Introduction to FreeRTOS
Part 1

FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port.

```
FreeRTOS
 |   |
 |   ├─Source  Directory containing the FreeRTOS source files
 |   |
 |   └─Demo    Directory containing pre-configured and port specific FreeRTOS demo projects
 |
FreeRTOS-Plus
 |
 ├─Source  Directory containing source code for some FreeRTOS+ ecosystem components
 |
 └─Demo    Directory containing demo projects for FreeRTOS+ ecosystem components
```

FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port.

FreeRTOSConfig.h: configure FreeRTOS.

```c
#ifndef _FREERTOSCONFIG_H
#define _FREERTOSCONFIG_H

#include "xparameters.h"
#define configUSE_PREEMPTION 1
#define configUSE_MUTEXES 1
#define INCLUDE_xSemaphoreGetMutexHolder 1
#define configUSE_RECURSIVE_MUTEXES 1
#define configUSE_COUNTING_SEMAPHORES 1
#define configUSE_TIMERS 1
#define configUSE_IDLE_HOOK 0
#define configUSE_TICK_HOOK 0
#define configUSE_DAEMON_TASK_STARTUP_HOOK 0
#define configUSE_MALLOC_FAILED_HOOK 1
...
```

FreeRTOS is supplied as a set of C source files. Some of the source files are common to all ports, while others are specific to a port.

```
FreeRTOS
│
└── Source
     │
     ├── tasks.c  FreeRTOS source file - always required
     ├── list.c  FreeRTOS source file - always required
     ├── queue.c  FreeRTOS source file - nearly always required
     ├── timers.c  FreeRTOS source file - optional
     ├── event_groups.c FreeRTOS source file - optional
     └── croutine.c  FreeRTOS source file - optional
```

Repository (Library) for freeRTOS

- A stand-alone board support package (BSP) is a library generated by the Xilinx SDK that is specific to a hardware design.

- It contains initialization code for bringing up the ARM CPUs in ZYNQ and also contains software drivers for all available ZYNQ peripherals.

The freeRTOS Repository

- The FreeRTOS port extends the stand-alone BSP to also include FreeRTOS source files

- After using this port in a Xilinx SDK environment, the user gets all the FreeRTOS source files in a FreeRTOS BSP library.

- This library uses the Xilinx SDK generated stand-alone BSP library.

Header Files

A source file that uses the FreeRTOS API must include ‘FreeRTOS.h’, followed by the header file that contains the prototype for the API function being used —

‘task.h’, ‘queue.h’, ‘semphr.h’, ‘timers.h’ or ‘event_groups.h’.

```
/* FreeRTOS includes. */
#include "FreeRTOS.h"
#include "task.h"
#include "queue.h"
#include "timers.h"
```

A Task

- Simple C Function
- A pointer to parameters (void*) as input
- Creates a forever loop (while (1))
- The tasks are controlled by the Scheduler (freeRTOS internal function)

Each task has his own Stack:

- Every variable you declare or memory allocate uses memory on the stack.
- The stack size of a task depends on the memory consumed by its local variables and function call depth.
- Please note that if your task (or function) uses printf, it consumes around 1024 bytes of stack.
- At minimum however, you would need at least 256 bytes + your estimated stack space above.
- If you don't allocate enough stack space, your CPU will run to an exception and/or freeze

A Task

```
void myTask (void *pvParameters){

    /* variables declaration */
    Int IvariableExample = 0;

    /* Task implemented as a infinite loop */
    for ( ;; )
        {
        /* Task Code here */
    }

    /* Function vTaskDelete () delete itself passing NULL parameter */
    vTaskDelete ( NULL );
}
```

Creating a Task

The Task function itself:

```
void ATaskFunction( void *pvParameters)
{
   // do initilisation
   while (1)
   {
// Task execution code
   }
}
```

Install the Task (in main.c):

```
portBASE_TYPE xTaskCreate(
    pdTASK_CODE    pvTaskCode,     // pointer to the Task
    Char*          pcName,         // String: name of Task for debug
    unsigned short usStackDepth,   // Stacksize
    Void*          pvParameters,   // pointer to Parameters
    unsigned short uxPriority,     // Priority
    XtaskHandle*   pxCreatedTask);// Pointer to receive Task handle
```

Return **pdPASS** or **pdFAIL** (when insufficient heap memory)

Example

```
void hello_world_task (void* p)
{
   while(1)
   {
    Printf(" Hello World!");
    vTaskDelay(1000);
   }
}

void main(void )
{
  XtaskCreate (hello_world_task, "TestTask", 512, NULL, 1, NULL);
  vTaskStartScheduler();
 //   never comes here
}
```

The main function in FreeRTOS based projects creates tasks.
FreeRTOS will let you multi-task based on your tasks and their priority.

Data Types

two port specific data types:
TickType_t and **BaseType_t** (both in portmacro.h).

TickType_t: FreeRTOS configures a periodic interrupt called the tick interrupt. The time between two tick interrupts is called the tick period. Times are specified as multiples of tick periods.

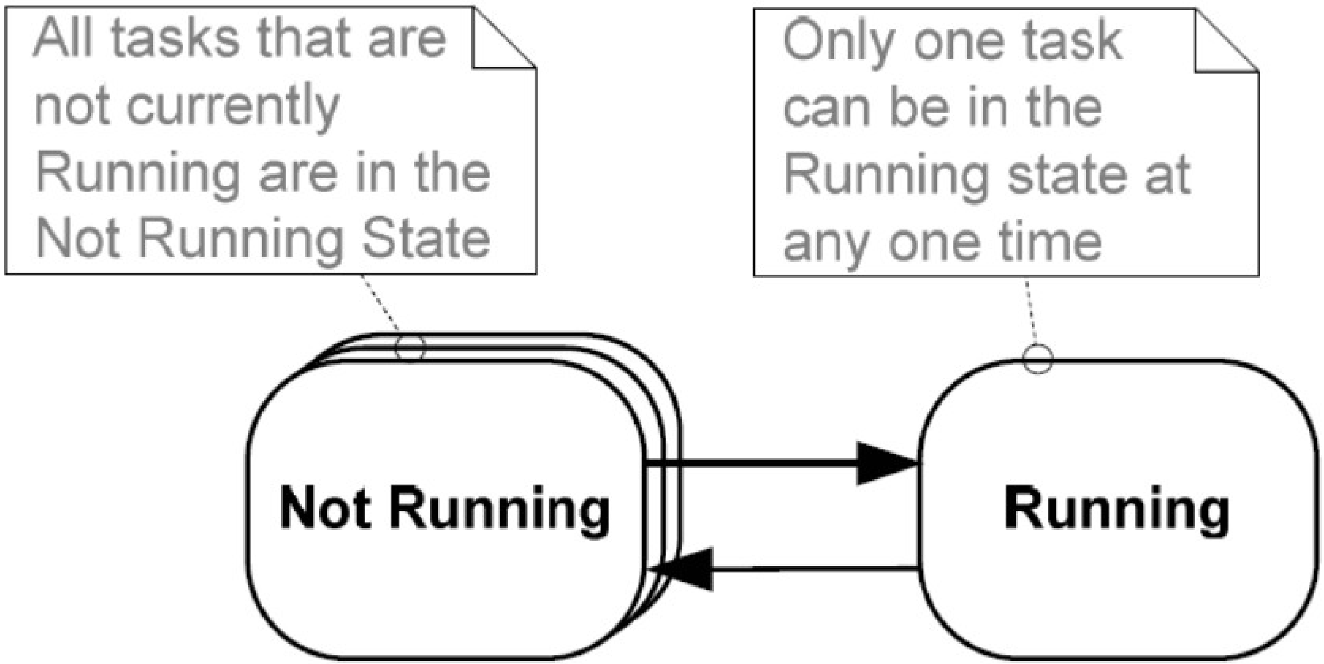
BaseType_t: is generally used for return types that can take only a very limited range of values, and for pdTRUE/pdFALSE type Booleans.

Function Names

Functions are prefixed with both the type they return, and the file they are defined within. For example:

- **v**TaskPrioritySet() returns a **void** and is defined within task.c.
- **x**QueueReceive() returns a **variable** of type BaseType_t and is defined within queue.c.
- **pv**TimerGetTimerID() returns a **pointer to void** and is defined within timers.c.

Top Level Task States



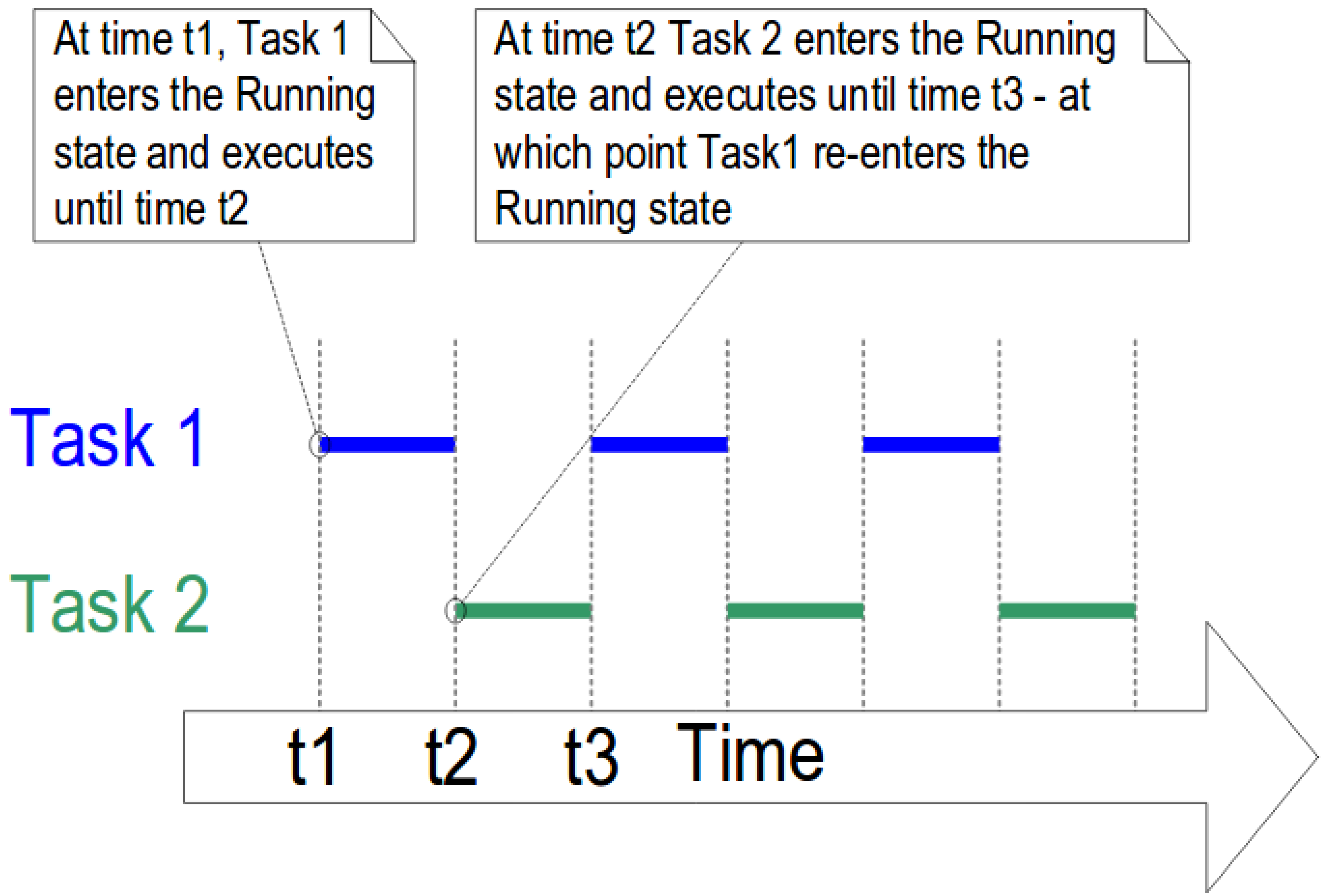
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Task running with the same priority

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul;
/*The string to print out is passed in via the parameter.*/
    pcTaskName = ( char * ) pvParameters;
/* As per most tasks, this task is implemented in an infiniteloop. */
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )/*Delay for a period. */
            {
        }
    }
}
```

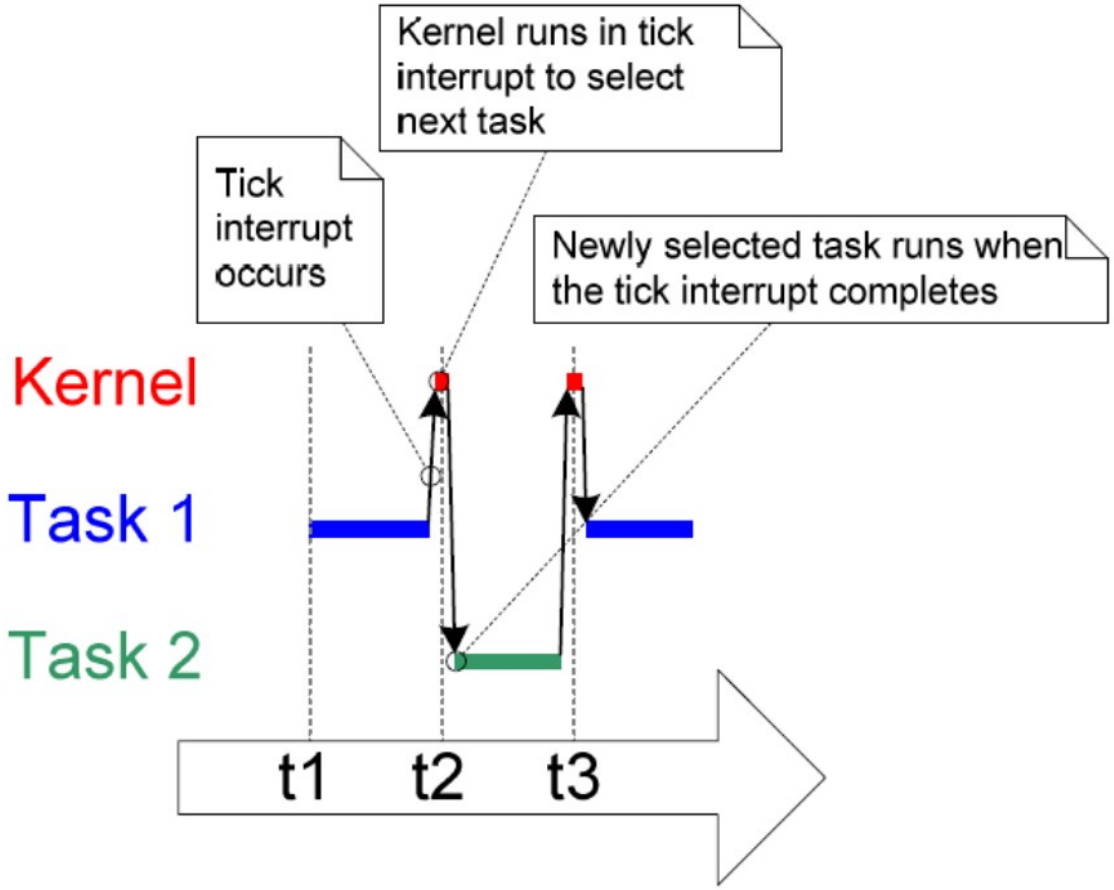
```
/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
/* Create one of the two tasks. */
    xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);

/* Create the second task from the SAME task implementation (vTaskFunction). Only the
value passed in the parameter is different. */
    xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,1,NULL);
/* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    for( ;; );
}
```

Task running with the same priority



Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Task running with the same priority



To select the next task to run, the scheduler itself must execute at each periodic interrupt, called ‘tick interrupt’.

Tick interrupt frequency, is configured by the application-defined configTICK_RATE_HZ contant (copilation time) within FreeRTOSConfig.h.

100Hz typical value

Time slice= 10ms

Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

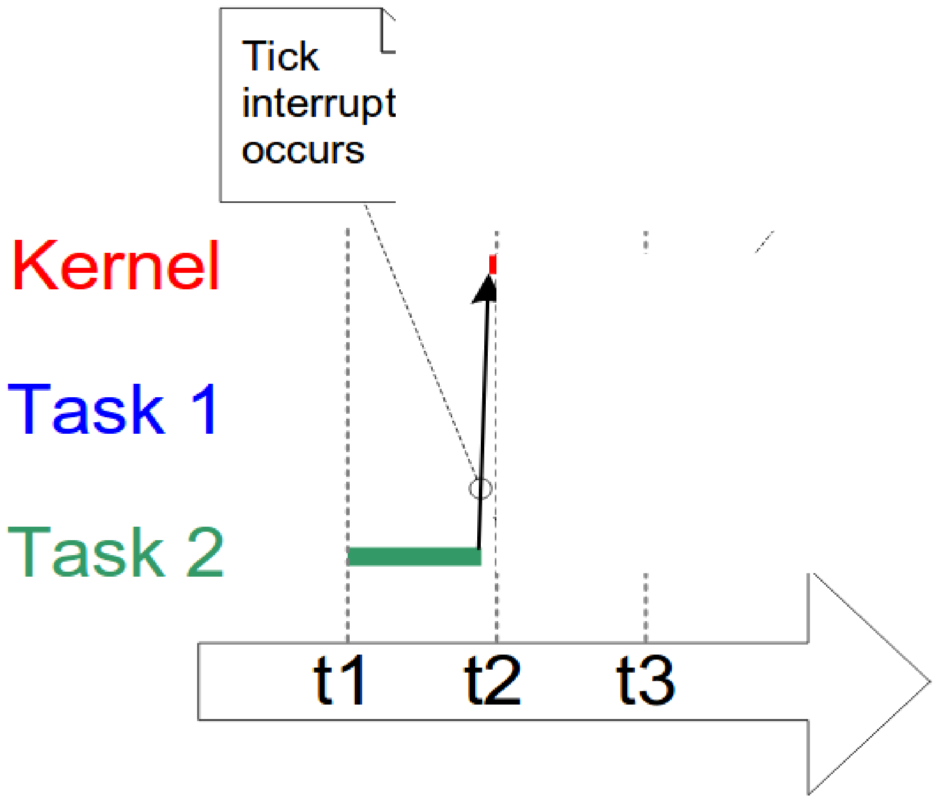
Task running with different priorities

```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    volatile uint32_t ul;
/*The string to print out is passed in via the parameter.*/
    pcTaskName = ( char * ) pvParameters;
/* As per most tasks, this task is implemented in an infiniteloop. */
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
        for( ul = 0; ul < mainDELAY_LOOP_COUNT; ul++ )/*Delay for a period. */
            {
        }
    }
}
```

```
/* main function */
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,"Task 1",1000,(void*)pcTextForTask1,1,NULL);

/* Create the second task with higher priority*/
    xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,2,NULL);
/* Start the scheduler so the tasks start executing. */
    vTaskStartScheduler();
    for( ;; );
}
```

Task running with different priorities

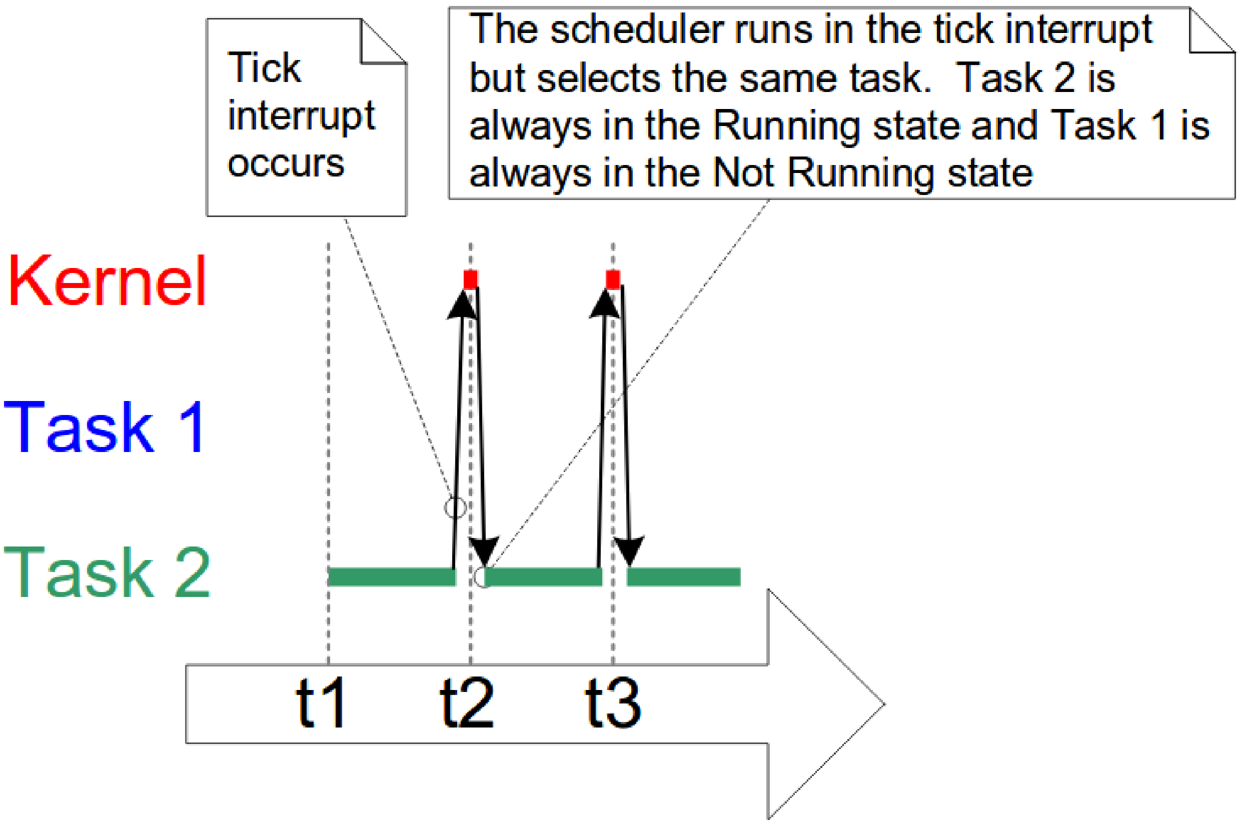
Task 2 with higher priority than task 1

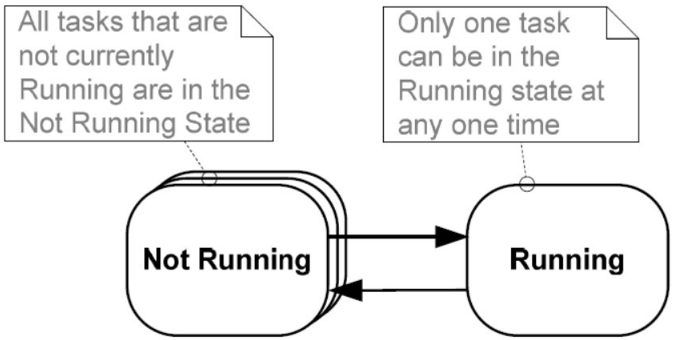


Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Task running with different priorities

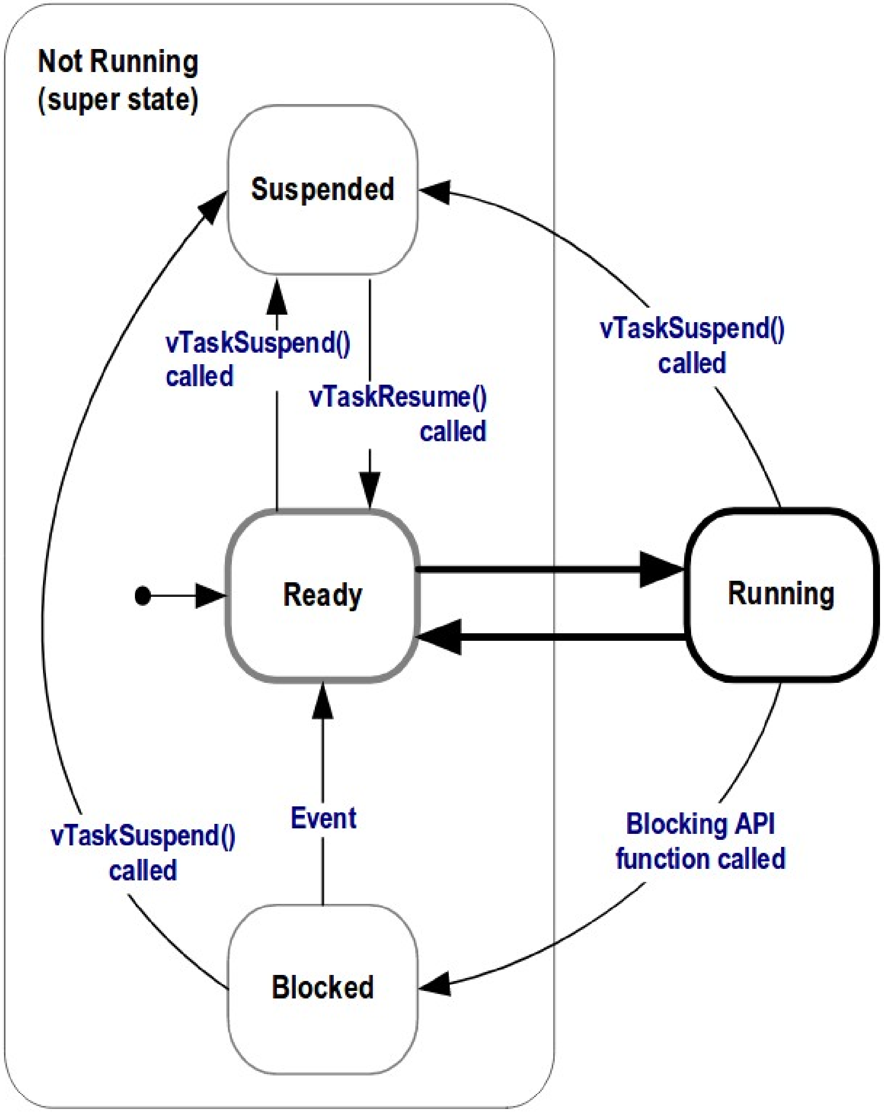
Task 2 with higher priority than task 1





Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the 'Not Running' State

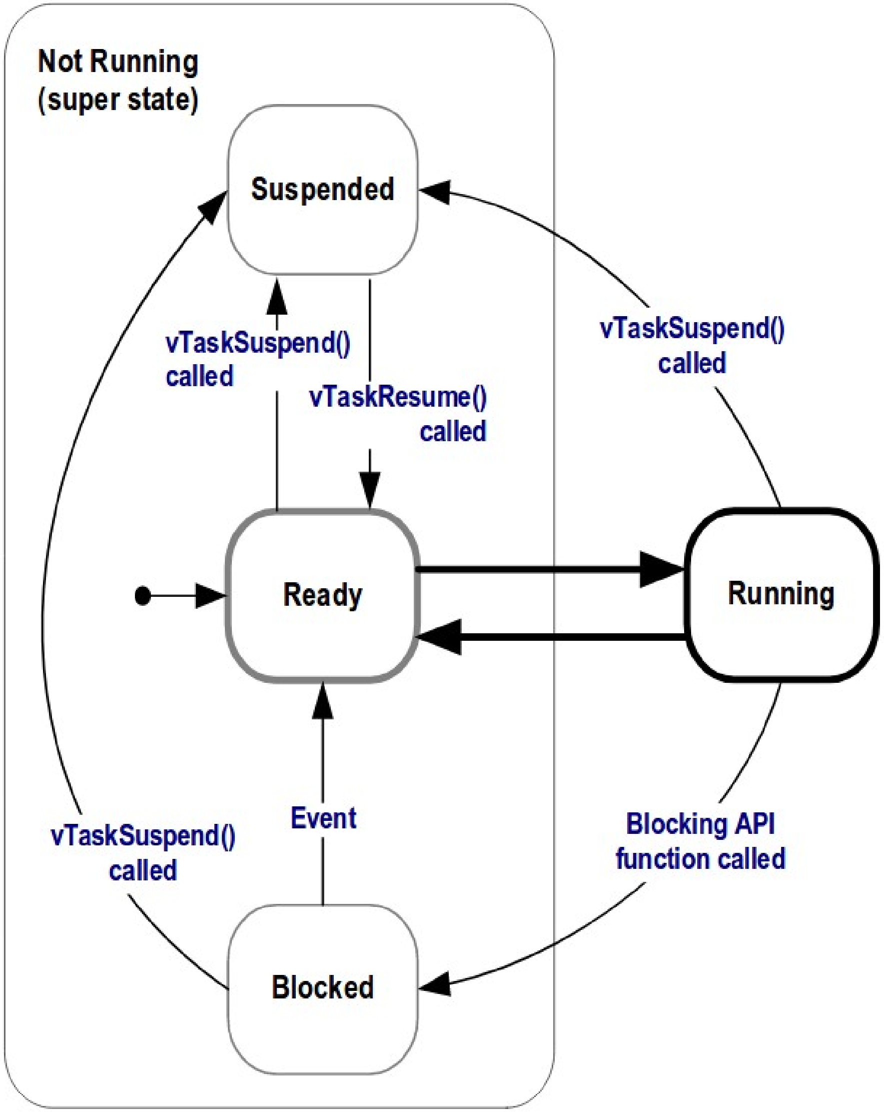


To make the tasks useful they must be re-written to be **event-driven**.

A task is triggered when an event occurs, and is not able to enter the Running state before that event has occurred.

Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the 'Not Running' State



When a task is waiting for an event is Blocked

To types of events

Temporal - Delays

Synchronization – Waiting for data in a queue

Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the ‘Not Running’ State

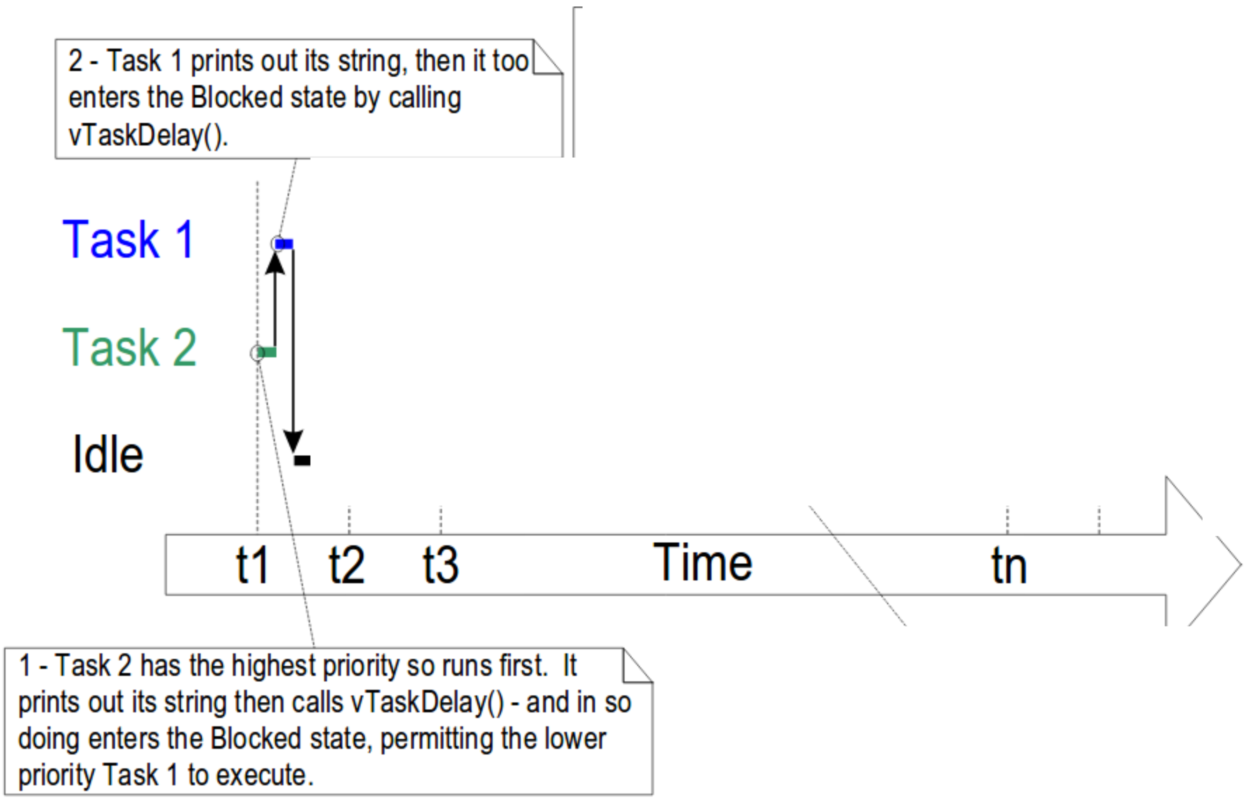
```
void vTaskFunction( void *pvParameters )
{
    char *pcTaskName;
    const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
    volatile uint32_t ul;
/*The string to print out is passed in via the parameter.*/
    pcTaskName = ( char * ) pvParameters;
/* As per most tasks, this task is implemented in an infiniteloop. */
    For( ;; )
    {
        vPrintString( pcTaskName );/* Print out the name of this task. */
        vTaskDelay(xDelay250ms);
    }
}
```

vTaskDelay() places the task into the Blocked state until the delay period has expired.

void vTaskDelay(portTickType xTicksToDelay);

Expanding the ‘Not Running’ State

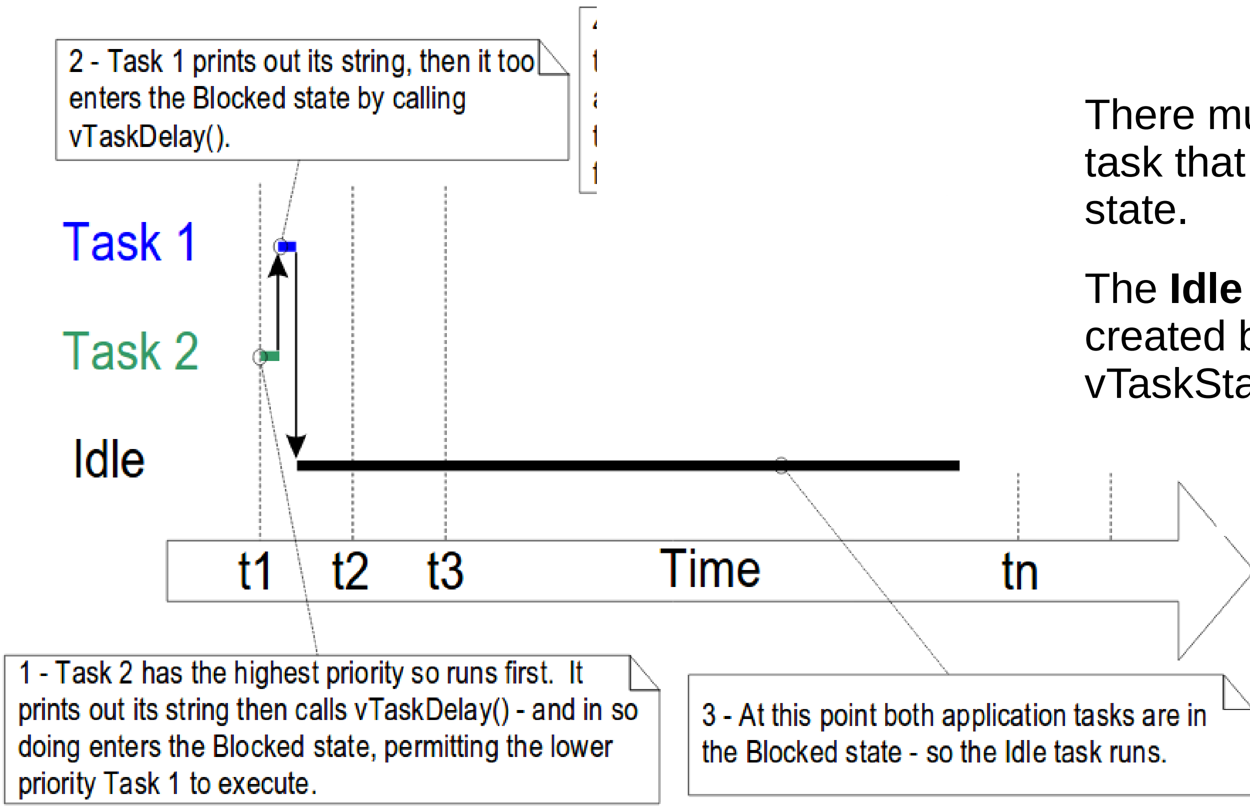
```
/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
     xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
     xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,2,NULL);
     vTaskStartScheduler();
     for( ;; );
}
```



Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the ‘Not Running’ State

```
/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
     xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
     xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,2,NULL);
     vTaskStartScheduler();
     for( ;; );
}
```



There must always be at least one task that can enter the Running state.

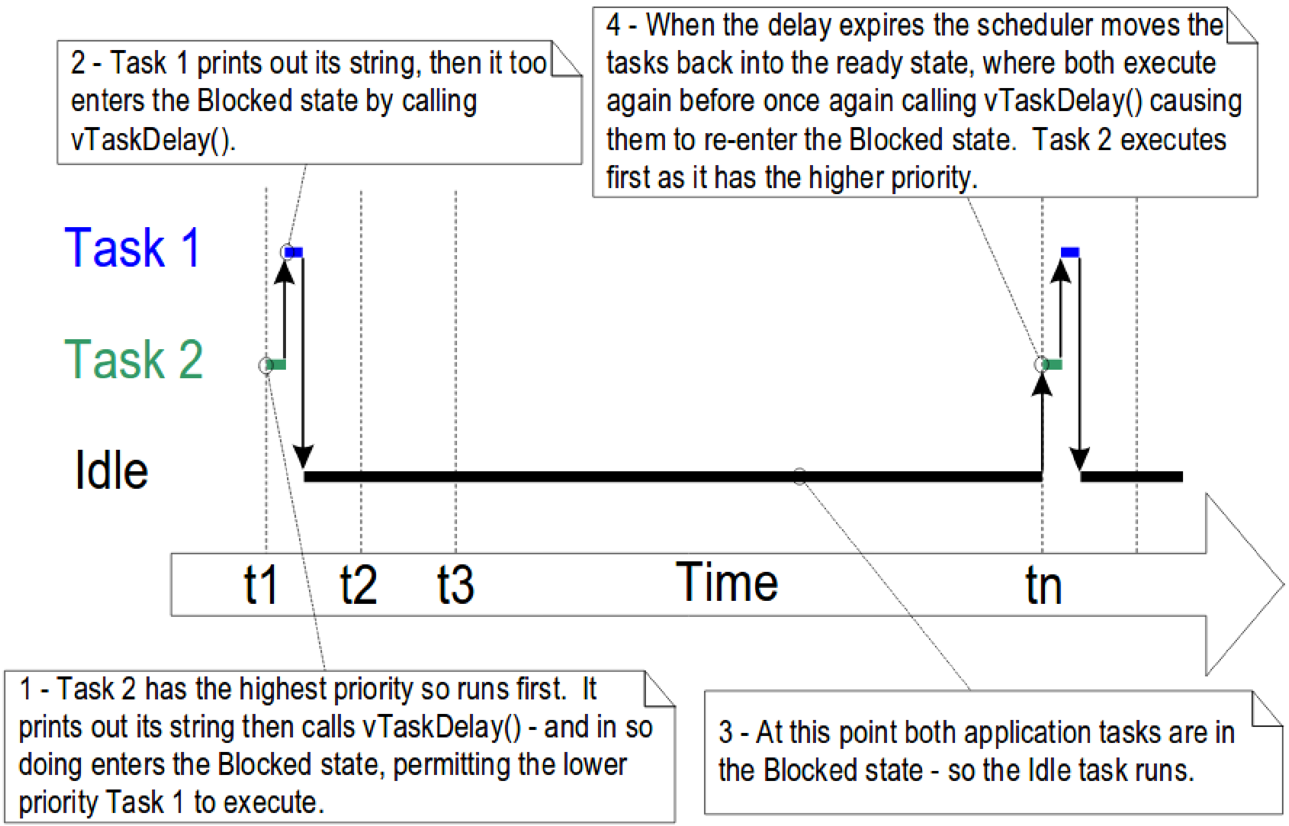
The **Idle task** is automatically created by the scheduler when vTaskStartScheduler() is called.

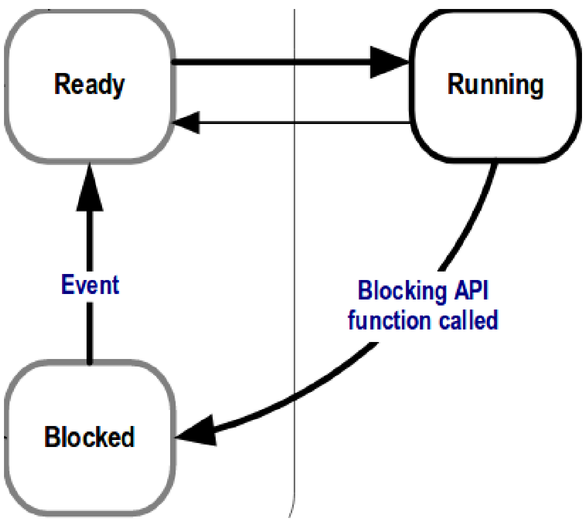
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the ‘Not Running’ State

More efficient implementation of tasks

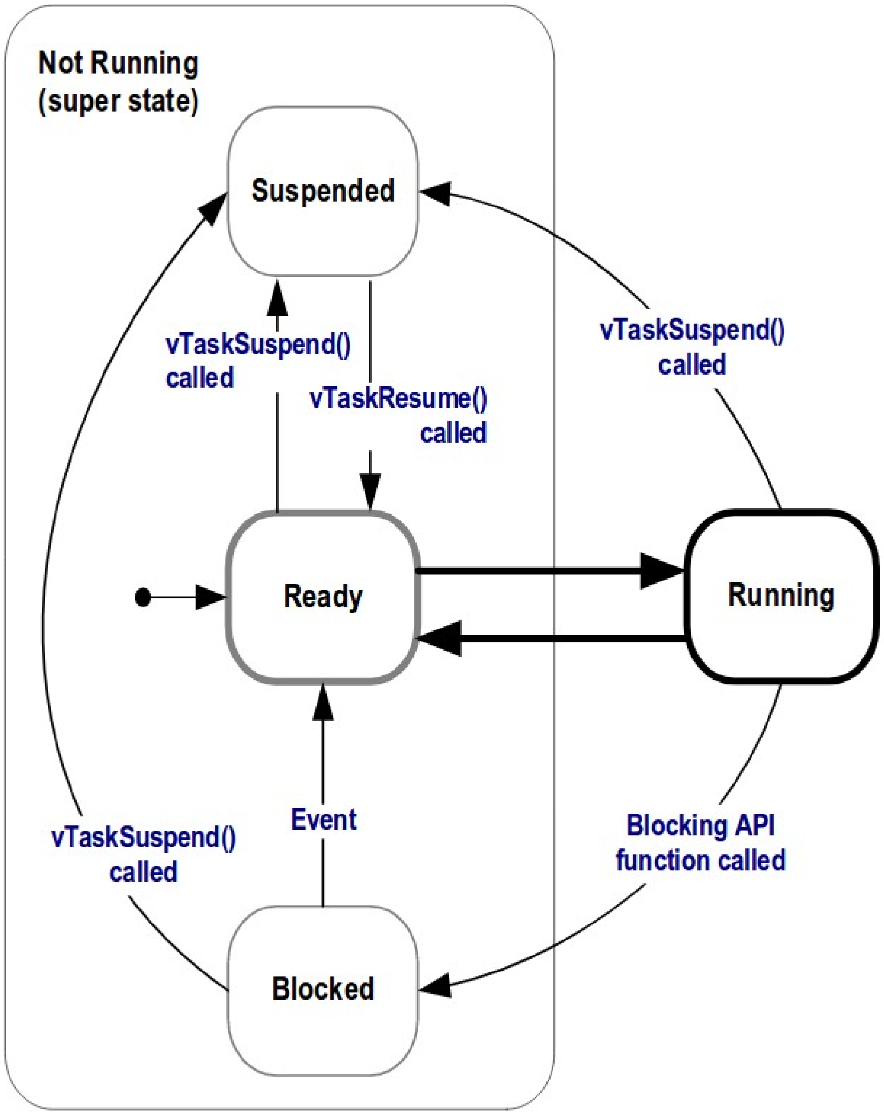
Less use of processor time





Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the 'Not Running' State



The Suspended State is also a sub-state of Not Running.

Tasks in the Suspended state are not available to the scheduler.

vTaskSuspend()API

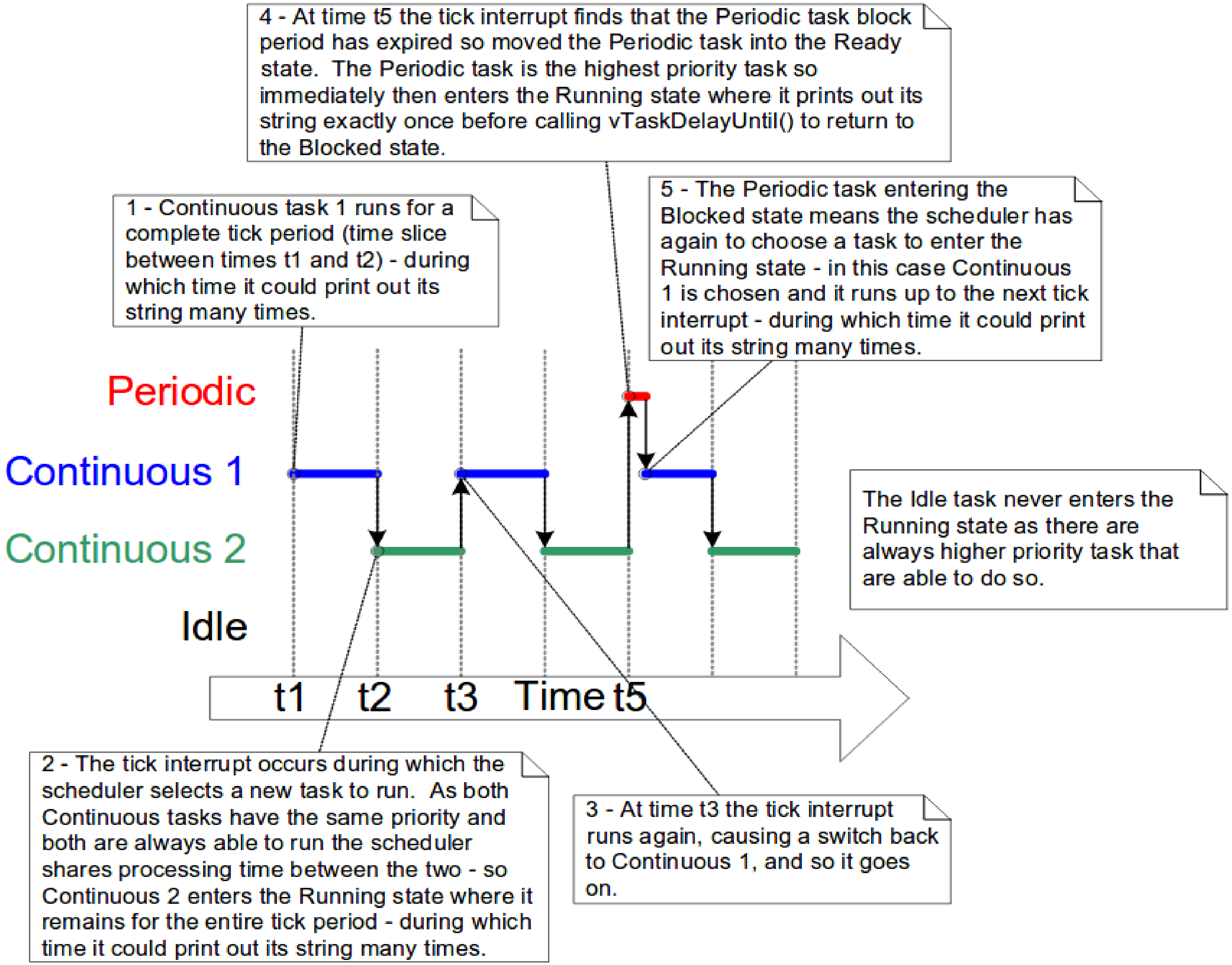
vTaskResume() or xTaskResumeFromISR() API functions.

Most applications do not use the Suspended state.

Combining blocking and non-blocking tasks

```
void vContinuousFunction( void *pvParameters )
{
     char *pcTaskName;
     volatile uint32_t ul;
     pcTaskName = ( char * ) pvParameters;
     For( ;; )
     {
          vPrintString( pcTaskName );/* Print out the name of this task. */
     }
}

void vPeriodicFunction( void *pvParameters ){
     char *pcTaskName;
     TickType_t xLastWakeTime;
     pcTaskName = ( char * ) pvParameters;
     xLastWakeTime = xTaskGetTickCount();/* current tickcount.*/
     for( ;; ){/* Print out the name of this task. */
          vPrintString( pcTaskName );
          vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ));
     }
}

/* main function */
Static const char *pcTextForTask1 ="Continuous task 1 running\r\n";
static const char *pcTextForTask2 ="Continuous task 2 running\r\n";
static const char *pcTextforperiodic ="Periodic task is running\r\n";
int main(void)
{
     xTaskCreate(vContinuousFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
     xTaskCreate(vContinuousFunction,"Task 2",1000,(void*)pcTextForTask2,1,NULL);
     xTaskCreate(vPeriodicFunction,”Task periodic”,1000,(void*)pcTextforperiodic,2,NULL);
     vTaskStartScheduler();
     for( ;; );
}
```

Combining blocking and non-blocking tasks



Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Other task related functions

- void **vTaskPrioritySet**(TaskHandle_t pxTask, UbaseType_t uxNewPriority);
  - pxTask: The handle of the task (last parameter of taskCreate function)
  - uxNewPriority: New priority to be set

- UbaseType_t **uxTaskPriorityGet**(TaskHandle_t pxTask);

- void **vTaskDelete**(TaskHandle_t pxTaskToDelete);
  - pxTaskToDelete: The handle of the task

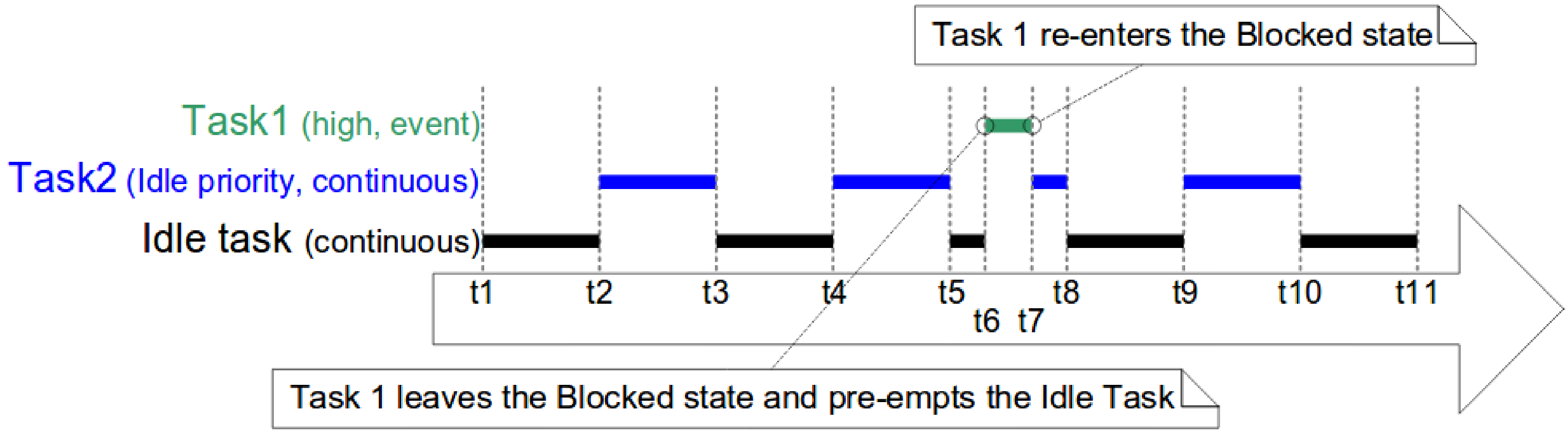
Scheduling Algorithms

- Round Robin Scheduling
- Fixed Priority Pre-emptive Scheduling with Time Slicing
  - Fixed priority: Do not change prorities assigned to tasks
  - Pre-emptive: Pre-empt immediately the running task if a task of hgher priority enters to Ready state
  - Time slicing: is used to share processing time between tasks of equal priority - Time between two RTO`S tick interrupts

Scheduling Algorithms

Configured in **FreeRTOSConfig.h**

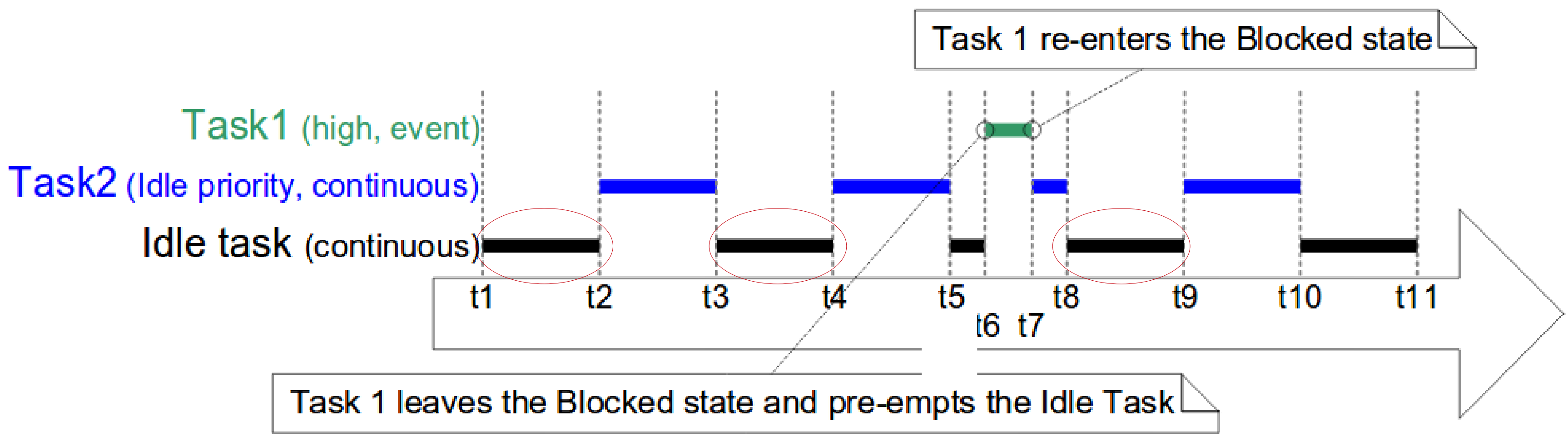
- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 1



Scheduling Algorithms

Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 1

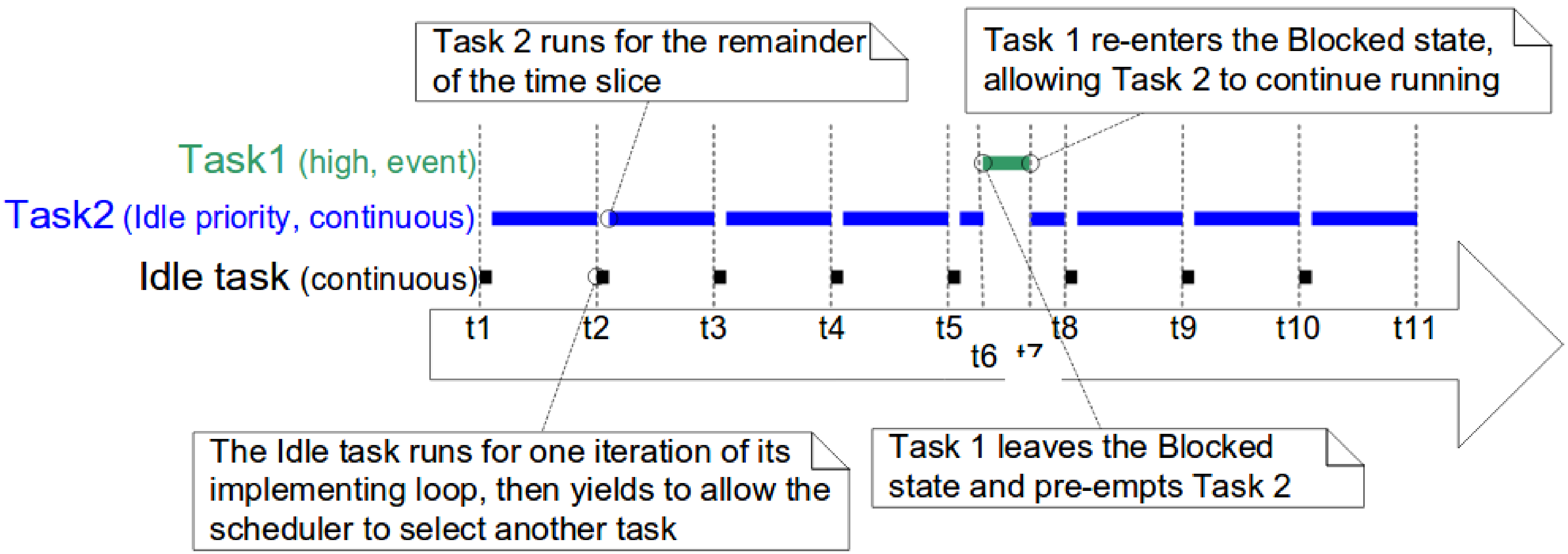


To much time for idle task

Scheduling Algorithms

Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 1
- **configIDLE_SHOULD_YIELD 1**



Scheduling Algorithms

- Round Robin Scheduling
- Fixed Priority Pre-emptive Scheduling with Time Slicing
- Fixed Priority Pre-emptive Scheduling **without Time Slicing**

Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 0

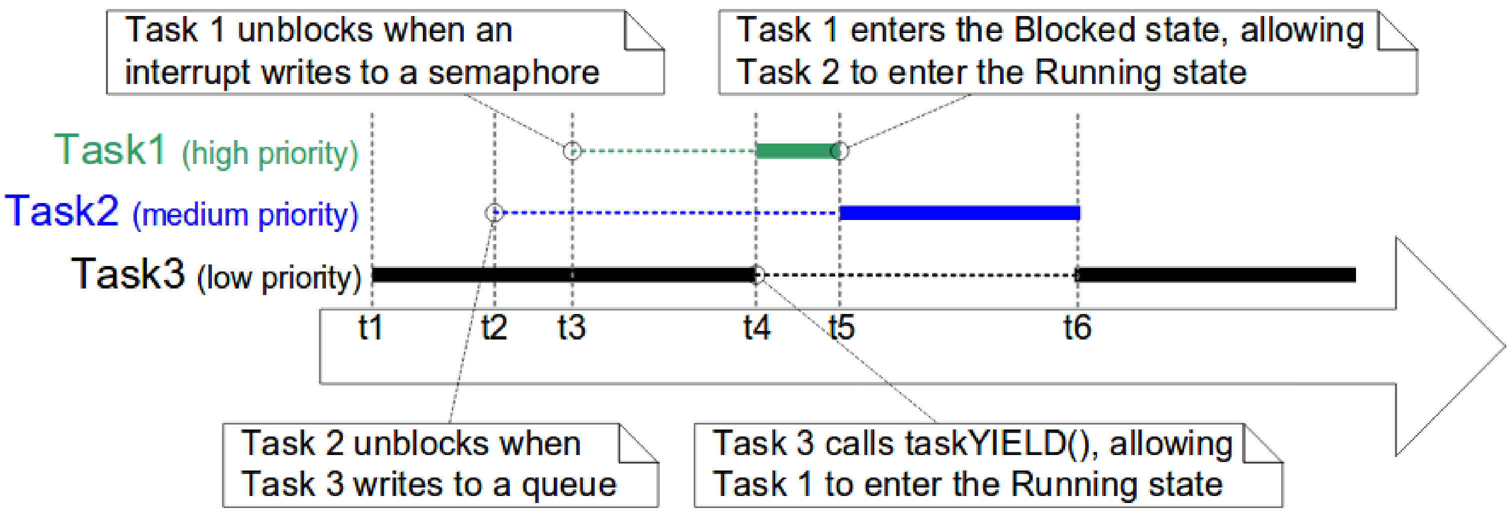
Scheduling Algorithms

- Round Robin Scheduling
- Fixed Priority Pre-emptive Scheduling with Time Slicing
- Fixed Priority Pre-emptive Scheduling without Time Slicing
- **Co-operative Scheduling**

Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION 0
- configUSE_TIME_SLICING any

Running state call **taskYIELD()** function to re-schedule



Queue Management

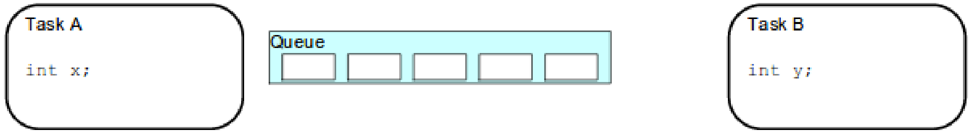
‘Queues’ provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism.

- Queues hold a finite number of fixed size data items
- Queues are normally used as First In First Out (FIFO) buffers

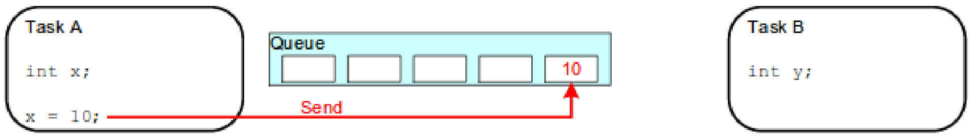
FreeRTOS use **queue by copy method.**

- Stack variable can be sent directly to a queue.
- Data can be sent to a queue without first allocating a buffer.
- The sending task and the receiving task are completely de-coupled.
- The RTOS takes complete responsibility for allocating the memory used to store data.

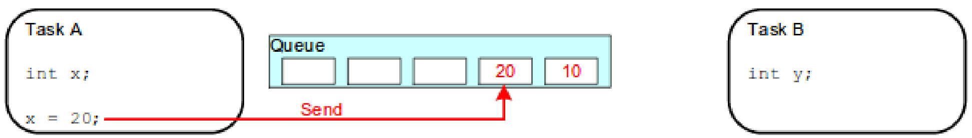
But still you can use queue by reference



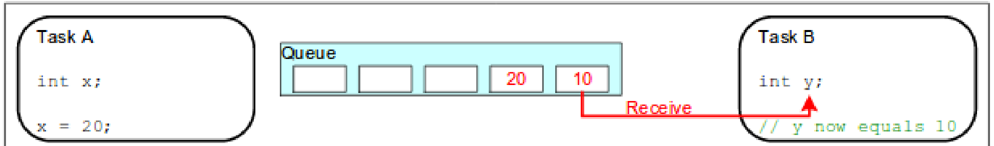
A queue is created to allow Task A and Task B to communicate. The queue can hold a maximum of 5 integers. When the queue is created it does not contain any values so is empty.



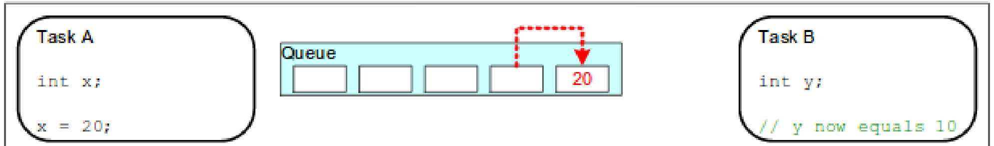
Task A writes (sends) the value of a local variable to the back of the queue. As the queue was previously empty the value written is now the only item in the queue, and is therefore both the value at the back of the queue and the value at the front of the queue.



Task A changes the value of its local variable before writing it to the queue again. The queue now contains copies of both values written to the queue. The first value written remains at the front of the queue, the new value is inserted at the end of the queue. The queue has three empty spaces remaining.



Task B reads (receives) from the queue into a different variable. The value received by Task B is the value from the head of the queue, which is the first value Task A wrote to the queue (10 in this illustration).



Task B has removed one item, leaving only the second value written by Task A remaining in the queue. This is the value Task B would receive next if it read from the queue again. The queue now has four empty spaces remaining.

Queue Management

Creating a queue
A queue must be explicitly created before it can be used.

QueueHandle_t **xQueueCreate**(UBaseType_t uxQueueLength, UbaseType_t 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.

Queue Management

Using a queue (writing)

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

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

≡ xQueueSend()

- xQueue: The handle of the queue
- pvItemToQueue: A pointer to the data to 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

Return:

- pdPASS – OK
- errQUEUE_FULL – Error, queue full

Queue Management

Using a queue (reading)

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

- xQueue: The handle of the queue
- pBuffer: A pointer to the memory into which the data will be copied
- xTicksToWait: The maximum amount of time the task should remain in the Blocked state to wait for a data to become available on the queue

Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue full

Queue Management

Receiving from multiples queues

FreeRTOS allows you to create a set of queues to manage different events from different sources to block a task (waiting) on read operation from multiple queues

The corresponding API is

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

The set of queue can be formed by several queues of different sizes and with different size of elements, bynary semaphores or counting semaphores.

The length of the queue is calculated from the sum of each queue size.

The constant configUSE_QUEUE_SETS in FreeRTOSConfig.h must be 1 to allow queue set creation

Queue Management

Adding elements to a set of queues

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

In main()
First create the set, then create the queues to be added and finally add queues to the set.

Removing elements from a set of queues

BaseType_t **xQueueRemoveFromSet**(QueueSetMemberHandle_t xQueueOrSemaphore,
QueueSetHandle_t xQueueSet);

Queue Management

Selecting elements from a set of queues

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

Keep the corresponding task blocked (waiting) until time defined is finished or until an event arrives to the queueset

Queue Management (example)

```
/* 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 queues */
#define ITEM_SIZE_QUEUE_1 sizeof( uint32_t )
#define ITEM_SIZE_QUEUE_2 sizeof( uint8_t )
/* define the total length of 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;
     uint8_t xReceivedFromQueue2;
/* Create a queue set large enough to hold an event for every added queue*/
     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. */
     xQueueAddToSet( xQueue1, xQueueSet );
     xQueueAddToSet( xQueue2, xQueueSet );
     xQueueAddToSet( xSemaphore, xQueueSet );
/* CONTINUED ON NEXT PAGE */
```

Queue Management (example)

```
/* CONTINUED FROM PREVIOUS PAGE */
    for( ;; )    {
/* Block to wait for something to be available from the queues or semaphore
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? */
        if( xActivatedMember == xQueue1 ){
            xQueueReceive( xActivatedMember, &xReceivedFromQueue1, 0 );
/*Receives/takes can use a block time of zero as they are guaranteed to pass
because xQueueSelectFromSet() only returned the handle when something was
available. */
            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 event*/
        }
    }
}
```

Queue Management

Other queue management functions

```
void vQueueDelete( TaskHandle_t    pxQueueToDelete );

QueueHandle_t xQueueCreateStatic(  UBaseType_t uxQueueLength,
                                   UBaseType_t uxItemSize,
                                   uint8_t *pucQueueStorageBuffer,
                                   StaticQueue_t *pxQueueBuffer );

UBaseType_t uxQueueMessagesWaiting( const QueueHandle_t xQueue );

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

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

BaseType_t   xQueueReset( QueueHandle_t xQueue );
```

Heap Memory Management

The dynamic memory allocation is used by FreeRTOS each time a kernel object is created, and the memory is released when the kernel object is deleted.

The FreeRTOS equivalent functions for malloc() and free() are

PvPortMalloc() and **vPortFree()**

Heap Memory Management

Heap allocation schemes indicated in files
heap_1.c, heap_2.c, heap_3.c, heap_4.c and heap_5

Heap_1

The heap_1 allocation scheme subdivides a simple array into smaller blocks, as calls to pvPortMalloc() are made. The array is called the FreeRTOS heap.

Small embedded dedicated systems that create task before the scheduler run. In this case the memory is dynamically allocated by the kernel before the application starts, and remain allocated until application finish.

The total memory array size is defined in FreeRTOSConfig.h file with

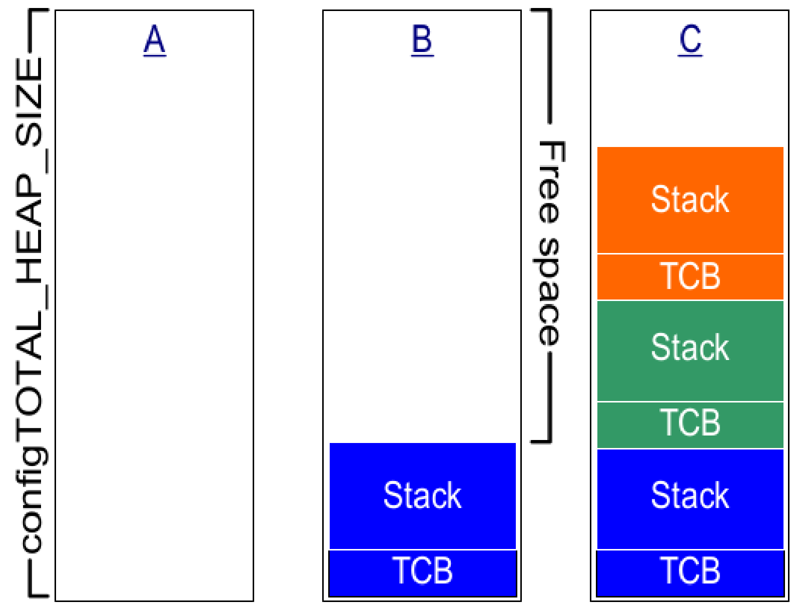
configTOTAL_HEAP_SIZE

Heap Memory Management

Heap allocation schemes

Heap_1

TCB: Task control block



Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

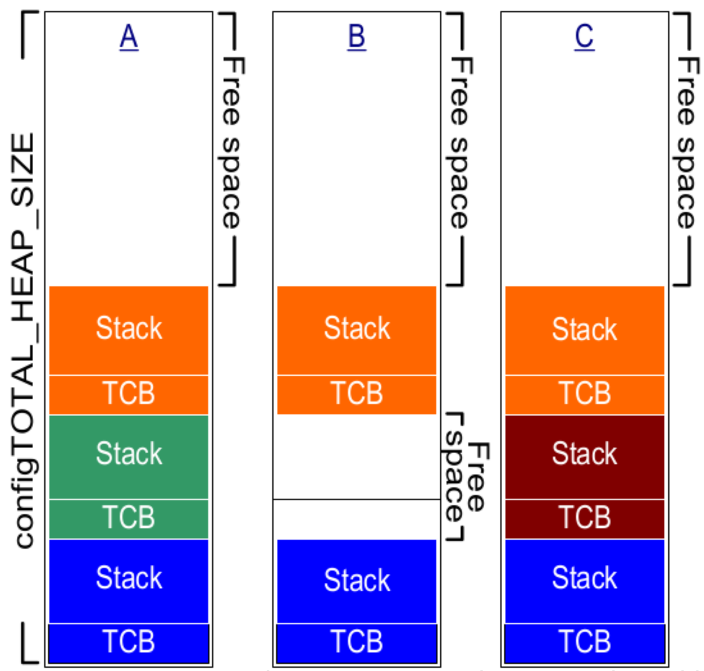
Heap Memory Management

Heap allocation schemes

Heap_2 Not recommended for new designs – Instead use Heap_4 scheme

It works dividing the memory heap array using the best fit algorithm to allocate memory when pvPortMalloc() function is called. The best fit algorithm ensure that the pvPortMalloc() will use the memory space that best fit with the requirements of the task.

Suitable for application y.



Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Heap Memory Management

Heap allocation schemes

Heap_3: configTOTAL_HEAP_SIZE has no effect in this mode.

It uses malloc() and free() functions to manage the memory and the size of the heap is defined by the linker.

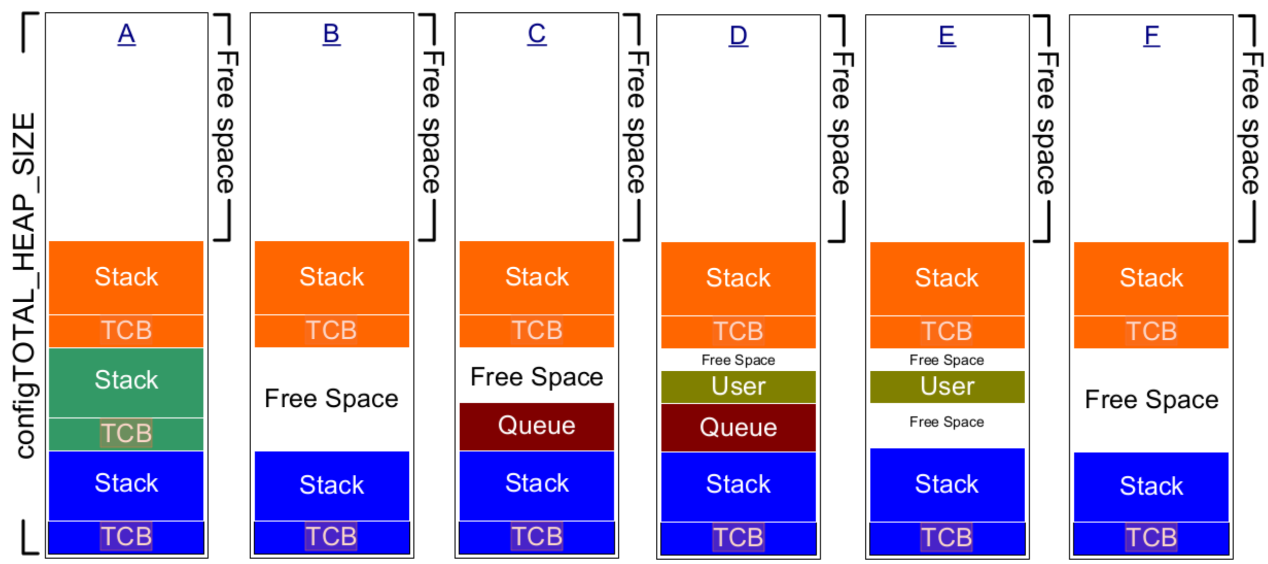
Heap Memory Management

Heap allocation schemes

Heap_4: Divides the array in smaller blocks, the total size is configured with configTOTAL_HEAP_SIZE.

Use best fit algorithm to allocate memory.

It combine adyacent blocks to reduce fragmentation.



Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Heap Memory Management

Heap allocation schemes

Heap_5: Can allocate memory from multiple and separated memory spaces.

Use best fit algorithm to allocate memory.

It combine adyacent blocks to reduce fragmentation.

Must be explicitly initialized before pvPortMalloc() can be called.
Heap_5 is initialized using the vPortDefineHeapRegions() API function.

This function defines the start address and the size of each separate portion of memory.

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

Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Software Timer Management

The Software Timer schedules the execution of a function at a set time in the future, or periodically with a fixed frequency. (Implemented and managed by the kernel).

The function executed by the software timer is called the software timer's callback function.

Software timer callback functions execute from start to finish, and exit in the normal way. They should be kept short, and must not enter the Blocked state.

To use it:
Add **timers.h** to your design
Set **configUSE_TIMERS** to **1** in FreeRTOSConfig.h

Software Timer Management

Timer is created, with a period, Type (oneshot/autoreload), the TimerID, and the name of the callback function).

The **period** is the amount of ticks until the callback function starts.
Onseshot: timer is executed only one tyme and the callback function start when period is finished and it is executed only once.
Autoreload: the timer restart after period is finished. The callback function is executed every time that period is finished.

```
TimerHandle_t   xTimerCreate(    const char * const pcTimerName,
                                     TickType_t xTimerPeriodInTicks,
                                     UBaseType_t uxAutoReload,
                                     void * pvTimerID,
                                     TimerCallbackFunction_t pxCallbackFunction );
```

The format to create Software Timer Callback Function is

```
void ATimerCallback( TimerHandle_t    xTimer );
```

Software Timer Management

The Software timer callback function is executed when the scheduler start, then a timer service from the kernel is executed (daemon).

Daemon priority and stack size are set by the **configTIMER_TASK_PRIORITY** and **configTIMER_TASK_STACK_DEPTH** constants in FreeRTOSConfig.h file.

The Software Timing Callback Function send commands (start, stop, reset) to the kernel daemon task through a timer command queue.

The length of the timer command queue is set using **configTIMER_QUEUE_LENGTH**

Software Timer Management

Two states Dormant and Running

Once the Timer has been created remains in the Dormant state until **xTimerStart()** or **xTimerReset()** function is called.

Once running, timer stop when **xTimerStop()** function is called or when Timer expire (Oneshot mode)

TimerHandle_t **xTimerStart**(TimerHandle_t xTimer, TickType_t xTicksToWait);

Return pdPASS if ok or pdFALSE if the timer command queue is full.

Software Timer Management (Example)

```
#define myTimer_period        pdMS_TO_TICKS (500)  /* Define Period */
int main(void){
     TimerHandle_t myTimer;
     BaseType_t     xmyTimerStarted;
     /*Create myTimer*/
     myTimer xTimerCreate(“mytimer”, /*for debug */
                              myTimer_period,
                              PdFalse, /*oneshot mode*/
                              0,
                              prvmyTymerCallback);
     if (myTymer !=NULL){
          xmyTimerStarted= xTimerStart(myTymer,0);
          if (xmyTimerStarted=pdPASS){
               vtaskStartScheduler();
          }
     }
/* As always, this line should not be reached. */
for( ;; );
}


/* Implementation of the myTimer callback function”*/
static void prvmyTimerCallback(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 );

}
```

Interrupt Management

Events

Embedded real-time systems have to take actions in response to events that originate from the environment.

How should they be detected? Interrupts, polling
What kind of processing needs to be done? Inside ISR, outside ISR

Interrupt priority vs task priority

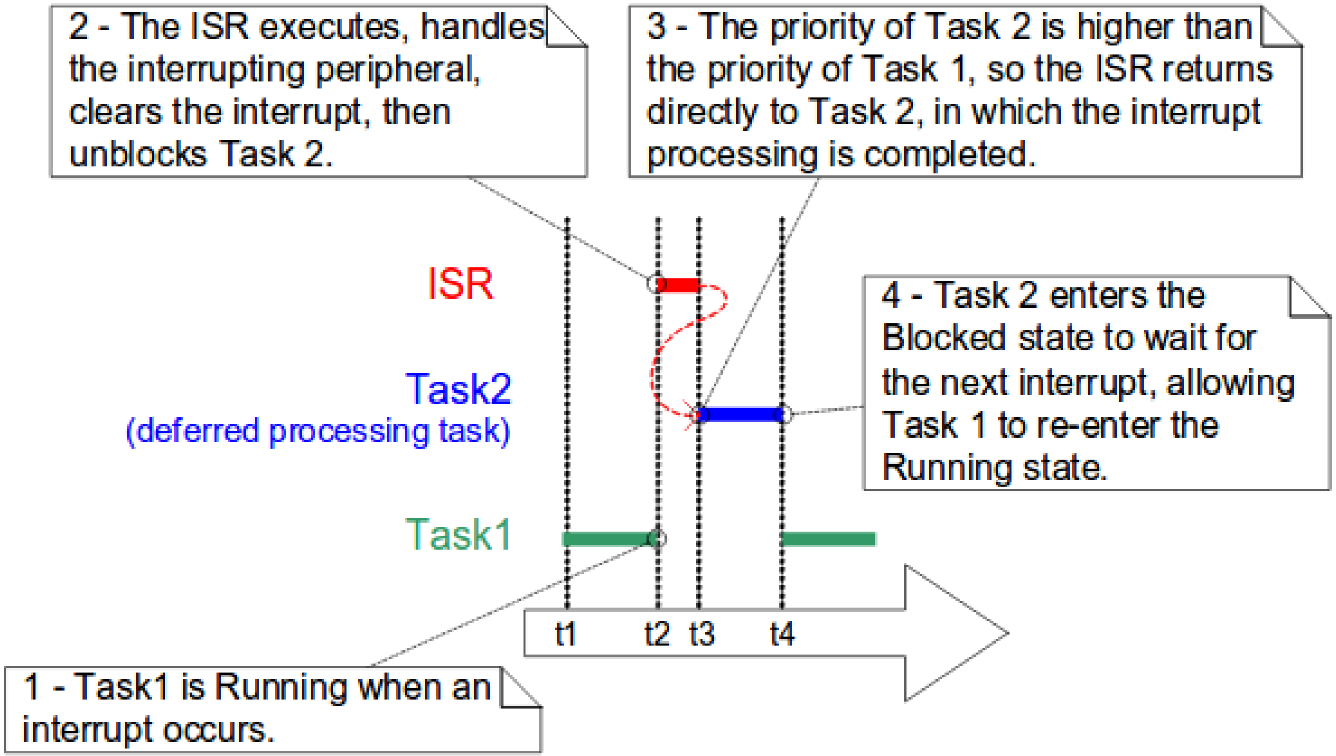
Lowest priority interrupt pre-empt highest priority task

Interrupt Safe APIFunction

FreeRTOS provides two versions of some API functions:
one for use from tasks,
and one for use from ISRs (“FromISR” appended to their name).

Interrupts should be deferred to a task

Interrupt Management



Interrupt Management

Binary Semaphores Used for Synchronization

The deferred processing task can be controlled using a ISR

- The ISR “gives” a semaphore to unblock the deferred task
- The deferred task “takes” the semaphore to enter in the blocked state

Interrupt Management

APIFunctions for managing semaphores

Creating a semaphore

SemaphoreHandle_t **xSemaphoreCreateBinary**(void);

Take

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

Give

BaseType_t **xSemaphoreGiveFromISR**(SemaphoreHandle_t xSemaphore,
BaseType_t *pxHigherPriorityTaskWoken);

Interrupt Management

Using a queue (writing) from an interrupt

```
BaseType_t  xQueueSendToFrontFromISR(QueueHandle_t  xQueue,
                                      void  *pvItemToQueue,
                                      BaseType_t   *pxHigherPriorityTaskWoken );

BaseType_t  xQueueSendToBackFromISR(QueueHandle_t  xQueue,
                                     void  *pvItemToQueue,
                                     BaseType_t   *pxHigherPriorityTaskWoken );
```

- xQueue: The handle of the queue
- pvItemToQueue: A pointer to the data to be copied into the queue
- pxHigherPriorityTaskWoken : a variable to inform the application writer that a context switch should be performed

.

Return:

- pdPASS – OK
- errQUEUE_FULL – Error, queue full

Interrupt Management

Using a queue (reading) from an ISR

BaseType_t **xQueueReceiveFromISR**(QueueHandle_t xQueue,
void *pvBuffer,
BaseType_t *pxHigherPriorityTaskWoken);

- xQueue: The handle of the queue
- pBuffer: A pointer to the memory into which the data will be copied
- PxHigherPriorityTaskWoken: a variable to inform the application writer that a context switch should be performed

.

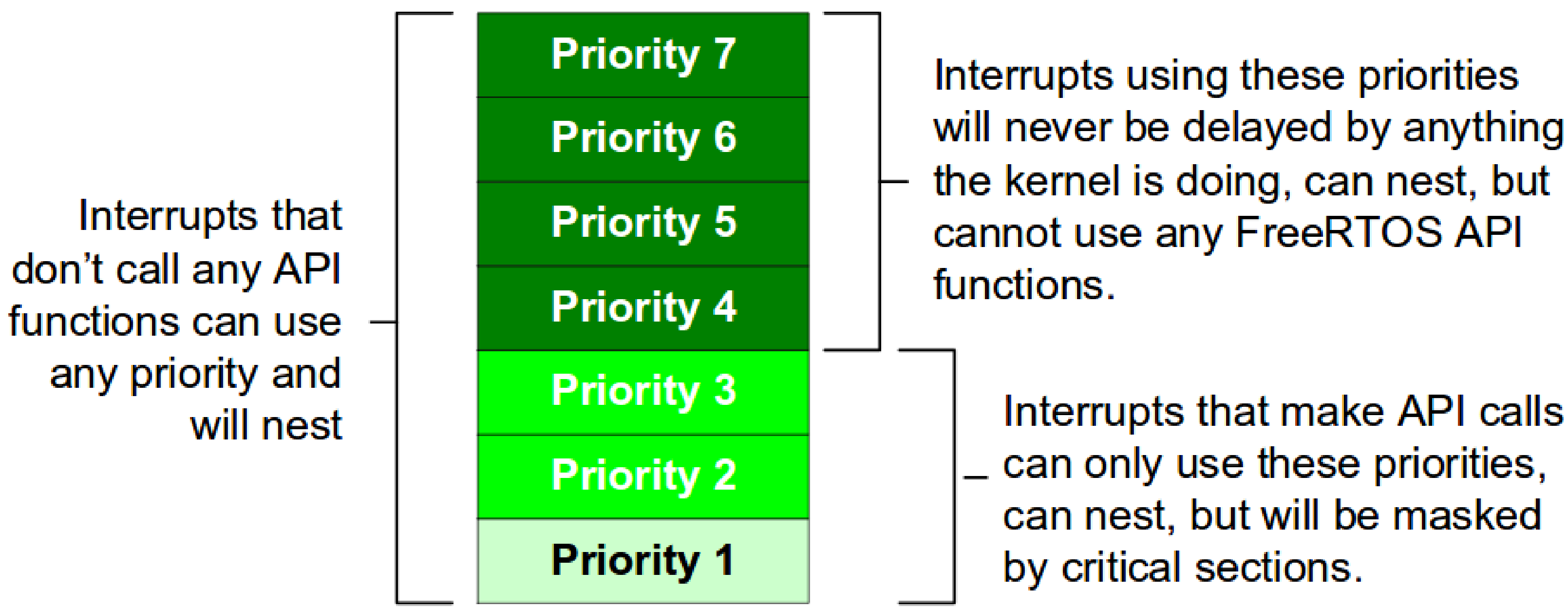
Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue full

Interrupt Management

Nested interrupts

configMAX_SYSCALL_INTERRUPT_PRIORITY = 3
configKERNEL_INTERRUPT_PRIORITY = 1



THANKS FOR YOUR ATTENTION

- QUESTIONS?