Reference Books

- Using the FreeRTOS Real-Time Kernel: A Practical Guide Cortex-M3 Edition

- SAFE**RTOS**™ User's Manual

What is Real-Time?

- “ Real time in operating systems:

  The ability of the operating system to provide a required level of service in a bounded response time.”

  - POSIX Standard 1003.1

Soft Real-Time

- In a soft real-time system, it is considered undesirable, but not catastrophic, if deadlines are occasionally missed.
- Also known as “best effort” systems
- Most modern operating systems can serve as the base for a soft real time systems
- Examples:
  - Multimedia transmission and reception
  - Networking, telecom (cellular) networks
  - Web sites and services
  - Computer games

Hard Real-Time

- A hard real-time system has time-critical deadlines that must be met; otherwise a catastrophic system failure can occur.
- Absolutely, positively, first time every time
- Requires formal verification/guarantees of being to always meet its hard deadlines (except for fatal errors).
- Examples:
  - Air traffic control
  - Vehicle subsystems control, e.g., airbags, brakes
  - Nuclear power plant control

Why Use a RTOS?

- Task prioritization can help ensure an application meets its processing deadlines
- Abstracting away timing information from applications
- Good Maintainability/Extensibility
- Good Modularity with tasks
- Well-defined interfaces support team development
- Easier testing with well-defined independent tasks
- Code reuse
- Improved efficiency with event-driven software
- Idle task when no applications wishing to execute
- Flexible interrupt handling
- Easier control over peripherals

FreeRTOS

- FreeRTOS is a real-time kernel (or real-time scheduler) targeting at hard real-time applications.
  - Simple
  - Portable
  - Royalty free
  - Concise
- Primarily written in C
- Few assembler functions
- Thumb mode supported

The Cortex-M3 Port of FreeRTOS

- The Cortex-M3 port includes all the standard FreeRTOS features:
  - Pre-emptive or co-operative scheduler operation
  - Very flexible task priority assignment
  - Queues
  - Binary semaphores
  - Counting semaphores
  - Recursive semaphores
  - Mutexes
  - Tick hook functions
  - Idle hook functions
  - Stack overflow checking
  - Trace hook macros

Coding Conventions

- Some project definitions in `ProjDefs.h`

Definition	Value
pdTRUE[a]	1
pdFALSE	0
pdPASS	1
pdFAIL	0

a. The 'pd' prefix denotes that the constant is defined within the ProjDefs.h header file. The ProjDefs.h header file also contains error code definitions that begin with the 'err' prefix.

- Port-dependent definitions

Definition	Value
portCHAR	char (type)
portLONG	long (type)
portSHORT	short (type)
portBASE_TYPE	Port-dependent [a]– defined to be the most efficient data type for the architecture
portMAX_DELAY	Port-dependent
portTickType	Port-dependent

Coding Conventions

- Naming conventions

Parameter Name	Type	Prefix
Variables	portCHAR	c
	portSHORT	s
	portLONG	l
	portBASE_TYPE	x
	structures, and so on	x
	void	v
Pointers	—	p
Unsigned variables	—	u

❖For example, a pointer to a short will have the prefix `ps`;a pointer to void will have the prefix `pv`;an unsigned short will have the prefix `us`

❖Function names are also prefixed with their return type using the same convention

Resources Used By FreeRTOS

- FreeRTOS makes use of
  - SysTick, PendSV, and SVC interrupts
  - These interrupts are not available for use by the application
- FreeRTOS has a very small footprint
  - A typical kernel build will consume approximately 6KB of Flash space and a few hundred bytes of RAM
  - Each task also requires RAM to be allocated for use as the task stack

FreeRTOS, OpenRTOS, SafeRTOS

- **FreeRTOS** uses a modified GPL license
  - FreeRTOS can be used in commercial applications
  - FreeRTOS itself remains open source
  - FreeRTOS users retain ownership of their intellectual property
- **OpenRTOS** shares the same code base as FreeRTOS
  - provided under standard commercial license terms
  - Not open source
  - provides IP infringement protection
- **SafeRTOS** has been developed compliance with various internationally recognized safety related standards.
  - SafeRTOS was originally derived from FreeRTOS
  - retains a similar usage model

Key Concepts in FreeRTOS

- 1. Task management
- 2. Queue management
- 3. Interrupt management
- 4. Resource management

1 Task Management

- Topics covered:
  - ❖How to implement tasks.
  - ❖How to create one or more instances of a task.
  - ❖How to use the task parameter.
  - ❖How to change the priority of a task that has already been created.
  - ❖How to delete a task.
  - ❖How to implement periodic processing.
  - ❖When the idle task will execute and how it can be used.

1.1 Tasks in FreeRTOS

- With FreeRTOS, application can be structured as a set of autonomous tasks
- Each task executes within its own context (e.g., stack) with no coincidental dependency on other tasks
- The scheduler is responsible for starting, stop, swapping in, and swapping out tasks

1.2 Tasks Functions

- Tasks are implemented as C functions
- The prototype of a task function must return void and take a void pointer parameter
- A task will typically execute indefinitely in an infinite loop: must never terminate by attempting to return to its caller
- If required, a task can delete itself prior to reaching the function end
- A single task function definition can be used to create any number of tasks
  - Each created task is a separate execution instance
  - Each created task has its own stack
  - Each created task has its own copy of any automatic variables defined within the task itself

1.2 Tasks Functions

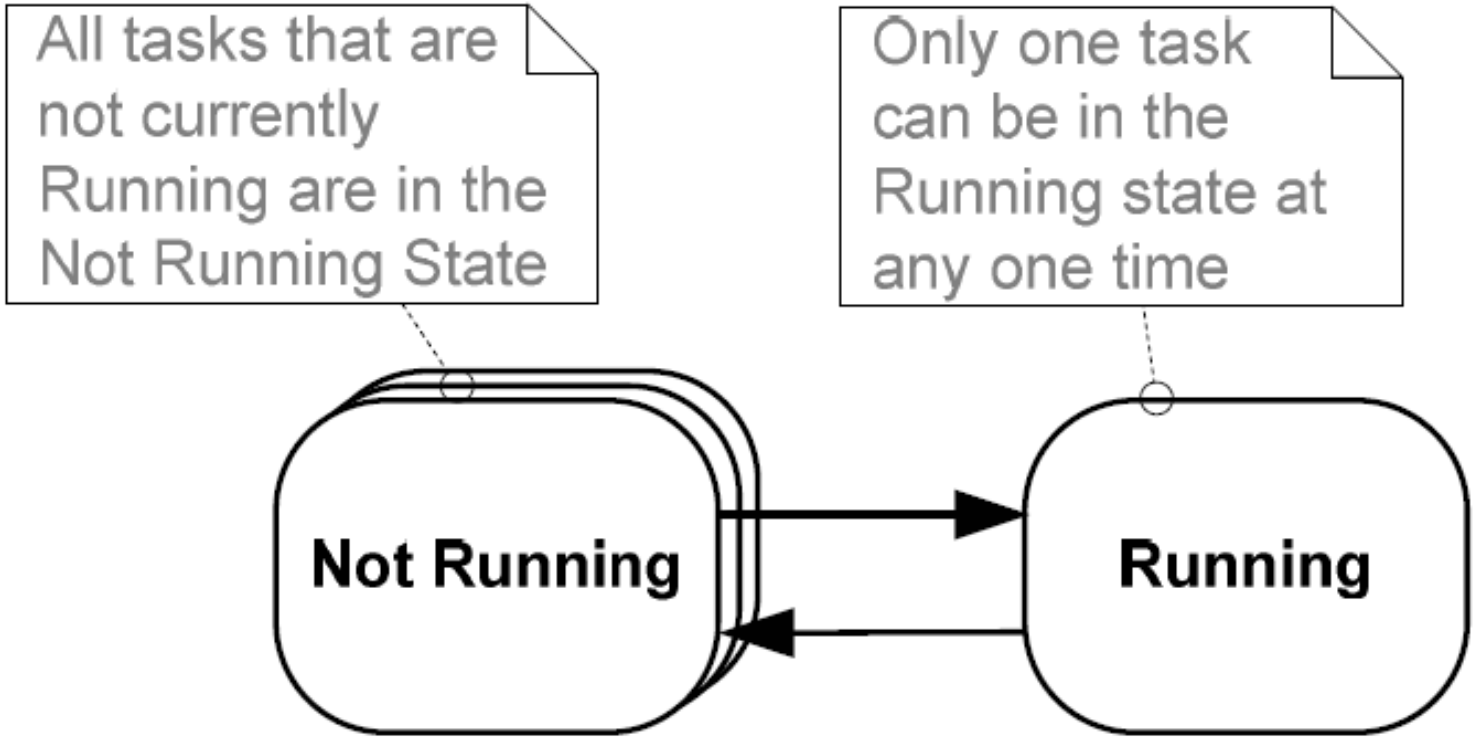
- The structure of a typical task function:

```
void ATaskFunction( void *pvParameters )
{
   /* Each instance of this task function will have its own copy of the iVariableExample
   variable. Except that if the variable was declared static – in which case only one copy of the
   variable would exist and would be shared by all created instance.*/
   int iVariableExample = 0;

   /* A task will normally be implemented as in 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 this function. The NULL parameter passed to the
   vTaskDelete() function indicates that the task to be deleted is the calling (this) task. */
   vTaskDelete( NULL );
}
```

1.3 Task States

- An application can consist of many tasks
- Only one task of the application can be executed at any given time on the microcontroller (single core)
- Thus, a task can exist in one of two states: Running or Not Running
- Only the scheduler can decide which task should enter the Running state
- A task is said to have been “switched in” or “swapped in” when transitioned from the Not Running to the Running state (“switched out” or “swapped out” when transitioned from the Running state to the Not Running state)
- The scheduler is responsible for managing the processor context:
  - Registers values
  - Stack contents

Task States and Transitions



1.4 Create a Task

- A task can be created by calling **`xTaskCreate()`**

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

- `pvTaskCode`
  is a pointer to the function that implement the task
- `pcName`
  is a descriptive name for the task, not used by FreeRTOS
- `usStackDepth`
  specifies the number of words the stack of this task can hold

❖`pvParameters`

is the parameter that can be passed to the task function (pointer to a complex data structure)

❖`uxPriority`

is the priority of the task (0 is the lowest priority, configMAX_PRIORITIES-1 is the highest priority)

❖`pxCreatedTask`

is the handler of the generated task, which can be used to reference the task within API calls

❖Returned value

There are two possible return values:

1. **pdTRUE**: when the task was created successfully
2. **errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY**: the task could not be created because there was insufficient heap memory available for FreeRTOS to allocate enough RAM to hold the task data structures and stack

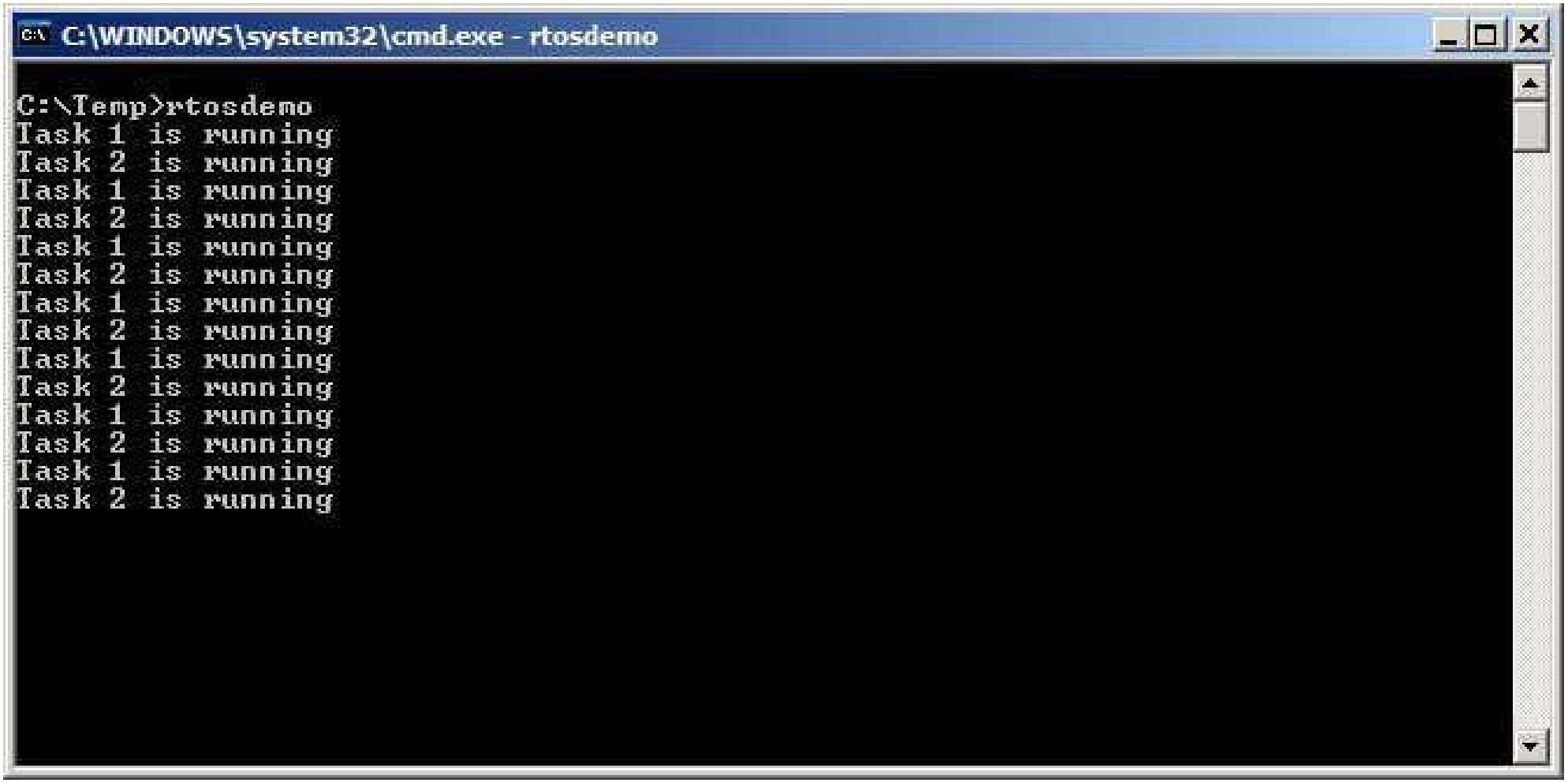
A Simple Printing Application Example

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

```
void vTask2( void *pvParameters )
{
   const char *pcTaskName = "Task 2 is running\r\n";
   volatile unsigned long ul;
   for( ;; )
   {
       vPrintString( pcTaskName );
       for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ ) {}
   }
}
```

```
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 - most small microcontrollers will use much
                             less stack than this. */
                    NULL,    /* We are not using the task parameter. */
                    1,       /* This task will run at priority 1. */
                    NULL );  /* We are not going to 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 5 provides more information on memory management. */
    for( ;; );
}
```

The Output Produced



1.5 Expanding the Not Running State

- The **Blocked** state : A task that is waiting for an event
- Tasks can enter the Blocked state to wait for two different types of event:
  - Temporal (time related) events where a delay period expiring or an absolute time being reached
    - For example, wait for 10 ms to pass
  - Synchronization events where the events originate from another task or interrupt
    - For example, wait for data to arrive on a queue
- It is possible for a task to block on a synchronization event with a timeout
  - For example, wait for a maximum of 10 ms for data to arrive on a queue

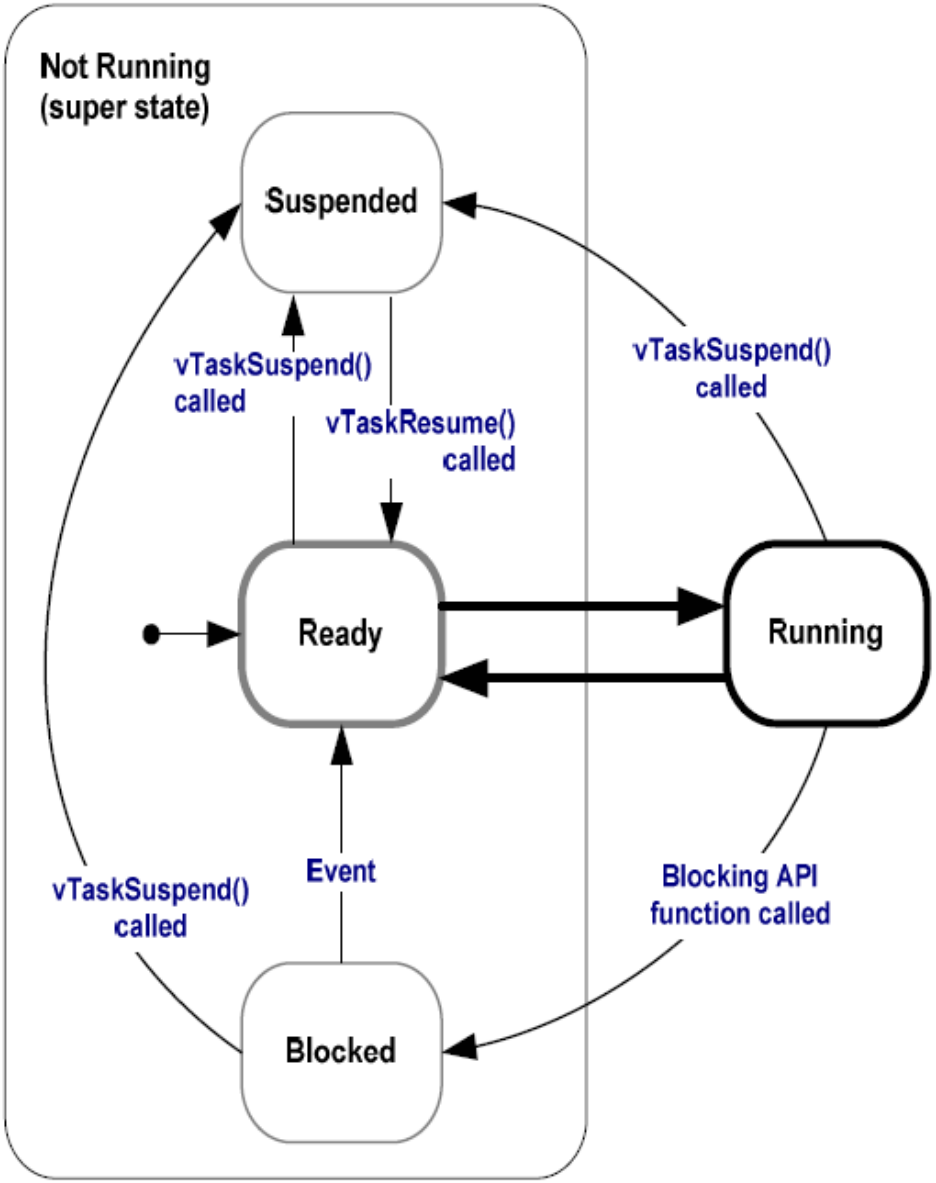
Expanding the Not Running State

- The **Suspended** state: tasks in the Suspended state are not available to the scheduler.
- The only way into the Suspended state is through a call to the `vTaskSuspend()` API function
- the only way out through a call to the `vTaskResume()` or `xTaskResumeFromISR()` API functions

Expanding the Not Running State

- The **Ready** State: tasks that are in the Not Running but are not Blocked or Suspended
- Tasks are able to run, and therefore 'ready' to run, but not currently in the Running state

Task State Transitions



Example: Using the Blocked state to create a delay

- The tasks in the previous examples generate delay using a null loop – polling a counter until it reaches a fixed value
  - Waste processor cycles
- An efficient method is to create a delay with a call to the **`vTaskDelay()`** API function
  - `vTaskDelay()` places the calling task into the Blocked state for a fixed number of tick interrupts
  - The task in the Blocked state will not use any processing time at all
- An alternative method is to use the **`vTaskDelayUntil()`** API function

`vTaskDelay()` Prototype

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

- `xTicksToDelay`

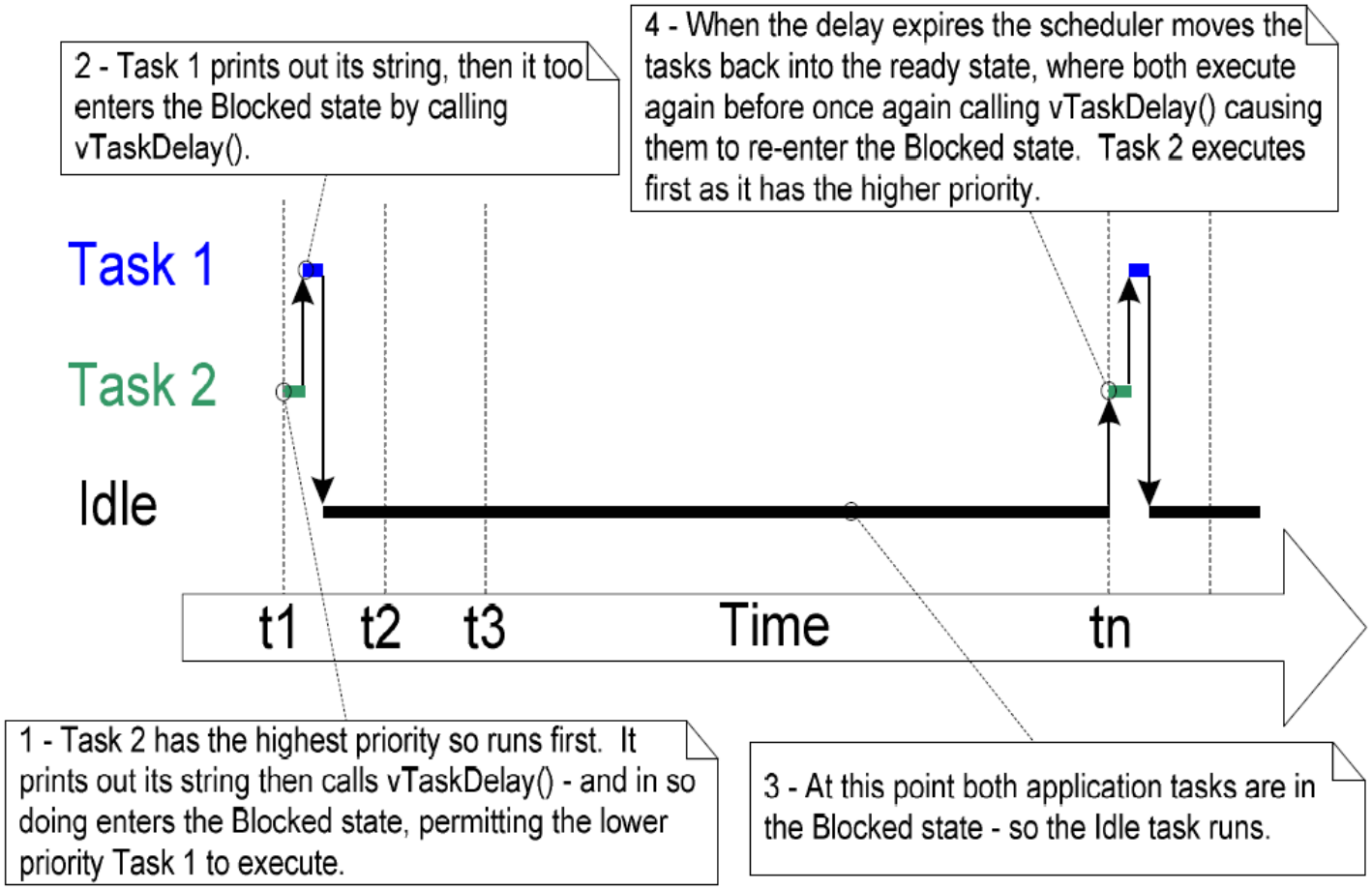
  is the number of tick interrupts that the calling task should remain in the Blocked state before being transitioned back into the Ready state.

  The constant **portTICK_RATE_MS** stores the time in milliseconds of a tick, which can be used to convert milliseconds into ticks.

Improved Printing Application Example Using `vTaskDelay()`

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

    /* 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 delay period is specified in 'ticks', but the constant
        portTICK_RATE_MS can be used to convert this to a more user friendly value
        in milliseconds.  In this case a period of 250 milliseconds is being
        specified. */
        vTaskDelay( 250 / portTICK_RATE_MS );
    }
}
```

Execution Sequence When Using `vTaskDelay()`



The idle task is created automatically when the scheduler is started to ensure there is always at least one task that is able to run (at least one task in the Ready state)

`vTaskDelayUntil()` Prototype

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

❖`pxPreviousWakeTime`

holds the time at which the task last left the Blocked state (was 'woken' up). This time is used as a reference point to calculate the time at which the task should next leave the Blocked state.

The variable pointed to by `pxPreviousWakeTime` is updated automatically within the `vTaskDelayUntil()` function and should be initialized by the application code first.

❖`xTicksToDelay`

`vTaskDelayUntil()` is normally being used to implement a task that executes periodically; the frequency being set by this value which is specified in 'ticks'

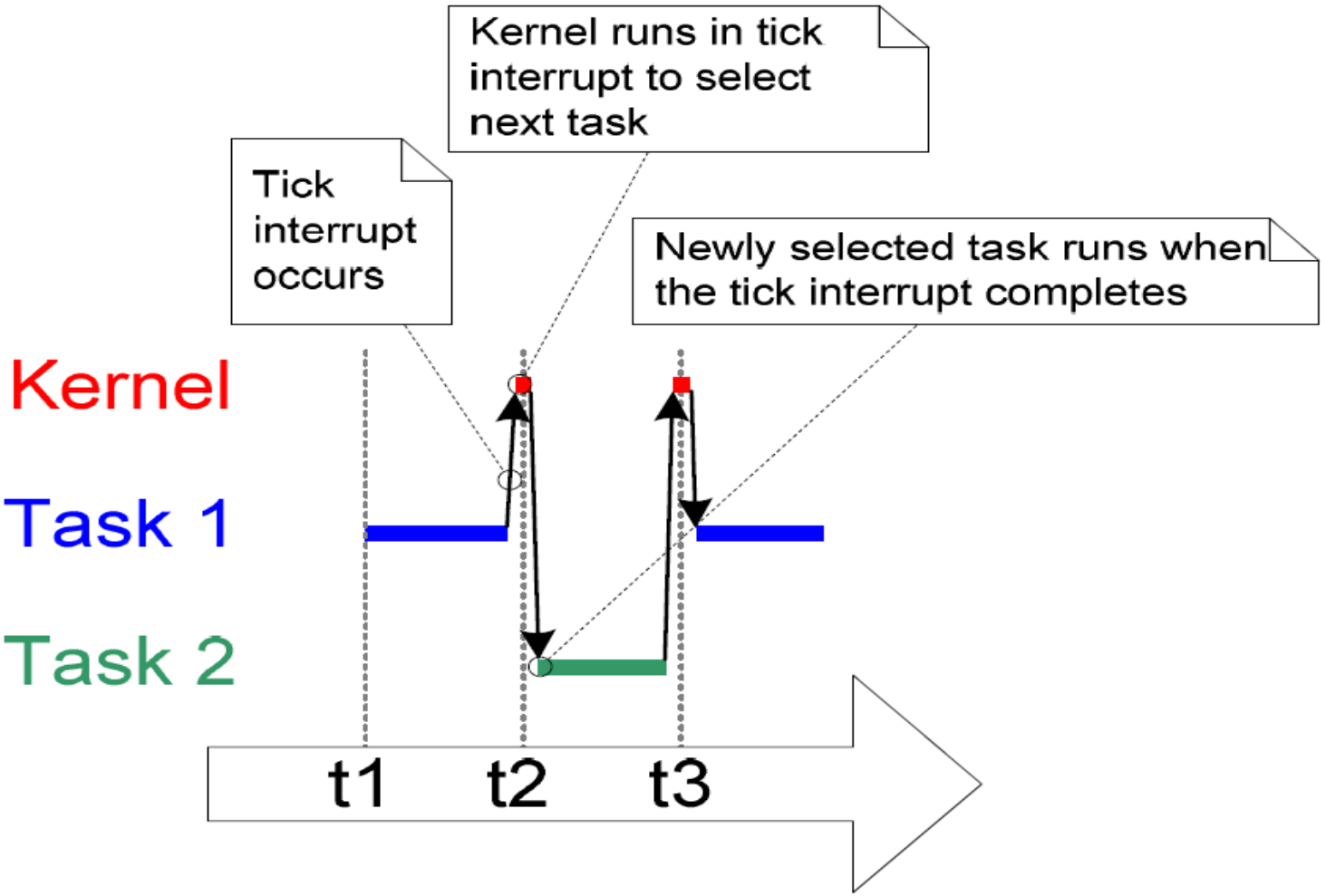
Improved Printing Application Example Using `vTaskDelayUntil()`

```
void vTaskFunction( void *pvParameters )
{
char *pcTaskName;
portTickType 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 updated automatically internally 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 exactly every 250 milliseconds.  As per
        the vTaskDelay() function, time is measured in ticks, and the
        portTICK_RATE_MS constant is used to convert milliseconds into ticks.
        xLastWakeTime is automatically updated within vTaskDelayUntil() so is not
        explicitly updated by the task. */
        vTaskDelayUntil( &xLastWakeTime, ( 250 / portTICK_RATE_MS ) );
    }
}
```

1.6 Task Priority

- Higher priorities task run before lower priority task
- Tasks with the same priority share CPU time (in time slices)
- The scheduler runs itself at the end of each time slice to select the next task to run
- The length of the time slice is set by the SysTick interrupt frequency
  - by configuring the **configTICK_RATE_HZ** while setting the compile configuration in FreeRTOSConfig.h

Execution Sequence of the Example



Change Task Priority

- The initial priority of a task is assigned when created by using `xTaskCreate()`
- The task priority can be queried by using the `xTaskPriorityGet()`
- The task priority can be changed by using `vTaskPrioritySet()`
- The range of priorities is configurable 0 - desired amount (**configMAX_PRIORITIES** in FreeRTOSConfig.h)
- Low numeric values denote low priority tasks (0 is the lowest priority)

vTaskPrioritySet() & uxTaskPriorityGet()

```
void vTaskPrioritySet( xTaskHandle pxTask, unsigned
                       portBASE_TYPE uxNewPriority );
```

❖pxTask

The handle of the task whose priority is being modified.

A task can change its own priority by passing NULL in place of a valid task handle

❖uxNewPriority

The priority to which the subject task is to be set

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

❖pxTask

The handle of the task whose priority is being queried.

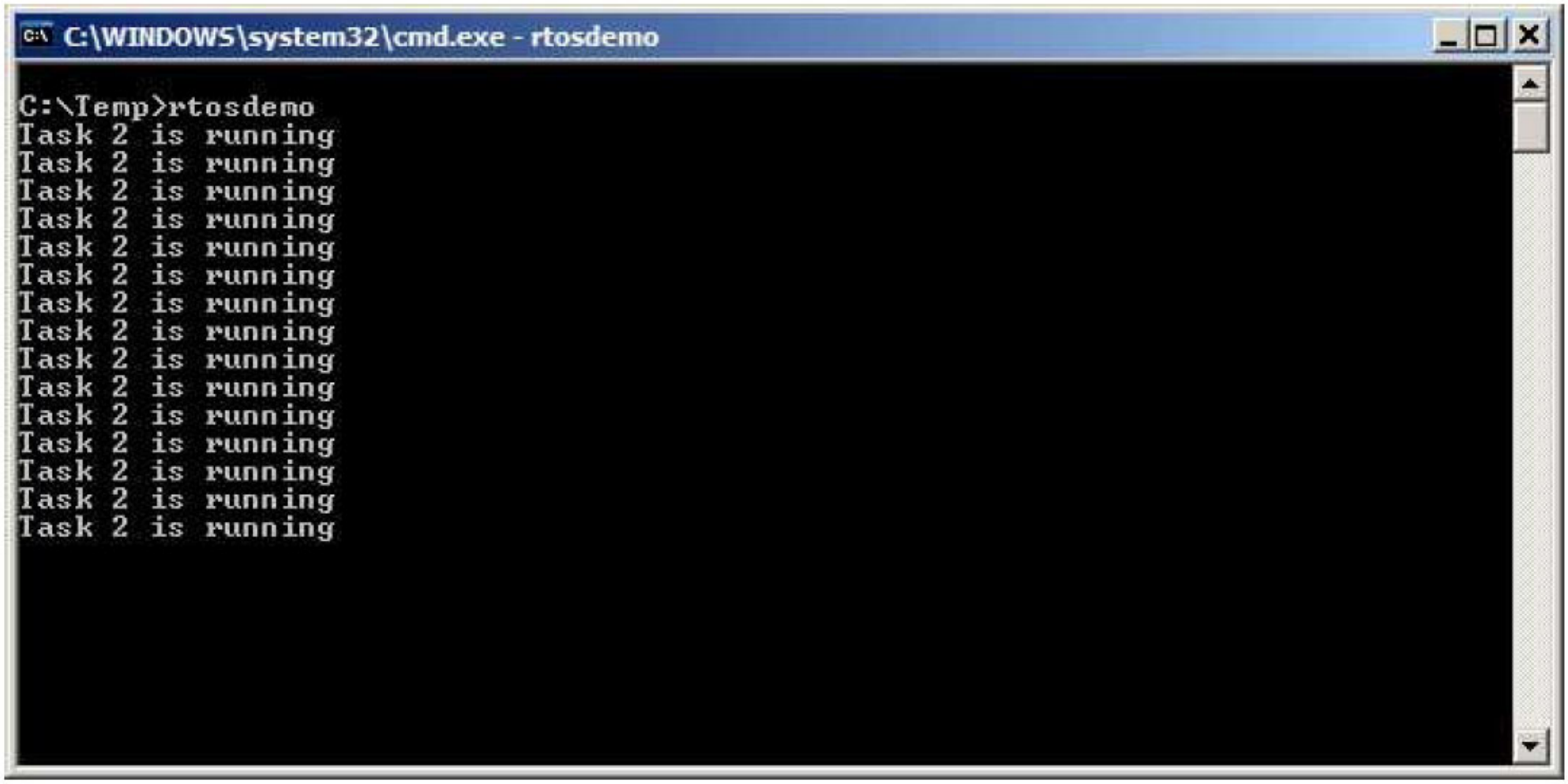
❖Returned value

The priority currently assigned to the task being queried.

Experiment with Priorities

```
/* 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\t\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. */
    xTaskCreate( vTaskFunction, "Task 2", 1000, (void*)pcTextForTask2, 2, NULL );

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

    return 0;
}
```

The Output Produced



1.7 Idle Task and Idle Task Hook Functions

- The idle task is executed when no application tasks are in Running and Ready state
- The idle task is automatically created by the scheduler when `vTaskStartScheduler()` is called
- The idle task has the lowest possible priority (priority 0) to ensure it never prevents a higher priority application task
- Application specific functionality can be directly added into the idle task via an idle hook (or call-back) function
  - An idle hook function is automatically called per iteration of the idle task loop
  - Can be utilized to execute low priority, background or continuous processing

1.9 Delete a Task

- A task can use the **`vTaskDelete()`** API function to delete itself or any other task
- The memory allocated by the kernel to a deleted task is freed by the idle task
- **`vTaskDelete()`** prototype:

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

❖`pxTaskToDelete`

The handle of the task that is to be deleted

A task can delete itself by passing NULL in place of a valid task handle

2 Queue Management

- Topics covered:
  - ❖How to create a queue.
  - ❖How a queue manages the data it contains.
  - ❖How to send data to a queue.
  - ❖How to receive data from a queue.
  - ❖What it means to block on a queue.
  - ❖The effect task priorities have when writing to and reading from a queue.

2.1 Characteristics of a Queue

- Queue is used for inter-communication among autonomous tasks
- A queue can hold a finite number of fixed size data items
- Normally, it is First In First Out (FIFO) buffers
- Writing data to a queue causes a byte for byte *copy* of data
- Reading data from a queue will remove the data
- Any number of tasks can write (read) to (from) the same queue

Blocking on Queue Reads

- If the queue is empty, a task attempting to read from the queue will be blocked
  - When new data is placed into the queue, the blocked task will automatically be moved from the Blocked state to the Ready state
  - When multiple tasks are blocked, the task with the highest priority will be unblocked
  - If the blocked tasks have equal priority, the task that has been waiting for data the longest will be unblocked
  - The blocked task will also be automatically moved to the Ready state if the specified block time expires before data becomes available.

Blocking on Queue Writes

- If the queue is already full, a task attempting to write to the queue will be blocked
  - When there is space available in the queue, the blocked task will automatically be moved from the Blocked state to the Ready state
  - When multiple tasks are blocked, the task waiting for space with the highest priority will be unblocked
  - If the blocked tasks have equal priority, the task that has been waiting for space the longest will be unblocked
  - The blocked task will also be automatically moved to the Ready state if the specified block time expires

2.2 Using a Queue

- A queue must be explicitly created before it can be used
- **`xQueueCreate()`** is used to create a queue and returns an xQueueHandle to reference the queue it creates

```
xQueueHandle xQueueCreate(    unsigned portBASE_TYPE uxQueueLength,
                              unsigned portBASE_TYPE uxItemSize );
```

- `uxQueueLength`
  The maximum number of items that the queue being created can hold at any one time
- `uxItemSize`
  The size in bytes of each data item that can be stored in the queue
- Return value
  If NULL is returned then the queue could not be created because there was insufficient heap memory available;
  A non-NULL value being returned indicates that the queue was created successfully.

Write to a Queue

- **`xQueueSendToFront()`** (**`xQueueSendToBack()`**) is used to send data to the back/tail (front/head) of a queue
- They should never be called from an interrupt service routine, instead, **`xQueueSendToFrontFromISR()`** (**`xQueueSendToBackFromISR()`**) should be used

```
portBASE_TYPE xQueueSendToBack(     xQueueHandle xQueue,
                                    const void * pvItemToQueue,
                                    portTickType xTicksToWait );
```

- `xQueue` The handle of the queue to which the data is being sent
- `pvItemToQueue` A pointer to the data that will be copied into the queue
- `xTicksToWait`

  The maximum amount of time the task should remain in the Blocked state to wait for space to become available on the queue, if queue is full
- Return value

  `pdPASS` will only be returned if data was successfully sent to the queue; `errQUEUE_FULL` will be returned if data could not be written to the queue because the queue was already full.

Read from a Queue

- **`xQueueReceive()`** is used to receive (read) an item from a queue and the item is removed from the queue
- **`xQueuePeek()`** is used to receive an item from a queue *without* the item being removed from the queue
- In ISRs, **`xQueueReceiveFromISR()`** API function should be used

```
portBASE_TYPE xQueueReceive(  xQueueHandle xQueue,
                              const void * pvBuffer,
                              portTickType xTicksToWait );
```

- `xQueue` The handle of the queue to which the data is being received
- `pvBuffer` A pointer to the memory into which the received data will be copied
- `xTicksToWait`

  The maximum amount of time the task should remain in the Blocked state to wait for data to become available on the queue, if queue is empty
- Return value

  `pdPASS` will only be returned if data was successfully read from the queue; `errQUEUE_EMPTY` is returned when data could not be read from the queue because the queue was already empty.

Query a Queue

- `uxQueueMessagesWaiting()` is used to query the number of items that are currently in a queue
- In ISRs, `uxQueueMessagesWaitingFromISR()` API function should be used

```
unsigned portBASE_TYPE uxQueueMessagesWaiting( xQueueHandle xQueue );
```

- `xQueue`
  The handle of the queue being queried
- Return value
  The number of items that the queue being queried is currently holding.

Example of Using Queue

```
/* Declare a variable of type xQueueHandle.  This is used to store the reference
to the queue that is accessed by all three tasks. */
xQueueHandle 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 long. */
    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();
    }
    /* 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 5 provides more information on memory management. */
    for( ;; );
}
```

```
static void vSenderTask( void *pvParameters )
{
long lValueToSend;
portBASE_TYPE xStatus;
lValueToSend = ( long ) pvParameters;
      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 as the queue should never contain more than
              one item! */
              vPrintString( "Could not send to the queue.\r\n" );
          }
          /* Allow the other sender task to execute.  taskYIELD() informs the
          scheduler that a switch to another task should occur now rather than
          keeping this task in the Running state until the end of the current time
          slice. */
          taskYIELD();
      }
}
```

```
static void vReceiverTask( void *pvParameters )
{
/* Declare the variable that will hold the values received from the queue. */
long lReceivedValue;
portBASE_TYPE xStatus;
const portTickType xTicksToWait = 100 / portTICK_RATE_MS;
   for( ;; )
   {
         /* 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 should remain in the Blocked state to wait for data to be available
         should the queue already be empty.  In this case the constant
         portTICK_RATE_MS is used to convert 100 milliseconds to a time specified in
         ticks. */
         xStatus = xQueueReceive( xQueue, &lReceivedValue, xTicksToWait );
         if( xStatus == pdPASS )
         {
             /* Data was successfully received from the queue, print out the received
             value. */
             vPrintStringAndNumber( "Received = ", lReceivedValue );
         }
    }
}
```

2.3 Working with Large Data

- If the size of data being stored in the queue is large, it is preferable to use *pointers* to the data rather than copy the data itself
- More efficient in both processing time and the amount of RAM required
- It is essential to ensure that both tasks do not modify the memory contents simultaneously
- Dynamically allocated memory should be freed by exactly one task and no tasks should attempt to access the memory after it has been freed

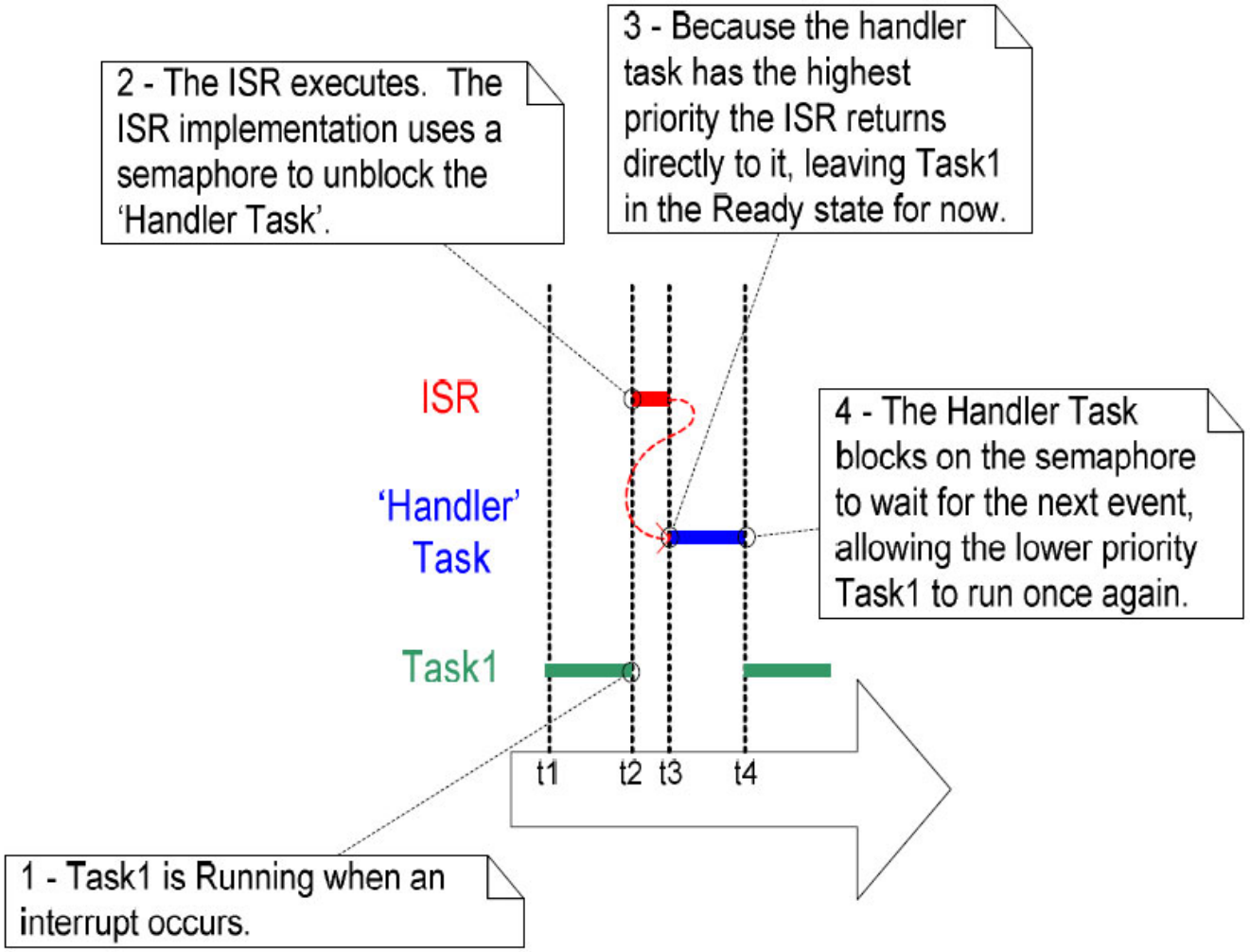
3 Interrupt Management

- Topics covered:
  - ❖Which FreeRTOS API functions can be used from within an interrupt service routine.
  - ❖How a deferred interrupt scheme can be implemented.
  - ❖How to create and use binary semaphores and counting semaphores.
  - ❖The differences between binary and counting semaphores.
  - ❖How to use a queue to pass data into and out of an interrupt service routine.

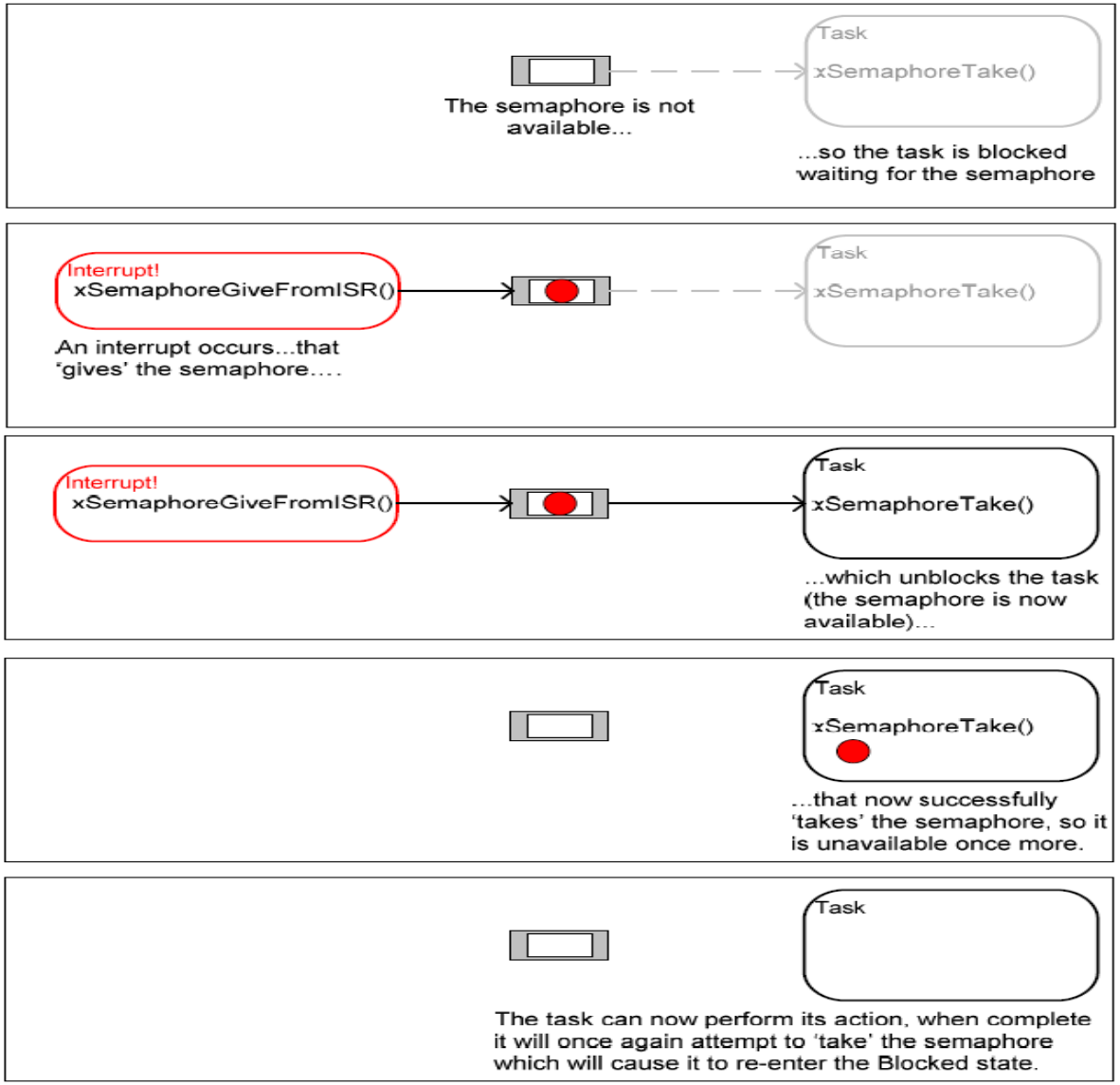
3.1 Deferred Interrupt Processing

- Binary Semaphores used for Synchronization
  - A Binary Semaphore can be used to unblock a task each time a particular interrupt occurs
  - The majority of the interrupt event processing can be implemented within the synchronized task in the ISR
  - Only a very fast and short portion remaining directly
  - The interrupt processing is said to have been 'deferred' to a 'handler' task
  - The handler task uses a blocking 'take' call to a semaphore and enters the Blocked state to wait for the event to occur
  - When the event occurs, the ISR uses a 'give' operation on the same semaphore to unblock the task

Interrupt and Handler Task



Interrupt and Handler Task



Create a Binary Semaphore

- Use the **`vSemaphoreCreateBinary()`** API function (macro) to create a binary semaphore

```
void vSemaphoreCreateBinary(xSemaphoreHandle xSemaphore);
```

- `xSemaphore`
  The semaphore being created

‘Take’ a Semaphore

- Use **`xSemaphoreTake()`** to ‘take’ a semaphore

```
portBASE_TYPE xSemaphoreTake(xSemaphoreHandle xSemaphore,
                             portTickType xTicksToWait );
```

- `xSemaphore`
  The semaphore being ‘taken’
- `xTicksToWait`
  The maximum amount of time the task should remain in the Blocked state;
  Setting xTicksToWait to portMAX_DELAY will cause the task to wait indefinitely
- Return value
  `pdPASS` will only be returned if it was successful in obtaining the semaphore
  `pdFALSE` if the semaphore was not available

‘Give’ a Semaphore

- Use **`xSemaphoreGive()`**(**`xSemaphoreGiveFromISR()`**)to ‘give’ a semaphore (when in an ISR)

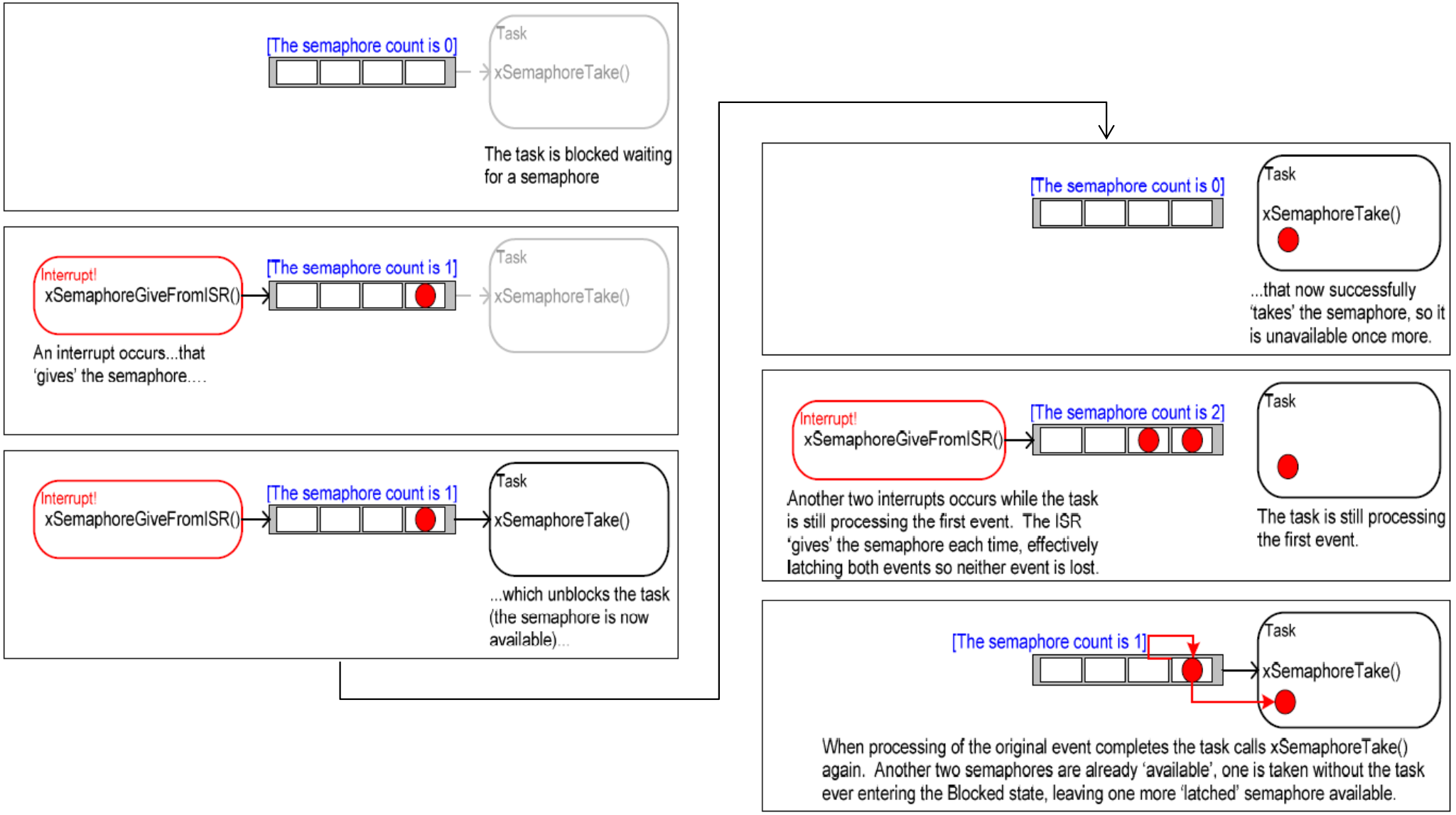
```
portBASE_TYPE xSemaphoreGiveFromISR(
                        xSemaphoreHandle xSemaphore,
                        portBASE_TYPE *pxHigherPriorityTaskWoken );
```

- `xSemaphore`
  The semaphore being ‘given’
- `pxHigherPriorityTaskWoken`
  If the handle task has a higher priority than the currently executing task (the task that was interrupted), this value will be set to `pdTRUE`
- Return value
  `pdPASS` will only be returned if successful
  `pdFAIL` if a semaphore is already available and cannot be given

3.2 Counting Semaphores

- When interrupts come at a speed faster than the handler task can process, events will be lost
- Counting semaphore can be thought of as queues that have a length of more than one
- Each time a counting semaphore is 'given', another space in its queue is used, count value is the length of the queue
- Counting semaphores are typically used for two things:
  - **Counting events:** the count value is the difference between the number of events that have occurred and the number that have been processed
  - **Resource management:** the count value indicates the number of resources available; a task must first obtains a semaphore before obtains the control of a resource; a task returns a semaphore when finishing with the resource

Using a counting semaphore to ‘count’ events



Create a Counting Semaphore

- Use **`xSemaphoreCreateCounting()`** to create a counting semaphore

```
xSemaphoreHandle xSemaphoreCreateCounting(
                    unsigned portBASE_TYPE uxMaxCount,
                    unsigned portBASE_TYPE uxInitialCount );
```

- `uxMaxCount`
  The maximum value the semaphore will count to
- `uxInitialCount`
  The initial count value of the semaphore after it has been created
- Return value
  If NULL is returned then the semaphore could not be created because there was insufficient heap memory available

  A non-NULL value being returned indicates that the semaphore was created successfully. The returned value should be stored as the handle to the created semaphore.

4 Resource Management

- Resources that are shared between tasks or between tasks and interrupts needs to be managed using a ‘mutual exclusion’ technique to ensure data consistency
- Topics covered:
  - ❖When and why resource management and control is necessary.
  - ❖What a critical section is.
  - ❖What mutual exclusion means.
  - ❖What it means to suspend the scheduler.
  - ❖How to use a mutex.
  - ❖How to create and use a gatekeeper task.
  - ❖What priority inversion is and how priority inheritance can lessen (but not remove) its impact.

4 Resource Management

- Resources that are shared between tasks or between tasks and interrupts needs to be managed using a ‘mutual exclusion’ technique to ensure data consistency
- Topics covered:
  - ❖When and why resource management and control is necessary.
  - ❖What a critical section is.
  - ❖What mutual exclusion means.
  - ❖What it means to suspend the scheduler.
  - ❖How to use a mutex.
  - ❖How to create and use a gatekeeper task.
  - ❖What priority inversion is and how priority inheritance can lessen (but not remove) its impact.

4.1 Basic Critical Sections

- Basic critical sections protect a region of code from access by other tasks and by interrupts
- Regions of code that are surrounded by calls to the macros **`taskENTER_CRITICAL()`** and **`taskEXIT_CRITICAL()`** respectively, e.g.,

```
/* 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
may still execute on FreeRTOS ports that allow interrupt nesting, but
only interrupts whose priority is above the value assigned to the
configMAX_SYSCALL_INTERRUPT_PRIORITY constant – and those interrupts are
not permitted to call FreeRTOS API functions. */
PORTA |= 0x01;

/* We have finished accessing PORTA so can safely leave the critical
section. */
taskEXIT_CRITICAL();
```

- A very crude method of providing mutual exclusion as all or partial interrupts are disabled

4.2 Suspending (Locking) the Scheduler

- Critical sections can also be created by suspending the scheduler
- A critical section that is too long to be implemented by simply disabling interrupts can instead be implemented by suspending the scheduler
- The scheduler is suspended by calling **`vTaskSuspendAll()`**

```
void vTaskSuspendAll( void );
```

- The scheduler is resumed by calling **`xTaskResumeAll()`**

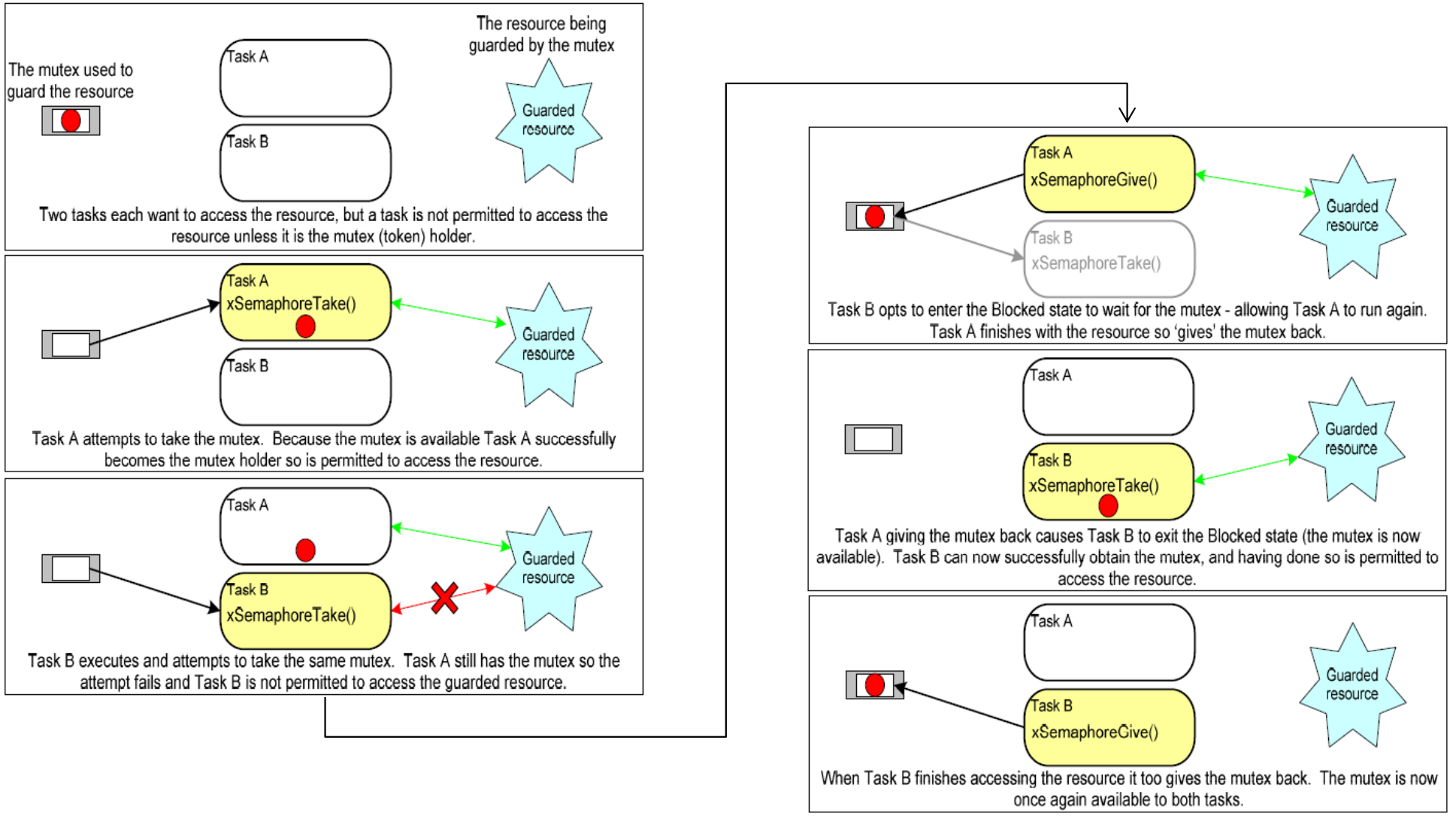
```
portBASE_TYPE xTaskResumeAll( void );
```

- Return value
  - `pdTRUE` if a previously pending context switch being performed before xTaskResumeAll() returns
  - `pdFALSE` otherwise

4.3 Mutexes

- A Mutex is a special type of binary semaphore that is used to control access to a resource that is shared between two or more tasks
- The mutex can be conceptually thought of as a token associated with the resource being shared
- For a task to legitimately access the resource it must first successfully ‘take’ the token
- When the token holder has finished with the resource it must ‘give’ the token back.

Mutual Exclusion Implemented Using a Mutex



Create a Mutex

- Use `xSemaphoreCreateMutex()` to create a mutex

```
xSemaphoreHandle xSemaphoreCreateMutex( void );
```

- Return value

  If NULL is returned then the mutex could not be created because there was insufficient heap memory available

  A non-NULL value being returned indicates that the mutex was created successfully. The returned value should be stored as the handle to the created mutex.

4.4 Priority Inversion & Priority Inheritance

- With a mutex, it is possible that a task with a higher priority has to wait for a task with a lower priority which hold the mutex to give up the control of the mutex
- A higher priority task being delayed by a lower priority task in this manner is called 'priority inversion'.
- Priority inheritance works by temporarily raising the priority of the mutex holder to that of the highest priority task that is attempting to obtain the same mutex
- The priority of the mutex holder is automatically reset back to its original value when it gives the mutex back
- Priority inheritance does not 'fix' priority inversion, it merely lessens its impact.

4.5 Deadlock

- Deadlock occurs when two tasks are both waiting for a resource that is held by the other, e.g.,

1) Task A executes and successfully takes mutex X.
2) Task A is pre-empted by Task B.
3) Task B successfully takes mutex Y before attempting to also take mutex X – but mutex X is held by Task A so is not available to Task B. Task B opts to enter the Blocked state to wait for mutex X to be released.
4) Task A continues executing. It attempts to take mutex Y – but mutex Y is held by Task B so is not available to Task A. Task A opts to enter the Blocked state to wait for mutex Y to be released.

4.6 Gatekeeper Tasks

- Gatekeeper tasks provide a clean method of implementing mutual exclusion without the worry of priority inversion or deadlock
- A gatekeeper task is a task that has sole ownership of a resource
- A task needing to access the resource can only do so indirectly by using the services of the gatekeeper