5.1 Introduction to Real-Time Multitasking

What is an RTOS? Tasks and Scheduling

A **Real-Time Operating System (RTOS)** is a specialized operating system designed for embedded systems that must process data and events within a strict, predictable timeframe. Unlike a desktop OS (like Windows or macOS) which prioritizes throughput and fairness, an RTOS prioritizes **determinism,** the ability to guarantee that a task will be completed within a specific deadline.

The fundamental unit of execution in an RTOS is a **Task**. You can think of a task as an independent function that runs in its own context, with its own stack and priority. For example, in a smart device, you might have one task for managing the Wi-Fi connection, another for reading sensor data, and a third for updating a display.

The core component of the RTOS is the **Scheduler**. Its job is to manage which task gets to use the CPU at any given moment. By rapidly switching between tasks (a process called a "context switch"), the scheduler creates the illusion that all tasks are running simultaneously, a concept known as **multitasking**. In a priority-based scheduler like the one in FreeRTOS, the scheduler will always ensure that if multiple tasks are ready to run, the one with the highest priority is the one that executes.

The Problem with `delay()`: Blocking vs. Non-Blocking Operations

In simple microcontroller programming (like a basic Arduino sketch), it is common to use a `delay()` function to control timing. For example, to blink an LED every second, you might write:

```
void loop() {
  digitalWrite(LED_PIN, HIGH);
  delay(1000); // Wait for 1 second
  digitalWrite(LED_PIN, LOW);
  delay(1000); // Wait for 1 second
}
```

This works, but it is extremely inefficient. During the `delay(1000)` call, the CPU is completely occupied doing nothing, it is stuck in a busy-wait loop, unable to perform any other work. This is known as a **blocking** operation. A blocking function halts the execution of its thread or task until a specific event occurs (in this case, the passage of time).

In a multitasking RTOS environment, blocking is the enemy of responsiveness. If one task calls `delay()`, it effectively puts the entire system on hold (unless a higher-priority task preempts it), preventing other, potentially critical, tasks from running. The goal in an RTOS is to use **non-blocking** operations. A task should perform its work and, if it needs to wait, it should inform the scheduler so that a lower-priority task can use the CPU in the meantime.

The Need for Asynchronous Events

The solution to the blocking problem is to design systems around **asynchronous events**. An asynchronous event is one that occurs independently of the main program flow, allowing the system to react to it without having to constantly wait and check for it.

An RTOS provides two primary mechanisms for handling asynchronous events:

1. **Hardware Interrupts:** These are signals sent directly from hardware peripherals to the CPU, demanding immediate attention. When an interrupt occurs, the CPU immediately pauses whatever it is doing, executes a special function called an **Interrupt Service Routine (ISR)**, and then resumes its previous work. This is ideal for high-priority, time-critical events triggered by the outside world, such as a button being pressed, data arriving on a communication bus, or a hardware timer reaching zero.
2. **Software Timers:** These are timers managed by the RTOS itself. You can ask the RTOS to execute a specific function (a "callback") at some point in the future, either once or repeatedly. This allows you to schedule application-level events without blocking. Instead of calling delay(), a task can start a software timer and then continue with other work or yield control of the CPU to other tasks. The RTOS will ensure the callback function is executed at the correct time.

By using timers and interrupts, you can build complex, responsive applications where tasks spend almost no time waiting and are instead driven by the occurrence of events.

Revision #1
Created 2025-08-28 12:41:21 UTC by GI
Updated 2025-08-28 12:43:08 UTC by GI