Module 2 - Task Management

Learning Objectives

After completing this module, students are expected to be able to:

- Understand the basic concepts of task scheduling in RTOS.
- Understand the types of task scheduling algorithms.
- Understand and apply FreeRTOS APIs to create and manage tasks on the ESP-32.

The Importance of Task Scheduling in IoT

One of the most important aspects in an RTOS is task scheduling since it defines the sequence in which many operations run on the CPU at the appropriate instant. Each process or task in the context of real-time applications has a particular function, such reading sensor data, handling images, or sending messages to other devices.

Scheduling takes great significance in applications using the **Internet of Things** (IoT) as many devices connect with one other and engage their surroundings using sensors and actuators. Usually having strict timing demands, IoT applications need dependability and rather quick reactions. An IoT greenhouse system, for instance, has to continuously monitor temperature and moisture in real time while also controlling ventilation or irrigation on time to keep ideal conditions.

Task scheduling in an RTOS is geared toward two primary objectives:

1. **Feasibility**: Every job needs to be finished by its deadline; nothing is left out.
2. **Optimality**: Optimize characteristics including CPU use, power consumption, and reaction time such that the system performs effectively.

Attaining both objectives is difficult since there are usually several duties with varying priorities, execution periods, due dates, and dependencies. In addition, task scheduling helps considerably with:

- Making sure projects fulfill Quality of Service (QoS) specifications including low latency, great throughput, reliability, and energy efficiency.
- Improving the use of CPU, memory, bandwidth, and power among other resources.
- Managing the workload on multicore systems.
- Adjusting to shifting environmental circumstances including fluctuating workloads, network conditions, et cetera.

Categories of Task Scheduling Algorithms

Several task-scheduling techniques available in a Real-Time Operating System (RTOS) have both benefits and drawbacks depending on the the needs of the system and the nature of the tasks under execution. Among the most often used algorithms are:

1. Run for Completion (RTC)

The simplest approach is the Run to Completion algorithm. Every job runs till finished before moving onto the next one. Once all activities are finished, the sequence repeats itself from the start.

Advantages	Simple and quick to put into practice.
Disadvantages	Other tasks influence the completion time of a job, therefore reducing the determinacy of the system.

2. Round Robin (RR)

Round Robin is similar to RTC, but a task does not have to complete all its work before releasing the CPU. When it is scheduled again, the task resumes from where it left off.

Advantages	• Provides a fairer time allocation to each task. • More adaptable than RTC.
Disadvantages	Still relies on how each job behaves and is unable to stop one from taking over the processor for too long.

3. Time Slice (TS)

A pre-emptive scheduling algorithm in which execution time is broken into minute pieces known as time slices or ticks (e.g., 1 ms). The scheduler picks one task from the whole task list to run every time an interrupt happens.

Advantages	Stops starvation (a job kept too long).

Disadvantages	Can lead to repeated context switching, hence raising system overhead.

4. Fixed Priority (FP)

Fixed Priority assigns a static priority based on urgency to every job. The scheduler always chooses the task with the highest priority to run first.

If:

- Many activities share the same priority; they are done in round robin.
- While another task is running, a higher-priority task emerges and instantly interrupts the current operation.

Advantages	Uncomplicated and efficient; regularly used in real-time applications.
Disadvantages	Less flexible in response to workload or shifting system conditions.

5. Earliest Deadline First (EDF)

Dynamic priority according to the deadline of each assignment is provided by the Earliest Deadline First method. The one with the shortest deadline is always done first.

- Two jobs with the same deadline are done in round robin.

Theoretically, EDF is ideal since it can plan any possible collection of jobs.

Advantages	Offers best performance in deadline compliance.
Disadvantages	Practically verifying execution can be somewhat challenging and difficult.

Introduction to FreeRTOS

Particularly for IoT uses, FreeRTOS is a well-known open-source RTOS kernel found in embedded systems.

FreeRTOS has three basic ideas guiding its design:

- **Simplicity**: Simple to grasp and put to use.
- **Portability**: It can operate on several different processor systems.
- **Flexibility**: Lets you change things depending on what you need.

Many architectures are supported by FreeRTOS, including ARM, AVR, PIC, MSP430, and ESP32. It also works with a variety of platforms, like Arduino, Raspberry Pi, and even AWS IoT.

Important aspects and services provided by FreeRTOS consist of:

1. Task

FreeRTOS lets you build and manage several tasks that may run simultaneously across several processor cores or on a single core.
Every task includes:

- Priority will decide the sequence of execution.
- Manage memory usage by stack size.
- Optional name meant to help with task management and bug fixes.

API allow users to create, remove, suspend, resume, postpone, or synchronize tasks.

2. Queue

For synchronization and inter-task communication, FreeRTOS offers queues.

- A data structure storing a set of objects is called a queue.
- Through queues, tasks can send and retrieve data.
- Queue can also be used to enforce access-control by means of semaphores and mutexes.

3. Timer

FreeRTOS has software timers that allow:

- Execution of a callback function periodically or one-shot.
- Timer objects with properties such as period, expiry time, and optional name.
- Timers can be created, deleted, started, stopped, and reset using the available APIs.

4. Event Groups

Event groups are used to signal between tasks or between tasks and interrupts. Characteristics include:

- Based on bit flags that can be set or cleared individually or together.
- Tasks can wait for one or more specific bits in an event group to be set before proceeding, using the provided APIs.

5. Notification

FreeRTOS provides task notifications for lightweight and fast communication between tasks or between tasks and interrupts. Key points:

- Notifications are 32-bit values sent to a task using APIs.
- They might serve as basic data values, mutexes, event flags, or even a semaphore.

Practical Sections

Setting Up FreeRTOS on ESP-32

Two cores in ESP-32 let this low-power microcontroller operate:

- `CPU0`: Handles BLE, Bluetooth, and Wi-Fi wireless protocols.
- `CPU1`: Executes code for user apps.

Installing and configuring the ESP-32 Arduino Core:
1. Obtain the most recent Arduino IDE and install it.
2. Launch Arduino IDE then go to File / Preferences. Enter in the field **Additional Boards Manager URLs**:

https://dl.espressif.com/dl/package_esp32_index.json

3. Go to Tools / Board / Boards Manager, look for **esp32**, and install the most recent release from Espressif Systems.
4. Go to Tools / Board / ESP32 Arduino and pick the right board (like **ESP32 Dev Module** or **ESP32 Wrover Module**).

All about FreeRTOS APIs

1. xTaskCreate()

- Purpose: Creates a new task and dynamically allocates the required memory. Returns a handle to the created task, or NULL if creation fails. Syntax:

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,
    const char * const pcName,
    const uint32_t usStackDepth,
    void * const pvParameters,
    UBaseType_t uxPriority,
    TaskHandle_t * const pvCreatedTask
);
```

Parameters

1. **pvTaskCode**: Pointer to the function that implements the task. The function must have a prototype of void vTaskCode(void * pvParameters).
2. **pcName**: Descriptive name of the task (helps with debugging).
3. **usStackDepth**: Stack size in words (not bytes) for the task.
4. **pvParameters**: Pointer to the arguments passed to the task function.
5. **uxPriority**: Priority of the task execution (higher number = higher priority).
6. **pvCreatedTask**: Pointer to the variable that will receive the created task handle.

Return Value

1. **pdPASS**: The task was successfully created and added to the ready list.
2. **errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY**: The task could not be created because there is not enough heap memory available.

---

2. xTaskCreatePinnedToCore()

- Purpose: Similar to xTaskCreate(), except that you can specify the core number on which the task will run. This is useful for performance reasons. Here is the syntax of the function:

```
BaseType_t xTaskCreatePinnedToCore(
TaskFunction_t pvTaskCode,
const char * const pcName,
const uint32_t usStackDepth,
void * const pvParameters,
UBaseType_t uxPriority,
TaskHandle_t * const pvCreatedTask,
const BaseType_t xCoreID
);
```

Parameters

Same as xTaskCreate(), except for:

1. **xCoreID**: The core number on which the task should run. Can be 0 or 1 for a dual-core target ESP, or any other valid number of cores for a multi-core target ESP, i.e. 2 cores, 3 cores, etc.

Return Value

Same as xTaskCreate().

---

3. vTaskDelete()

- Purpose: Delete a task and free the memory allocated by it; delete other tasks. The syntax of this function is as follows:

```
void vTaskDelete(TaskHandle_t xTask);
```

Parameters

1. **xTask**: The handler of the task to be deleted. Passing NULL will delete the calling task.

Return Value

None.

---

4. vTaskDelay()

- Purpose: Causes the calling task to block for the specified number of ticks (ms). The syntax of this function is as follows:

```
void vTaskDelay(const TickType_t xTicksToDelay);
```

Parameters

1. **xTicksToDelay**: The number of ticks to delay. One tick = the unit of time specified by the configTICK_RATE_HZ configuration constant in FreeRTOSConfig.h.

Return Value

None.

---

5. vTaskDelayUntil()

- Purpose: This function blocks the calling task for a specified period of time, relative to the time the function was last called. In other words, it can be used when you want a task to run with a fixed frequency. The syntax of this function is as follows:

```
void vTaskDelayUntil(TickType_t * const pxPreviousWakeTime, const TickType_t xTimeIncrement);
```

Parameters

1. **pxPreviousWakeTime**: Pointer to a TickType_t variable that stores the time when the task was last unblocked. This variable must be initialized with the current time before the first call to the vTaskDelayUntil() function. The function will update the variable with the current time after each call.
2. **xTimeIncrement**: The time period between executions (cycle time) in ticks. The task will be unblocked at times (pxPreviousWakeTime + xTimeIncrement), (pxPreviousWakeTime +

xTimeIncrement2), and so on.

Return Value

None.

---

6. vTaskSuspend()

- Purpose: This function temporarily suspends a task, preventing it from being scheduled until it is reactivated by another task. The syntax of this function is as follows:

```
void vTaskSuspend(TaskHandle_t xTaskToSuspend);
```

Parameters

1. **xTaskToSuspend**: The handle of the task to be suspended. Passing a NULL value will suspend the calling task.

Return Value

None.

---

7. vTaskResume()

- Purpose: This function reactivates a task that has been paused by vTaskSuspend(). The syntax of this function is as follows:

```
void vTaskResume(TaskHandle_t xTaskToResume);
```

Parameters

1. **xTaskToResume**: The handle of the task to be reactivated.

Return Value

None.

---

8. vTaskPrioritySet()

- Purpose: Change the priority of a task. The syntax of this function is as follows:

```
void vTaskPrioritySet(TaskHandle_t xTask, UBaseType_t uxNewPriority);
```

Parameters

1. **xTask**: The handle of the task whose priority will be changed. Passing a NULL value will change the priority of the calling task.
2. **uxNewPriority**: The new priority for the task.

Return Value

None.

---

9. uxTaskPriorityGet()

- Purpose: Returns the priority of a task. The syntax of this function is as follows:

```
UBaseType_t uxTaskPriorityGet(TaskHandle_t xTask);
```

Parameters

1. **xTask**: The handle of the task whose priority is to be obtained. Passing a NULL value will return the priority of the calling task.

Return Value

- The priority of the task.

---

10. eTaskGetState()

- Purpose: Returns the status of a task. The syntax of this function is as follows:

```
eTaskState eTaskGetState(TaskHandle_t xTask);
```

Parameters

1. **xTask**: The handle of the task whose status is to be obtained.

Return Value

1. **xTask**: The handle of the task whose status is to be returned.

The following are the possible task states:

Status	Description
eRunning	The task is running.
eReady	The task is ready to run.

eBlocked	The task is blocked, waiting for an event.
eSuspended	The task is temporarily suspended.
eDeleted	The task has been deleted.
eInvalid	The task marker is invalid.

Additional References

- [1]“Introduction to RTOS Part 3 - Task Scheduling | Digi-Key Electronics,” www.youtube.com. https://www.youtube.com/watch?v=95yUbClyf3E.
- [2]“Introduction to RTOS - Solution to Part 3 (Task Scheduling),” DigiKey, 2021. https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-3-task-scheduling/8fbb9e0b0eed4279a2dd698f02ce125f.