Introduction to FreeRTOS

Characteristics of freeRTOS (Operating System)

- FreeRTOS is a “Embedded Operating System” for Embedded MicroController Software that provides multitasking facilities.
- Open Source
- Introduces minimum overhead (1%-4% CPU Time)
- Takes up little memory space (~6KB Flash)

FreeRTOS features:

- Dynamic Task creation
- Priority-based multitasking capability
- Queues to communicate between multiple tasks
- Semaphores – mutex - to manage shared resource between multiple tasks
- Utilities to view CPU utilization, stack utilization etc.

Supported CPUs (Ports):

- http://www.freertos.org/RTOS_ports.html

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

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.

```
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
```

Building FreeRTOS

tasks.c and list.c: FreeRTOS source code common to all the FreeRTOS ports and they are located directly in the FreeRTOS/Source directory

In addition to these two files, the following source files are located in the same directory: **queue.c and timer.c**

queue.c provides both queue and semaphore services. queue.c is nearly always required.

timers.c provides software timer functionality. It need only be included in the build if software timers are actually going to be used

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.

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’.

TASKS

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

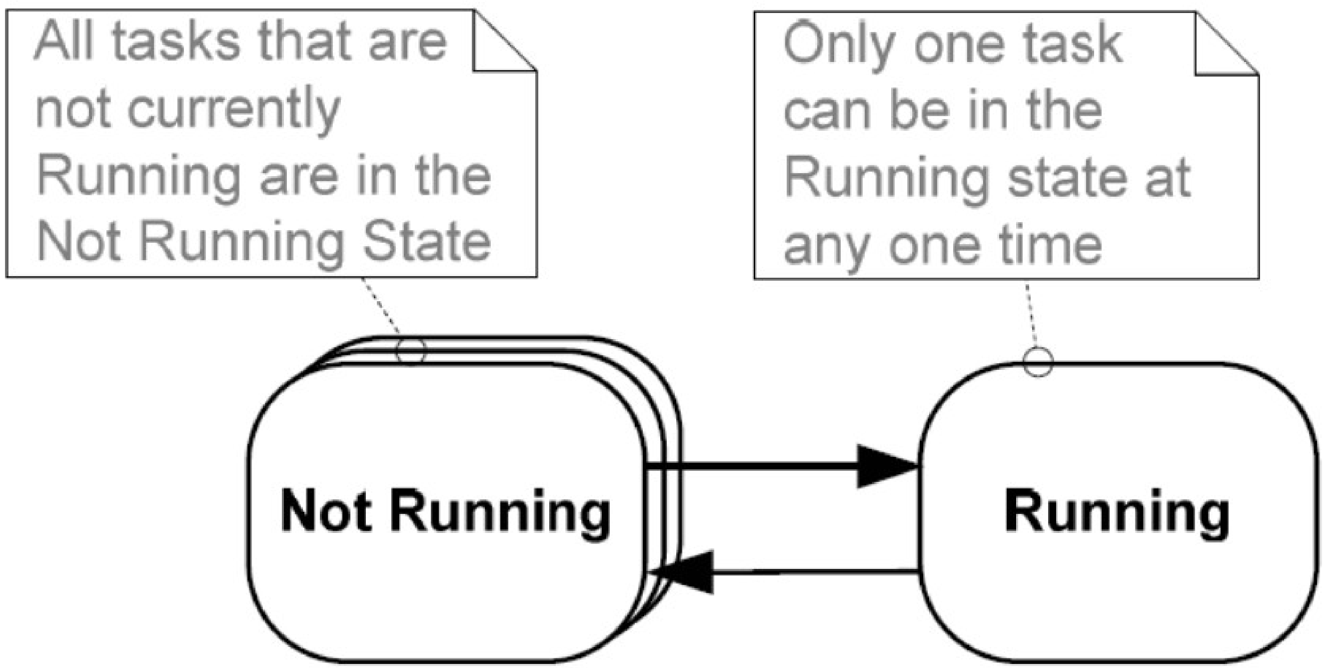
```
void myTask (void *pvParameters){

    /* variables declaration */
    Int iVariableExample = 0;

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

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

Top Level Task States



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

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
}
```

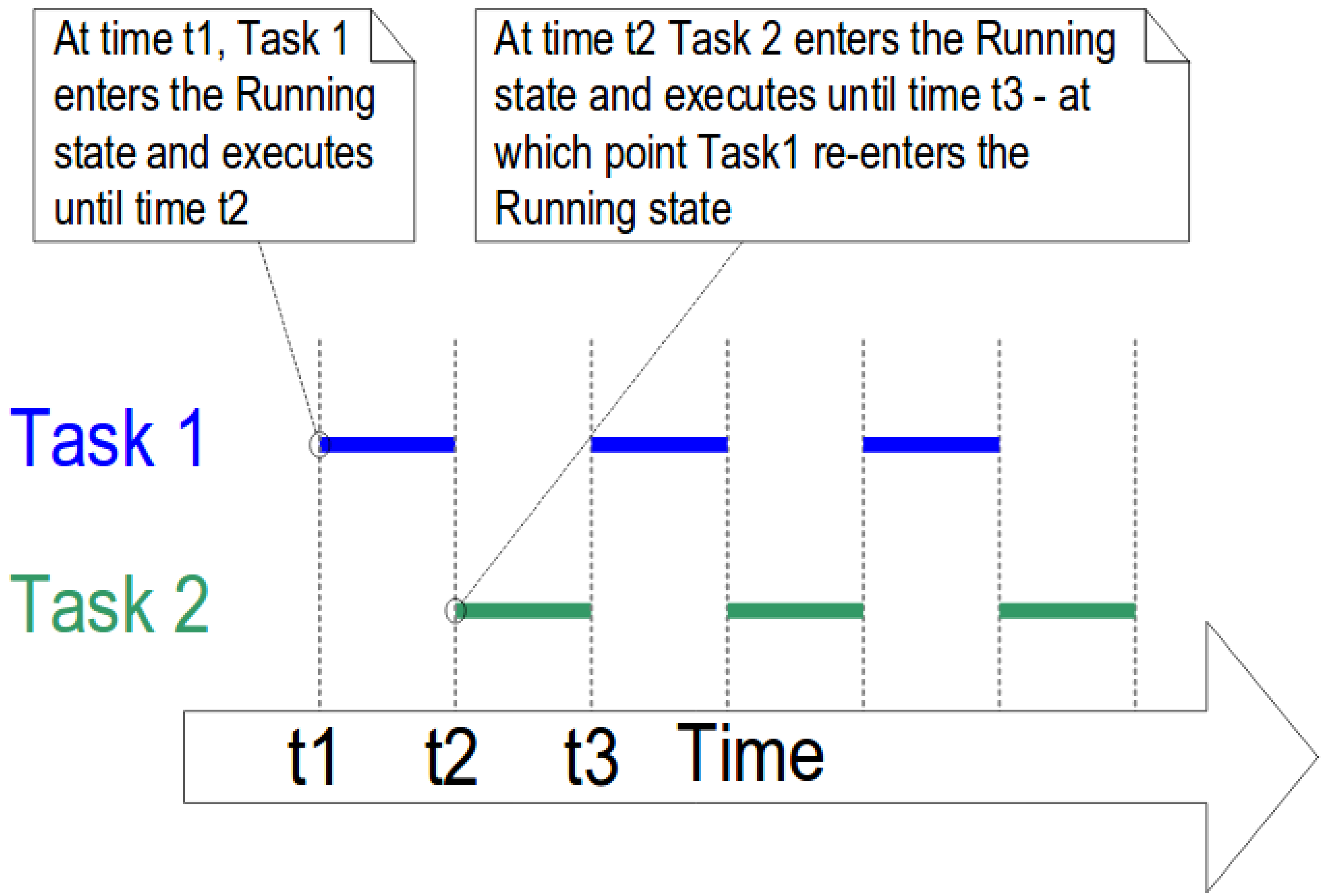
The main function in FreeRTOS based projects creates tasks.
FreeRTOS will let you multi-task based on your tasks and their priority.

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. */
               {
          }
     }
}
```

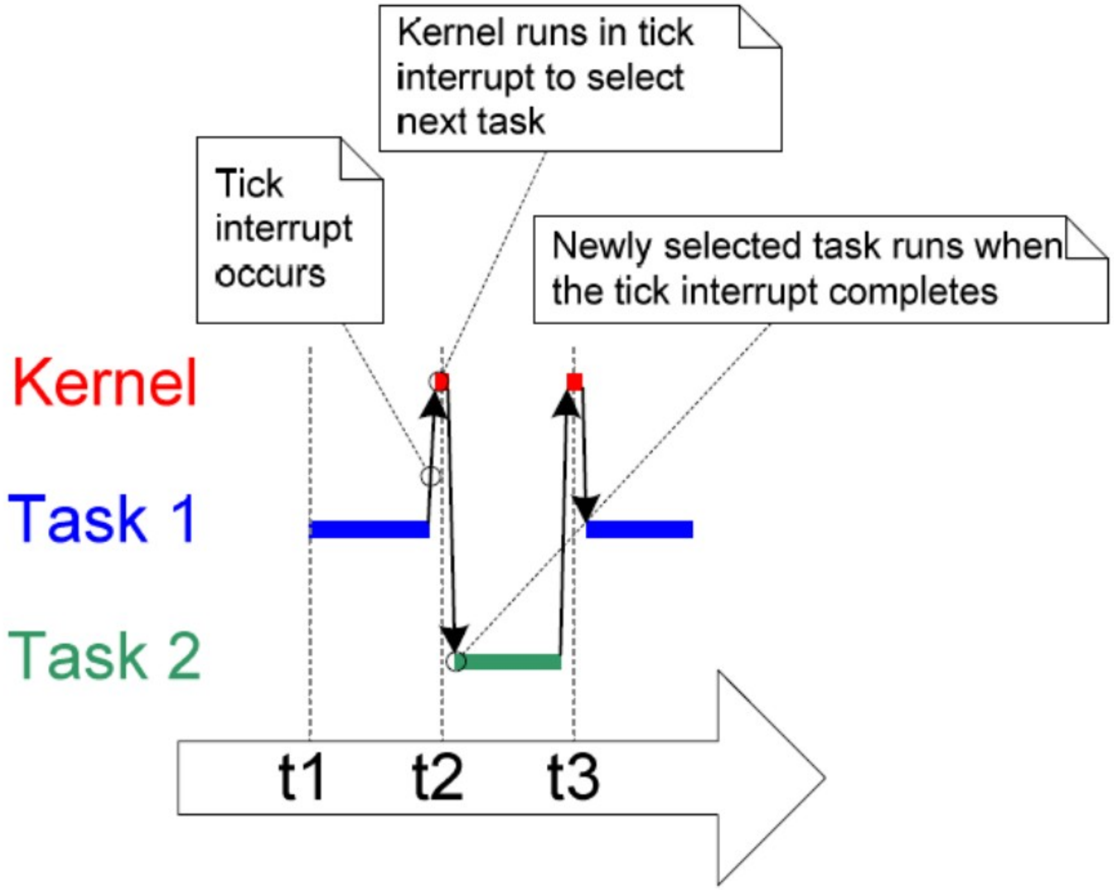
```
/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
/* Create one of the two tasks. */
     xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
/* Create the second task from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

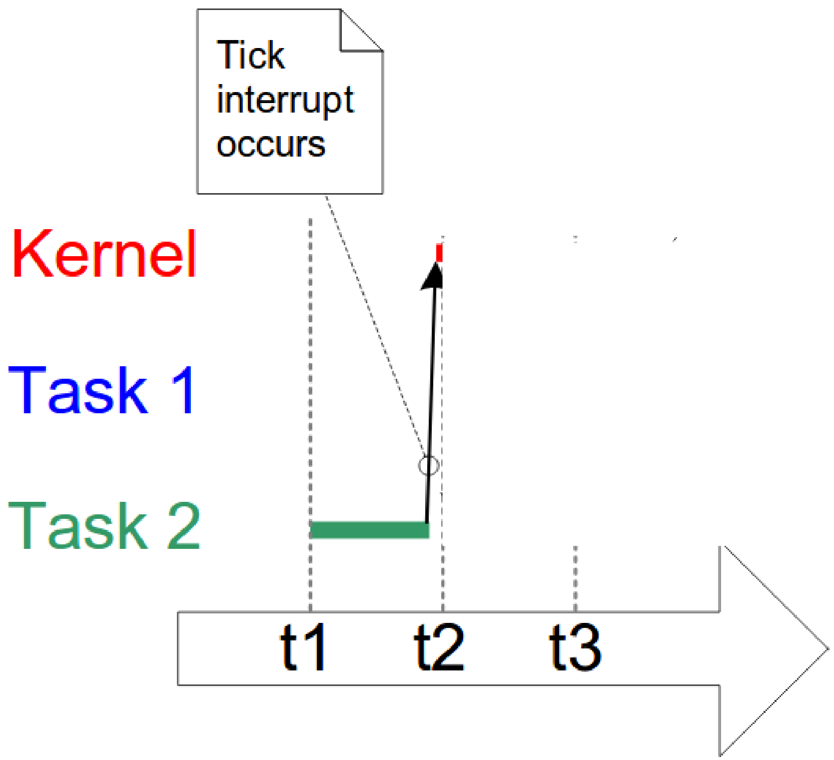
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

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

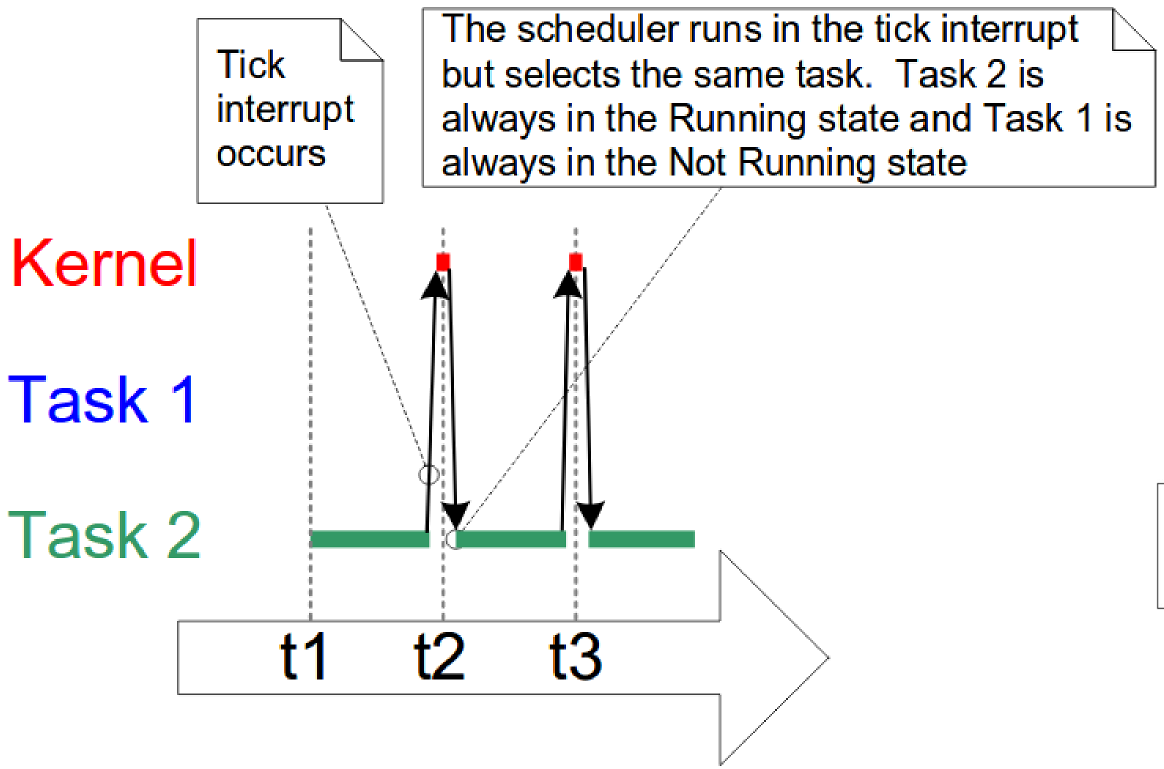


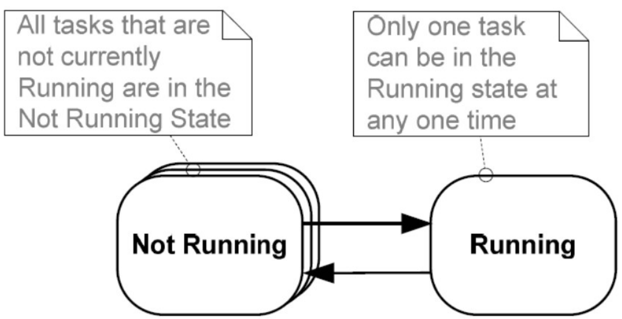
Task 2 with higher priority than task 1

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

Task running with different priorities

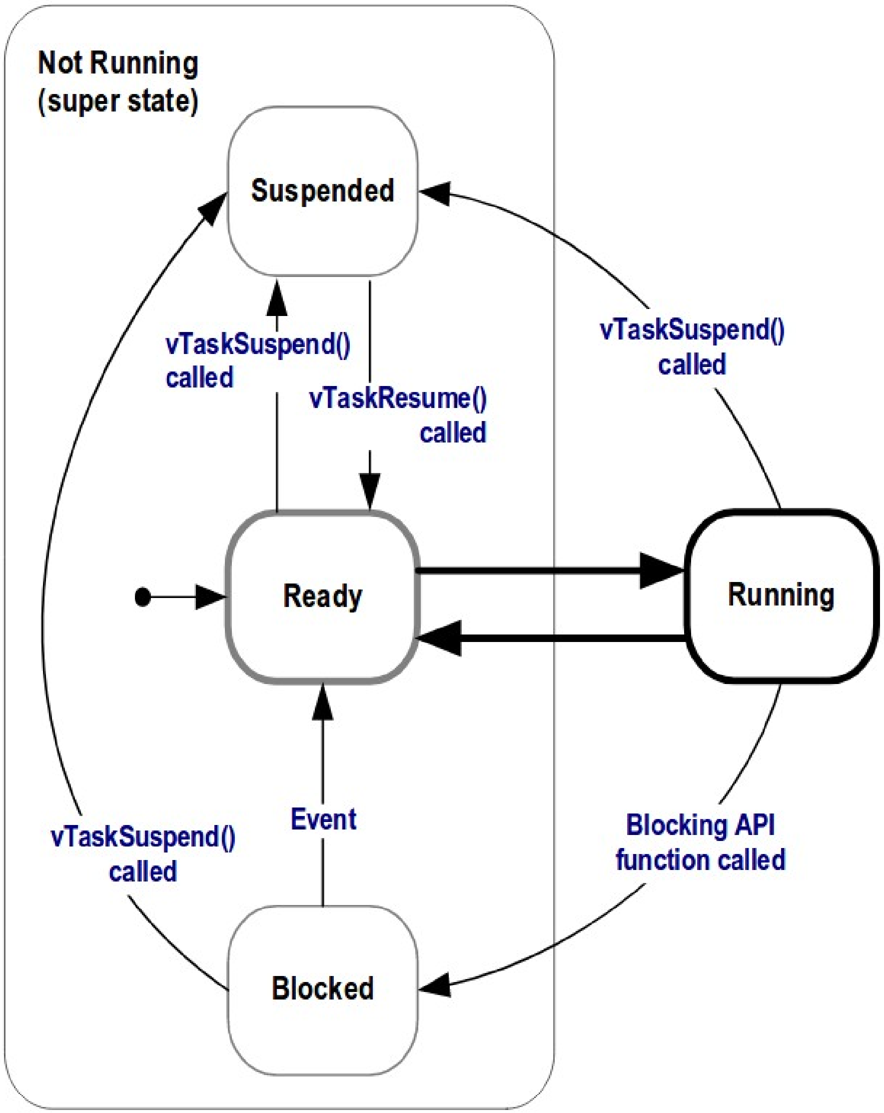
Task 2 with higher priority than task 1





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

Expanding the 'Not Running' State

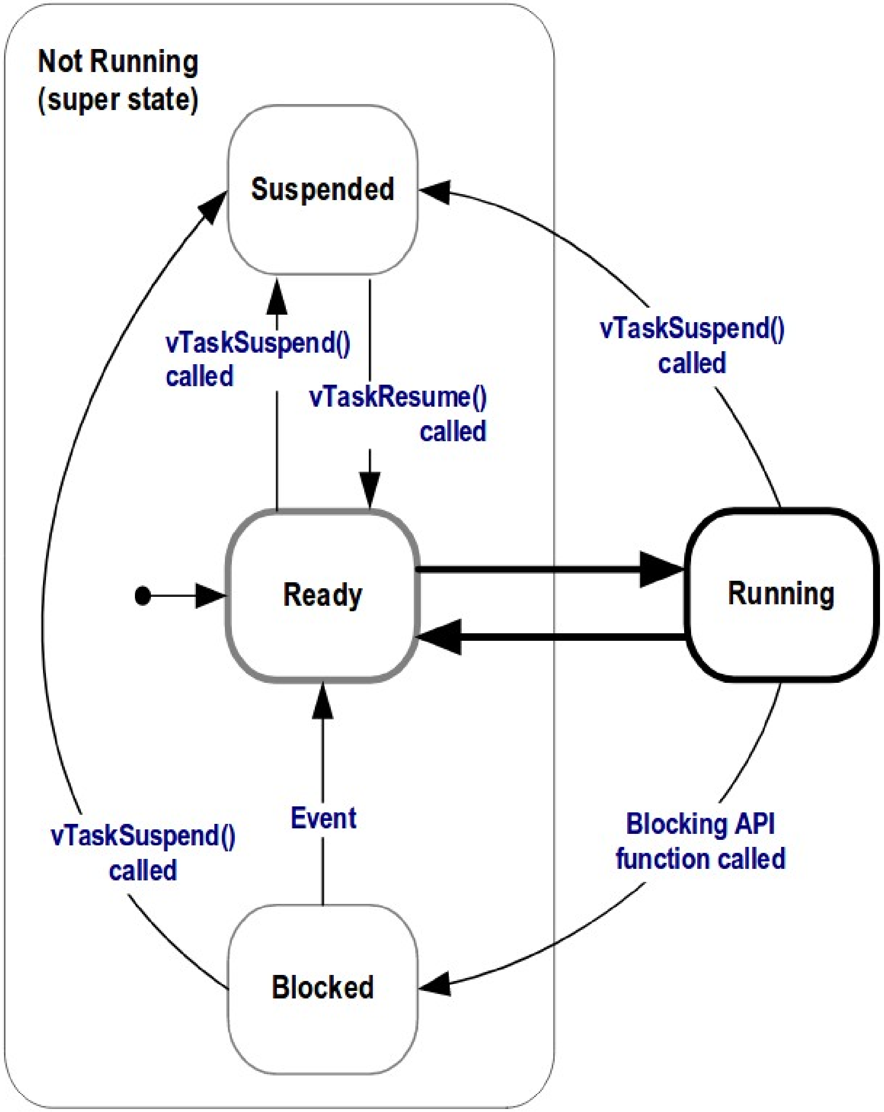


To make the tasks useful they must be re-written to be **event-driven**.

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

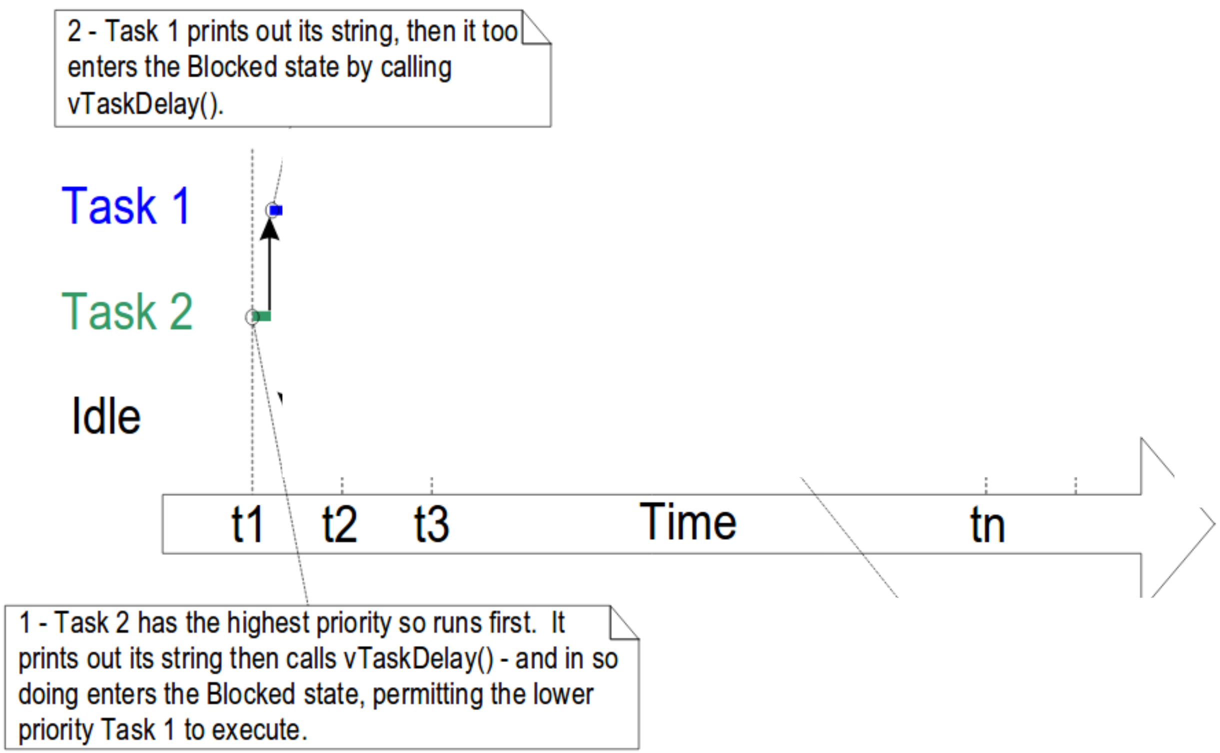
```
void vTaskFunction( void *pvParameters )
{
     char *pcTaskName;
     const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
     volatile uint32_t ul;
/*The string to print out is passed in via the parameter.*/
     pcTaskName = ( char * ) pvParameters;
/* As per most tasks, this task is implemented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

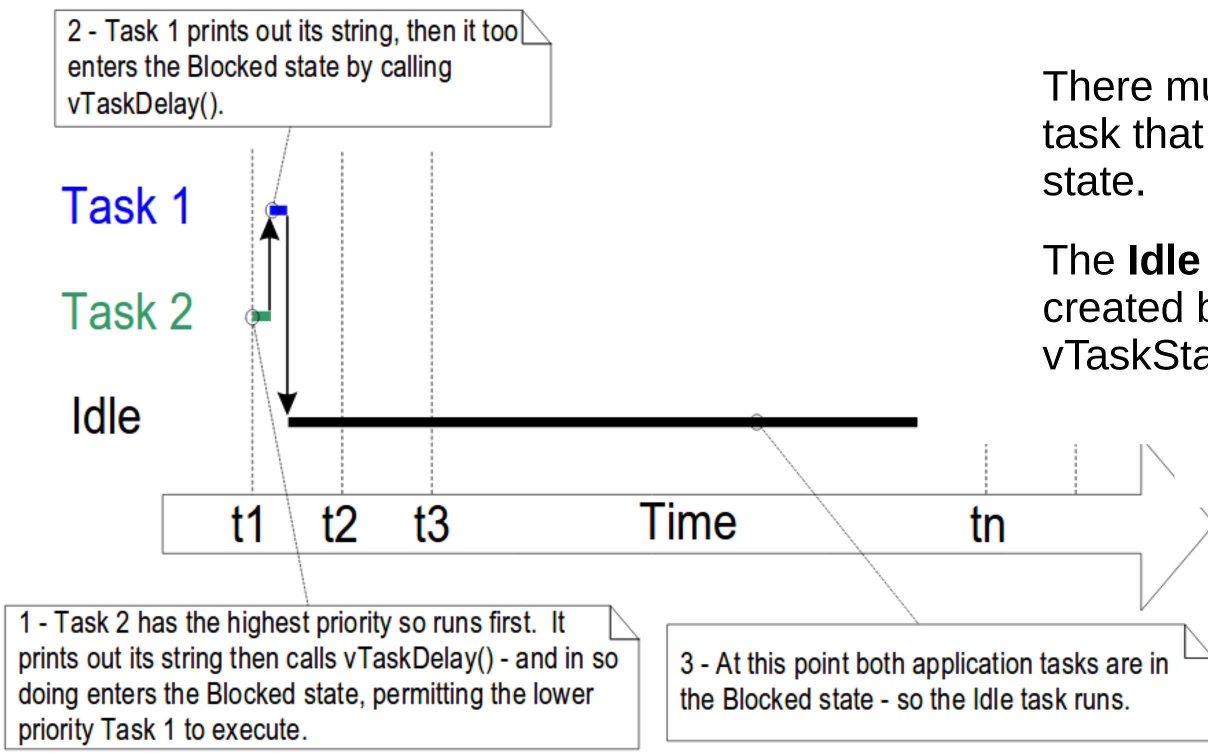
```
/* main function */
Static const char *pcTextForTask1 ="Task 1 is running\r\n";
static const char *pcTextForTask2 ="Task 2 is running\r\n";
int main(void)
{
      xTaskCreate(vTaskFunction,"Task 1",1000,(void*)pcTextForTask1,1,NULL);
      xTaskCreate(vTaskFunction,"Task 2",1000,(void*)pcTextForTask2,2,NULL);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

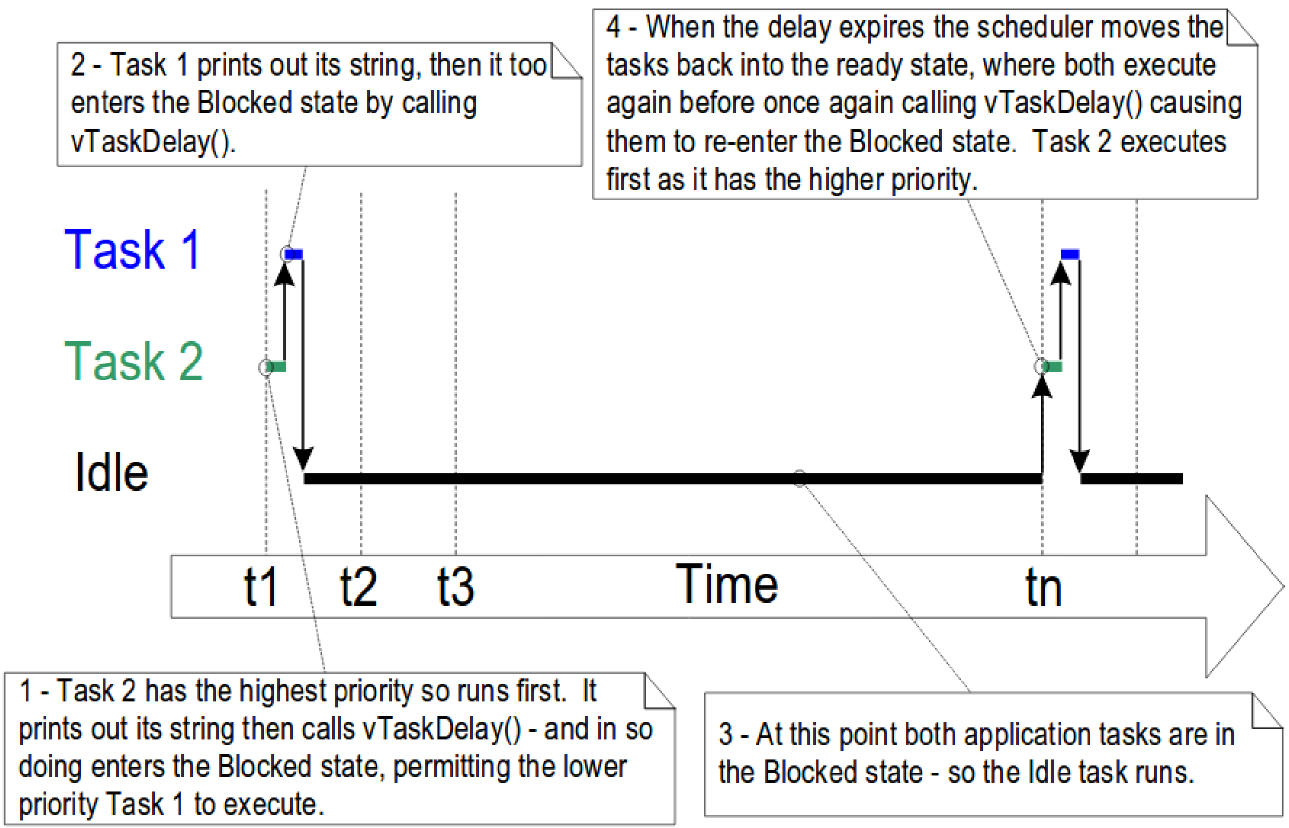


There must always be at least one task that can enter the Running state.

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

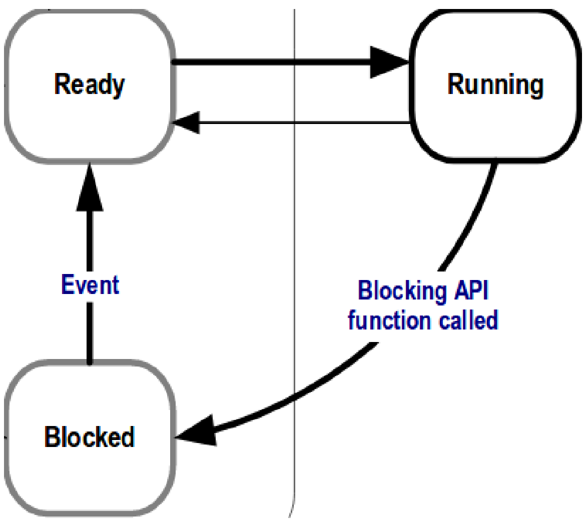
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the 'Not Running' State



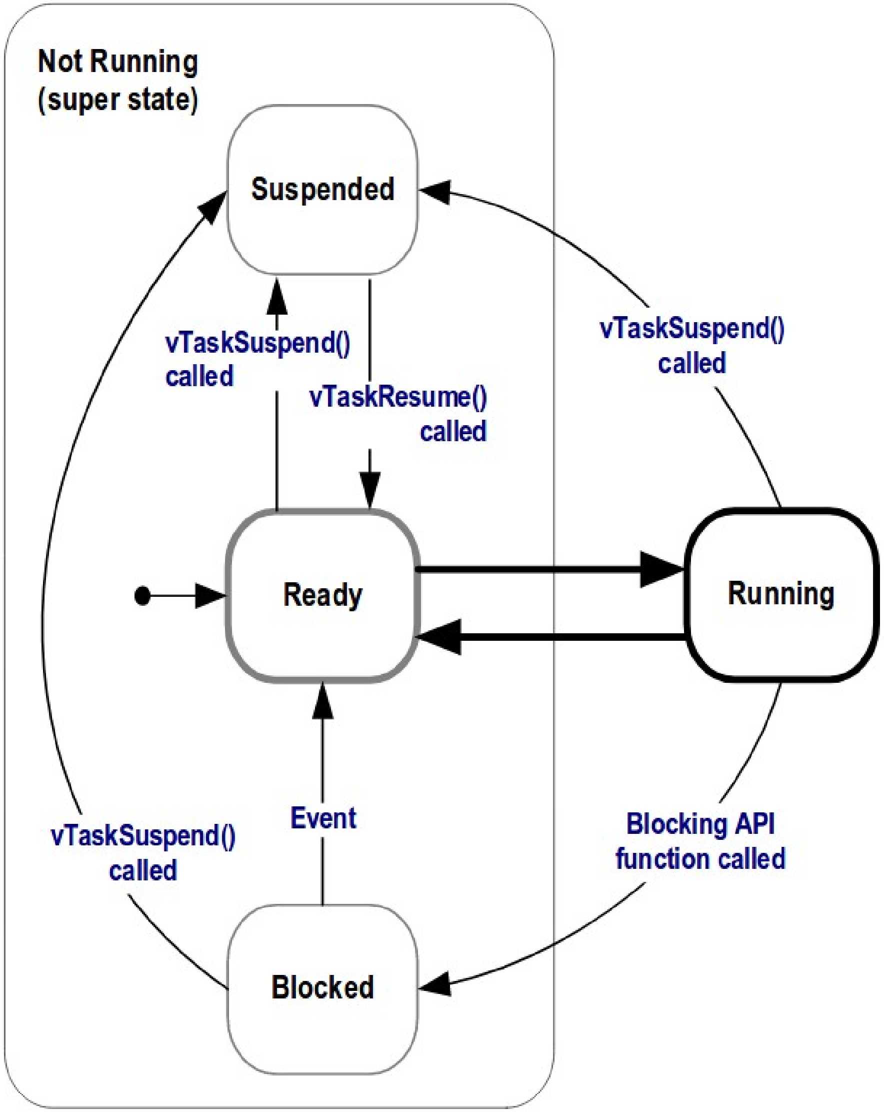
More efficient implementation of tasks

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.

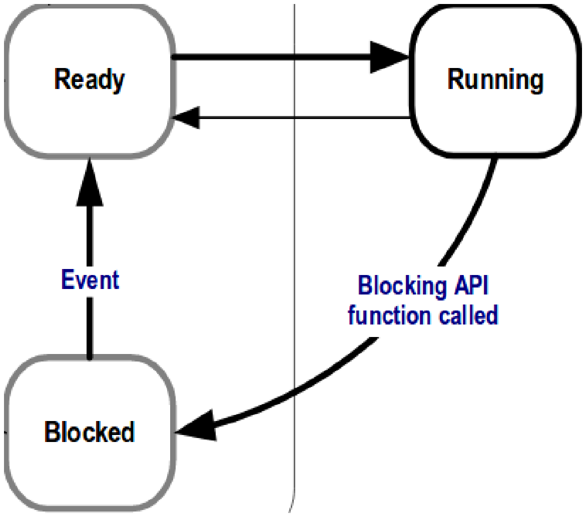
Most applications do not use the Suspended state.

Executing periodic tasks

```
const TickType_t xDelay250ms = pdMS_TO_TICKS( 250 );
vTaskDelay(xDelay250ms);
```

number of tick interrupts that the calling task will remain in the Blocked state

Using vTaskDelay() does not guarantee that the frequency at which they run is fixed,



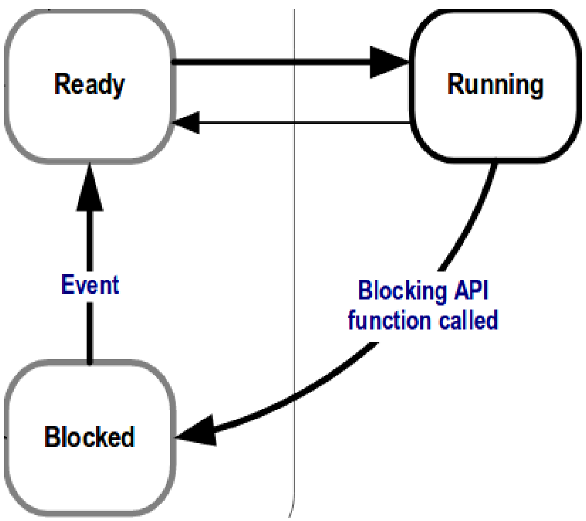
Executing periodic tasks

void vTaskDelayUntil(TickType_t * pxPreviousWakeTime, TickType_t xTimeIncrement);

Time of last left of the Blocked state

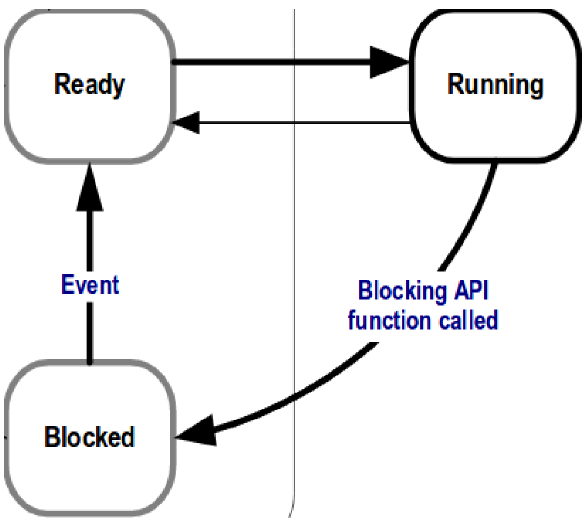
Time in number of ticks

The parameters to **vTaskDelayUntil()** specify **the exact tick count value** at which the calling task should be moved **from the Blocked state into the Ready state.**



Executing periodic tasks

```
void vTaskFunction( void *pvParameters ){
    char *pcTaskName;
    TickType_t xLastWakeTime;
    pcTaskName = ( char * ) pvParameters;
    xLastWakeTime = xTaskGetTickCount();/* current tickcount.*/
    for( ;; ){/* Print out the name of this task. */
        vPrintString( pcTaskName );
    /*This task should execute every 250 milliseconds exactly.*/
        vTaskDelayUntil( &xLastWakeTime, pdMS_TO_TICKS( 250 ));
    }
}
```

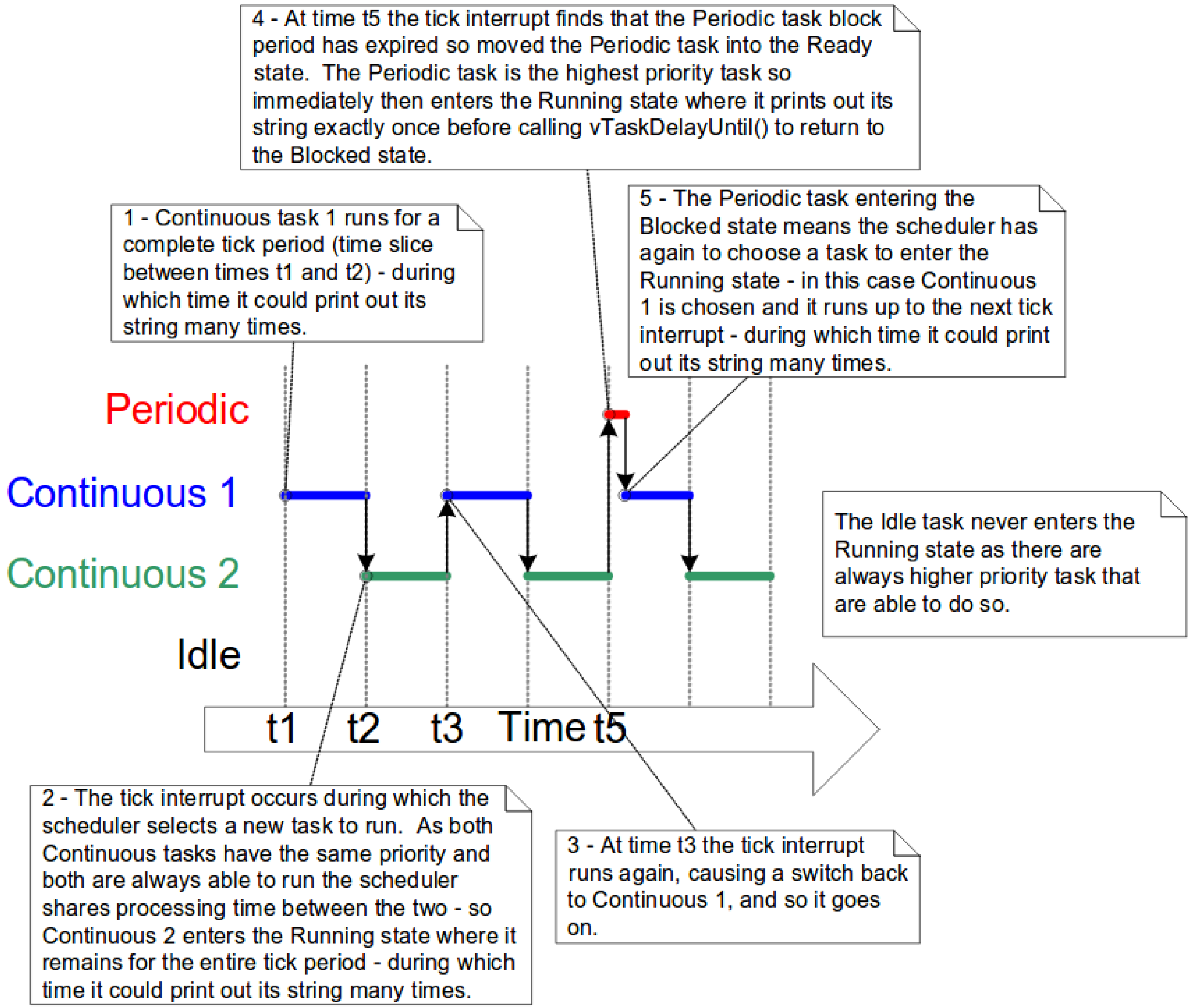


Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

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 handler 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 handler of the task
  - Have to Select INCLUDE_VtaskDelete in FreeRtosConfig.h file
- void **vTaskSuspend**(TaskHandle_t   pxTaskToSuspend);
  - PxTaskToSuspend: Handler of the task. With NULL means the task itself

Scheduling Algorithms

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

Configured in **FreeRTOSConfig.h**

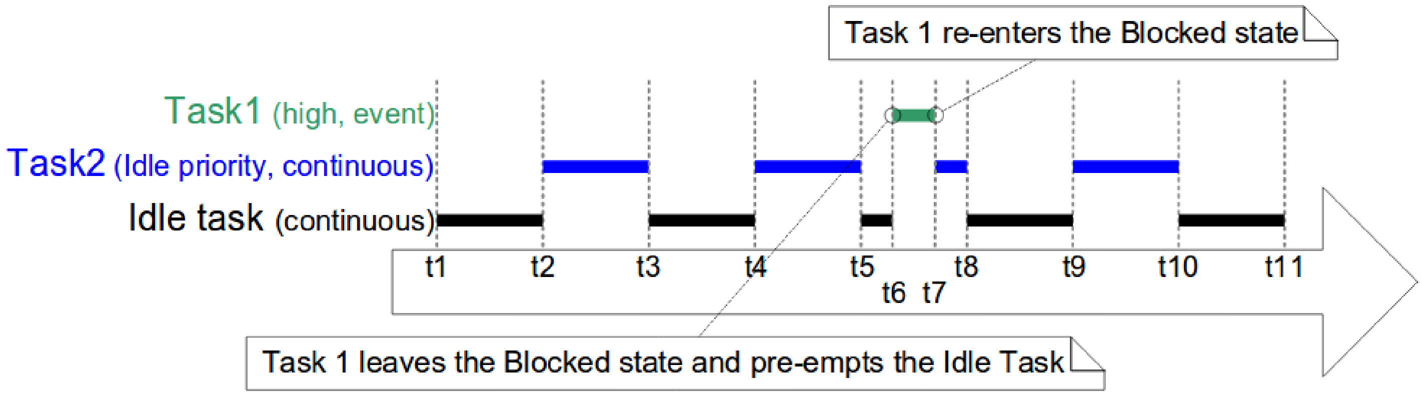
- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 1

Scheduling Algorithms

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

Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 1

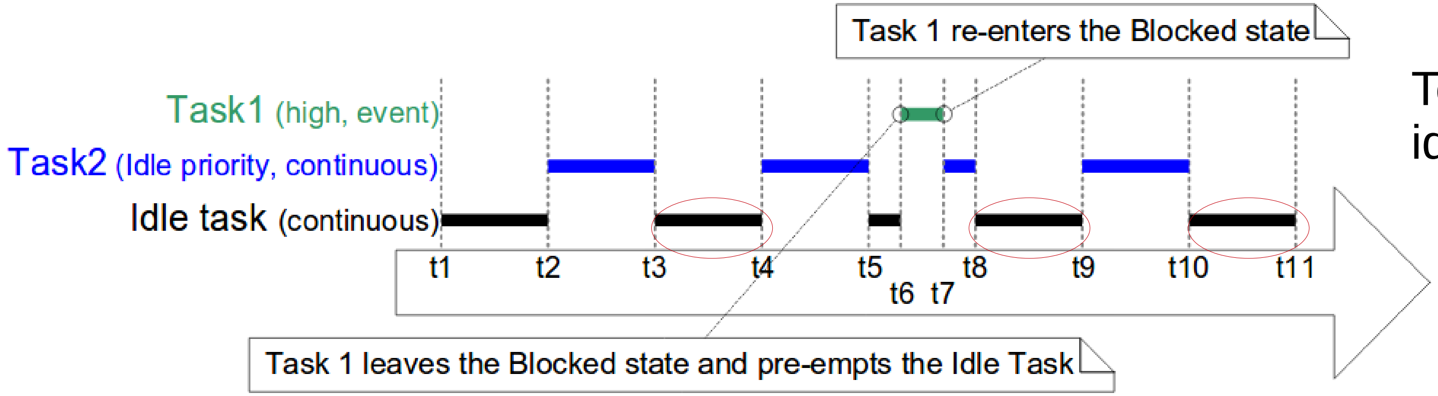


Scheduling Algorithms

- 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 RTOS tick interrupts

Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION 1
- configUSE_TIME_SLICING 1



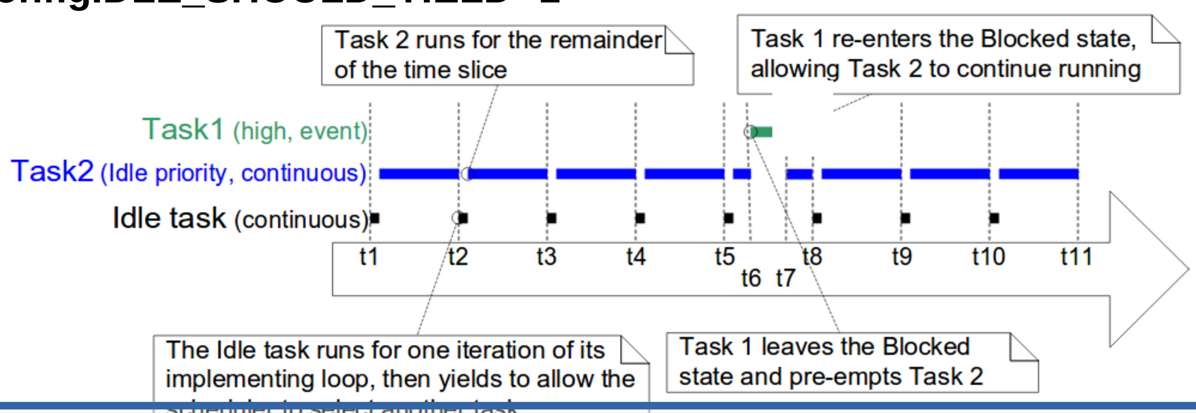
To much time for idle task

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 RTOS tick interrupts

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

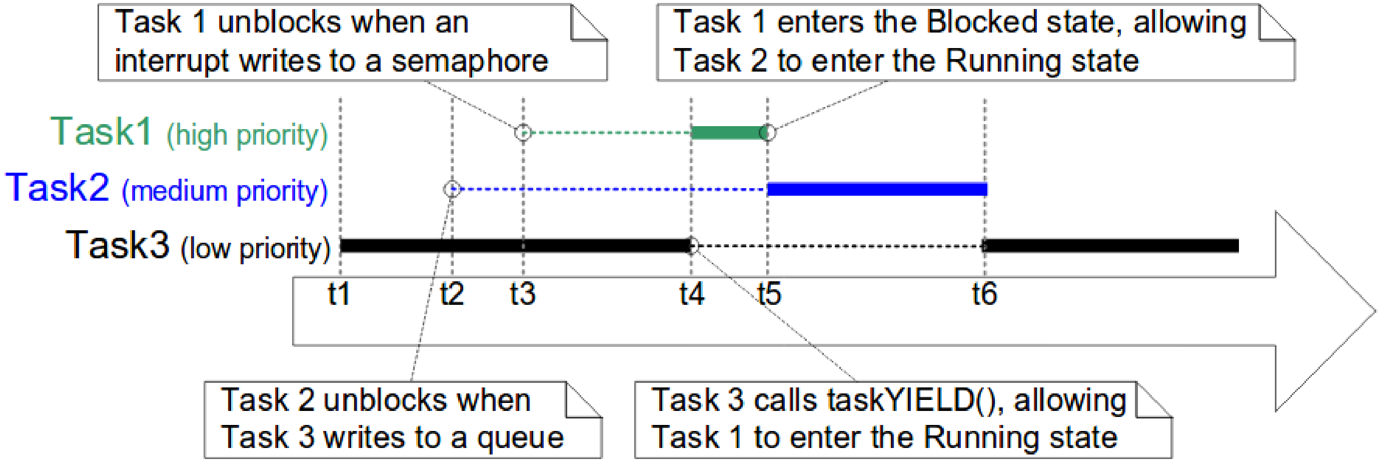
Scheduling Algorithms

- 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



Synchronization and Communications between tasks

FreeRtos provides with different mechanisms to share information between tasks and to control the access to shared resources

- Queues.
- Binary Semaphores
- Counting Semaphores
- Mutexes
- Recursive Mutexes
- Interrupts

Queues

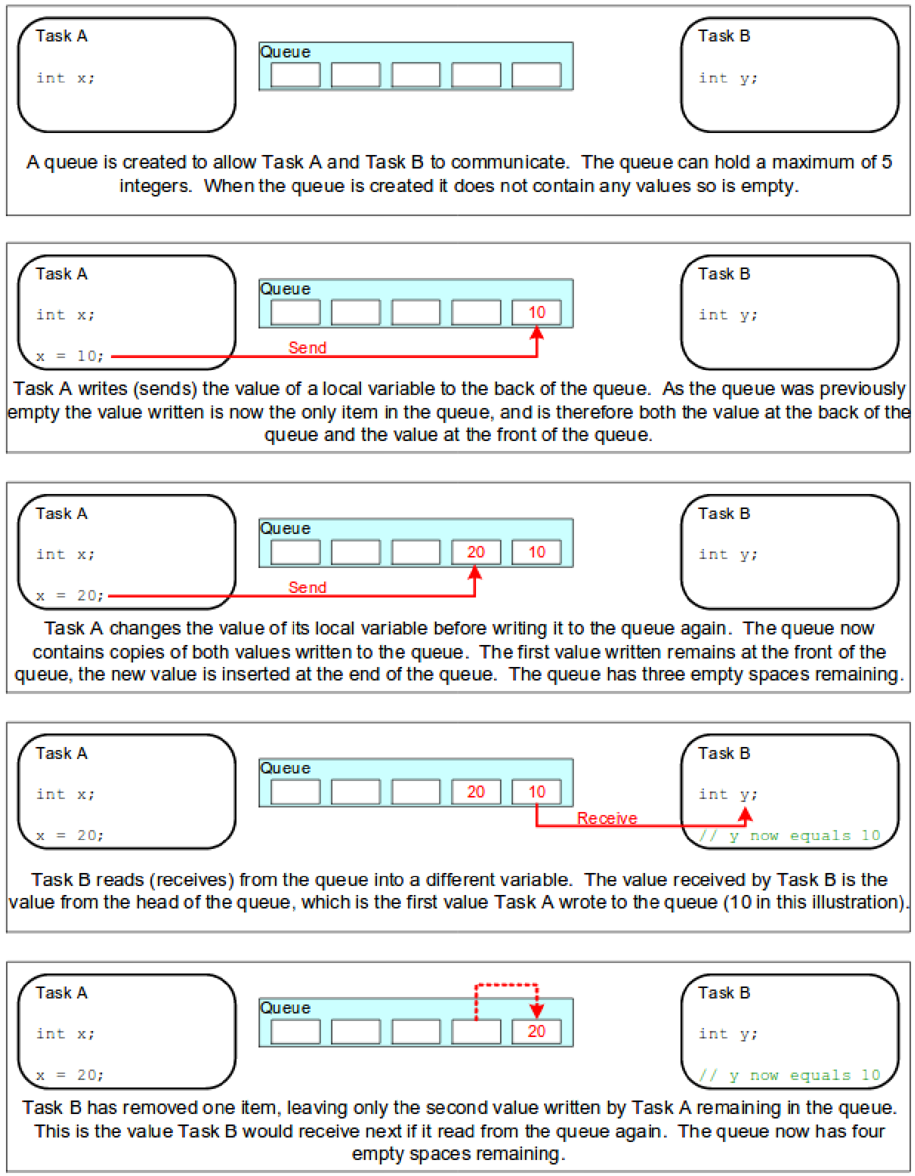
'Queues' provide a task-to-task, task-to-interrupt, and interrupt-to-task communication mechanism.

- 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.

Queue Management



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

Writing in a queue

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

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

Acts as a LIFO

```
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

Reading in a queue

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

- xQueue: The handle of the queue
- pvBuffer: A pointer to the buffer where the read value will be copied to.
- xTicksToWait: The maximum amount of time the task should wait for available data.

Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue empty

Queue Management

Reading in a queue

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

- xQueue: The handle of the queue
- pvBuffer: A pointer to the buffer where the read value will be copied to.
- xTicksToWait: The maximum amount of time the task should wait for available data.

Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue empty

Example

```
if (xQueueReceive (MyQueue, &valueFromQueue, portMAX_DELAY) ==pdPASS) {
Serial.println (valueFromQueue);
```

Queue Management

Reading in a queue

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

- xQueue: The handle of the queue
- pvBuffer: A pointer to the buffer where the read value will be copied to.
- xTicksToWait: The maximum amount of time the task should wait for available data.

Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue empty

After reading an element in a queue, this element is normally removed from it; however, an other read function given in allows to read an element without having it to be deleted from the queue.

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

Queue Management

- xQueueCreate
- xQueueCreateStatic
- vQueueDelete
- xQueueSend
- xQueueSendFromISR
- xQueueSendToBack
- xQueueSendToBackFromISR
- xQueueSendToFront
- xQueueSendToFrontFromISR
- xQueueReceive
- xQueueReceiveFromISR
- uxQueueMessagesWaiting
- uxQueueMessagesWaitingFromISR
- uxQueueSpacesAvailable
- xQueueReset
- xQueuePeek
- xQueuePeekFromISR
- vQueueAddToRegistry
- pcQueueGetName
- vQueueUnregisterQueue
- xQueueIsQueueEmptyFromISR
- xQueueIsQueueFullFromISR
- xQueueOverwrite
- xQueueOverwriteFromISR

Binary Semaphores

Used to control the access of shared resources

Can be seen as a queue of one element

A semaphore can be taken by only one task. If another task try to take the semaphore it will be blocked until the owner or the semaphore gives them.

Binary Semaphores

API Functions for managing semaphores

Creating a semaphore

SemaphoreHandle_t **xSemaphoreCreateBinary**(void);

Take

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

Give

BaseType_t **xSemaphoreGive(** SemaphoreHandle_t xSemaphore);

Give asemaphore from ISR

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

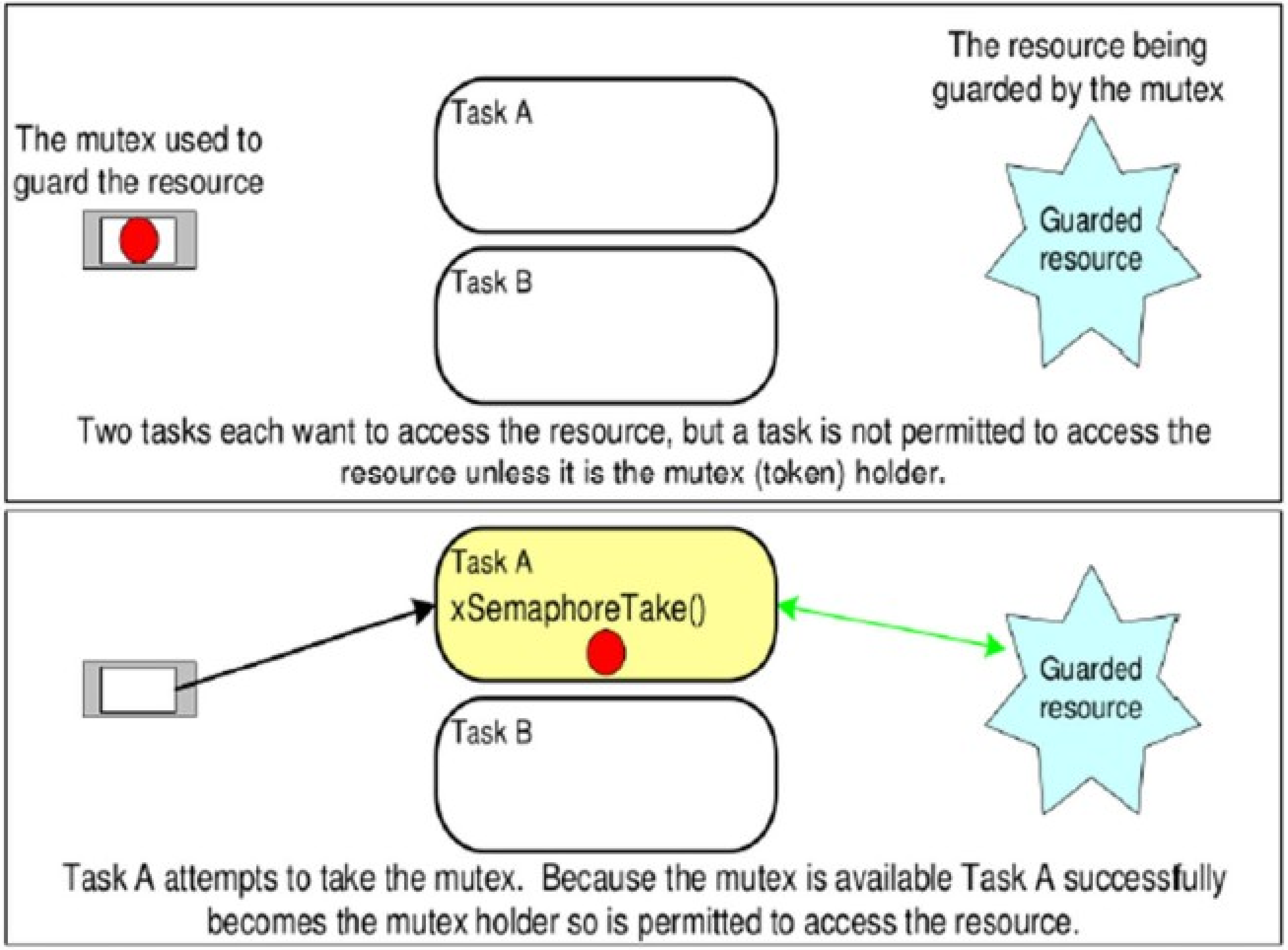
Mutex

The Mutex es un special kind of binary semaphore to control the access to the same resource for two of more tasks.

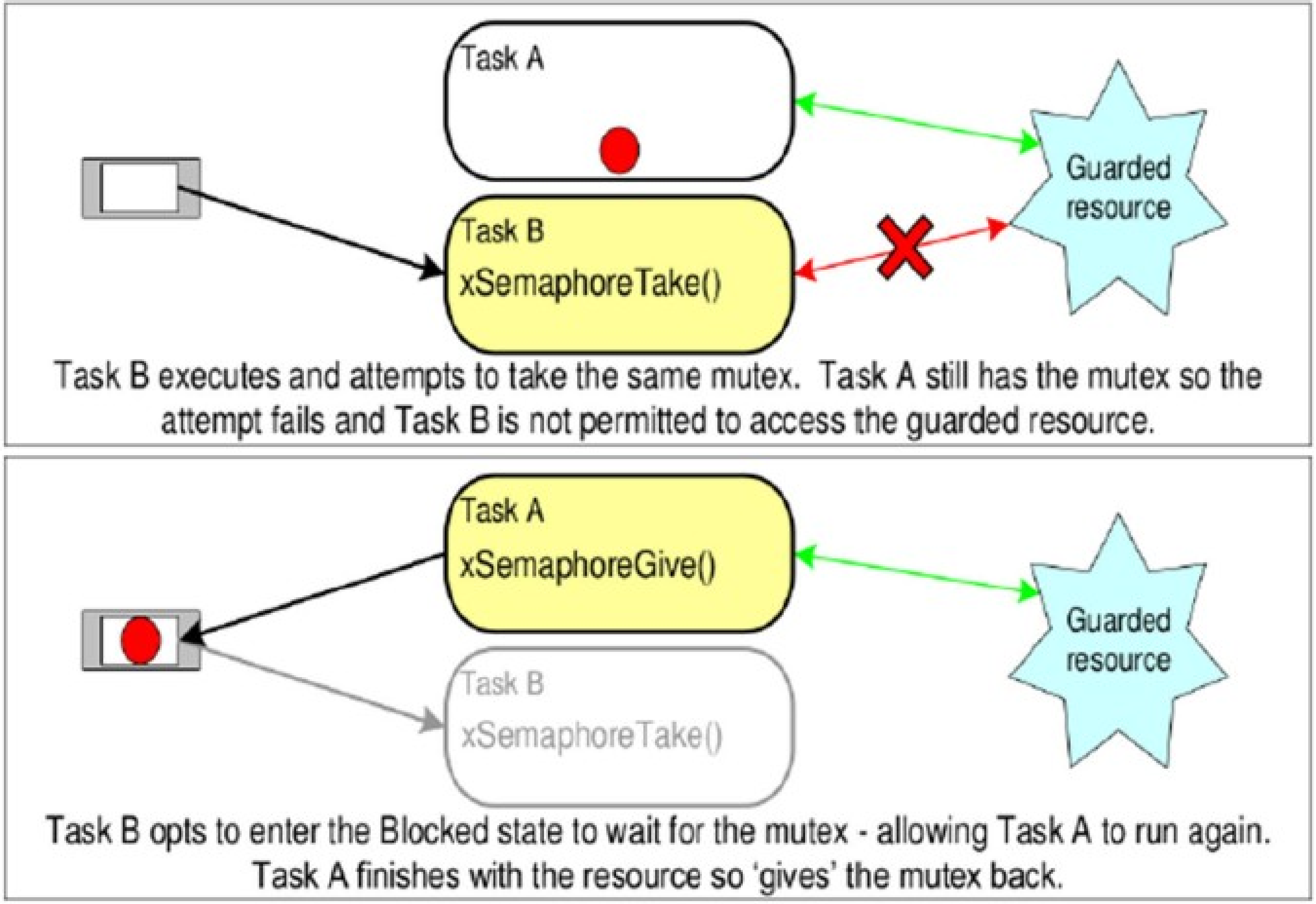
It Includes a priority inheritance mechanism.

While the binary semaphores are the best option for synchronization between tasks or between tasks and interruptions, mutexes are the best option for simple **mut**ual **ex**clusion implementation.

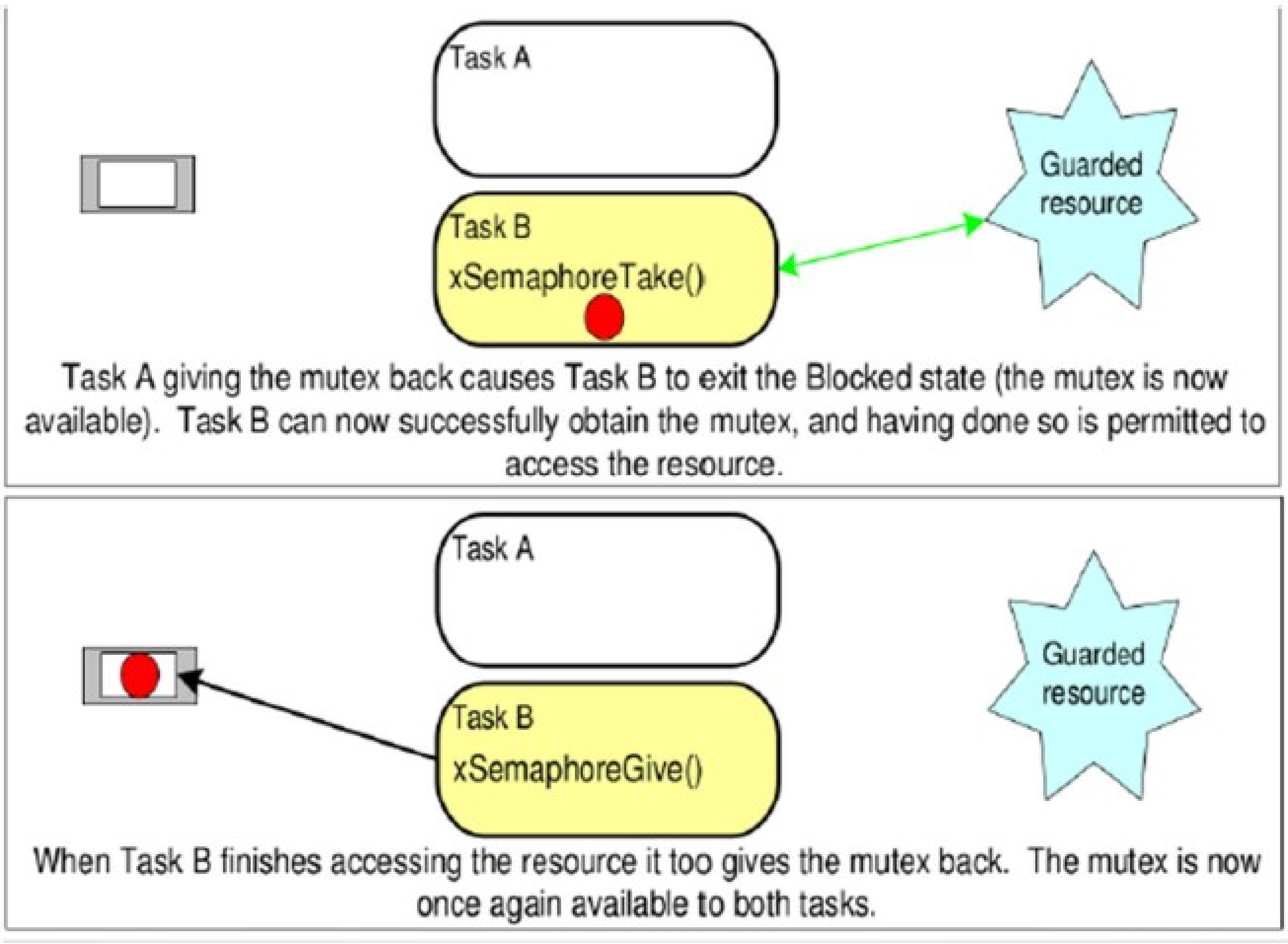
Mutex


The resource being guarded by the mutex
Task A
The mutex used to guard the resource
Guarded resource
Task B
Two tasks each want to access the resource, but a task is not permitted to access the resource unless it is the mutex (token) holder.
Task A
xSemaphoreTake()
Guarded resource
Task B
Task A attempts to take the mutex. Because the mutex is available Task A successfully becomes the mutex holder so is permitted to access the resource.

Mutex


Task A
Guarded
resource
Task B
xSemaphoreTake()
Task B executes and attempts to take the same mutex. Task A still has the mutex so the attempt fails and Task B is not permitted to access the guarded resource.
Task A
xSemaphoreGive()
Guarded
resource
Task B
xSemaphoreTake()
Task B opts to enter the Blocked state to wait for the mutex - allowing Task A to run again. Task A finishes with the resource so 'gives' the mutex back.

Mutex


Task A
Task B
xSemaphoreTake()
Guarded resource
Task A giving the mutex back causes Task B to exit the Blocked state (the mutex is now available). Task B can now successfully obtain the mutex, and having done so is permitted to access the resource.
Task A
Task B
xSemaphoreGive()
Guarded resource
When Task B finishes accessing the resource it too gives the mutex back. The mutex is now once again available to both tasks.

Interrupt Manangement

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

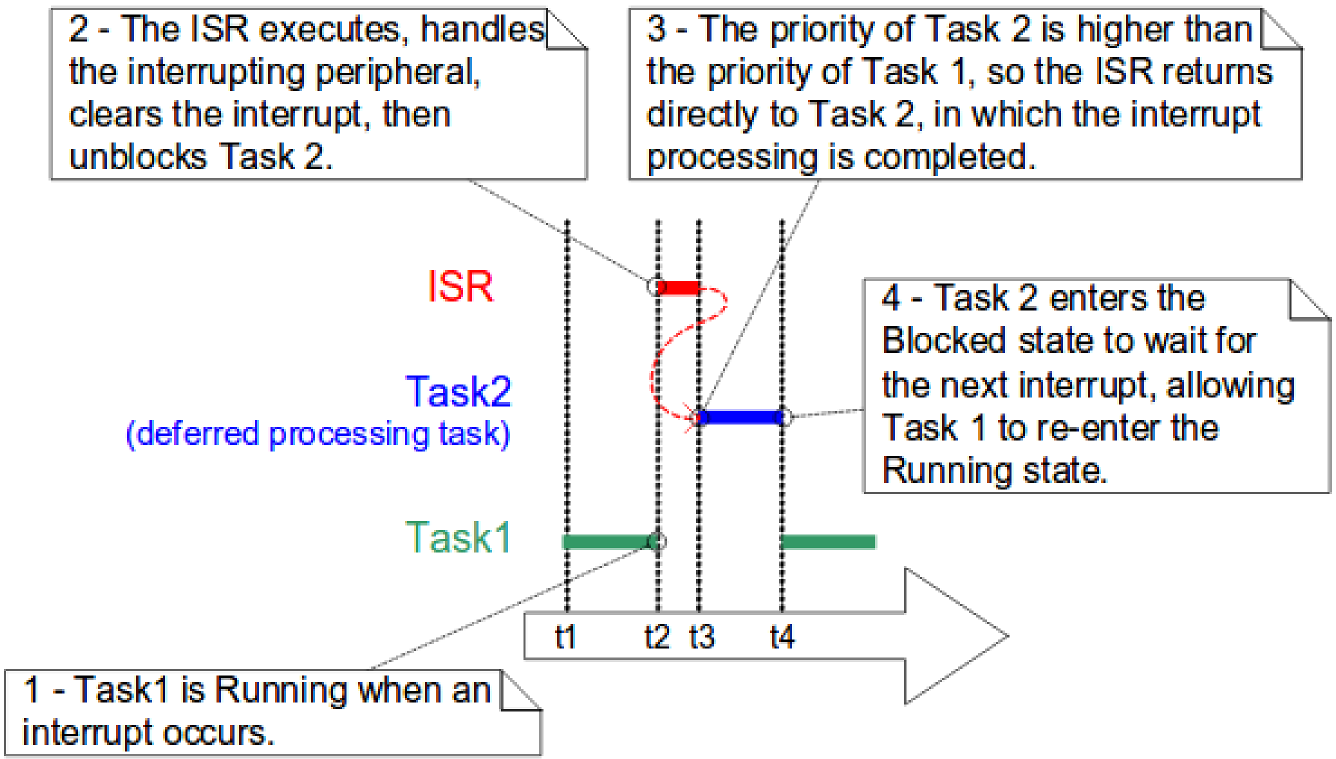
Lowest priority interrupt pre-empt highest priority task

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).

I**nterrupts should be deferred to a task**

Interrupt Manangement



Interrupt Manangement

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 Manangement

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 Manangement

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

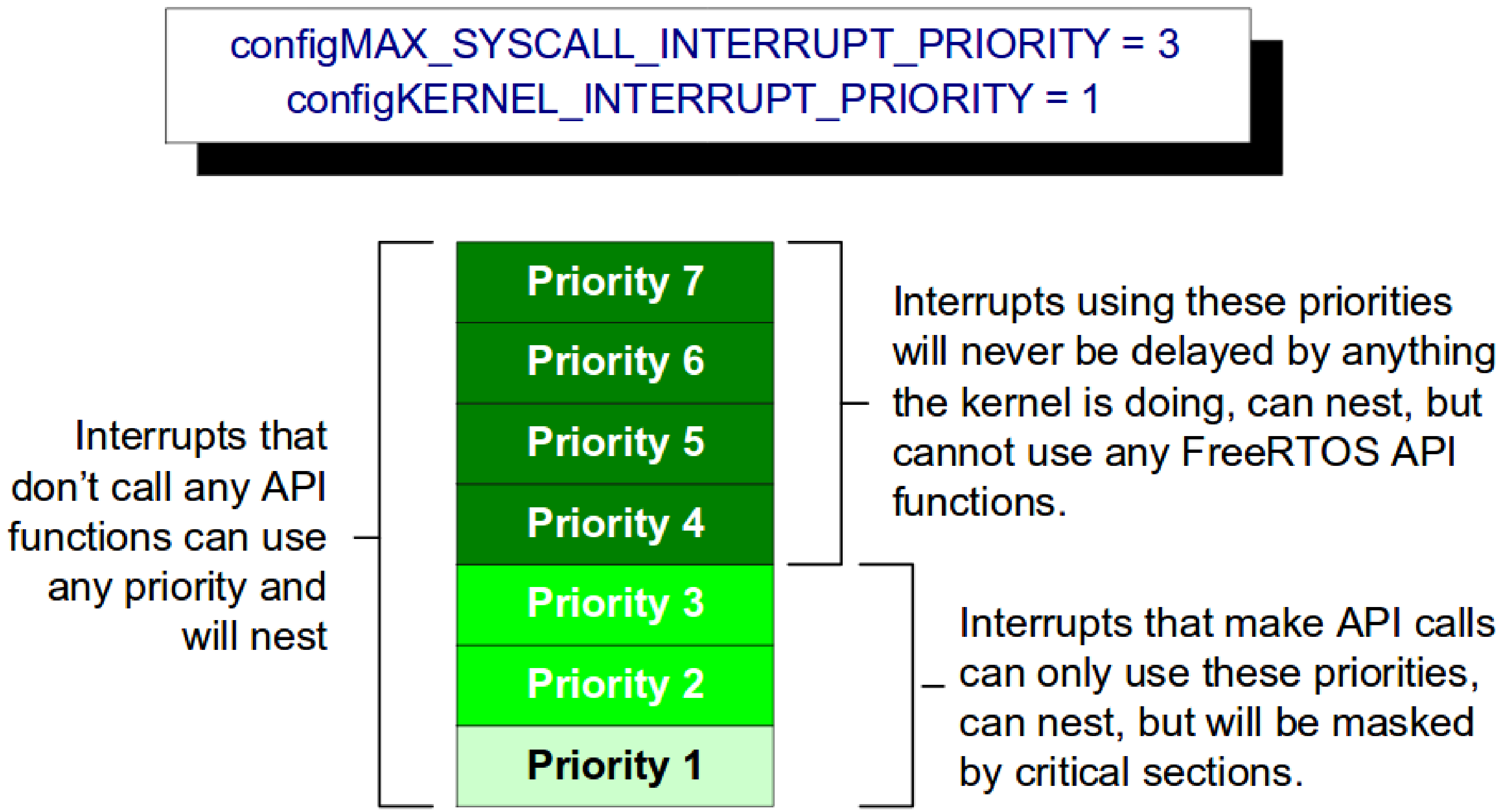
.

Return:

- pdPASS – OK
- errQUEUE_EMPTY – Error, queue full

Interrupt Manangement

Nested interrupts



Any Question?

THANKS AND GOOD LUCK!!