24EC504 – Unit III – Part 1: Real-Time Operating Systems and Task Scheduling

1. Real-Time Operating Systems (RTOS) and FreeRTOS

1.1 What is an RTOS?

A Real-Time Operating System is built for environments where many events, mostly external to the computer, must be accepted and processed within a short, guaranteed time or before a fixed deadline. Typical domains: industrial control, telephone switching, flight control and real-time simulation.

- Responds to tasks quickly and predictably.
- Ensures tasks finish within a fixed time limit (deadline).
- Used where timing is critical: robots, medical devices, automobiles.
- Supports multitasking with precise, deterministic scheduling.
- Examples: air-traffic control, command and control systems, airline reservation, cardiac pacemakers, networked multimedia, robotics.

1.2 Types of RTOS

Type	Deadline behaviour	Example	Typical use
Hard real-time	Critical tasks must finish within a strict bound; a missed deadline is a total system failure	A welding robot on a car line must weld exactly on time	Scientific experiments, medical imaging, industrial control, weapon systems, robots, air-traffic control
Soft real-time	Timing is relaxed; a missed deadline degrades quality of service, not the whole system	Multimedia and digital audio	Priority-based preemptive scheduling is typical: the CPU always goes to the highest-priority process
Firm real-time	Deadlines must still be respected, but an occasional miss has a small, tolerable impact (reduced output quality)	Multimedia applications	–

1.3 Purpose and Uses

The main goal of an RTOS is to perform critical tasks on time, finishing processes within strict deadlines while handling multiple tasks concurrently. Uses: defense systems (RADAR), air-

traffic control, networked multimedia, medical devices (pacemakers) and stock-trading applications.

1.4 RTOS vs General-Purpose OS

General-purpose OS	RTOS
Complex	Simple
Best effort	Guaranteed response
Fairness-oriented	Strict timing constraints
Average bandwidth	Minimum and maximum limits defined
Unknown components	Components known in advance
Unpredictable behaviour	Predictable, deterministic
Plug-and-play	Purpose-built, upgraded in a controlled way

1.5 Features of an RTOS

- **Determinism:** tasks run within a predictable, fixed time frame.
- **Multitasking:** schedules multiple tasks and switches rapidly, giving the illusion of parallelism.
- **Memory management:** stops tasks interfering with each other's memory, improving stability and security.
- **Minimal latency:** interrupt and task-switch latencies kept as low as possible.
- **Priority-based scheduling:** higher-priority tasks always run before lower-priority ones.
- **Resource allocation:** manages memory, CPU time and other resources efficiently.
- **Interrupt handling:** responds quickly through the RTOS API with minimal time in interrupts.
- **Task synchronization:** inter-task communication (ITC) such as semaphores and message queues for safe sharing.

1.6 Advantages and Disadvantages

Advantages	Disadvantages
Maximum utilization of devices and resources	Very few tasks run simultaneously, to avoid errors
Fast task shifting: context switch about 10 µs in older systems, about 3 µs in modern ones	System resources required can be expensive
Priority given to the running application over queued ones	Scheduling algorithms are complex and hard to design
Small code size, embeddable in transport	Needs specific device drivers and interrupt

Advantages	Disadvantages
controllers and similar systems	signals to respond quickly
Designed to run reliably with minimal errors	Setting thread priority is tricky
Very well-managed memory allocation	Only minimal task switching is performed

1.7 FreeRTOS: a Practical RTOS Kernel

FreeRTOS is one of the most widely used real-time kernels for microcontrollers: free, open-source (MIT licence), with a very small RAM/ROM footprint and ports to 40+ architectures (ARM Cortex-M, ESP32, PIC, AVR, RISC-V). It is the de-facto standard teaching and industry RTOS for embedded and IoT projects.

Key kernel objects:

Object	Purpose
Tasks	Independently scheduled code units, each with its own stack and priority
Queues	Inter-task and task-to-ISR data transfer (FIFO by default)
Semaphores (binary/counting) and mutexes	Synchronization and mutual exclusion
Software timers	Run a callback after a set period without dedicating a task
Event groups	Let a task wait on a combination of event bits
Task notifications	Lightweight, fast alternative to semaphores/queues for simple signalling

Task states:

- **Running:** executing on the CPU.
- **Ready:** able to run, waiting for CPU time.
- **Blocked:** waiting for an event, delay, semaphore or queue.
- **Suspended:** explicitly removed from scheduling until resumed.

Scheduling policy: FreeRTOS uses a preemptive, priority-based scheduler by default: the highest-priority ready task always runs. Optional time-slicing (round robin) shares CPU among tasks of equal priority. A periodic tick interrupt (for example SysTick on Cortex-M) drives the scheduler.

Common API calls:

Call	Use
xTaskCreate() / vTaskDelete()	Create and delete a task
vTaskDelay() / vTaskDelayUntil()	Block a task for a period
xQueueSend() / xQueueReceive()	Send/receive data through a queue
xSemaphoreTake() / xSemaphoreGive()	Acquire/release a semaphore or mutex
vTaskStartScheduler()	Start the RTOS scheduler

FreeRTOS is configured through a project-specific FreeRTOSConfig.h, which sets tick rate, heap scheme (heap_1 to heap_5), stack sizes and enabled features.

2. Task Scheduling and Synchronization

2.1 What is Task Scheduling?

Task scheduling is how an OS/RTOS decides which task runs next, when, and for how long. An embedded system usually has one processor, so tasks cannot truly run at once; the scheduler switches the CPU between them rapidly, creating the illusion of parallel execution. **The scheduler is the brain of the RTOS.**

2.2 Tasks, Threads and Processes

- **Task:** the smallest unit of work scheduled by the RTOS, doing one function (read a sensor, blink an LED, update an LCD, send UART data).
- **Thread:** an independent execution path within a program; threads share the same memory space.
- **Process:** a complete running application that owns its memory and resources.

2.3 The Scheduler

The scheduler is the RTOS component that selects the next task. Its responsibilities are allocating CPU time, keeping fairness, meeting deadlines and maximizing utilization. The rule set it follows is the **scheduling algorithm**.

2.4 Scheduling Algorithms

Algorithm	Working principle	Characteristics	Advantages	Disadvantages
Run-to-completion (RTC)	Task 1 runs until it finishes, then Task 2, and so on	Non-preemptive; single stack; simplest	Easy to implement, minimal memory, portable	A long task blocks all others; poor response time
Round robin (cooperative)	Tasks take turns (A → B → C → D → repeat); a task releases the CPU by calling an RTOS service	Cooperative; context saved; task resumes where it left off	Fair CPU allocation, better than RTC, easy	Responsiveness depends on cooperation; no priority
Time-slice	Each task gets a fixed time slice (for example 10 ms each); on expiry the scheduler switches automatically	Fixed slices	Automatic switching	–
Priority	Each task has a priority; the highest-priority ready task always runs	The scheduler reruns when the current task suspends, when a higher-priority task becomes ready, or when an interrupt readies one	Important events served first	–
Composite	Combines techniques, typically priority plus time slicing: same-priority tasks share the CPU by time slice	Used by many commercial RTOSs	Better fairness, responsiveness and CPU utilization	–

2.5 Task Suspension and Task States

A suspended task is temporarily prevented from running. Three types:

- **Pure suspend:** suspended manually; resumed by another task or an ISR.

- **Sleep suspend:** suspended for a fixed time; resumes automatically on timeout.
- **Blocking suspend:** waits for a resource; resumes when it becomes available.

Generically a task moves between four states: **Ready** (able to run), **Running** (on the CPU), **Blocked/Waiting** (waiting for an event, delay or resource) and **Suspended/Terminated** (removed from scheduling). The scheduler moves tasks between these states correctly and predictably.