**FreeRTOS**

* very basic handle of tasks and memory management.
* nothing is provided for network communication, drivers for external hardware, or access to a filesystem.
* preemptive tasks.
* a support for 23 microcontroller architectures.
* compiled with various C compiler (some ports are compiled with gcc, others with openwatcom or borland c++).

1. **Tasks** :

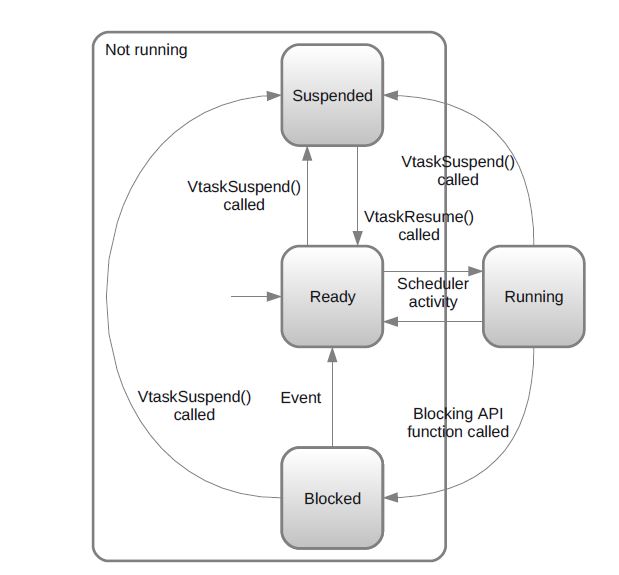
* It allows an unlimited number of tasks to run at the same time and no limitation about their priorities as long as used hardware can afford it.
* It can handle both cyclic and acyclic tasks.
* Functions to manage tasks :

1- Task creation & destruction

1. Priority management
2. Delay/Resume

* States of a task:

A task is either running or not running. Not running can be either ready ( which is a task that can run but is waiting for the processor to be available ) or blocked ( a task that is delayed or waiting for another task due to synchronization issues) or suspended.



* Creation of a task :

Using vTaskCreate(). This function takes as argument the following list:

* pvTaskCode: a pointer to the function where the task is implemented.
* pcName: given name to the task.
* usStackDepth: length of the stack for this task in words.
* pvParameters: a pointer to arguments given to the task.
* uxPriority: priority given to the task, a number between 0 and MAX\_PRIORITIES – 1.
* pxCreatedTask: a pointer to an identifier that allows to handle the task. If the task does not have to be handled in the future, this can be leaved NULL.
* Deletion of a task :

Using vTaskDelete(NULL).

1. **Scheduling :**

* It decides when a task in the ready state has to be run at a given time.
* Priority scheduling and if they have the same priority round robin is applied where q is the clock tick which is in TICK\_RATE\_HZ constant in the config file. ( every clock tick the scheduler decides which task to run)
* More frequent tasks take higher priority
* Event based or continuous tasks are preempted by periodic tasks.
* Maximum number of priorities is defined in MAX\_PRIORITIES constant. If a higher value is given to a task, then FreeRTOS cuts it to MAX\_PRIORITIES – 1.
* Functions like vTaskPriorityGet() & vTaskPrioritySet() are used to assign and know priorities of the tasks.
* No mechanism supported to prevent starvation, It’s the programmer’s responsibility to let low priority tasks run like idle tasks.

1. **Queue Management :**

* They are a means to store a finite length of fixed size data.
* They don’t belong to any task in particular.
* It is normally a FIFO ( elements are read in the order they have been written in ) but can be LIFO according to the writing method.
* **Reading in a queue:**
* When a task reads from a queue, it is moved to BLOCKED state and then back to READY state when data is written by another task.
* A task can specify max. waiting time for the queue to allow it to read, after this time it switches back to READY state. (xTicksToWait)

If INCLUDE\_vTaskSuspend if defined to 1 in FreeRTOSConfig.h and xTicksToWait equals MAX\_DELAY, then the task has no time limit to wait.

* Higher priority tasks read first from queue, if the same priority then the one who requested first is chosen.
* After reading an element in a queue, this element is normally removed from it; however, an other read function (xQueuePeek() ) allows to read an element without having it to be deleted from the queue.
* pdPASS is returned if a value was sucessfully read before max. time to wait is reached. If not, errQUEUE\_EMPTY is returned.

**Writing to a queue :**

* same rules for reading.
* It can send to back ( FIFO) or send to front ( LIFO).

**Creating a queue:**

You should specify length of the queue ( number of elements) and width ( size of each element).  
It returns NULL if the queue wasn’t created due to lack of memory available.

1. **Resource Management :**

**Binary semaphores:**

* Binary semaphores are the simplest effective way to synchronize tasks.
* A binary semaphore can be seen as a queue which contains only one element.
* Taking and giving a semaphore is like reading and writing to a queue.

**Mutexes :**

* Mutexes are designed to prevent mutual exclusion or deadlocking.
* A mutex is used similarly to a binary semaphore, except the task which take the semaphore must give it back. This can be achieved by having a token associated with the resource to access. A task holds the token, works with the resource then gives back the token.
* It helps against priority inversion phenomenon. When several tasks asks for a mutex, the mutex holder's priority is set to the highest waiting task priority.
* It increases complexity.

**Counting Semaphores:**

* A counting semaphore is a semaphore that can be taken several (but limited) times before it becomes unavailable.
* Its value is increased as the semaphore is given, and decreased when is is taken.
* It is comparable to a queue with a certain amount of elements.
* When created, a counting semaphore can be initialized to be available an arbitrary number of times.
* Returned value is NULL if the semaphore was not created, because of a lack of memory.

1. **Handling Interrupts**

\* Constants available in FreeRTOSConfig.h. :

\* **configKERNEL\_INTERRUPT\_PRIORITY** sets the interrupt priority level for the tick interrupt.

**\* configMAX\_SYSCALL\_INTERRUPT\_PRIORITY** defines the highest interrupt level available to interrupts that use interrupt safe FreeRTOS API functions.

If this constant is not defined, then any interrupt handler function that makes a use of FreeRTOS API must execute at configKERNEL\_INTERRUPT\_PRIORITY

* Any interrupt whose priority level is greater than configMAX\_SYSCALL\_INTERRUPT\_PRIORITY or configKERNEL\_INTERRUPT\_PRIORITY if configMAX\_SYSCALL\_INTERRUPT\_PRIORITY is not defined, will never be preempted by the kernel, but are forbidden to use FreeRTOS API functions.

1. Using binary semaphores
2. Critical sections either by

Suspend interrupts :

protects from any context change, either from a scheduler operation, or an interrupt event.

It must be kept as short as possible

1. taskENTER\_CRITICAL() and stop it using taskEXIT\_CRITICAL().
2. Stop the scheduler:

create a critical section consists in preventing any task from preempting it, but let interrupts to do their job.

Functions used :

vTaskSuspendAll()

xTaskResumeAll()

**6 – Memory Management :**

* Prototypes for memory allocation and dellocation :

void \*pvPortMalloc( size\_t xWantedSize)

void pvPortFree( void \*pv)

* FreeRTOS provides three different ways to allocate memory :

1. **Memory allocated once for all :**

It allocates all tasks, queues and semaphores, then start the scheduler and run the entire application, so need for the function to free the memory.

It allocates a simple array whose size is the TOTAL\_HEAP\_SIZE in FreeRTOSConfig.h, and divides it in smaller parts which are allocated for the tasks. This makes the application to appear to consume a lot of memory, even before any memory allocation.

1. **Constant sized and numbered memory**

An application can require to allocate and deallocation dynamically memory

Used If in every tasks' life cycle, number of variables and its size remains constant.

It has VportFree().

Causes a lot of fragmentation in memory. (like the best fit memory allocation)

1. **Free memory allocation and deallocation**

This implementation wraps the two functions, but make them thread safe by suspending the scheduler while allocating or deallocating.

\*\* FreeRTOS+Nabto is a small piece of C code that, when integrated into an embedded networked device, allows that device to be remotely accessed and controlled using a rich web based user interface or intelligent data acquisition system.

http://www.freertos.org/FreeRTOSPlus/Nabto/what\_is\_freertos\_plus\_nabto.shtml