<u>Introduction to RTOS</u>:

- A Real Time Operating System also known as RTOS is an operating system which is intended to fulfills the requirement of real time application.
- It is able to process data as comes in, typically without buffering delays.
- RTOS is the combination of calling predefined functions.
- The key factors in Real Time Operating System are minimum interrupt latency and minimum threads switching latency.
- The Real Time Operating System is valued more for how quickly and how predictably it responds to complete the tasks in given period of time.

<u>There are three types of RTOS</u>:

- **Hard RTOS:** bound to complete task in given deadline
- **Firm RTOS:** bound of deadline but if they miss the deadline it is acceptable but not in the case of Hard RTOS.
- **Soft RTOS:** not bound of any deadline.
- **Examples of RTOS:**
  1. LynxOS
  2. RTLinux
  3. VxWorks
  4. FreeRTOS
  5. OSE
  6. QNX
  7. Windows CE
- **Why RTOS are required:**
  - ➢ When we write good embedded software we do not need RTOS but when its complexity and size increases RTOS is always beneficial for the reasons listed below:
    - ▪ Abstract out timing information
    - ▪ Maintainability/Extensibility
    - ▪ Modularity
    - ▪ Cleaner interfaces
    - ▪ Easier testing (in some cases)
    - ▪ Code reuse
    - ▪ Improved efficiency

  - Idle time
  - Flexible interrupt handling
  - Mixed processing requirements
  - Easier control over peripherals
- **These are the advantages of RTOS but there are also some disadvantages listed as below:**
  - Low Priority Tasks
  - Precision of code
  - Limited Tasks
  - Complex Algorithms
  - Device driver and interrupt signals
  - Thread Priority
  - Expensive
  - Not easy to program

Introduction to FreeRTOS:

- FreeRTOS is an free and open-source Real-Time Operating system developed by Real Time Engineers Ltd.
- Its design has been developed to fit on very small embedded systems and implements only a very minimalist set of functions:
  - Very basic handle of tasks and memory management,
  - Just sufficient API concerning synchronization, and
- Absolutely nothing is provided for network communication, drivers for external hardware, or access to a filesystem.
- However, among its features are the following characteristics:
  - Preemptive tasks, a support for 23 micro-controller architectures by its developer,
  - A small footprint(4.3Kbytes on an ARM7 after compilation), written in C and compiled with various C compiler (some ports are compiled with gcc, oth ers with openwatcom or borland c++).
  - It also 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.
  - Finally, it implements queues, binary, counting semaphores and mutexes.

- There are also SafeRTOS and OpenRTOS available online which are similar to FreeRTOS.

Tasks:

A task in FreeRTOS:

- FreeRTOS allows an unlimited number of tasks to be run as long as hardware and memory can handle it.
- As a real time operating system, FreeRTOS is able to handle both cyclic and acyclic tasks.
- In RTOS, a task is defined by a simple C function, taking a void* parameter and returning nothing (void).
- Several functions are available to manage tasks:
- task creation (vTaskCreate()),
- destruction (vTaskDelete()),
- priority management (uxTaskPriorityGet(), vTaskPrioritySet()) or
- delay/resume ((vTaskDelay(), vTaskDelayUntil(),
- vTaskSuspend(),
- vTaskResume(),
- vTaskResumeFromISR().
- More options are available to user, for instance to create a critical sequence or monitor the task for debugging purpose.

Life Cycle of a Task:

- This section will describe more precisely how can a task evolve from the moment it is created to when it is destroyed.
- In this context, we will consider to be available only one microcontroller core, which means only one calculation, or only one task, can be run at a given time.
- Any given taskcan be in one of two simple states : “running” or “not running”.

As we suppose there is only one core, only one task can be running at a given time;
all other tasks are in the “not running” task.

- Figure 1 gives a simplified representation of this life cycle.
- When a task changes its state from“Not running” to running, it is said “swapped in” or “switched in”.

- When a task changes its state from “running” to “not running” it is called “Swapped out” or “Switched out”.

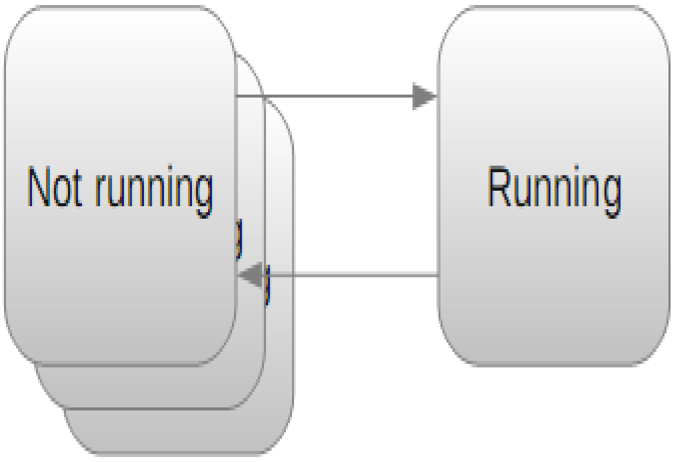


Figure 1: Simplified life cycle of a task: Only one task can be "running" at a given time, whereas the "not running state can be expanded".

- As there are several reasons for a task not to be running, the “Not running” state can be expanded as shows Figure 2.

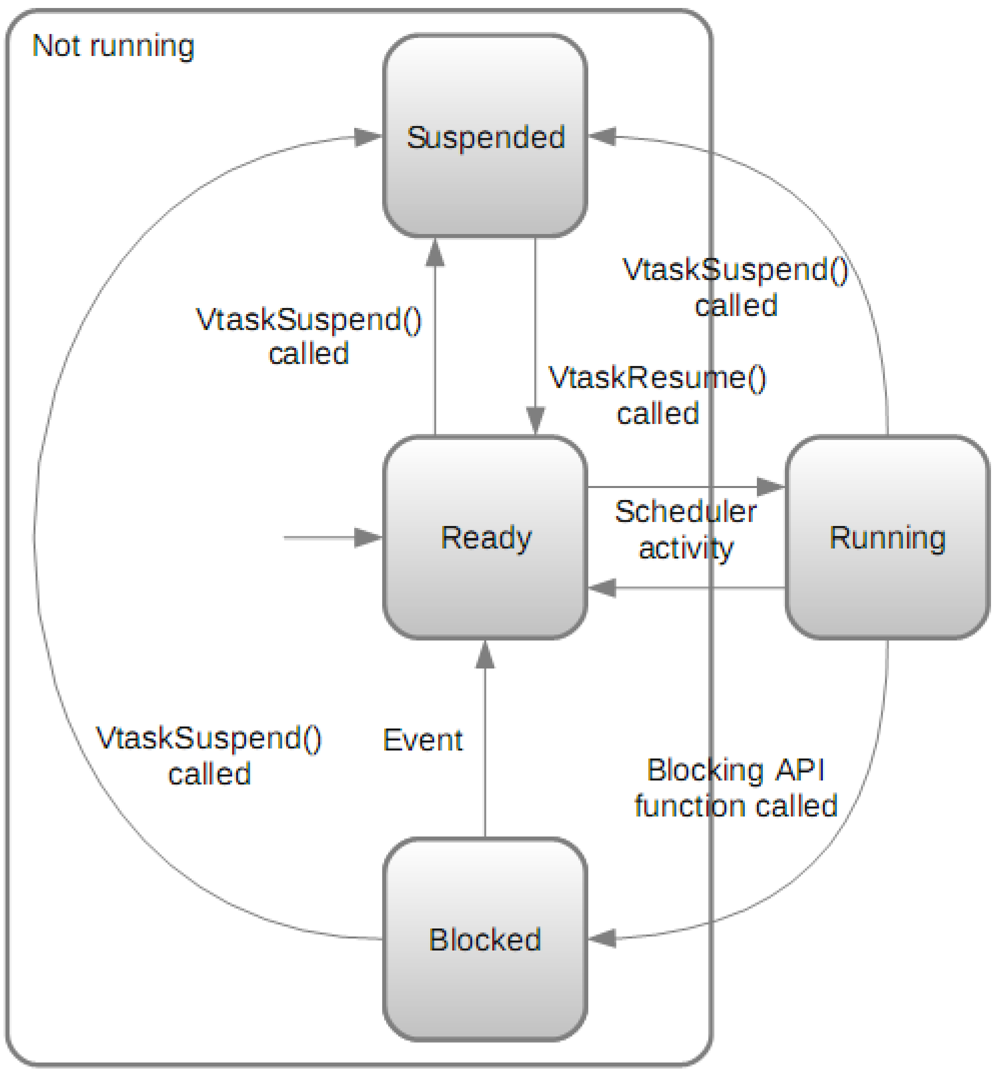


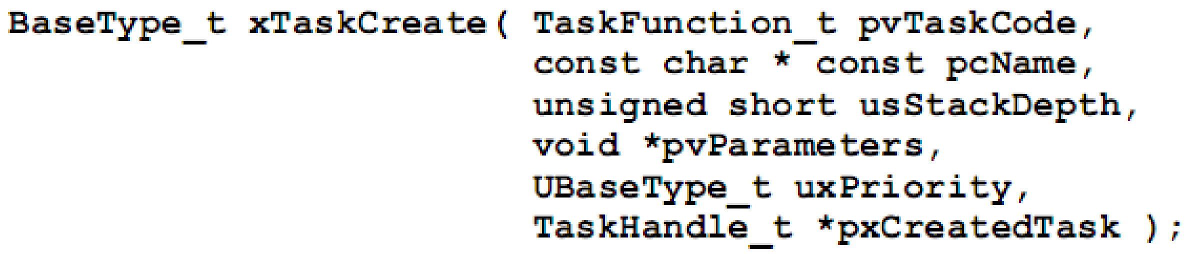
Figure 2: Lifecycle of a Task

- A task can be preempted because of a more priority task, because it has been delayed or because it waits for a event.
- When a task can runs but is waiting for the processor to be available, its state is said "**Ready**".This can happen when a task has it needs everything to run but there is a more priority task running at this time.
- When a task is delayed or is waiting for anotherrtask a task is said to be "**Blocked**".
- Finally, a call to vTaskSuspend() and vTaskResume() or xTaskResumeFromISR() makes the task going in and out the state "**Suspend**".
- It is important to underline that a if a task can leave by itself the "Running" state (delay, suspend or wait for anevent), only the scheduler can "switch in" again this task .

- When a task wants to run again, its state turns to “Ready” anonly the scheduler can choose which “Ready” task is run at a given time.

<u>Creation and Deletion of the Task</u>:

A task can be created using xTaskCreate() as shown in figure below:



```
BaseType_t xTaskCreate( TaskFunction_t pvTaskCode,
                        const char * const pcName,
                        unsigned short usStackDepth,
                        void *pvParameters,
                        UBaseType_t uxPriority,
                        TaskHandle_t *pxCreatedTask );
```

Figure 3: Task Creation Function

This function takes the argument list as:

pvTaskCode: The pvTaskCode parameter is simply a pointer to the function (in effect, just the function name) that implements the task.

pcName: A descriptive name for the task. This is mainly used to facilitate debugging, but can also be used in a call to xTaskGetHandle() to obtain a task handle.

usStackDepth: Each task has its own unique stack that is allocated by the kernel to the task when the task is created. The usStackDepth value tells the kernel how large to make the stack. The value specifies the number of words the stack can hold, not the number of bytes.

pvParameters: Task functions accept a parameter of type ‘pointer to void’ (void*). The value assigned to pvParameters will be the value passed into the task.

uxPriority: Defines the priority at which the task will execute. Priorities can be assigned from 0, which is the lowest priority, to (configMAX_PRIORITIES – 1), which is the highest priority.

pxCreatedTask: pxCreatedTask can be used to pass out a handle to the task being created. This handle can then be used to reference the task in API calls that, for example, change the task priority or delete the task. If your application has no use for the task handle, then pxCreatedTask can be set to NULL.