**Contents**

[**I.** **OVERVIEW** 2](#_Toc104389940)

[**1.** **Real time operation system – RTOS** 2](#_Toc104389941)

[**1.1.** **What is RTOS** 2](#_Toc104389942)

[**1.2.** **Features of RTOS** 2](#_Toc104389943)

[**1.3.** **How does the RTOS work** 2](#_Toc104389944)

[**1.4.** **Soft real real-time and hard real-time** 3](#_Toc104389945)

[**2.** **Overview about FreeRTOS** 4](#_Toc104389946)

[**2.1.** **Why FreeRTOS** 4](#_Toc104389947)

[**2.2.** **Structure of freeRTOS project** 4](#_Toc104389948)

[**II.** **FREERTOS** 6](#_Toc104389949)

[**1.** **Task** 6](#_Toc104389950)

[**1.1.** **Create a task** 6](#_Toc104389951)

[**1.2.** **State of task** 7](#_Toc104389952)

[**1.3.** **Task priorities** 8](#_Toc104389953)

[**1.4.** **Scheduling tasks** 9](#_Toc104389954)

[**1.5.** **Multitasking** 11](#_Toc104389955)

[**1.6.** **Communication of task** 12](#_Toc104389956)

[**2.** **Interrupt** 12](#_Toc104389957)

[**3.** **Communication and synchronization** 13](#_Toc104389958)

[**3.1.** **Queues** 13](#_Toc104389959)

[**3.2.** **Semaphores** 14](#_Toc104389960)

[**3.3.** **Mutex** 15](#_Toc104389961)

[**4.** **Memmory** 17](#_Toc104389962)

[**4.1.** **Allocate memory** 17](#_Toc104389963)

[**4.2.** **Stack memory** 18](#_Toc104389964)

[**4.3.** **Heap memory** 18](#_Toc104389965)

[**5.** **Software timer** 19](#_Toc104389966)

[**5.1.** **Create software timer** 20](#_Toc104389967)

[**5.2.** **Start and reset software timer** 20](#_Toc104389968)

[**5.3.** **Change period software timer** 21](#_Toc104389969)

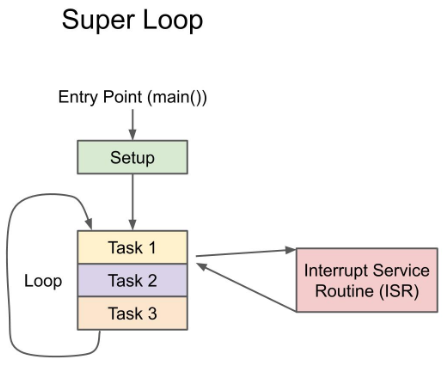
[**6.** **Coding style** 22](#_Toc104389970)

1. **OVERVIEW**
2. **Real time operation system – RTOS**
   1. **What is RTOS**

An operating system- OS is a computer program that supports a computer's basic functions, and provides services to other programs (or applications) that run on the computer. The services provided by the operating system make writing the applications faster, simpler, and more maintainable.

Real Time Operating Systems (RTOSs) have scheduler which designed to provide a predictable execution pattern. This is particularly of interest to embedded systems as embedded systems often have real time requirements. A real time requirements is one that specifies that the embedded system must respond to a certain event within a strictly defined time (the deadline). A guarantee to meet real time requirements can only be made if the behaviour of the operating system's scheduler can be predicted

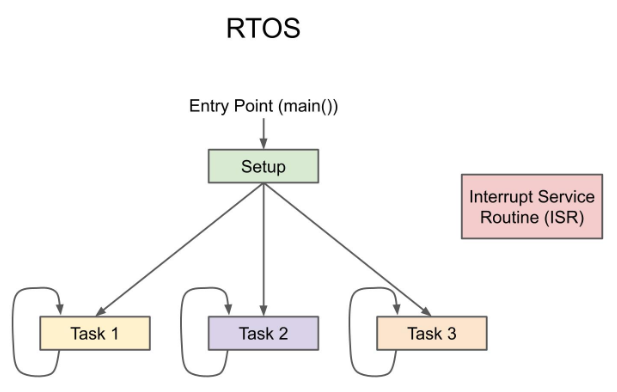
* 1. **Features of RTOS**
* Occupy very less memory
* Consume fewer resources
* Response times are highly predictable
* Unpredictable environment
* The Kernel saves the state of the interrupted task ad then determines which task it should run next.
* The Kernel restores the state of the task and passes control of the CPU for that task.
  1. **How does the RTOS work**

Super loop architecture or bare-metal programming is one of the most popular ways to program a microcontroller. Here, there is no operating system, and the structure is fairly simple there is a main() function where set up any variables, drivers, libraries, etc. and then perform one or more periodic tasks in a while(true) loop

The super loop application is easy to implement and easy to debug. You can even add in interrupts by using interrupt service routines (ISRs) that cause the program to halt and execute some arbitrary code when an external event occurs

If you have multiple tasks to accomplish during the main loop, you generally execute them in a round-robin fashion. The problem comes in when you start adding so many tasks that some start missing deadlines or preventing other features from working. This is where an RTOS can help. Rather than execute everything in a round-robin fashion, you can essentially execute everything concurrently.

You can still have interrupts--they would halt whatever task was being run at the moment to execute the ISR and then return execution to the task.



* 1. **Soft real real-time and hard real-time**

Real-time systems are typically categorized as hard real-time systems or soft real-time systems.

In a hard real-time system, the deadline for generating an output or action must be met otherwise the system fails to perform its intended function. For example, in an aircraft, if the wing control actuators fail to respond to the pilot’s input within the specific time, the aircraft fails to descend for landing.

The time constraints in a soft real-time system are not as harsh. The output result of computation may still be useful even if it does not meet its timing cut-off. The video broadcast, for example, is a soft real-time system. If a video frame is delayed or dropped, the quality of the video is degraded, but the video may still be visible

1. **Overview about FreeRTOS**
   1. **Why FreeRTOS**

* FreeRTOS is a market-leading real--time operating system (RTOS) for microcontrollers and small microprocessors
* FreeRTOS supports more than 40 architectures including highly popular ARM architectures. The FreeRTOS code is written in C programming language that allows easy porting on new or custom architectures.
* FreeRTOS is truly free. It is an MIT-licensed open-source project and can be embedded in commercial products without any requirement to expose the product’s proprietary source code. It has existed since the year 2003 and it is actively maintained by Amazon.
* Provides a single and independent solution for many different architectures and development tools.
* Is feature rich and still undergoing continuous active development.
* Has a minimal ROM, RAM and processing overhead. Typically an RTOS kernel binary image will be in the region of 6K to 12K bytes.
* Is very scalable, simple and easy to use. The core of the RTOS kernel is contained in only 3 C files. The majority of the many files included in the .zip file download relate only to the numerous demonstration applications.
* Is well established with a large and ever growing user base.
* Contains a pre-configured example for each port. No need to figure out how to setup a project - just download and compile!
* Has an excellent, monitored, and active free support forum.
* Has the assurance that commercial support is available should it be required.
* Provides ample documentation.
* FreeRTOS offers a smaller and easier real time processing alternative for applications where eCOS, embedded Linux (or Real Time Linux) and even uCLinux won't fit, are not appropriate, or are not available.
  1. **Structure of freeRTOS project**

The FreeRTOS package can be get from *https://github.com/FreeRTOS/FreeRTOS/releases*

* 1. ***Source files***

As a minimum, the following source files must be included in your project:

* *FreeRTOS/Source/tasks.c*
* *FreeRTOS/Source/queue.c*
* *FreeRTOS/Source/list.c*
* *FreeRTOS/Source/portable/[compiler]/[architecture]/port.c.*
* *FreeRTOS/Source/portable/MemMang/heap\_x.c where 'x' is 1, 2, 3, 4 or 5.*
  1. ***Optional Source Files***
* If you need software timer functionality, then add *FreeRTOS/Source/timers.c* to your project.
* If you need event group functionality, then add *FreeRTOS/Source/event\_groups.c* to your project
* If you need stream buffer or message buffer functionality, then add *FreeRTOS/Source/stream\_buffer.c* to your project.
  1. ***Header Files***

The following directories must be in the compiler's include path (the compiler must be told to search these directories for header files):

* *FreeRTOS/Source/include*
* *FreeRTOS/Source/portable/[compiler]/[architecture].*
* Whichever directory contains the *FreeRTOSConfig.h* file to be used.

Depending on the port, it may also be necessary for the same directories to be in the assembler's include path.

* 1. ***Configuration File***

Every project also requires a file called *FreeRTOSConfig.h. FreeRTOSConfig.h* tailors the RTOS kernel to the application being built. **It is therefore specific to the application, not the RTOS, and should be located in an application directory, not in one of the RTOS kernel source code directories.**

If *heap\_1, heap\_2, heap\_4* or *heap\_5* is included in your project, then the *FreeRTOSConfig.h* definition *configTOTAL\_HEAP\_SIZE* will dimension the FreeRTOS heap. Your application will not link if *configTOTAL\_HEAP\_SIZE* is set too high.

The *FreeRTOSConfig.h* definition *configMINIMAL\_STACK\_SIZE* sets the size of the stack used by the idle task. If *configMINIMAL\_STACK\_SIZE* is set too low, then the idle task will generate stack overflows. It is advised to copy the *configMINIMAL\_STACK\_SIZE* setting from an official FreeRTOS demo provided for the same microcontroller architecture.

To build and run a freeRTOS project, it similar with other C applications.

1. **FREERTOS**
2. **Task**
   1. **Create a task**

A task should have the following structure:

*void vATaskFunction( void \*pvParameters )*

*{*

*for( ;; )*

*{*

*if( WaitForEvent( EventObject, TimeOut ) == pdPASS )*

*{ -- Handle event here. -- }*

*else*

*{ -- Clear errors, or take actions here. -- }*

*}*

*vTaskDelete( NULL );*

*}*

Tasks are created by calling xTaskCreate() or xTaskCreateStatic(), and deleted by calling vTaskDelete().

* **xTaskCreate()**

*BaseType\_t xTaskCreate( TaskFunction\_t pvTaskCode,*

*const char \* const pcName,*

*configSTACK\_DEPTH\_TYPE usStackDepth,*

*void \*pvParameters,*

*UBaseType\_t uxPriority,*

*TaskHandle\_t \*pxCreatedTask*

*);*

*configSUPPORT\_DYNAMIC\_ALLOCATION* must be set to 1 in *FreeRTOSConfig.h*, or left undefined (in which case it will default to 1), for this RTOS API function to be available.

* **xTaskCreateStatic()**

*TaskHandle\_t xTaskCreateStatic( TaskFunction\_t pxTaskCode,*

*const char \* const pcName,*

*const uint32\_t ulStackDepth,*

*void \* const pvParameters,*

*UBaseType\_t uxPriority,*

*StackType\_t \* const puxStackBuffer,*

*StaticTask\_t \* const pxTaskBuffer );*

*configSUPPORT\_STATIC\_ALLOCATION* must be set to 1 in FreeRTOSConfig.h for this RTOS API function to be available.

Each task requires RAM that is used to hold the task state, and used by the task as its stack. If a task is created using xTaskCreate() then the required RAM is automatically allocated from the FreeRTOS heap. If a task is created using xTaskCreateStatic() then the RAM is provided by the application writer, so it can be statically allocated at compile time

* **vTaskDelete()**

*void vTaskDelete( TaskHandle\_t xTask );*

*INCLUDE\_vTaskDelete* must be defined as 1 for this function to be available. See the RTOS Configuration documentation for more information

* 1. **State of task**

A task can exist in one of the following states:

* *Running*

When a task is actually executing it is said to be in the Running state. It is currently utilising the processor. If the processor on which the RTOS is running only has a single core then there can only be one task in the Running state at any given time.

* *Ready*

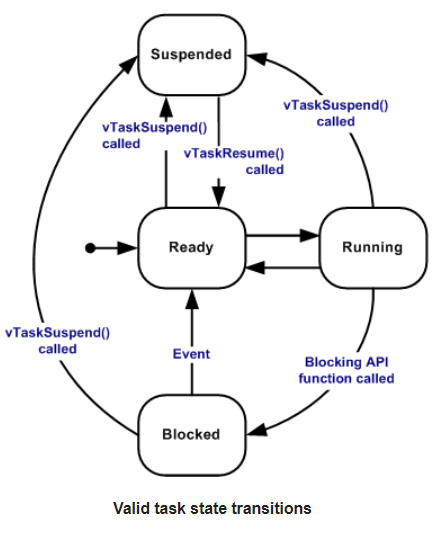
Ready tasks are those that are able to execute (they are not in the Blocked or Suspended state) but are not currently executing because a different task of equal or higher priority is already in the Running state.

* *Blocked*

A task is said to be in the Blocked state if it is currently waiting for either a temporal or external event. For example, if a task calls vTaskDelay() it will block (be placed into the Blocked state) until the delay period has expired - a temporal event. Tasks can also block to wait for queue, semaphore, event group, notification or semaphore event. Tasks in the Blocked state normally have a 'timeout' period, after which the task will be timeout, and be unblocked, even if the event the task was waiting for has not occurred. Tasks in the Blocked state do not use any processing time and cannot be selected to enter the Running state.

* *Suspended*

Like tasks that are in the Blocked state, tasks in the Suspended state cannot be selected to enter the Running state, but tasks in the Suspended state do not have a time out. Instead, tasks only enter or exit the Suspended state when explicitly commanded to do so through the vTaskSuspend() and xTaskResume() API calls respectively

* 1. **Task priorities**

Each task is assigned a priority from 0 to ( configMAX\_PRIORITIES - 1 ), where configMAX\_PRIORITIES is defined within *FreeRTOSConfig.h*.

If the port in use implements a port optimised task selection mechanism that uses a 'count leading zeros' type instruction (for task selection in a single instruction) and *configUSE\_PORT\_OPTIMISED\_TASK\_SELECTION* is set to 1 in *FreeRTOSConfig.h*, then *configMAX\_PRIORITIES* cannot be higher than 32. In all other cases configMAX\_PRIORITIES can take any value within reason - but for reasons of RAM usage efficiency should be kept to the minimum value actually necessary.

Low priority numbers denote low priority tasks. The idle task has priority zero (tskIDLE\_PRIORITY).

The FreeRTOS scheduler ensures that tasks in the Ready or Running state will always be given processor (CPU) time in preference to tasks of a lower priority that are also in the ready state. In other words, the task placed into the Running state is always the highest priority task that is able to run.

Any number of tasks can share the same priority. If *configUSE\_TIME\_SLICING* is not defined, or if *configUSE\_TIME\_SLICING* is set to 1, then Ready state tasks of equal priority will share the available processing time using a time sliced round robin scheduling scheme.

* 1. **Scheduling tasks**

The scheduling algorithm is the software routine that decides which RTOS task should be in the Running state.

* There can only be one task in the Running state per processor core at any given time.
* AMP (asymmetric multicore) is where each processor core *runs its own instance of FreeRTOS.*
* SMP (symmetric multicore) is where there is *one instance of FreeRTOS that schedules RTOS tasks across multiple cores.*

**The default RTOS scheduling policy for Single-core**

By default, FreeRTOS uses a fixed-priority preemptive scheduling policy, with round-robin time-slicing of equal priority tasks:

* "**Fixed priority**" means the scheduler will not permanently change the priority of a task, although it may temporarily boost the priority of a task due to priority inheritance.
* "**Preemptive**" means the scheduler always runs the highest priority RTOS task that is able to run, regardless of when a task becomes able to run. For example, if an interrupt service routine (ISR) changes the highest priority task that is able to run, the scheduler will stop the currently running lower priority task and start the higher priority task - even if that occurs within a time slice. In this case, the lower priority task is said to have been "preempted" by the higher priority task.
* "**Round-robin**" means tasks that share a priority take turns entering the Running state.
* "**Time sliced**" means the scheduler will switch between tasks of equal priority on each tick interrupt - the time between tick interrupts being one time slice. (The tick interrupt is the periodic interrupt used by the RTOS to measure time.)

We should use a prioritised preemptive scheduler to avoiding task starvation. A consequence of always running the highest priority task that is able to run is that a high priority task that never enters the Blocked or Suspended state will permanently starve all lower priority tasks of any execution time. That is one reason why, normally, it is best to create tasks that are event-driven.

* **Configuring the RTOS scheduling policy**

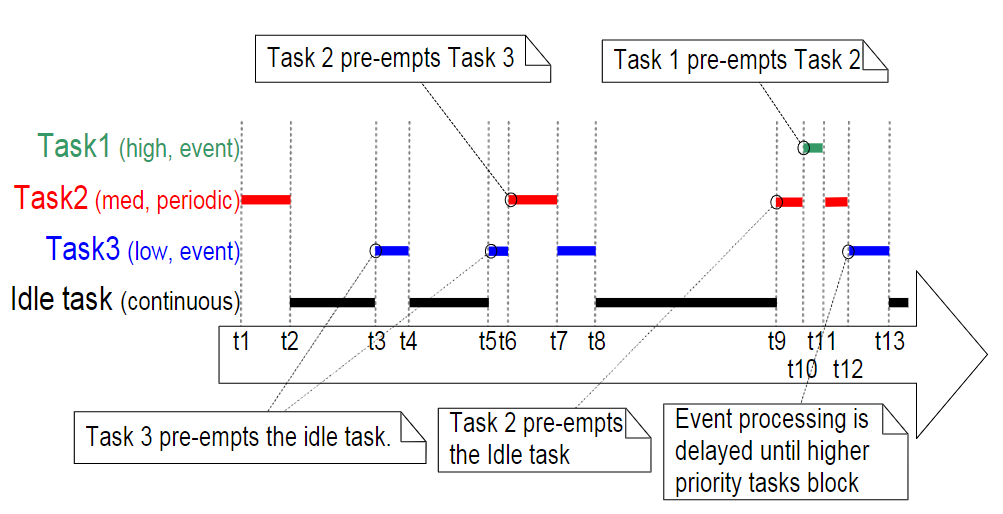
The following *FreeRTOSConfig.h* settings change the default scheduling behaviour:

* *configUSE\_PREEMPTION*
* If *configUSE\_PREEMPTION* is 0 then preemption is off and a context switch will only occur if the Running state task enters the Blocked or Suspended state, the Running state task calls taskYIELD(), or an interrupt service routine (ISR) manually requests a context switch.
* *configUSE\_TIME\_SLICING*

If *configUSE\_TIME\_SLICING* is 0 then time slicing is off, so the scheduler will not switch between equal priority tasks on each tick interrupt.

* **Example Prioritized Pre-emptive Scheduling with Time Slicing**

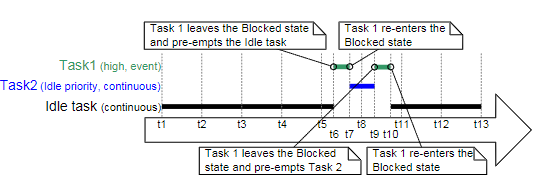
The configuration:

* *configUSE PREEMPTION* set to 1
* *****configUSE TIME SLICING* set to 1
* **Example Prioritized Pre-emptive Scheduling (without Time Slicing)**

The configuration:

* *configUSE PREEMPTION* set to 1
* *configUSE TIME SLICING* set to 0

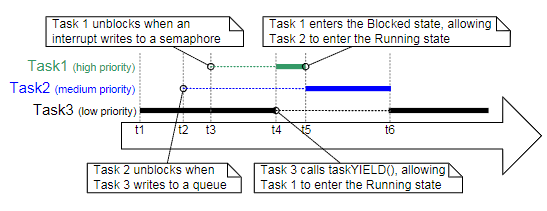
If time slicing is not used, then the scheduler will only select a new task to enter the Running state when either:

* A higher priority task enters the Ready state
* The task in the Running state enters the Blocked or Suspended state
* **Example Co-operative Scheduling**

The configuration:

* *configUSE PREEMPTION* set to 0
* *configUSE TIME SLICING* set to any value

When the co-operative scheduler is used, a context switch will only occur when the Running state task enters the Blocked state, or the Running state task explicitly yields (manually requests a re-schedule)

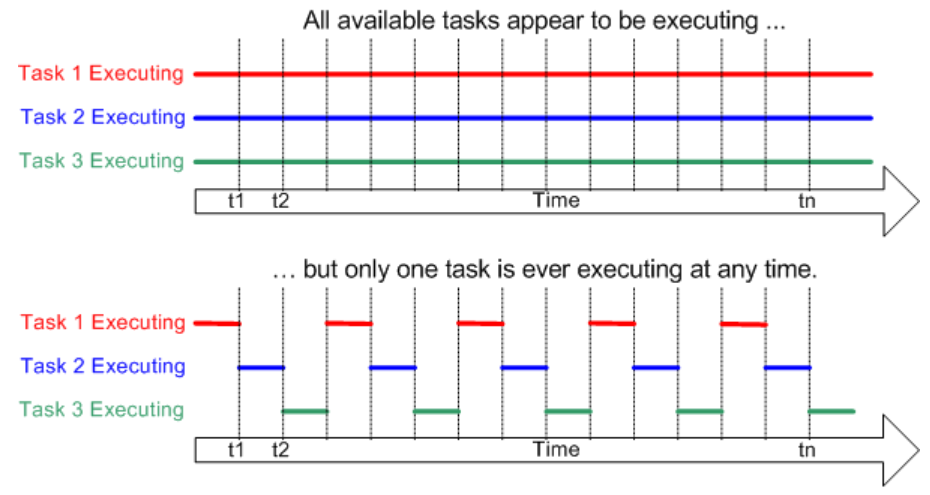
by calling taskYIELD(). Tasks are never pre-empted, so time slicing cannot be used.

* 1. **Multitasking**

Each executing program is a task (or thread) under control of the operating system. If an operating system can execute multiple tasks in this manner it is said to be multitasking.

The use of a multitasking operating system can simplify the design of what would otherwise be a complex software application:

* The multitasking and inter-task communications features of the operating system allow the complex application to be partitioned into a set of smaller and more manageable tasks.
* The partitioning can result in easier software testing, work breakdown within teams, and code reuse.
* Complex timing and sequencing details can be removed from the application code and become the responsibility of the operating system.

A conventional processor can only execute a single task at a time - but by rapidly switching between tasks a multitasking operating system can make it appear as if each task is executing concurrently.

* 1. **Communication of task**

Inter-task Communication can be one of following

* RTOS Task Notifications
* Stream and Message Buffers
* Queues
* Binary Semaphores
* Counting Semaphores
* Mutexes
* Recursive Mutexes

1. **Interrupt**

Interrupts have higher priorities than other Tasks. Therefore, it Interrupts should not wait for a mutex, semaphore, and other resources and should be executed as soon as it occurs. Otherwise, it may cause issues. Defer processing of interrupts through other tasks is a possible countermeasure to minimize the processing time of ISR as soon as possible.

if a variable (such as a global) is updated inside an ISR, it should declare with the “volatile” qualifier. This lets the compiler know that the “volatile” variable can change outside the current thread of execution.

* Interrupt handler is not directly invoked by FreeRTOS Kernel, porting layer handles this

In theory, FreeRTOS can work without interrupts. To switch tasks, interrupts are not needed. However, that’s not very useful.

* Porting layer does below
* Setup (any) one timer to be configured at configTICK\_RATE\_HZ, typically 1ms
* Timer ISR is in porting layer
* When the timer ISR happens the porting layer calls xTaskIncrementTick() FreeRTOS API to maintain the FreeRTOS timer tick state
* Porting layer also implements the common interrupt entry and exit logic
* Porting layer also implements functions to protect the critical section of FreeRTOS via the below APIs:
* *portENTER\_CRITICAL*, disable interrupt, track nesting of calls
* *portEXIT\_CRITICAL*, reenable interrupt, if nesting call depth is 0
* *portSET\_INTERRUPT\_MASK\_FROM\_ISR*, interrupt disable and return old interrupt state
* *portCLEAR\_INTERRUPT\_MASK\_FROM\_ISR*, restore interrupt state

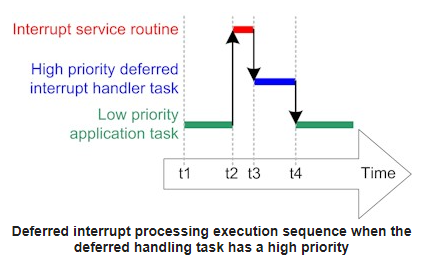
**Handle interrupt**

There are 2 types of handling interrupt:

* ***Standard ISR Processing***

Standard ISR processing will typically involve recording the reason for the interrupt, clearing the interrupt, then performing any processing necessitated by the interrupt, all within the ISR itself.

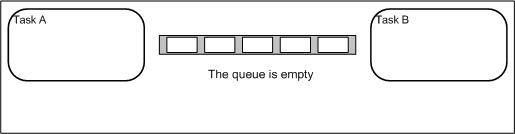
* ***Deferred Interrupt Processing***

Deferred interrupt processing will typically involve recording the reason for the interrupt and clearing the interrupt within the ISR, but then unblocking an RTOS task so the processing necessitated by the interrupt can be performed by the unblocked task, rather than within the ISR.

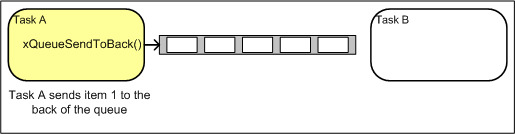
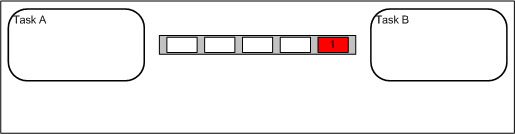
An application will only benefit from deferring interrupt processing if the processing:

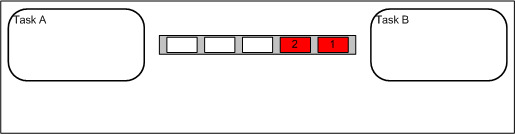
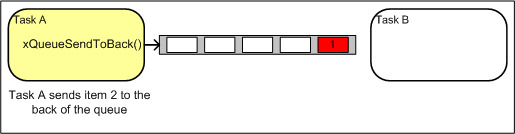
* Needs to perform lengthy operations, or
* Would benefit from using the full RTOS API, rather than just the ISR safe API, or
* Needs to perform an action that is not deterministic, within reasonable bounds.

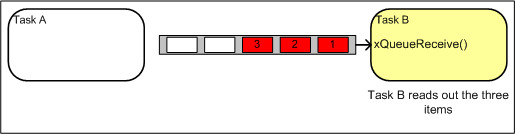
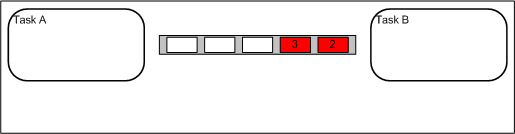
1. **Communication and synchronization**
   1. **Queues**

Queues are the primary form of inter-task communications. They can be used to send messages between tasks, and between interrupts and tasks. In most cases they are used as thread safe FIFO (First In First Out) buffers with new data being sent to the back of the queue, although data can also be sent to the front.

(1)



(2) (3)

 (4) (5)

(6) (7)

* **Blocking on Queues**

Queue API functions permit a block time to be specified.

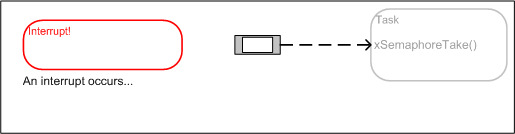
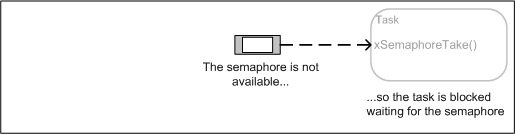
* When a task attempts to read from an empty queue the task will be placed into the Blocked state (so it is not consuming any CPU time and other tasks can run) until either data becomes available on the queue, or the block time expires.
* When a task attempts to write to a full queue the task will be placed into the Blocked state (so it is not consuming any CPU time and other tasks can run) until either space becomes available in the queue, or the block time expires.
* If more than one task block on the same queue then the task with the highest priority will be the task that is unblocked first.
  1. **Semaphores**

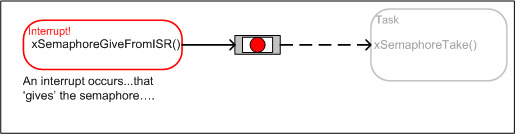
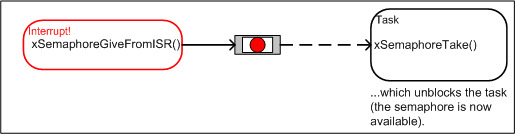
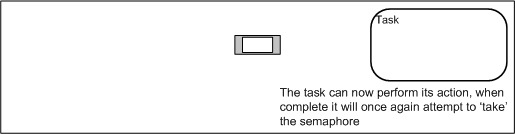
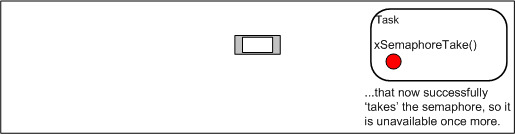
There are 2 types of FreeRTOS semaphores: **Binary Semaphores** and **Counting Semaphores**

* 1. **Binary Semaphores**

Binary semaphores are used for both mutual exclusion and synchronisation purposes.

Binary semaphores and mutexes are very similar but have some subtle differences:

* Mutexes include a priority inheritance mechanism, binary semaphores do not. This makes binary semaphores the better choice for implementing synchronisation (between tasks or between tasks and an interrupt)
* ****Mutexes is the better choice than binary semaphores for implementing simple mutual exclusion.

1. **** (2)
2. **** (4)

(5) (6)

* 1. **Counting Semaphores**

Just as binary semaphores can be thought of as queues of length one, counting semaphores can be thought of as queues of length greater than one (users of the semaphore are not interested in the data that is stored in the queue - just whether or not the queue is empty or not)

Counting semaphores are typically used for two things:

* **Counting events.**

An event handler will 'give' a semaphore each time an event occurs (incrementing the semaphore count value), and a handler task will 'take' a semaphore each time it processes an event (decrementing the semaphore count value). The count value is therefore the difference between the number of events that have occurred and the number that have been processed. In this case it is desirable for the count value to be zero when the semaphore is created.

* **Resource management.**

In this usage scenario the count value indicates the number of resources available. To obtain control of a resource a task must first obtain a semaphore - decrementing the semaphore count value. When the count value reaches zero there are no free resources. When a task finishes with the resource it 'gives' the semaphore back - incrementing the semaphore count value. In this case it is desirable for the count value to be equal the maximum count value when the semaphore is created.

* 1. **Mutex**

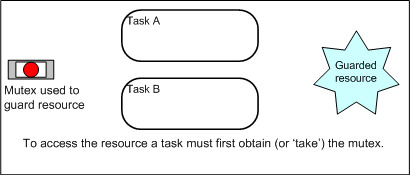
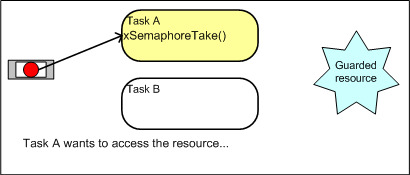
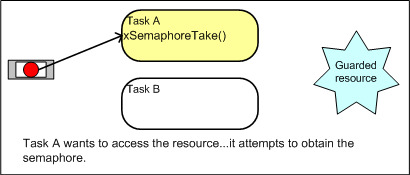
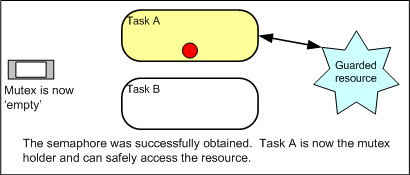
Mutexes are binary semaphores that include a priority inheritance mechanism. Whereas binary semaphores are the better choice for implementing synchronisation (between tasks or between tasks and an interrupt), mutexes are the better choice for implementing simple mutual exclusion

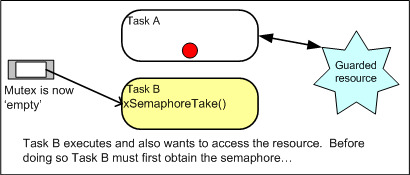
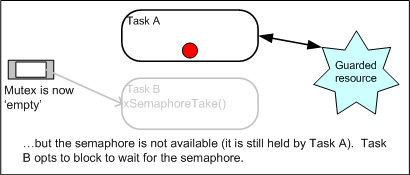
When used for mutual exclusion the mutex acts like a token that is used to guard a resource. When a task wishes to access the resource it must first obtain ('take') the token. When it has finished with the resource it must 'give' the token back - allowing other tasks the opportunity to access the same resource.

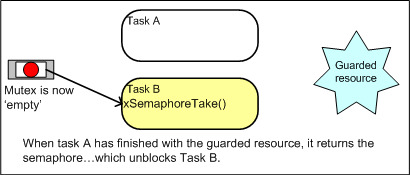
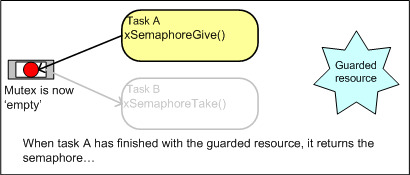
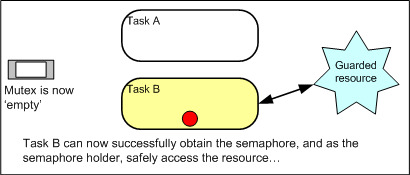
Unlike binary semaphores however - mutexes employ priority inheritance. This means that if a high priority task blocks while attempting to obtain a mutex (token) that is currently held by a lower priority task, then the priority of the task holding the token is temporarily raised to that of the blocking task. This mechanism is designed to ensure the higher priority task is kept in the blocked state for the shortest time possible, and in so doing minimise the 'priority inversion' that has already occurred.

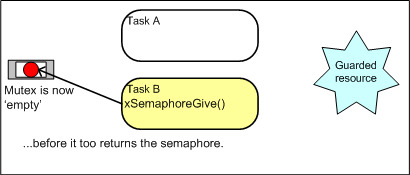
Mutexes should not be used from an interrupt because:

* They include a priority inheritance mechanism which only makes sense if the mutex is given and taken from a task, not an interrupt.
* An interrupt cannot block to wait for a resource that is guarded by a mutex to become available.

1.  (2)

 (3) (4)

 (5) (6)

 (7) (8)

(9) (10)

Above is process using a mutex to guard access to a shared resource

1. **Memmory**
   1. **Allocate memory**

FreeRTOS V9.0.0 (latest version is v10.4.0) and onwards gives the application writer the ability to instead provide the memory themselves, allowing the following objects to optionally be created without any memory being allocated dynamically:

* Tasks
* Software Timers
* Queues
* Event Groups
* Binary Semaphores
* Counting Semaphores
* Recursive Semaphores
* Mutexes

Whether it is preferable to use static or dynamic memory allocation is dependent on the application, and the preference of the application writer. Both methods have pros and cons, and both methods can be used within the same RTOS application.

* ***Creating an RTOS Object Using Dynamically Allocated RAM***

Creating RTOS objects dynamically has the benefit of greater simplicity, and the potential to minimise the application's maximum RAM usage:

* Fewer function parameters are required when an object is created.
* The memory allocation occurs automatically, within the RTOS API functions.
* The application writer does not need to concern themselves with allocating memory themselves.
* The RAM used by an RTOS object can be re-used if the object is deleted, potentially reducing the application's maximum RAM footprint.
* RTOS API functions are provided to return information on heap usage, allowing the heap size to be optimised.
* The memory allocation scheme used can be chosen to best suite the application, be that *heap\_1.c* for simplicity and determinism often necessary for safety critical applications, *heap\_4.c* for fragmentation protection, heap\_5.c to split the heap across multiple RAM regions, or an allocation scheme provided by the application writer themselves.

We must set *configSUPPORT\_DYNAMIC\_ALLOCATION t*o 1 to using this method.

* ***Creating an RTOS Object Using Statically Allocated RAM***

Creating RTOS objects using statically allocated RAM has the benefit of providing the application writer with more control:

* RTOS objects can be placed at specific memory locations.
* The maximum RAM footprint can be determined at link time, rather than run time.
* The application writer does not need to concern themselves with graceful handling of memory allocation failures.
* It allows the RTOS to be used in applications that simply don't allow any dynamic memory allocation (although FreeRTOS includes allocation schemes that can overcome most objections).

We must set *configSUPPORT\_STATIC\_ALLOCATION* to 1 to using this method.

* 1. **Stack memory**

Each task maintains its own stack. **Stack overflow** is a very common cause of application instability. FreeRTOS provides two optional mechanisms that can be used to assist in the detection and correction of just such an occurrence. The application must provide a stack overflow hook function if *configCHECK\_FOR\_STACK\_OVERFLOW* is not set to 0. The hook function must be called *vApplicationStackOverflowHook()*.

Stack overflow checking introduces a context switch overhead so its use is only recommended during the development or testing phases.

* **Method 1**
* It is likely that the stack will reach its greatest (deepest) value ***after the RTOS kernel has swapped the task out of the Running state*** because this is when the stack will contain the task context. At this point the RTOS kernel can check that the processor stack pointer remains within the valid stack space. The stack overflow hook function is called if the stack pointer contain a value that is outside of the valid stack range.
* This method is ***quick but not guaranteed*** to catch all stack overflows.
* Set *configCHECK\_FOR\_STACK\_OVERFLOW* to 1 to use this method.
* **Method 2**
* When a task is first created its stack is filled with a known value.
* When swapping a task out of the Running state the RTOS kernel can check the last 16 bytes within the valid stack range to ensure that these known values have not been overwritten by the task or interrupt activity.
* The stack overflow hook function is called should any of these 16 bytes not remain at their initial value.
* This method is less efficient than **Method 1**, but still fairly fast. It is very likely to catch stack overflows but is still not guaranteed to catch all overflows.
* Set *configCHECK\_FOR\_STACK\_OVERFLOW* to 2 to use this method.
  1. **Heap memory**

The RTOS kernel needs RAM each time a task, queue, mutex, software timer, semaphore or event group is created

If RTOS objects are created dynamically then the standard C library *malloc()* and *free()* functions can sometimes be used for the purpose, but they have some limitations

* They are not always available on embedded systems
* They take up valuable code space
* They are not thread safe
* They are not deterministic (the amount of time taken to execute the function will differ from call to call)

One embedded / real time system can have very different RAM and timing requirements to another - so a single RAM allocation algorithm will only ever be appropriate for a subset of applications.

To get around this problem, FreeRTOS keeps the memory allocation API in its portable layer. The portable layer is outside of the source files that implement the core RTOS functionality, allowing an application specific implementation appropriate for the real time system being developed to be provided.

When the RTOS kernel requires RAM, instead of calling *malloc(),* it instead calls *pvPortMalloc().* When RAM is being freed, instead of calling *free(),* the RTOS kernel calls *vPortFree().*

The FreeRTOS provided 5 implementations which contained in a 5 source files (**heap\_1.c, heap\_2.c, heap\_3.c, heap\_4.c and heap\_5.c**). They are located in the *Source/Portable/MemMang* directory of the main RTOS source code download

* **heap\_1** - the very simplest, does not permit memory to be freed. This implementation is less useful
* **heap\_2** - permits memory to be freed, but does not coalescence adjacent free blocks. This implementation is now considered legacy as the newer **heap\_4** is preferred.
* **heap\_3** - simply wraps the standard *malloc()* and *free()* for thread safety.
* **heap\_4** - coalescences adjacent free blocks to avoid fragmentation. Includes absolute address placement option.
* **heap\_5** - as per **heap\_4**, with the ability to span the heap across multiple non-adjacent memory

1. **Software timer**

Timers (in embedded systems) allow us to delay the execution of some function or execute a function periodically. These can be hardware timers, which are unique to the architecture, or software timers that are based on some running code or the RTOS tick timer.

A software timer allows a function to be executed at a set time in the future.The function executed by the timer is called the timer’s callback function.The time between a timer being started and its callback function being executed is called the timer’s period.The FreeRTOS kernel provides an efficient software timer implementation because:

* It does not execute timer callback functions from an interrupt context.
* It does not consume any processing time unless a timer has actually expired.
* It does not add any processing overhead to the tick interrupt.
* It does not walk any link list structures while interrupts are disabled

In FreeRTOS, there are a few ways to delay the execution of a function:

* vTaskDelay() allows us block the currently running task for a set amount of time (given in ticks).
* You can also perform a non-blocking delay by comparing xTaskGetTickCount() with some known timestamp
* Many microcontrollers (and microprocessors) include one or more hardware timers. These can be configured (often by setting various registers) to count up or down and trigger an interrupt service routine (ISR) when they expire (or reach a particular number).
* Software timers exist in code and are not hardware dependent (except for the fact that the RTOS tick timer usually relies on a hardware timer).

The *configUSE\_TIMERS* configuration constant must be set to 1 for xTimerStart() to be available.

* 1. **Create software timer**

When you include the FreeRTOS timer library, it will automatically run a timer service task (also known as a “timer daemon”) separately from all your other tasks. This service task is in charge of managing all of the timers you set and calling the various callback functions. API function calls communicate with the timer service task through a queue to ensure that all commands are received.

When a timer is created, you assign a function (a “callback function”) that is called whenever the timer expires. Note that timers are dependent on the tick timer, which means you can never create a timer with less resolution than the tick (1 ms by default). Additionally, you can set the software timer to be “one-shot” (executes the callback function once after the timer expires) or “auto-reload” (executes the callback function periodically every time the timer expires).

* 1. **Start and reset software timer**

1. **Start software timer**

* xTimerStart() starts a timer that was previously created using the xTimerCreate() API function.
* If the timer had already been started and was already in the active state, then xTimerStart() has equivalent functionality to the xTimerReset() API function.

Starting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n 'ticks after xTimerStart() was called, where 'n' is the timers defined period.

*BaseType\_t xTimerStart( TimerHandle\_t xTimer,*

*TickType\_t xBlockTime );*

**Parameters**:

* ***xTimer***: The handle of the timer being started/restarted.
* ***xBlockTime***: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the start command to be successfully sent to the timer command queue, should the queue already be full when xTimerStart() was called. xBlockTime is ignored if xTimerStart() is called before the RTOS scheduler is started.

**Returns:**

* ***pdFAIL*** will be returned if the start command could not be sent to the timer command queue even after xBlockTime ticks had passed.
* ***pdPASS*** will be returned if the command was successfully sent to the timer command queue.

1. **Reset software timer**

* xTimerReset() re-starts a timer that was previously created using the xTimerCreate() API function. I
* If the timer had already been started and was already in the active state, then xTimerReset() will cause the timer to re-evaluate its expiry time so that it is relative to when xTimerReset() was called.
* If the timer was in the dormant state then xTimerReset() has equivalent functionality to the xTimerStart() API function.

Resetting a timer ensures the timer is in the active state. If the timer is not stopped, deleted, or reset in the mean time, the callback function associated with the timer will get called 'n' ticks after xTimerReset() was called, where 'n' is the timers defined period.

*BaseType\_t xTimerReset( TimerHandle\_t xTimer,*

*TickType\_t xBlockTime );*

* 1. **Change period software timer**

In real time operating systems, we use software timers to perform some task within a specified rate using a callback function. The rate of callback of function execution is determined by the period of the software timer. In the last tutorial, we have created software timers using FreeRTOS xTimerCreate() API function

* FreeRTOS provide API function to change software timer period during run-time that is xTimerChangePeriod() API Function.
* xTimerChangePeriod() changes the period of a timer that was previously created using the xTimerCreate() API function.
* If the timer is in running state, then the timer will expire according to its new period.
* If the timer is not running, it will recalculate its period and enter the running state.
* The *configUSE\_TIMERS* configuration constant must be set to 1 for xTimerChangePeriod() to be available

*BaseType\_t xTimerChangePeriod( TimerHandle\_t xTimer,*

*TickType\_t xNewPeriod,*

*TickType\_t xBlockTime );*

**Parameters:**

* ***xTimer***: The handle of the timer that is having its period changed.
* ***xNewPeriod***: The new period for xTimer.
* ***xBlockTime***: Specifies the time, in ticks, that the calling task should be held in the Blocked state to wait for the change period command to be successfully sent to the timer command queue, should the queue already be full when xTimerChangePeriod() was called. xBlockTime is ignored if xTimerChangePeriod() is called before the RTOS scheduler is started.

1. **Coding style**
2. **Naming conventions**

* Variables
* Variables of type uint32\_t are prefixed ul, where the 'u' denotes 'unsigned' and the 'l' denotes 'long'.
* Variables of type uint16\_t are prefixed us, where the 'u' denotes 'unsigned' and the 's' denotes 'short'.
* Variables of type uint8\_t are prefixed uc, where the 'u' denotes 'unsigned' and the 'c' denotes 'char'.
* Variables of non stdint types are prefixed x. Examples include BaseType\_t and TickType\_t, which are portable layer defined typedefs for the natural or most efficient type for the architecture and the type used to hold the RTOS tick count respectively.
* Unsigned variables of non stdint types have an additional prefix u. For example variables of type UBaseType\_t (unsigned BaseType\_t) are prefixed ux.
* Variables of type size\_t are also prefixed x.>
* Enumerated variables are prefixed e
* Pointers have an additional prefixed p, for example a pointer to a uint16\_t will have prefix pus.
* In line with MISRA guides, unqualified standard char types are only permitted to hold ASCII characters and are prefixed c.
* In line with MISRA guides, variables of type char \* are only permitted to hold pointers to ASCII strings and are prefixed pc.
* Functions
* File scope static (private) functions are prefixed with prv.
* API functions are prefixed with their return type, as per the convention defined for variables, with the addition of the prefix v for void.
* API function names start with the name of the file in which they are defined. For example vTaskDelete is defined in tasks.c, and has a void return type.
* Macros
* Macros are pre-fixed with the file in which they are defined. The pre-fix is lower case. For example, configUSE\_PREEMPTION is defined in FreeRTOSConfig.h.
* Other than the pre-fix, macros are written in all upper case, and use an underscore to separate words.

1. **Data Types**

* Char :In line with MISRA guides, unqualified char types are permitted, but only when they are used to hold ASCII characters.
* char \*: In line with MISRA guides, unqualified character pointers are permitted, but only when they are used to point to ASCII strings.
* TickType\_t: If configUSE\_16\_BIT\_TICKS is set to non-zero (true), then TickType\_t is defined to be an unsigned 16-bit type. If configUSE\_16\_BIT\_TICKS is set to zero (false), then TickType\_t is defined to be an unsigned 32-bit type.
* BaseType\_t: This is defined to be the most efficient, natural, type for the architecture.
* UBaseType\_t: This is an unsigned BaseType\_t.
* StackType\_t: Defined to the type used by the architecture for items stored on the stack. Normally this would be a 16-bit type on 16-bit architectures and a 32-bit type on 32-bit architectures, although there are some exceptions. Used internally by FreeRTOS.