# **Embedded Systems Essentials with Arm: Get Practical with Hardware**

## Module 3

## KV3: RTOS—Tasks and Scheduling

Taking what we learned about RTOSs and tasks, we can now explore how these tasks are implemented in an RTOS. Tasks and multi-tasking are also known as threads or multi-threading, but for the sake of this video, we will be using the term “tasks.”

A task is a program strand or section which has a clear and distinct purpose and outcome within an embedded program. ​Multi-tasking describes when there are a number of tasks that need to be performed, simultaneously or concurrently. ​

The tasks are usually not all of equal importance. Therefore, we recognize that different tasks can have different priorities. A high-priority task should have the right to execute before a low-priority task. ​

Some tasks are time-triggered, and others event-triggered. As the name suggests, a time-triggered task occurs on completion of a certain period of time, and is usually periodic. Similarly, event-triggered tasks occur when a certain event takes place. *​*

This brings us to time slicing, and the clock tick. To help manage time systematically, operating systems use a technique to divide time into so-called slices, so that different tasks can be executed in different slices. In this case the OS is driven by a periodic interrupt, often called the clock “tick.” ​

This can be confused with the microcontroller clock oscillator, because both play an essential part in managing timing aspects of the microcontroller. For example, the oscillator could be running at twenty MegaHertz, with the clock tick running at one-hundred Hertz, which means once every ten milliseconds.

Now that we know the importance of timing, tasks and priorities, we can look at how an RTOS schedules these functions. The scheduler determines which task is to be executed next, and for how long. The performance of an RTOS relies on the scheduling strategy, and the four most common kinds of strategies are:

1. Cyclic Scheduling
2. Round Robin Scheduling
3. Preemptive Prioritized Scheduling, and
4. Cooperative Scheduling

In Cyclic Scheduling, each task is allowed to run to completion, then followed by the next task. This is an example of non-preemptive scheduling, it is very similar to the behavior we would get from a program structured as a “super loop”.

In Round Robin Scheduling, tasks are selected for execution in a fixed sequence. On each clock tick, the current task is discontinued, and the next is allowed to start execution. ​

The scheduler must preserve the context of the task being left, which means all its flags, registers, and so on. On its return, the task can then pick up activity seamlessly. A memory stack is reserved for each task for this purpose.​

All tasks are treated as of equal importance, or equal priority, and wait in turn for their slot of CPU time. ​

As tasks are not allowed to run to completion, this is an example of a preemptive scheduler.​

In the Preemptive Prioritized Scheduling​ strategy, tasks are given priorities. High-priority tasks are allowed to complete before time is given to lower priority. ​

The scheduler is still run by a clock tick, and on every tick, it checks whether a task of higher priority is ready, and should be prioritized. If a low-priority task is being executed, it will be replaced by one of higher priority. ​

Although this kind of scheduling represents classic RTOS action, it has disadvantages such as being memory-intensive, context switching takes time, and tasks must be specifically written in order to be switched at any time.

An alternative to this is called Cooperative Scheduling​. It also relies on tasks being specifically written, but with the function of giving up its CPU access at its own discretion. The task gives up control at a moment of its choosing, so it can manage its context saving, and the central overhead is not required.

Because of this, cooperative scheduling is unlikely to be quite as responsive to tight deadlines as preemptive scheduling, but it needs less memory. This ability to minimize memory needs and switch tasks fast is important in small systems.​

All these scheduling strategies require tasks to go through different states. They are:

1. Running: the current task, one at a time.
2. Ready: Tasks that are ready to run. When a running task has ceased execution, through pre-emption, the next ready task with the highest priority becomes the running task.​
3. Waiting: sometimes called blocked, are tasks that are waiting for an event.
4. Inactive: also called dormant, are tasks that are not yet created, or which have terminated, and consume no system resources.

In the next lesson, we’ll explore the Mbed RTOS and identify functions provided by the Mbed API.

**Visit arm.com for more information.**