strongly
    recommended *not* to use xQueueOverwrite() on queues that can
    contain more than one value, and doing so will trigger an assertion
    if configASSERT() is defined. */
    xQueue = xQueueCreate( 1, sizeof( unsigned long ) );

    /* Write the value 10 to the queue using xQueueOverwrite(). */
    ulVarToSend = 10;
    xQueueOverwrite( xQueue, &ulVarToSend );

    /* Peeking the queue should now return 10, but leave the value 10 in
    the queue.  A block time of zero is used as it is known that the
    queue holds a value. */
    ulValReceived = 0;
    xQueuePeek( xQueue, &ulValReceived, 0 );

    if( ulValReceived != 10 )
    {
        /* Error, unless another task removed the value. */
    }

    /* The queue is still full.  Use xQueueOverwrite() to overwrite the
    value held in the queue with 100. */
    ulVarToSend = 100;
    xQueueOverwrite( xQueue, &ulVarToSend );

    /* This time read from the queue, leaving the queue empty once more.
    A block time of 0 is used again. */
    xQueueReceive( xQueue, &ulValReceived, 0 );

    /* The value read should be the last value written, even though the
    queue was already full when the value was written. */
    if( ulValReceived != 100 )
    {
        /* Error unless another task is using the same queue. */
    }

    /* ... */
}
```

3.24 uxQueueSpacesAvailable()

```
#include "FreeRTOS.h"
#include "queue.h"

UBaseType_t uxQueueSpacesAvailable( const QueueHandle_t xQueue );
```

Listing 144 uxQueueSpacesAvailable() function prototype

Summary

Returns the number of free spaces that are available in a queue. That is, the number of items that can be posted to the queue before the queue becomes full.

Example

```
void vAFunction( QueueHandle_t xQueue )
{
UBaseType_t uxNumberOfFreeSpaces;

    /* How many free spaces are currently available in the queue referenced by the
    xQueue handle? */
    uxNumberOfFreeSpaces = uxQueueSpacesAvailable( xQueue );
}
```

Listing 145 Example use of uxQueueSpacesAvailable()

6.5 Counting Semaphores

Counting semaphores are typically used for two things:

1. Counting events[1]

Counting semaphores that are used to count events are created with an initial count value of zero.

2. Resource management.

In this scenario, the count value indicates the number of resources available. To obtain control of a resource, a task must first obtain a semaphore—decrementing the semaphore's count value. When the count value reaches zero, there are no free resources. When a task finishes with the resource, it 'gives' the semaphore back—incrementing the semaphore's count value.

---

[1] It is more efficient to count events using a direct to task notification than it is using a counting semaphore. Direct to task notifications are not covered until Chapter 9.

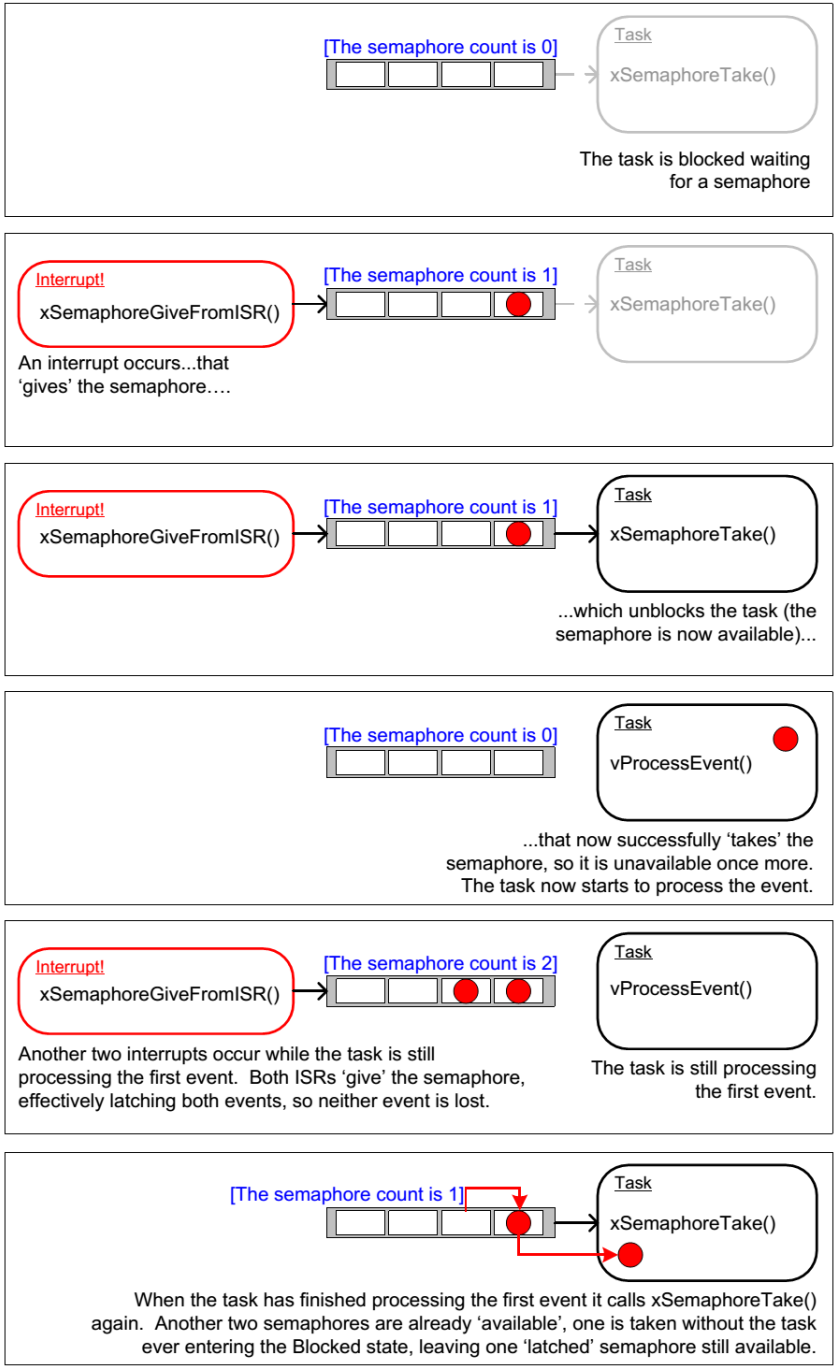


Figure 55. Using a counting semaphore to 'count' events

4.4 xSemaphoreCreateCounting()

```
#include "FreeRTOS.h"
#include "semphr.h"

SemaphoreHandle_t xSemaphoreCreateCounting(    UBaseType_t uxMaxCount,
                                               UBaseType_t uxInitialCount );
```

Listing 152 xSemaphoreCreateCounting() function prototype

4.16 xSemaphoreTake()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreTake( SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait );
```

Listing 173 xSemaphoreTake() function prototype

4.13 xSemaphoreGive()

```
#include "FreeRTOS.h"
#include "semphr.h"

BaseType_t xSemaphoreGive( SemaphoreHandle_t xSemaphore );
```

Listing 167 xSemaphoreGive() function prototype

8.2 Characteristics of an Event Group

Event Groups, Event Flags and Event Bits

- If configUSE_16_BIT_TICKS is 1, then each event group contains 8 usable event bits.
- If configUSE_16_BIT_TICKS is 0, then each event group contains 24 usable event bits.

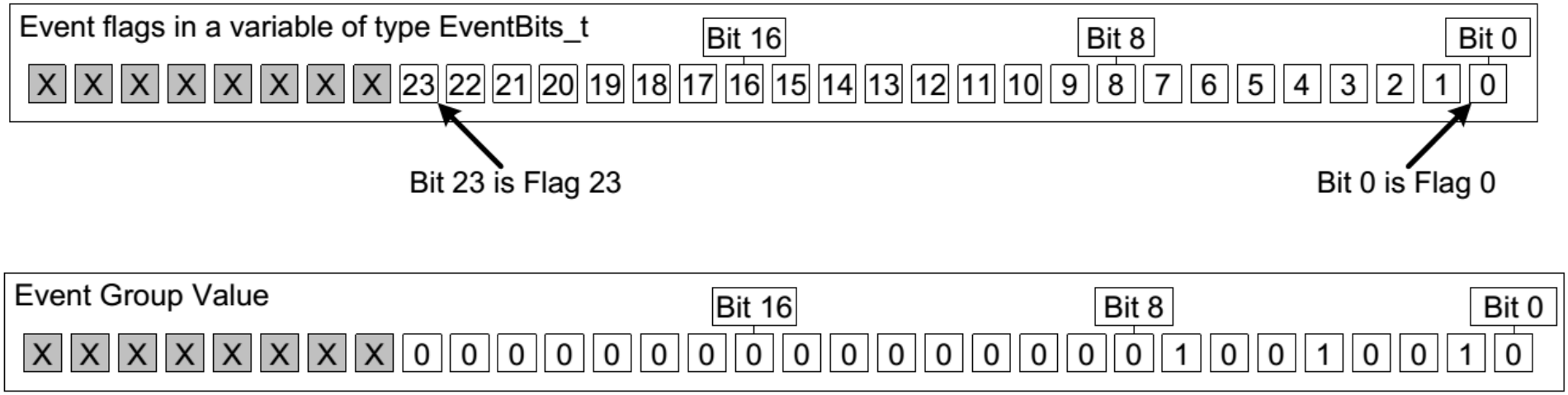


Figure 72 An event group in which only bits 1, 4 and 7 are set, and all the other event flags are clear, making the event group's value 0x92

xEventGroupCreate()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventGroupHandle_t xEventGroupCreate( void );
```

Listing 222 xEventGroupCreate() function prototype

```
/* Declare a variable to hold the created event group. */
EventGroupHandle_t xCreatedEventGroup;

/* Attempt to create the event group. */
xCreatedEventGroup = xEventGroupCreate();

/* Was the event group created successfully? */
if( xCreatedEventGroup == NULL )
{
    /* The event group was not created because there was insufficient
    FreeRTOS heap available. */
}
else
{
    /* The event group was created. */
}
```

Listing 223 Example use of xEventGroupCreate()

xEventGroupWaitBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupWaitBits( const EventGroupHandle_t xEventGroup,
                                 const EventBits_t uxBitsToWaitFor,
                                 const BaseType_t xClearOnExit,
                                 const BaseType_t xWaitForAllBits,
                                 TickType_t xTicksToWait );
```

Listing 235 xEventGroupWaitBits() function prototype

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;
const TickType_t xTicksToWait = pdMS_TO_TICKS( 100 );

    /* Wait a maximum of 100ms for either bit 0 or bit 4 to be set within
    the event group.  Clear the bits before exiting. */
    uxBits = xEventGroupWaitBits(
             xEventGroup,   /* The event group being tested. */
             BIT_0 | BIT_4, /* The bits within the event group to wait for. */
             pdTRUE,        /* BIT_0 and BIT_4 should be cleared before returning. */
             pdFALSE,       /* Don't wait for both bits, either bit will do. */
             xTicksToWait );/* Wait a maximum of 100ms for either bit to be set. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* xEventGroupWaitBits() returned because both bits were set. */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* xEventGroupWaitBits() returned because just BIT_0 was set. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* xEventGroupWaitBits() returned because just BIT_4 was set. */
    }
    else
    {
        /* xEventGroupWaitBits() returned because xTicksToWait ticks passed
        without either BIT_0 or BIT_4 becoming set. */
    }
}
```

Listing 236 Example use of xEventGroupWaitBits()

xEventGroupSetBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupSetBits( EventGroupHandle_t xEventGroup,
                                const EventBits_t uxBitsToSet );
```

Listing 229 xEventGroupSetBits() function prototype

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    /* Set bit 0 and bit 4 in xEventGroup. */
    uxBits = xEventGroupSetBits(
                                xEventGroup,    /* The event group being updat
                                BIT_0 | BIT_4 );/* The bits being set. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* Both bit 0 and bit 4 remained set when the function returned. */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 remained set when the function returned, but bit 4 was
        cleared.  It might be that bit 4 was cleared automatically as a
        task that was waiting for bit 4 was removed from the Blocked
        state. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 remained set when the function returned, but bit 0 was
        cleared.  It might be that bit 0 was cleared automatically as a
        task that was waiting for bit 0 was removed from the Blocked
        state. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 remained set.  It might be that a task
        was waiting for both of the bits to be set, and the bits were cleared
        as the task left the Blocked state. */
    }
}
```

Listing 230 Example use of xEventGroupSetBits()

xEventGroupClearBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupClearBits( EventGroupHandle_t xEventGroup,
                                  const EventBits_t uxBitsToClear );
```

Listing 218 xEventGroupClearBits() function prototype

Return Value

The value of the bits in the event group before any bits were cleared.

```
#define BIT_0 ( 1 << 0 )
#define BIT_4 ( 1 << 4 )

void aFunction( EventGroupHandle_t xEventGroup )
{
EventBits_t uxBits;

    /* Clear bit 0 and bit 4 in xEventGroup. */
    uxBits = xEventGroupClearBits(
                                xEventGroup,    /* The event group being updated. */
                                BIT_0 | BIT_4 );/* The bits being cleared. */

    if( ( uxBits & ( BIT_0 | BIT_4 ) ) == ( BIT_0 | BIT_4 ) )
    {
        /* Both bit 0 and bit 4 were set before xEventGroupClearBits()
        was called.  Both will now be clear (not set). */
    }
    else if( ( uxBits & BIT_0 ) != 0 )
    {
        /* Bit 0 was set before xEventGroupClearBits() was called.  It will
        now be clear. */
    }
    else if( ( uxBits & BIT_4 ) != 0 )
    {
        /* Bit 4 was set before xEventGroupClearBits() was called.  It will
        now be clear. */
    }
    else
    {
        /* Neither bit 0 nor bit 4 were set in the first place. */
    }
}
```

Listing 219 Example use of xEventGroupClearBits()

xEventGroupGetBits()

```
#include "FreeRTOS.h"
#include "event_groups.h"

EventBits_t xEventGroupGetBits( EventGroupHandle_t xEventGroup );
```

Listing 227 xEventGroupGetBits() function prototype