Introduction to FreeRTOS
Part 1

Characteristics of freeRTOS (Operating System)

- FreeRTOS is a "Embedded Operating System" for Embedded MicroController
- Software that provides multitasking facilities.
- FreeRTOS allows to run multiple tasks
- and has a simple scheduler to switch between tasks.

FreeRTOS features:
- Priority-based multitasking capability
- Queues to communicate between multiple tasks
- Semaphores to manage resource sharing 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

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'.

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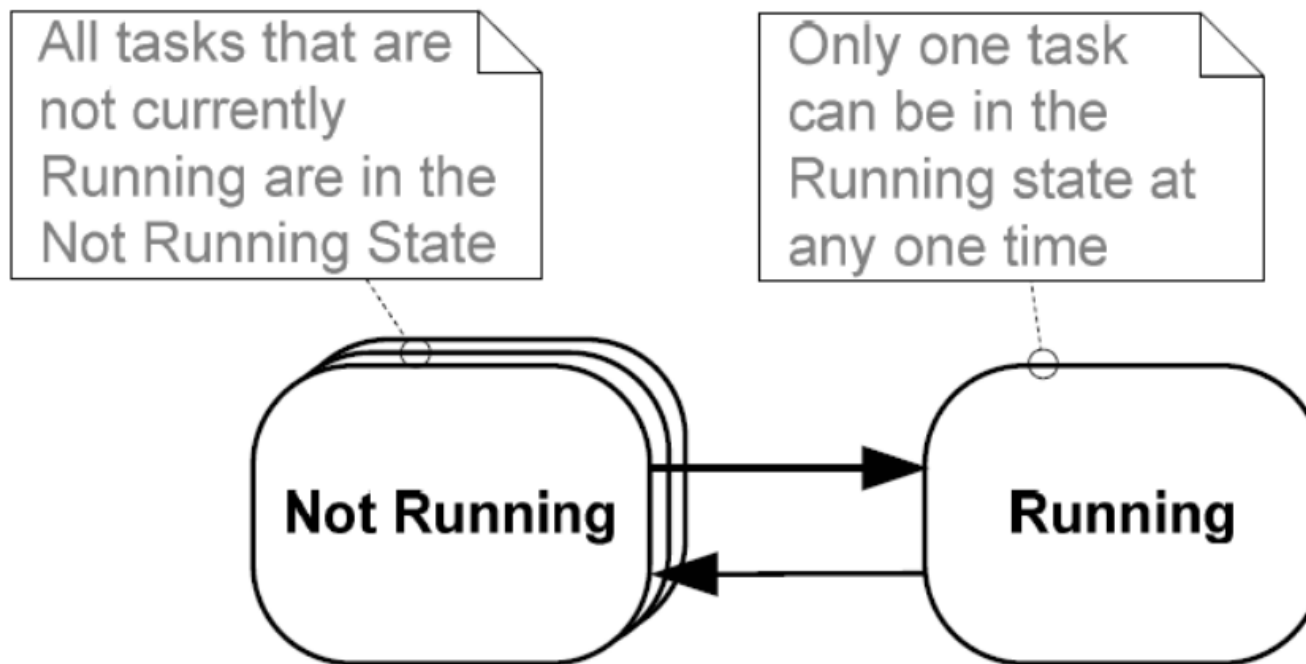
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Creating a Task

The Task function itself:

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

Install the Task (in main.c):

```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

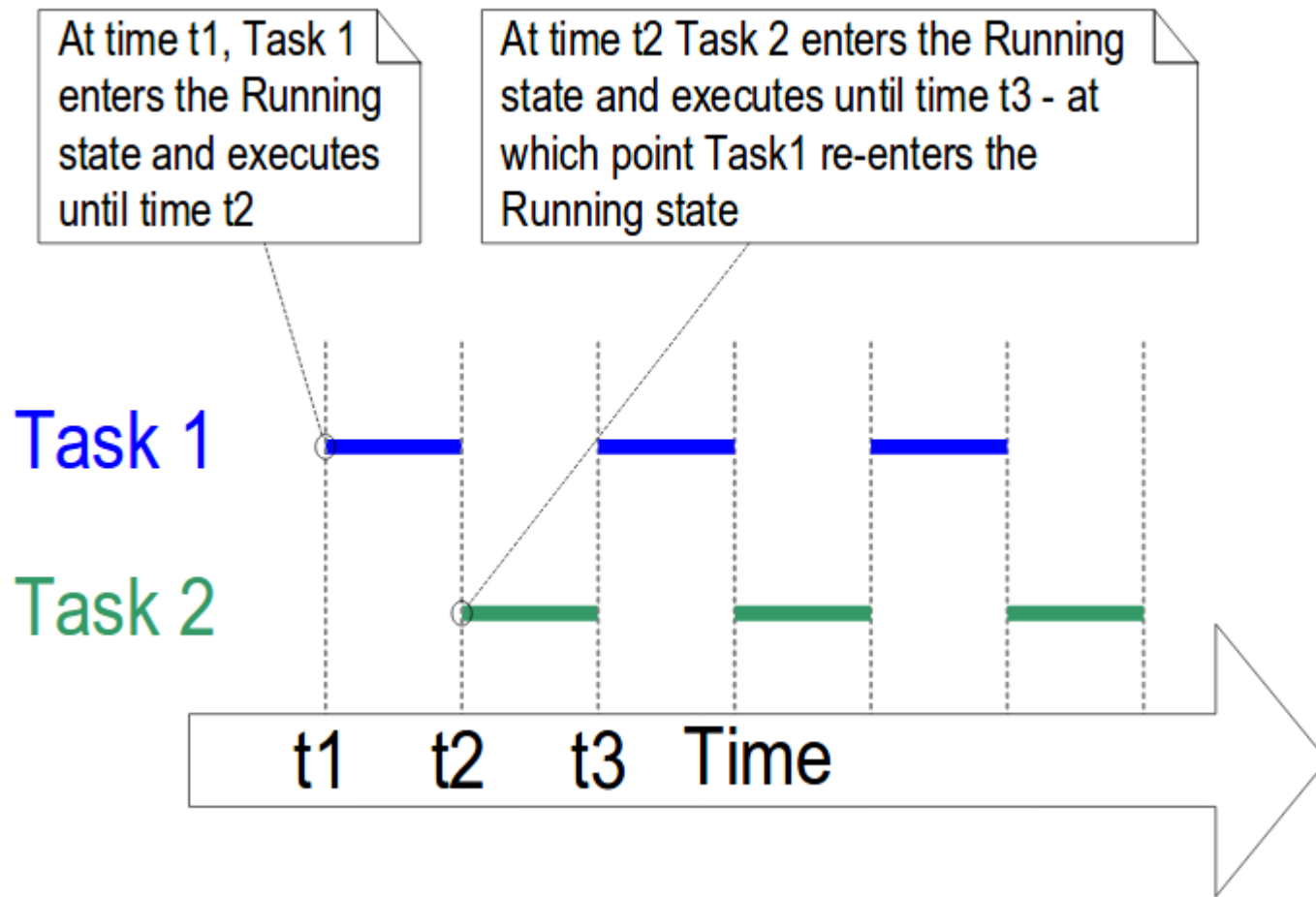Functions are prefixed with both the type they return, and the file they are defined within. For example:

- **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.

Task running with the same priority

```c
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. */
        {
        }
    }
}
```

```c
/* 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

Kernel runs in tick interrupt to select next task

Tick interrupt occurs

Newly selected task runs when the tick interrupt completes

Kernel

Task 1

Task 2

t1   t2   t3

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

```c
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. */
        {
        }
    }
}
```

```c
/* 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( ;; );
}
```
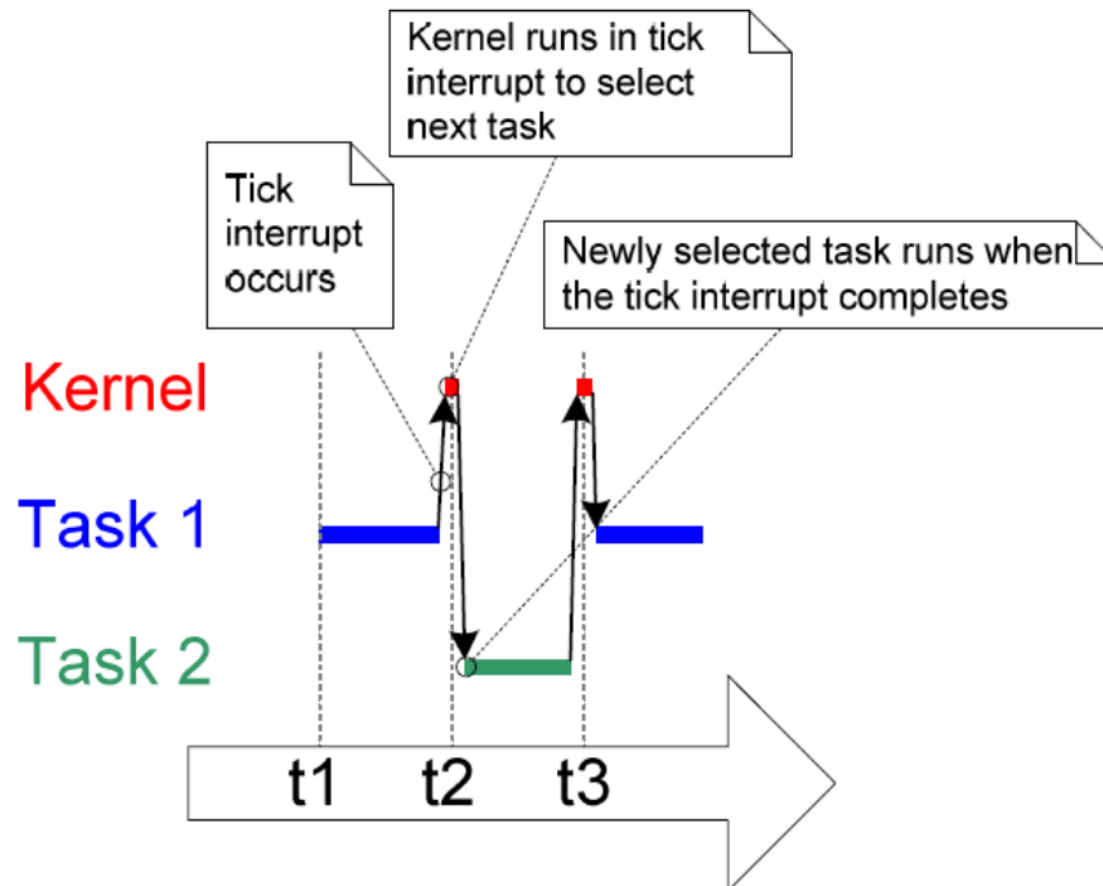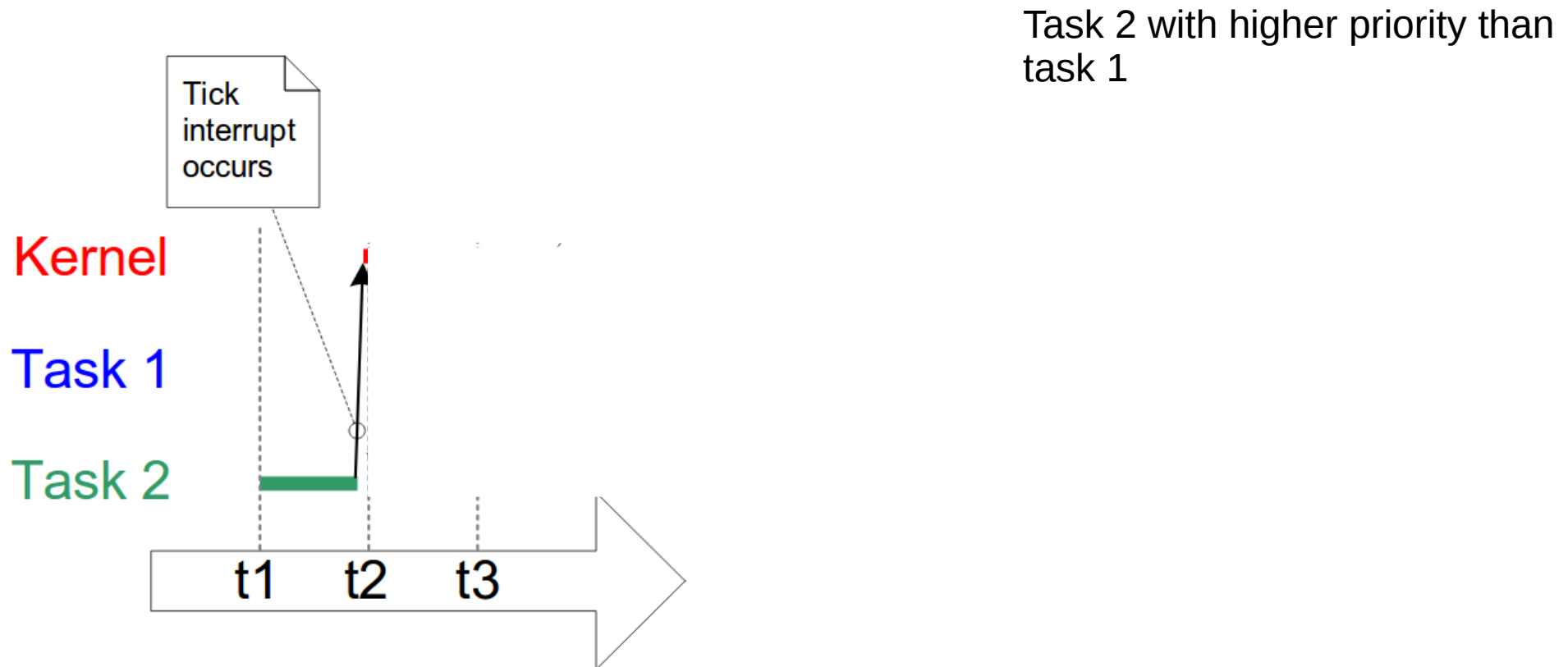
Task running with different priorities

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

Tick interrupt occurs

The scheduler runs in the tick interrupt but selects the same task. Task 2 is always in the Running state and Task 1 is always in the Not Running state

Kernel

Task 1

Task 2

t1   t2   t3

All tasks that are not currently Running are in the Not Running State

Only one task can be in the Running state at any one time

Not Running   Running
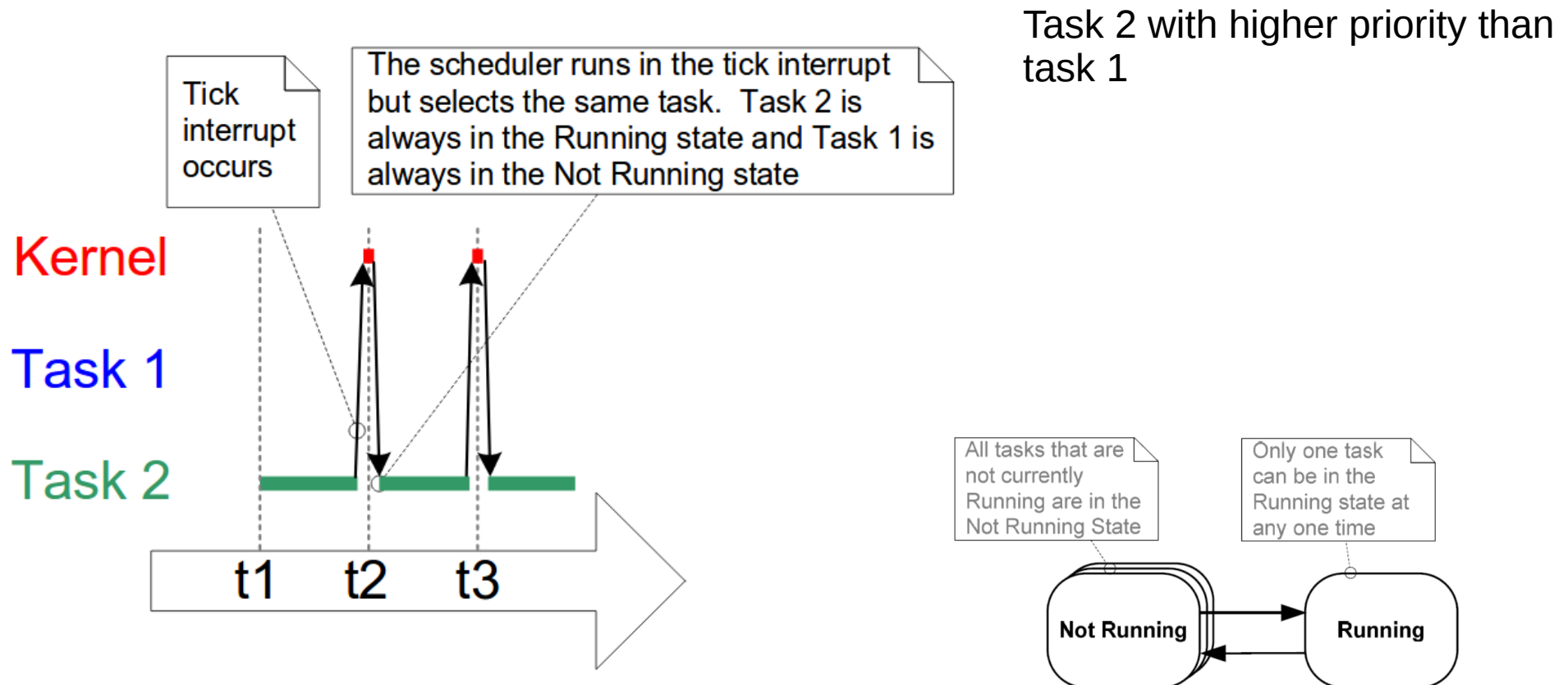
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

Expanding the 'Not Running' State
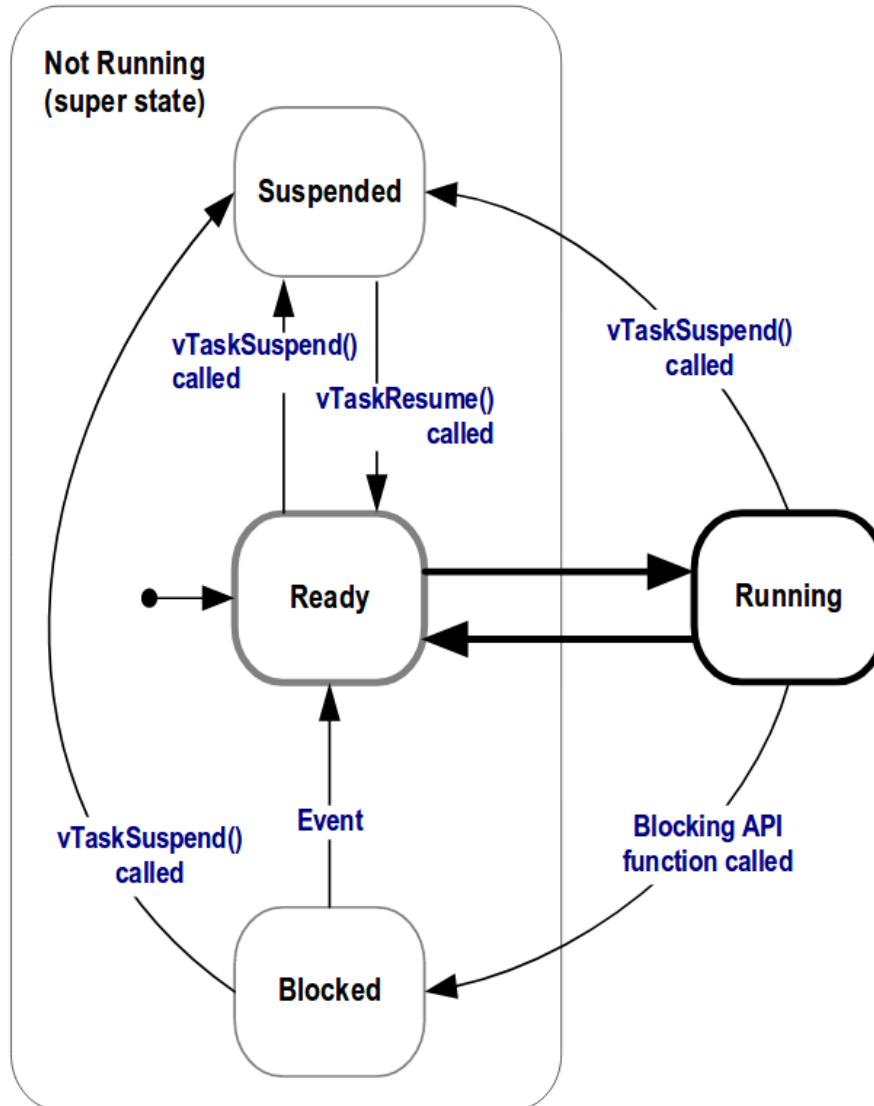
To make the tasks useful they must be re-written to be **event-driven**.

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
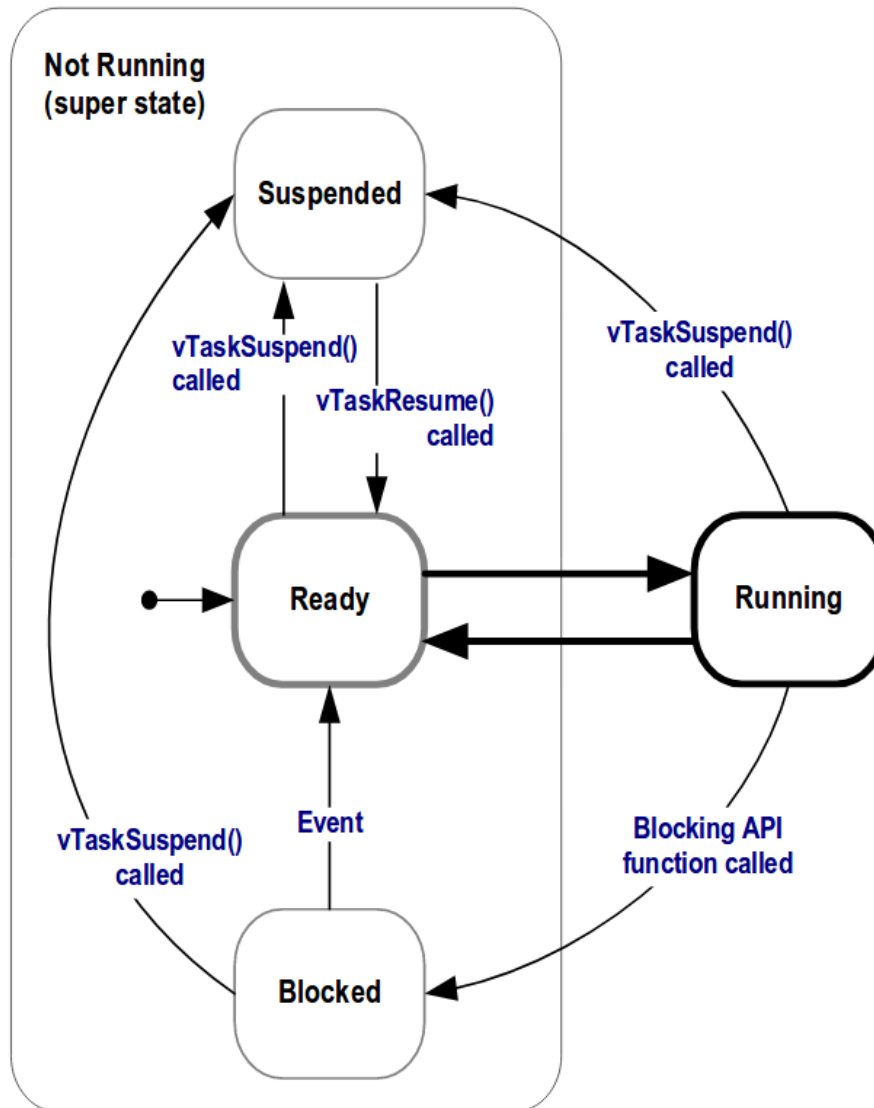
When a task is waiting for an event is Blocked

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

```c
/* 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( ;; );
}
```

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().

Task 1

Task 2

Idle

t1   t2   t3   Time   tn

1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.

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( ;; );
}
```
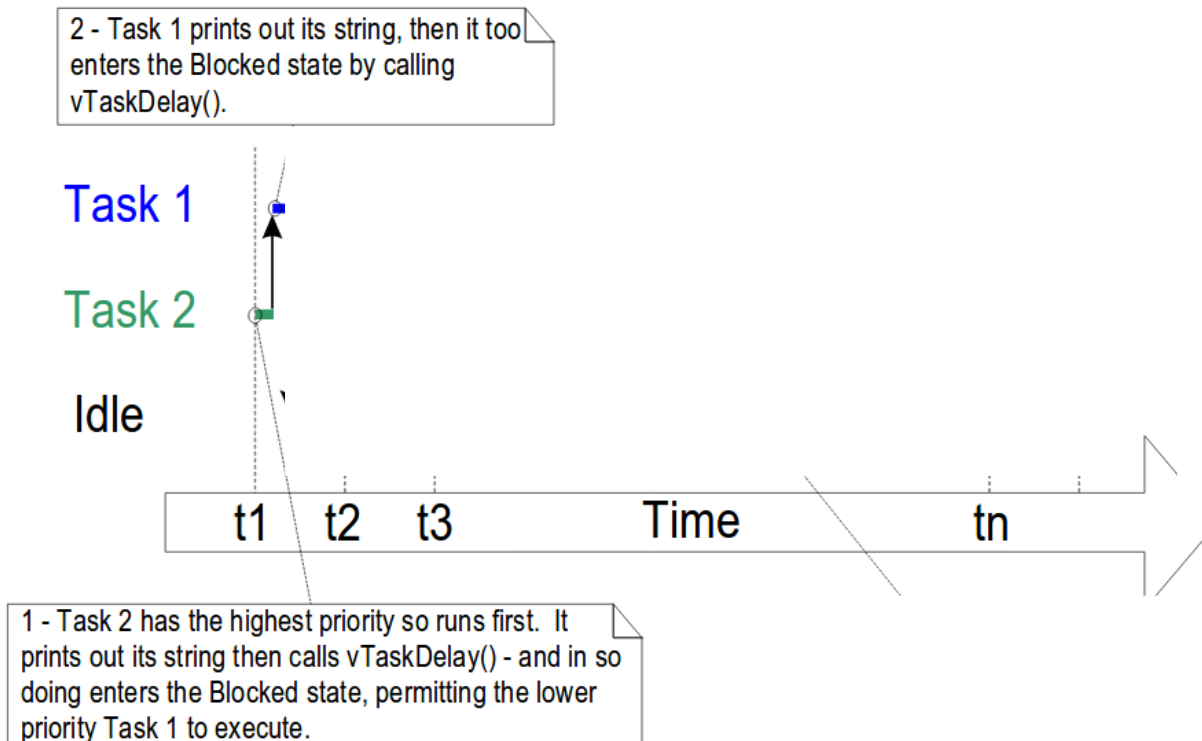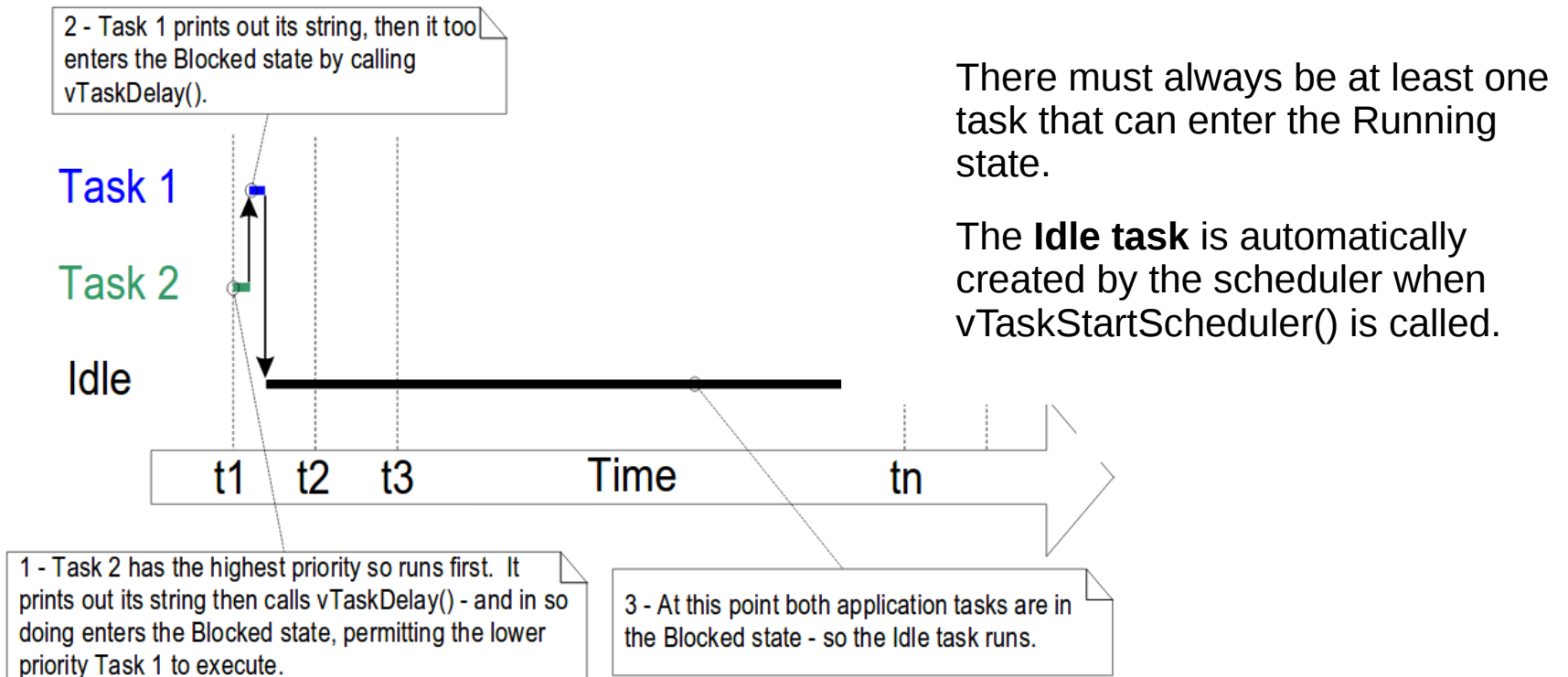

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().

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.

1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.

3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

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

Expanding the 'Not Running' State

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().

4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.

More efficient implementation of tasks

Less use of processor time

Task 1

Task 2

Idle

t1 t2 t3   Time   tn

1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.

3 - At this point both application tasks are in the Blocked state - so the Idle task runs.
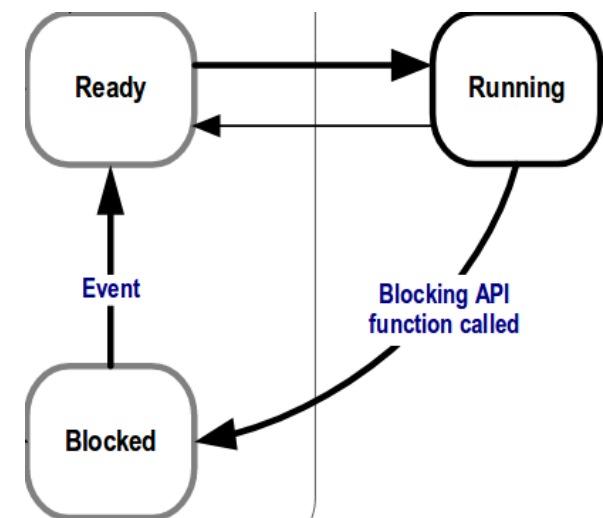
Ready

Running

Event

Blocking API function called

Blocked

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

Expanding the 'Not Running' State

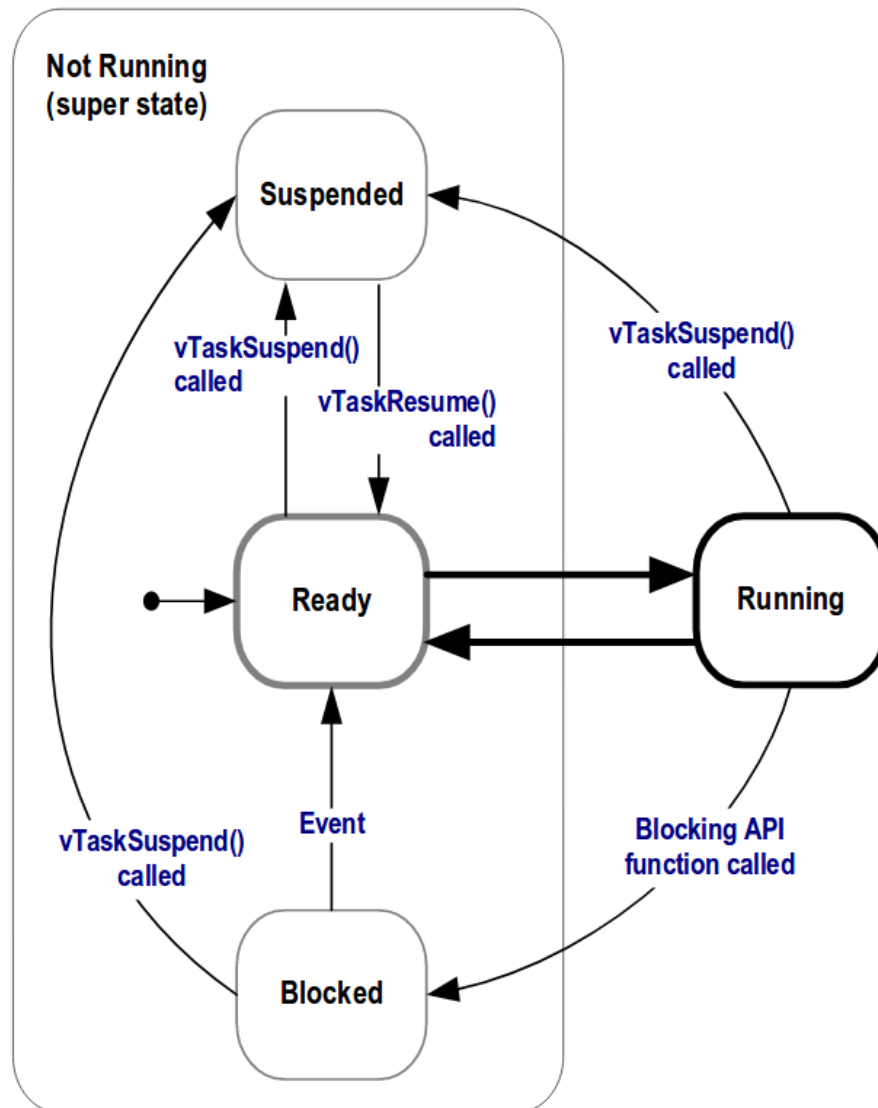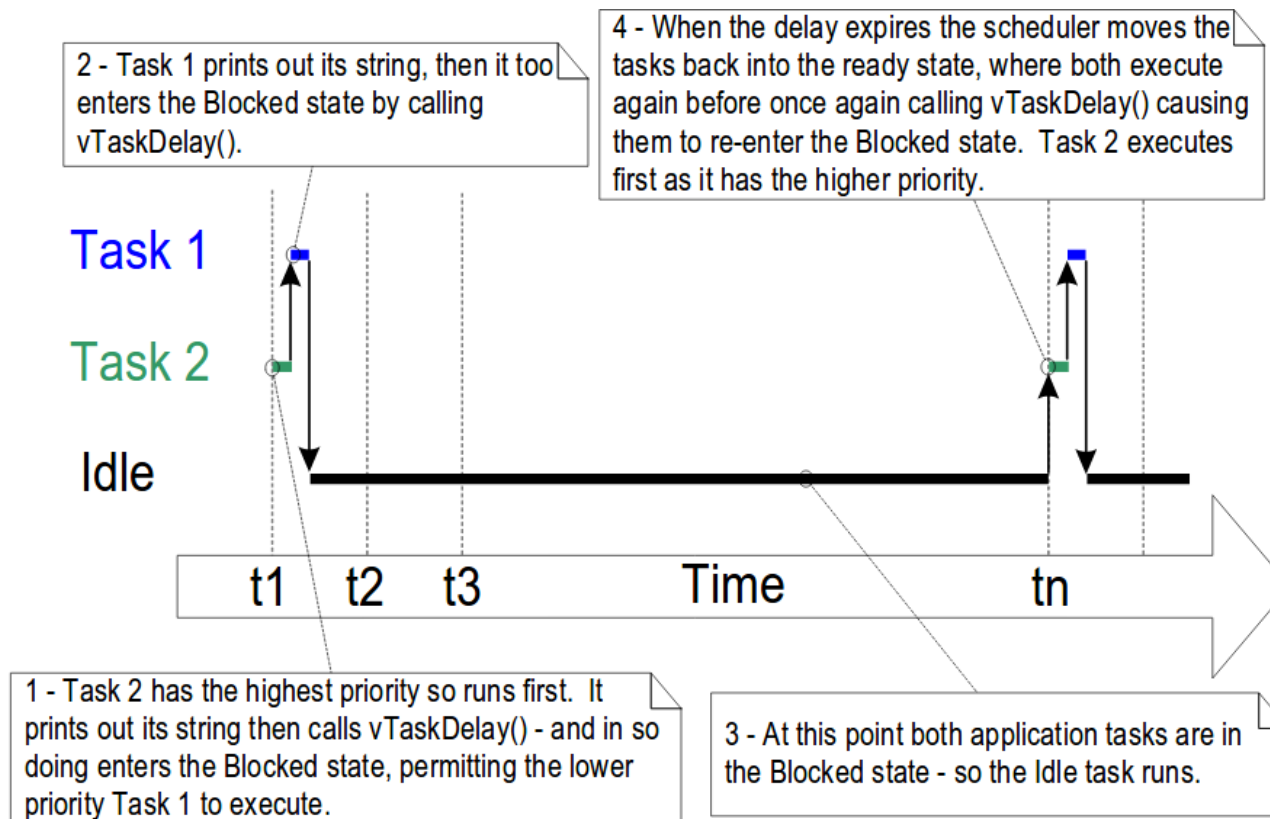The Suspended State is also a sub-state of Not Running.

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

vTaskSuspend()API
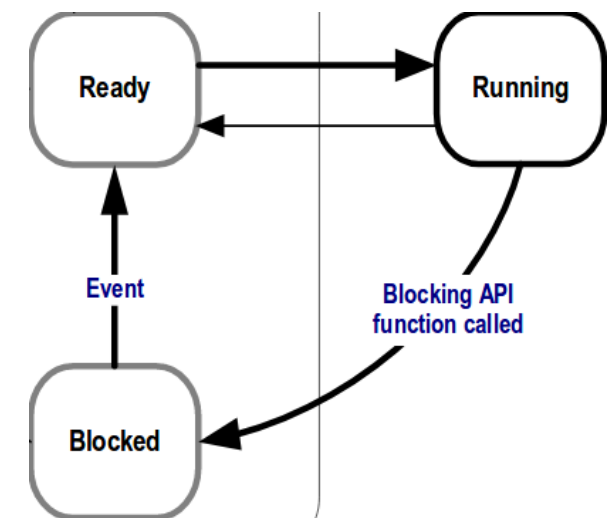
vTaskResume() or xTaskResumeFromISR() API functions.

Most applications do not use the Suspended state.

Expanding the 'Not Running' State

2 - Task 1 prints out its string, then it too enters the Blocked state by calling vTaskDelay().

4 - When the delay expires the scheduler moves the tasks back into the ready state, where both execute again before once again calling vTaskDelay() causing them to re-enter the Blocked state. Task 2 executes first as it has the higher priority.

More efficient implementation of tasks

Less use of processor time

Task 1

Task 2

Idle

t1   t2   t3   Time   tn

1 - Task 2 has the highest priority so runs first. It prints out its string then calls vTaskDelay() - and in so doing enters the Blocked state, permitting the lower priority Task 1 to execute.

3 - At this point both application tasks are in the Blocked state - so the Idle task runs.

Ready

Running

Event

Blocked

Blocking API function called

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

UNSL-San Luis 2017   –   Julio Dondo Gazzano

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,
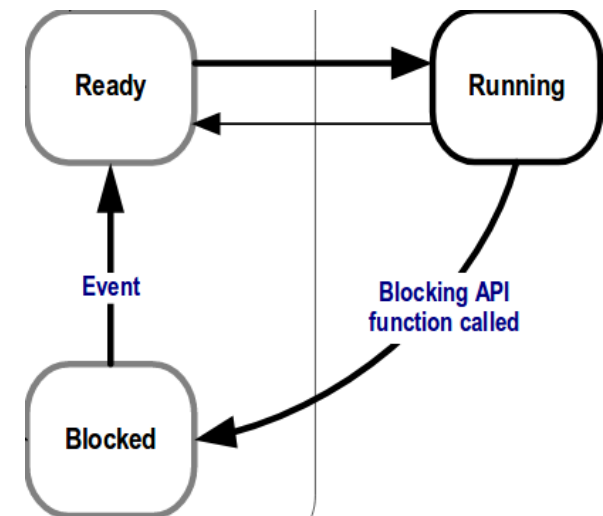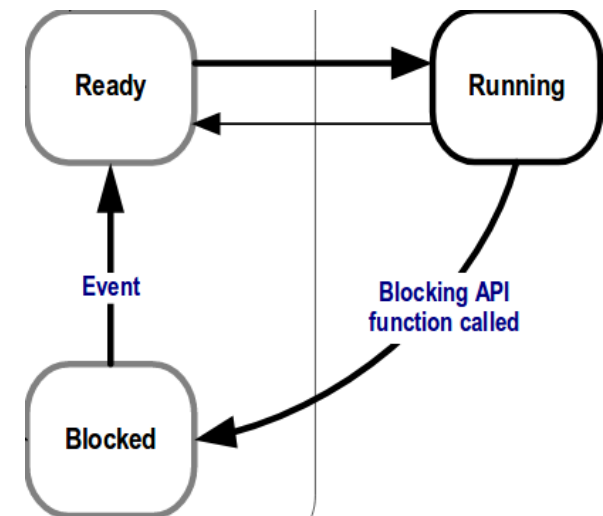
Executing periodic tasks

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 ));
    }
}
```
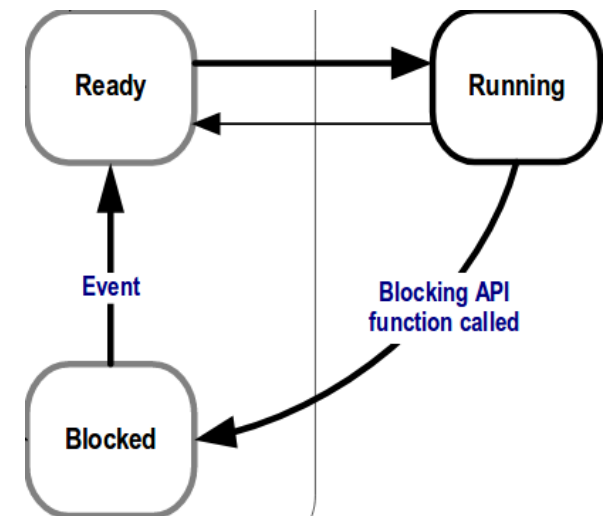

Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

## Combining blocking and non-blocking tasks

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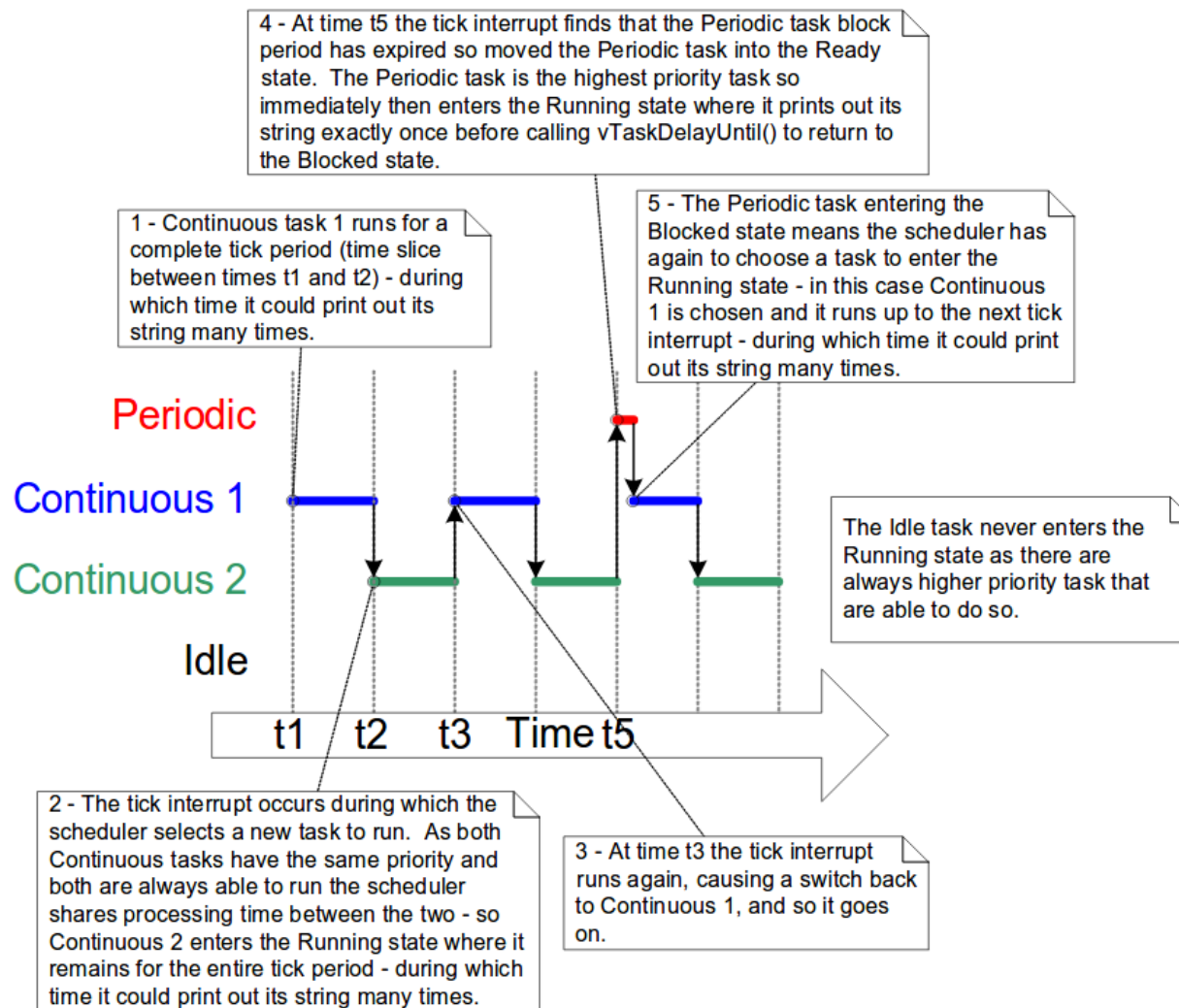
Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

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 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

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
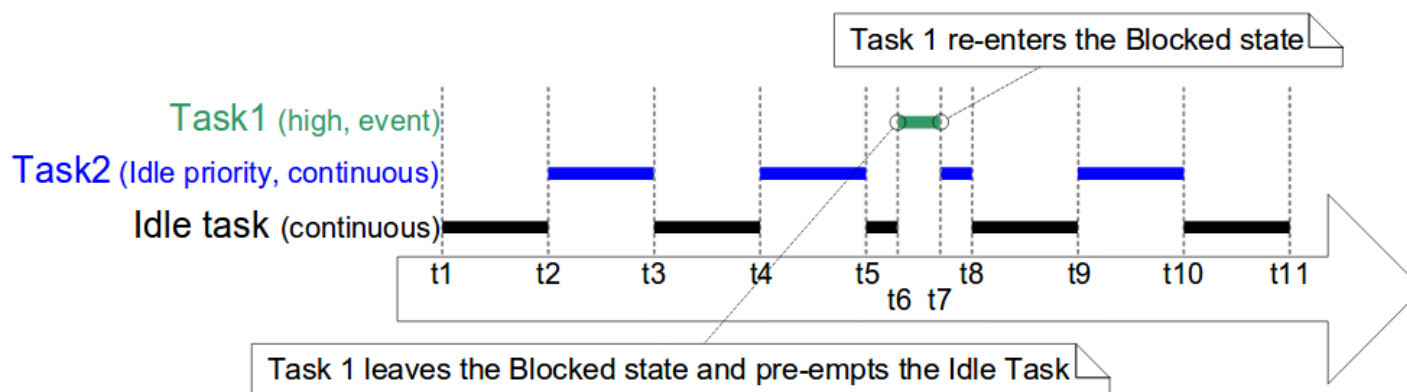
Configured in **FreeRTOSConfig.h**

- configUSE_PREEMPTION   1

- configUSE_TIME_SLICING   1

Task 1 re-enters the Blocked state

Task1 (high, event)
Task2 (Idle priority, continuous)
Idle task (continuous)

t1   t2   t3   t4   t5   t8   t9   t10   t11
t6 t7

To much time for idle task

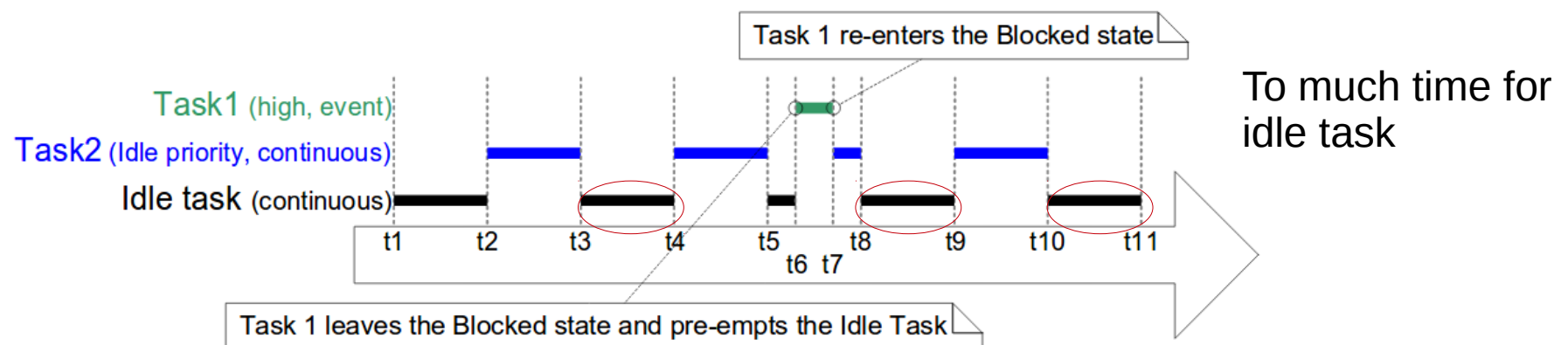Task 1 leaves the Blocked state and pre-empts the Idle Task

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**

Task 2 runs for the remainder of the time slice

Task 1 re-enters the Blocked state, allowing Task 2 to continue running

Task1 (high, event)

Task2 (Idle priority, continuous)

Idle task (continuous)

t1 t2 t3 t4 t5 t8 t9 t10 t11
t6 t7

The Idle task runs for one iteration of its implementing loop, then yields to allow the scheduler to select another task

Task 1 leaves the Blocked state and pre-empts Task 2

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**
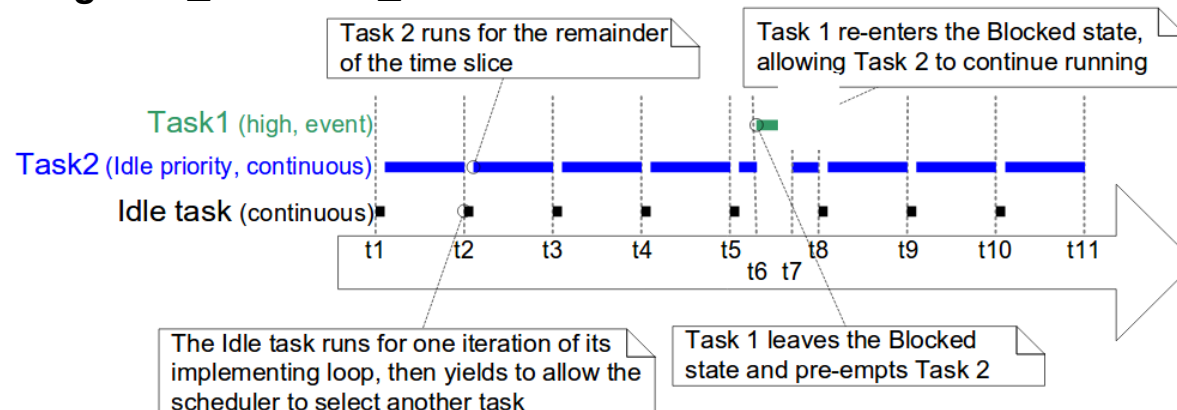
- configUSE_PREEMPTION   1

- 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

Task 1 unblocks when an interrupt writes to a semaphore

Task 1 enters the Blocked state, allowing Task 2 to enter the Running state

Task1 (high priority)
Task2 (medium priority)
Task3 (low priority)

t1   t2   t3   t4 t5   t6

Task 2 unblocks when Task 3 writes to a queue

Task 3 calls taskYIELD(), allowing Task 1 to enter the Running state
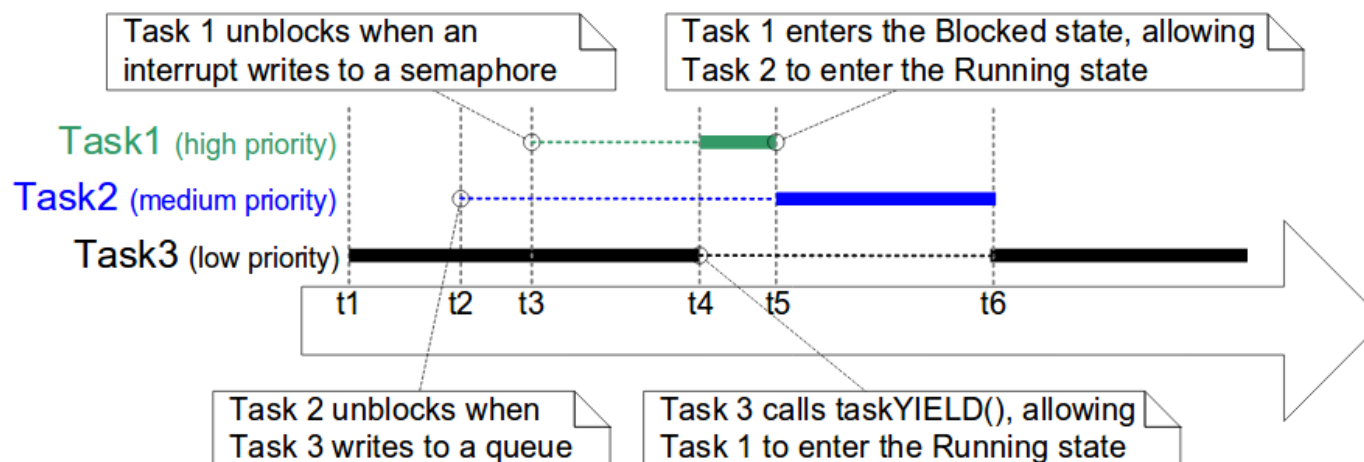
Heap Memory Management

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 ths case the memory is dynamically allocated by the kernel befor the application start, and remain allocated until application finish.
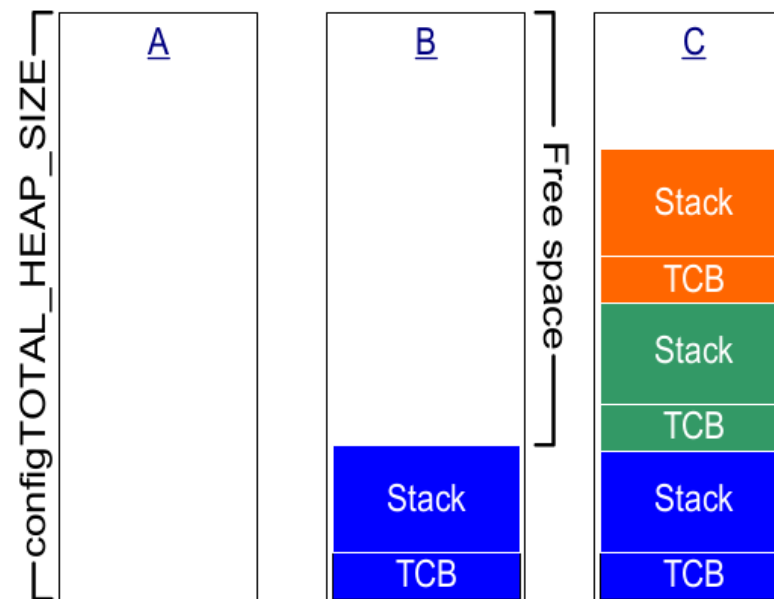
The total memory array size is defined in FreeRTOSConfig.h file with

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.
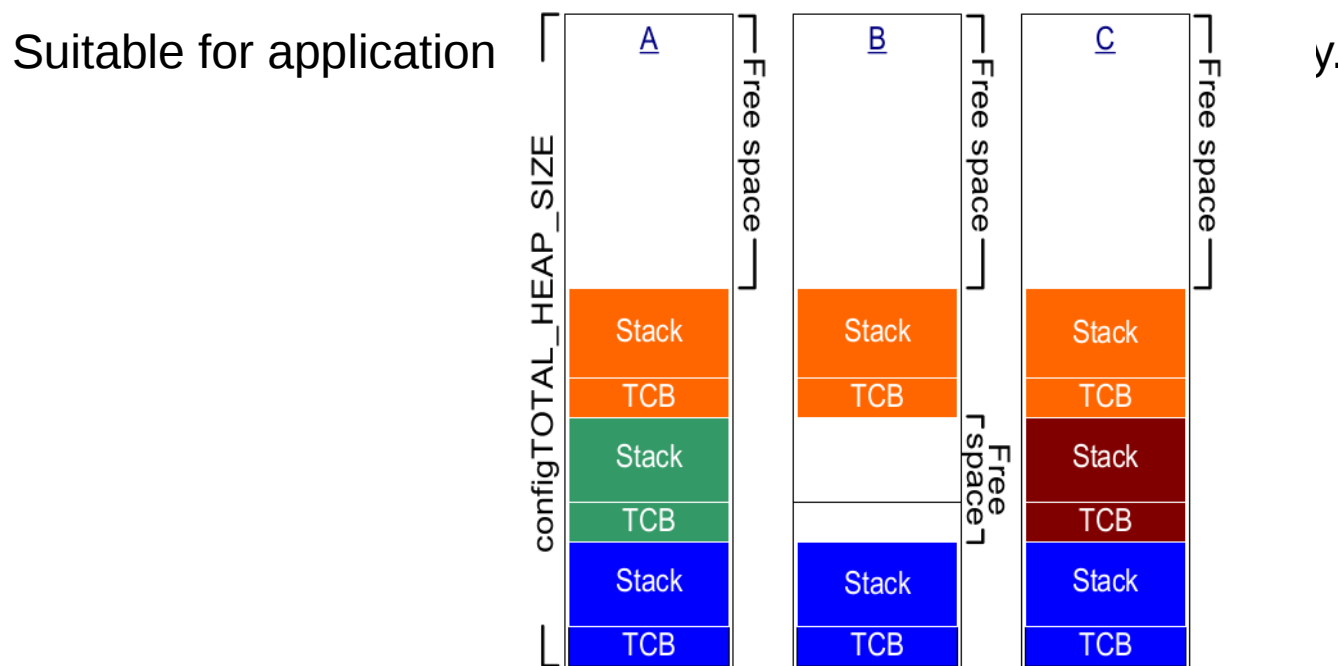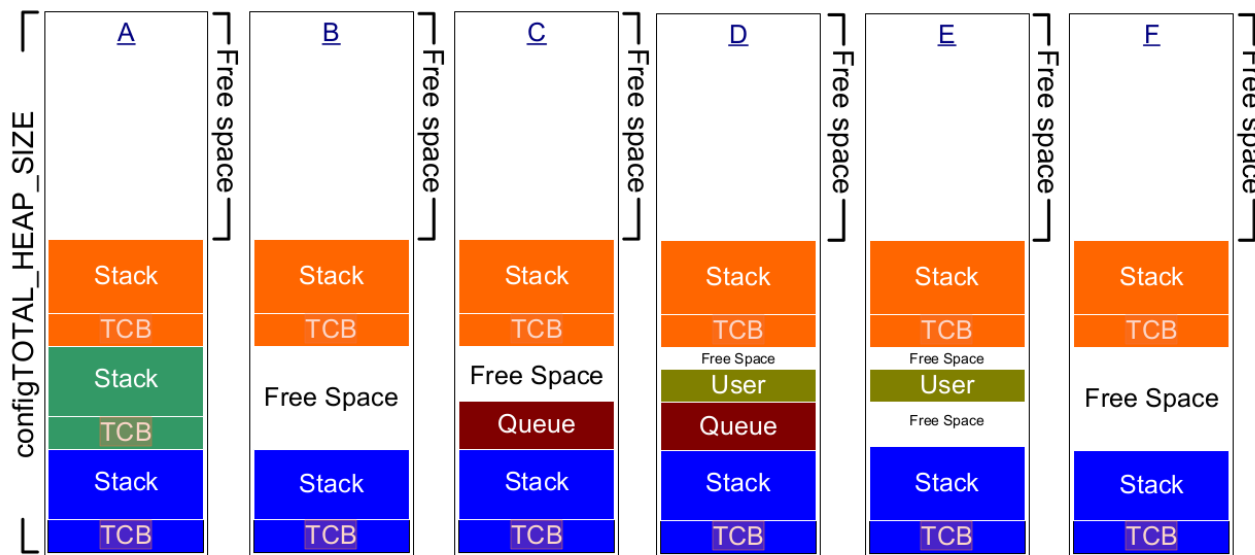

Source:Mastering the FreeRTOS™ Real Time Kernel A Hands-On Tutorial Guide- Richard Barry

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

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

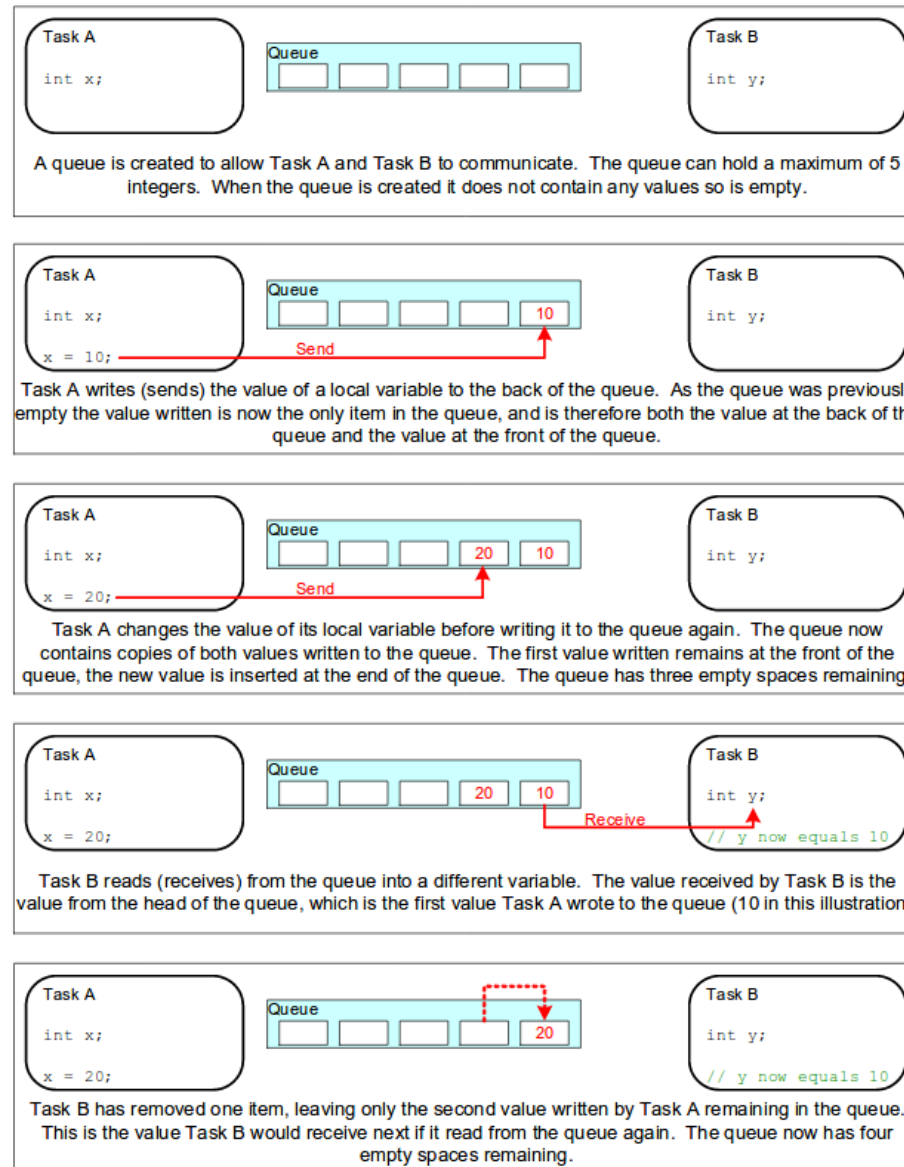- Queues are normally used as First In First Out (FIFO) buffers

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

Queue Management

Task A | Queue | Task B
int x; | | int y;

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 | Queue [| | | | 10] | Task B
int x; | | int y;
x = 10; —— Send

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 | Queue [| | | 20 | 10] | Task B
int x; | | int y;
x = 20; —— Send

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 A | Queue [| | | 20 | 10] | Task B
int x; | | int y;
x = 20; | Receive | // y now equals 10

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 A | Queue [| | | | 20] | Task B
int x; | | int y;
x = 20; | | // y now equals 10

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,   ≡   xQueueSend()
                                                             const void *   pvItemToQueue,
                                                             TickType_t   xTicksToWait);

- 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);

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 );

}
```

QUESTIONS?