Lecture 8

Embedded Systems

RTOS III

Associated Prof. Wafaa Shalash

Lect. 8

Class Rules

- Be in class on time,
- Listen to instructions and explanations.
- Talk to your classmates only when there is an activity.
- Use appropriate and professional language.
- Keep your mobile silent.

Lecture Topics

- RTOS kernel
- KeyFunctions of the RTOS Kernel
- What is RTOS task? Scheduler
- The Heart of RTOS
- Preemptive vs Non-Preemptive Scheduling

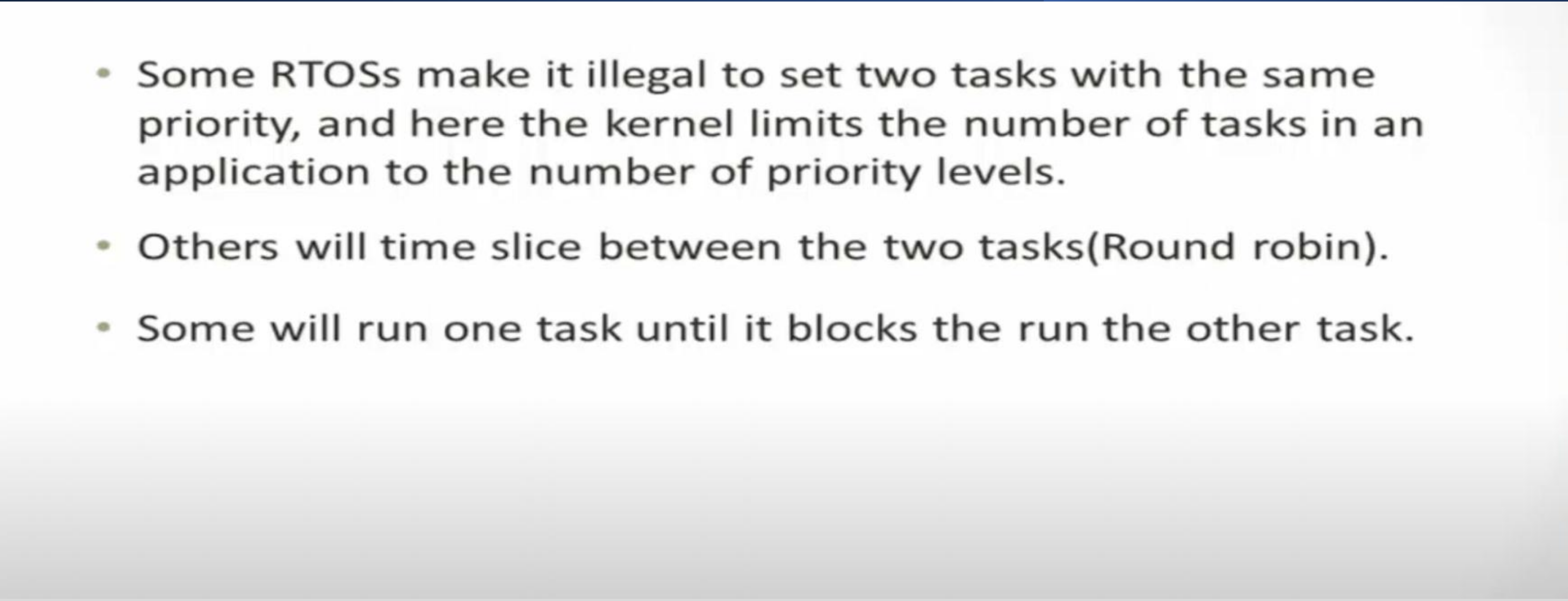
RTOS schedular

- Scheduler the core Component of any RTOS kernel,
- Its a set of algorithms that determines which task executes when.
- It's keeping track on the status of each task, and decides which to run.
- In Most RTOSs the developers is the one who sets the priority of each task, regarding to this priority the scheduler will decide which task will run.
- The scheduler assumes that you knew what you where doing while setting tasks priority.
- A bad design for tasks priority, may leads to a high priority task hogs the processor for long time, this is called ***CPU starvation.***

RTOS schedular (Cont.)

- It’s keeping track on the status of each task, and decides which to run.
- Scheduler has no control on tasks on the blocked status.
- If tasks are blocked the scheduler waits an event to unblock this tasks, like an external interrupt from pushing a button.
- If no events happened, surely it’s a bad design from your side.

What if two tasks have the same priority ?



- Some RTOSs make it illegal to set two tasks with the same priority, and here the kernel limits the number of tasks in an application to the number of priority levels.
- Others will time slice between the two tasks(Round robin).
- Some will run one task until it blocks the run the other task.

What if two tasks have the same priority ?

1. Round-Robin (Time-Slicing) Scheduling

- **How it works:**
  - Tasks of the same priority share CPU time in fixed time slices (quantum).
  - The RTOS scheduler switches between them in a circular order.
- **Example:**
  - Task A & Task B (both priority 5) run alternately for, say, 10ms each.
- **Pros:**
  - Fair CPU allocation.
- **Cons:**
  - Overhead due to frequent context switches.
- **Used in:** FreeRTOS
- `(configUSE_TIME_SLICING enabled), Micrium µC/OS.`

What if two tasks have the same priority ?

2. FIFO (First-In-First-Out) Scheduling

- **How it works:**
  - The task that enters the **ready state first** runs until it blocks or yields.
- **Example:**
  - If Task A was ready before Task B (same priority), Task A runs to completion (or until it waits).
- **Pros:**
  - Simple, predictable.
- **Cons:**
  - A misbehaving task can hog CPU.
- **Used in:** VxWorks, some configurations of FreeRTOS (`configUSE_TIME_SLICING disabled`).

What if two tasks have the same priority ?

3. Priority Inversion Handling (Optional)
•If a lower-priority task holds a resource needed by a same-priority task:
  •Some RTOS (e.g., FreeRTOS with Priority Inheritance) temporarily boost the lower task’s priority.

4. Custom Scheduling Policies
•Some RTOS allow user-defined schedulers (e.g., weighted round-robin).

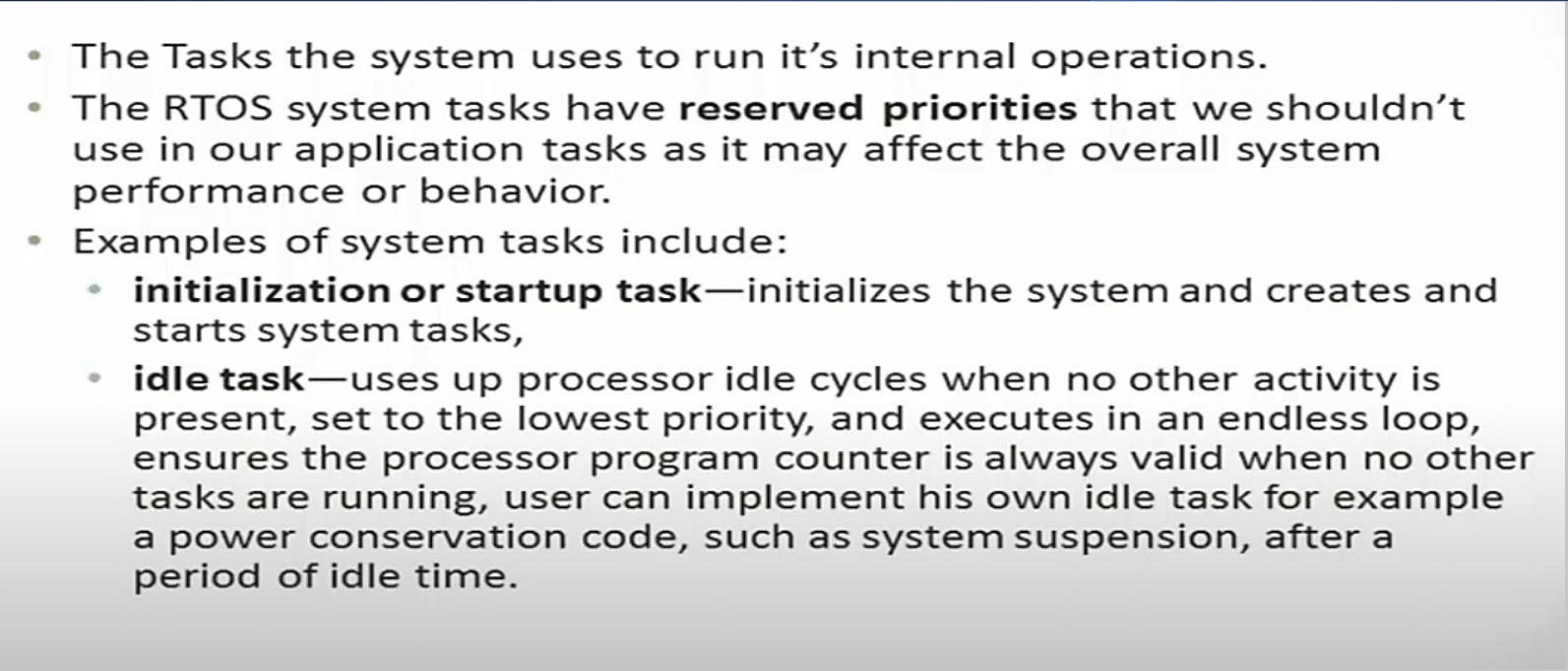
Example (FreeRTOS)

// FreeRTOS configuration:

#define configUSE_TIME_SLICING 1 // Enable round-robin for same-priority tasks

If disabled (0), tasks run in FIFO order.

System Tasks



- The Tasks the system uses to run it’s internal operations.
- The RTOS system tasks have **reserved priorities** that we shouldn’t use in our application tasks as it may affect the overall system performance or behavior.
- Examples of system tasks include:
  - **initialization or startup task**—initializes the system and creates and starts system tasks,
  - **idle task**—uses up processor idle cycles when no other activity is present, set to the lowest priority, and executes in an endless loop, ensures the processor program counter is always valid when no other tasks are running, user can implement his own idle task for example a power conservation code, such as system suspension, after a period of idle time.

System Tasks

- **Logging Task**
- **Exception chandelling tasks**
- **Debug agent task**

System Tasks

- n a **Real-Time Operating System (RTOS)**, a **system task** (also called a **kernel task** or **daemon task**) is a special task managed by the RTOS kernel to handle critical system functions. These tasks are not created by the user but are essential for the OS's internal operations.

Key Characteristics of System Tasks

- **Managed by the Kernel**
  - Automatically created and scheduled by the RTOS.
  - Typically run at **high priority** (sometimes the highest).
- **Purpose**
  - Handle low-level operations like:
    - **Task scheduling** (e.g., the **scheduler task**).
    - **Memory management** (e.g., garbage collection in some RTOS).
    - **Interrupt servicing** (e.g., deferred interrupt handlers).
    - **Timer management** (e.g., the **tick task** in FreeRTOS).
    - **Idle task** (runs when no other task is active).
- **Execution Trigger**
  - Runs in response to **events**, **interrupts**, or **time-based triggers** (e.g., the RTOS tick).
-

Common System Tasks in RTOS

System Task	Function	Example RTOS
Idle Task	Runs when no other task is active; may handle power-saving (e.g., WFI).	FreeRTOS, Zephyr, µC/OS
Tick Task	Manages system time (e.g., vTaskDelay(), timeouts).	FreeRTOS (xTimerTask)
Scheduler Task	Decides which user task runs next (if not handled in the kernel ISR).	VxWorks, QNX
Deferred ISR Task	Processes interrupt bottom halves (to reduce ISR latency).	Linux (RT-Preempt), Zephyr
Garbage Collector	Reclaims memory (in RTOS with dynamic allocation).	Some embedded JVM RTOS

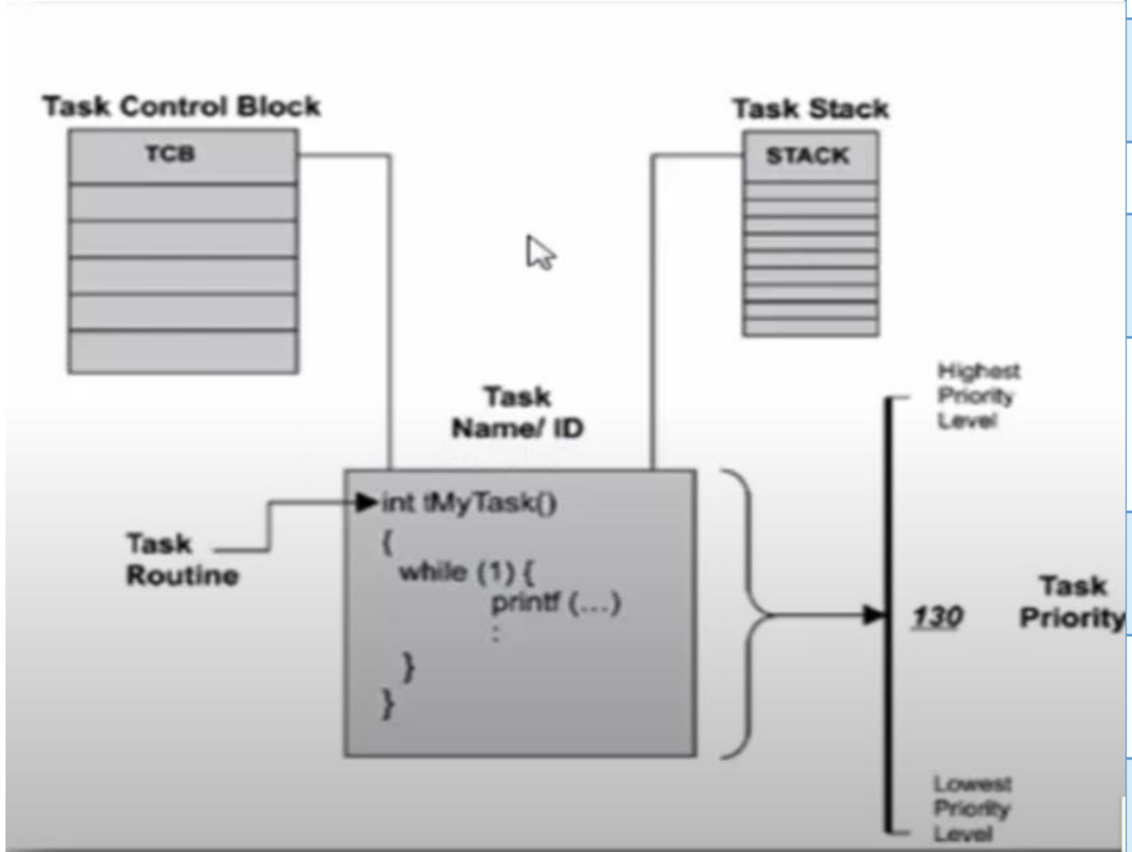
How System Tasks Differ from User Tasks

Aspect	System Task	User Task
Creation	Auto-created by the kernel.	Manually created by the developer.
Priority	Usually high (or hidden from users).	User-defined priority.
Purpose	Maintains OS functionality.	Implements application logic.
Modification	Typically non-configurable.	Fully customizable.

- **Why Are System Tasks Important?**
- **Ensures RTOS stability** (e.g., prevents deadlocks in scheduling).
- **Reduces interrupt latency** (by deferring non-critical work).
- **Manages resources** (e.g., memory, timers).

Task Object Data

- In FreeRTOS, each task is represented by a **Task Control Block (TCB)**, a data structure that holds all the task's metadata. Below is a detailed breakdown of the key components:



FIELD	DESCRIPTION
pxTopOfStack	Pointer to the task's current stack location.
pxStack	Pointer to the start of the task's stack.
pcTaskName	Human-readable task name (e.g., "LED_Task").
uxPriority	Task priority (0 = lowest, configMAX_PRIORITIES-1 = highest).
xGenericListItem	Used to place the task in Ready/Suspended/Event lists.
xEventListItem	Links tasks to event groups or queues.
uxBasePriority	Original priority (used for priority inheritance).
ulRunTimeCounter	Accumulates runtime stats (if configGENERATE_RUN_TIME_STATS =1).
pxTaskTag	User-assigned identifier (for debugging).
pvThreadLocalStorage	Pointer to task-specific data (like thread-local storage in OS).
pxEndOfStack	Marks the stack's end (for overflow checks)

Simple FreeRTOS example with two tasks that blink LEDs at different rates, demonstrating task creation

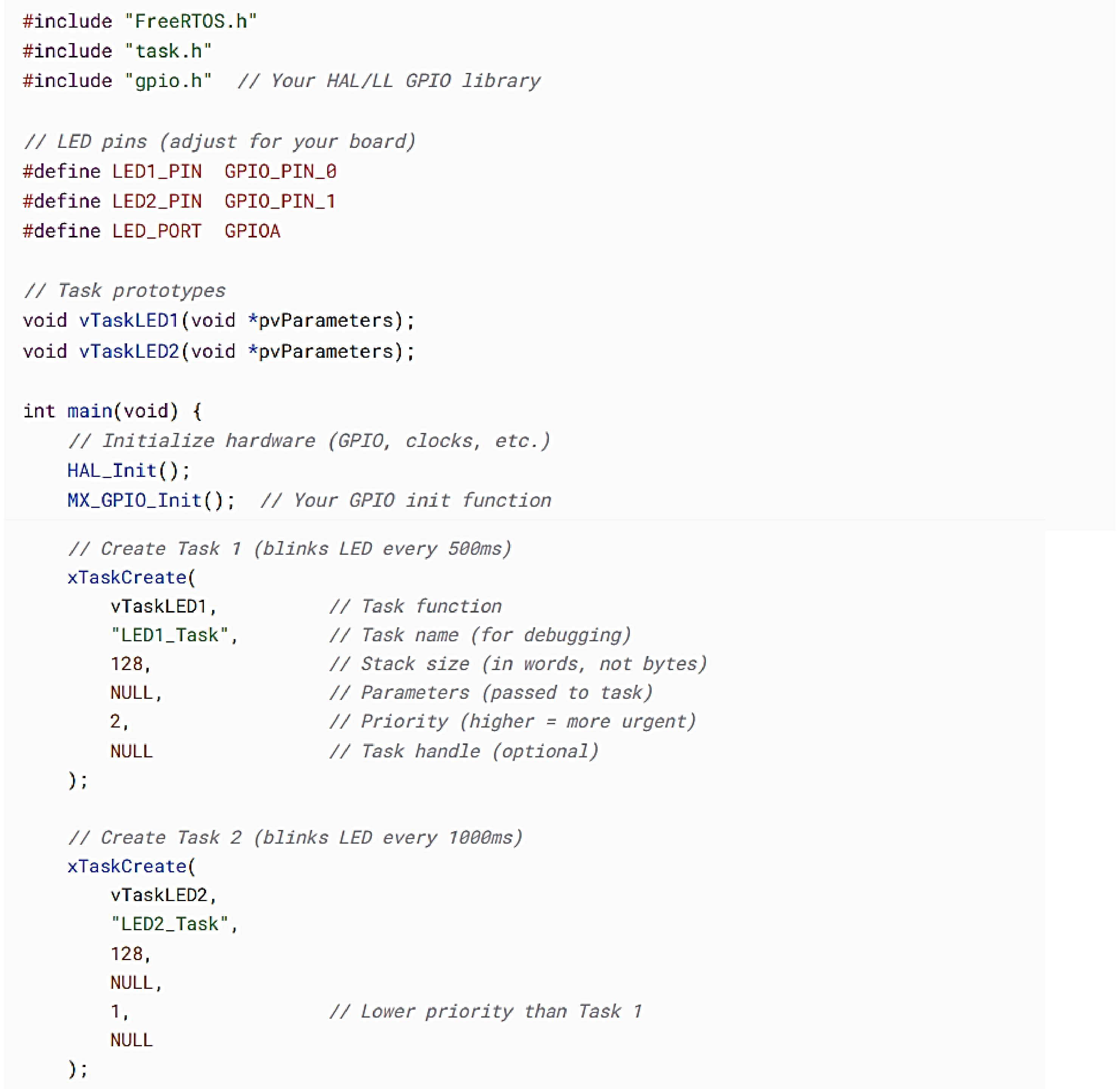
1. Hardware Setup

Target: STM32 (or any FreeRTOS-supported board)

Components:

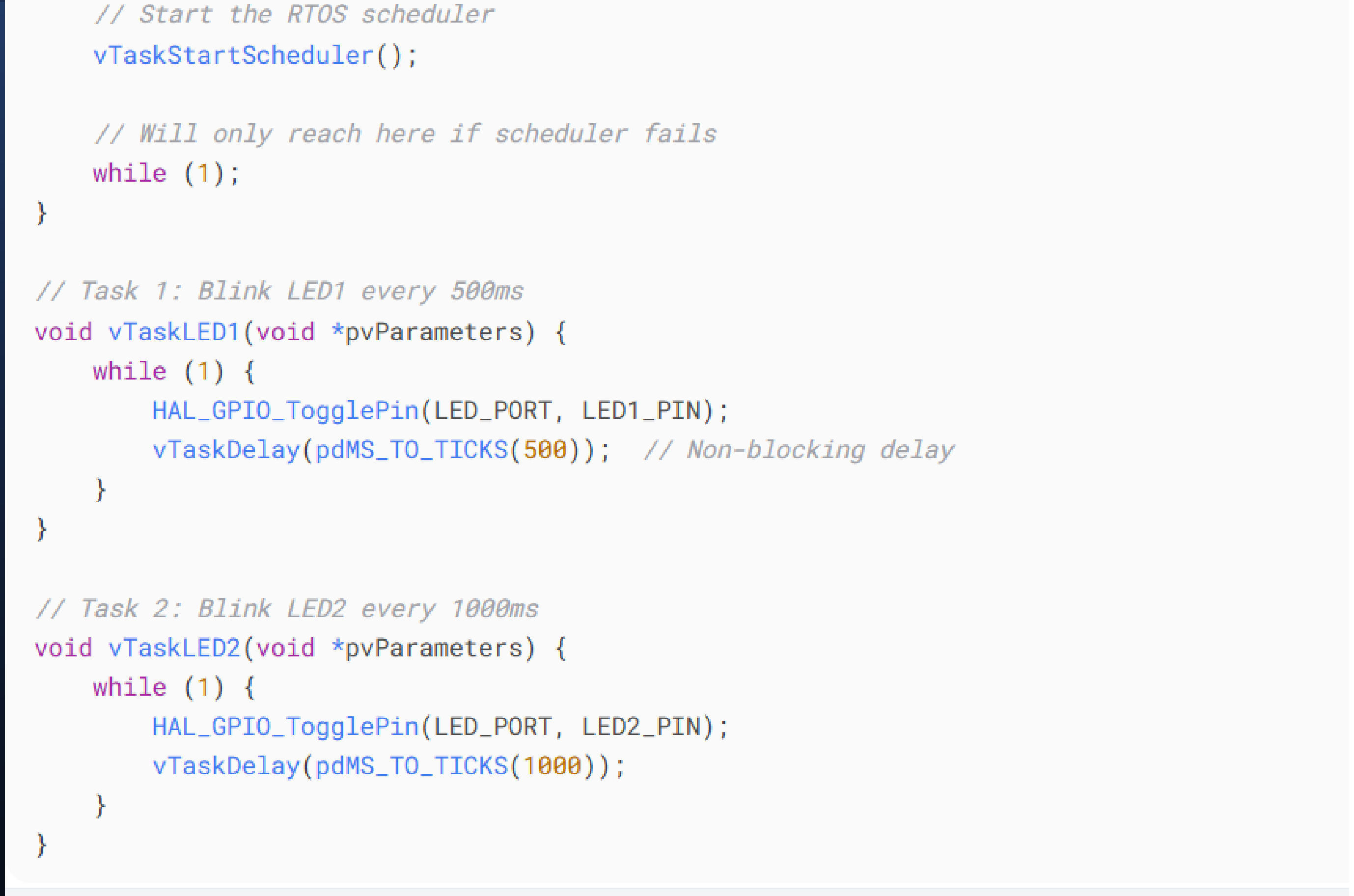
- 2 LEDs (GPIO pins: LED1_PIN, LED2_PIN)
- Optional: Serial console for debug messages (printf).

FreeRTOS Code Example



```c
#include "FreeRTOS.h"
#include "task.h"
#include "gpio.h"  // Your HAL/LL GPIO library

// LED pins (adjust for your board)
#define LED1_PIN  GPIO_PIN_0
#define LED2_PIN  GPIO_PIN_1
#define LED_PORT  GPIOA

// Task prototypes
void vTaskLED1(void *pvParameters);
void vTaskLED2(void *pvParameters);

int main(void) {
    // Initialize hardware (GPIO, clocks, etc.)
    HAL_Init();
    MX_GPIO_Init();  // Your GPIO init function

    // Create Task 1 (blinks LED every 500ms)
    xTaskCreate(
        vTaskLED1,          // Task function
        "LED1_Task",        // Task name (for debugging)
        128,                // Stack size (in words, not bytes)
        NULL,               // Parameters (passed to task)
        2,                  // Priority (higher = more urgent)
        NULL                // Task handle (optional)
    );

    // Create Task 2 (blinks LED every 1000ms)
    xTaskCreate(
        vTaskLED2,
        "LED2_Task",
        128,
        NULL,
        1,                  // Lower priority than Task 1
        NULL
    );
```

FreeRTOS Code Example



```c
    // Start the RTOS scheduler
    vTaskStartScheduler();

    // Will only reach here if scheduler fails
    while (1);
}

// Task 1: Blink LED1 every 500ms
void vTaskLED1(void *pvParameters) {
    while (1) {
        HAL_GPIO_TogglePin(LED_PORT, LED1_PIN);
        vTaskDelay(pdMS_TO_TICKS(500));  // Non-blocking delay
    }
}

// Task 2: Blink LED2 every 1000ms
void vTaskLED2(void *pvParameters) {
    while (1) {
        HAL_GPIO_TogglePin(LED_PORT, LED2_PIN);
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

3. Key Concepts Explained

Task Creation

- xTaskCreate() defines:
  - Function pointer (vTaskLED1).
  - Stack size (e.g., 128 words = 512 bytes on 32-bit systems).
  - Priority (2 > 1, so Task 1 preempts Task 2 if both are ready).

Non-Blocking Delay

- vTaskDelay(pdMS_TO_TICKS(500)) yields CPU to other tasks.

Scheduler

- vTaskStartScheduler() launches RTOS (never returns unless error).

Task Priorities

- Task 1 (priority 2) runs before Task 2 (priority 1) if both are ready.

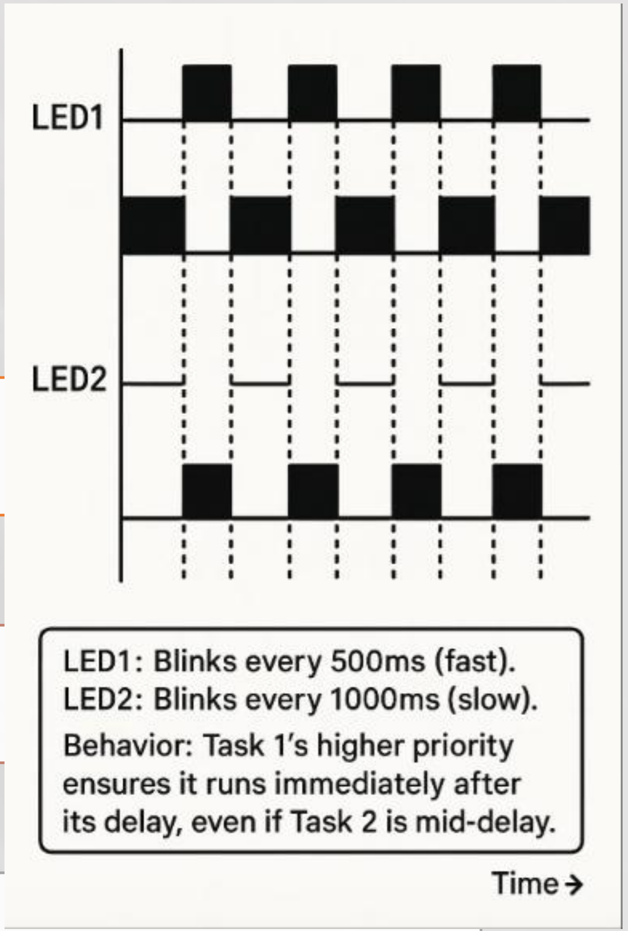
4. Expected Output

LED1: Blinks every **500ms** (fast).

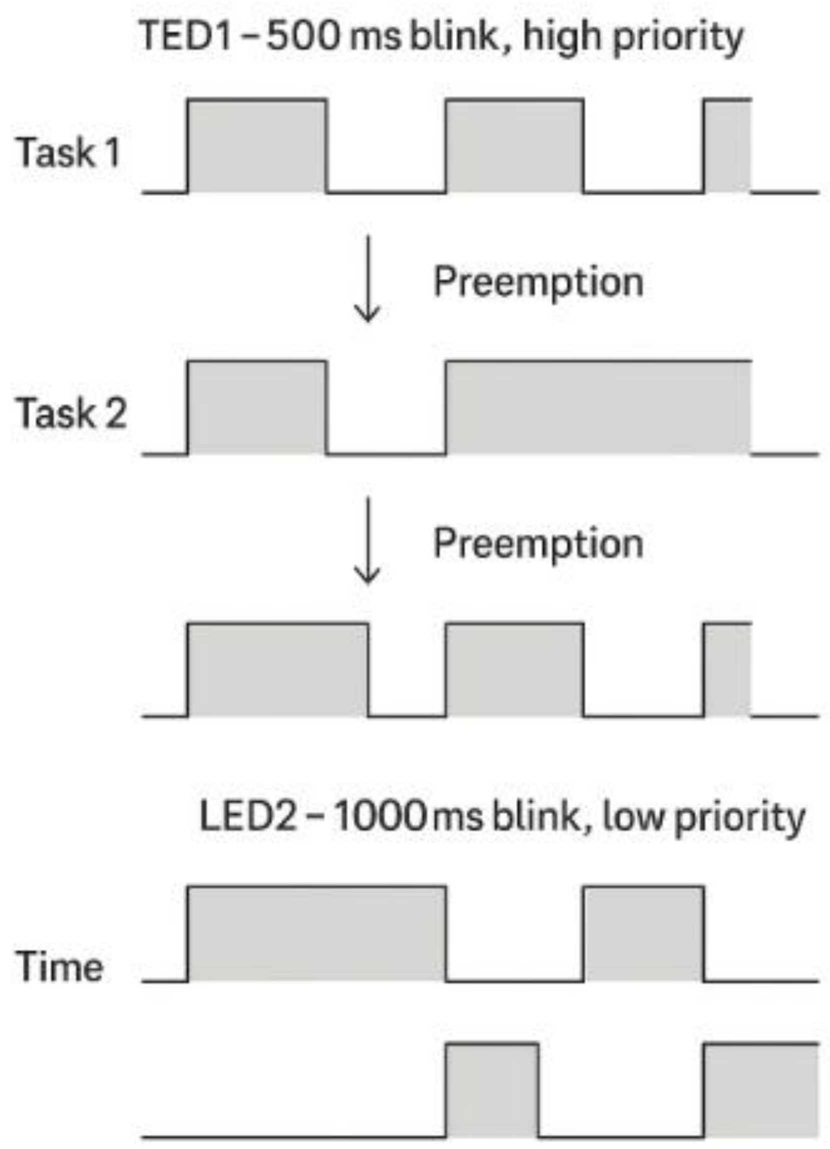
LED2: Blinks every **1000ms** (slow).

Behavior:

- Task 1's higher priority ensures it runs immediately after its delay, even if Task 2 is mid-delay.



LED1: Blinks every 500ms (fast).
LED2: Blinks every 1000ms (slow).
Behavior: Task 1's higher priority ensures it runs immediately after its delay, even if Task 2 is mid-delay.


TED1 – 500 ms blink, high priority
Task 1
Preemption
Task 2
Preemption
LED2 – 1000 ms blink, low priority
Time