# Module 1 - Introduction to SMP with RTOS

# 1.1 Learning Objectives

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

- Understand the difference between a General-Purpose Operating System (GPOS) and a Real-Time Operating System (RTOS)
- Understand the differences and benefits of Multi-Threading on a Microcontroller
- Understand the specifications of the ESP-32 as a microcontroller for IoT purposes

## What Will We Learn?

In previous courses, you may already be familiar with Operating Systems (OS) and the use of Microcontrollers for Cyber-Physical Systems.

In those two cases, you used Linux, a General-Purpose Operating System (GPOS), and programmed using Arduino with a bare-metal architecture (running directly on the hardware without an OS).

In this IoT practicum, we will introduce the Real-Time Operating System (RTOS), a crucial architecture in embedded systems and IoT applications.

# 1.2 Introduction to RTOS

## GPOS

The types of OS we often use (Windows, Linux, Mac, Android, iOS) can be classified as GPOS, which, as the name suggests, are designed for general purposes and typically utilize a GUI or CLI as the human interaction interface.

GPOS systems are designed to run multiple processes simultaneously, generally supported by multitasking and multi-threading, allowing the user to run several tasks at once. In general, timing deadlines for tasks in a GPOS are not crucial, and delays in tasks can be tolerated as long as they are not noticeable to the user[1]. For example, when a user opens a PDF document while listening to music on Spotify, both applications can run concurrently. If the system experiences a slight delay, such as the PDF page rendering taking longer or the audio buffering for a moment, this is still tolerable and does not significantly disrupt the user experience.

GPOS is non-deterministic, meaning the OS does not guarantee that a task will be fully completed within its allocated time (non-strict deadlines) [2]. This is not an issue for everyday GPOS applications that do not require strict timing certainty.

## RTOS

Unlike GPOS, an RTOS plays a critical role in certain real-world applications. Imagine you are designing a car's airbag system, where the system must process sensor data with extreme speed and accuracy during a collision, then immediately deploy the airbag within microseconds. Even a slight delay could be fatal, thus requiring an operating system capable of guaranteeing real-time responses and consistently meeting strict deadlines.

Generally, an RTOS is designed to run on a microcontroller that does not have a user interface (GUI / CLI). The main advantage of an RTOS is its deterministic scheduling method, meaning the start time of a task can be known before it begins [1]. This ensures the timeliness of task execution, allowing the system to respond to events consistently, which is often crucial in IoT or embedded systems applications.

# 1.3 Microcontroller Architecture

Besides the differences in the type of OS used, there are also differences in the microcontrollers used. In this IoT lab, the ESP-32 microcontroller is used, which differs from the Arduino Uno used in the Embedded Systems lab. Look at the table below for a comparison between the two microcontrollers. [3]

What-Are-the-Advantages-of-EPP32-Over-Arduino-UNO

In addition to the increase in RAM and wireless communication modules, a key difference between the ESP32 and the Arduino UNO is the number of cores.

This means the ESP32 can achieve true parallelism in executing its tasks.

## Why is this important?

First, let's look at the program structure used on the Arduino Uno, which lacks parallelism and runs on bare metal, often using a "Superloop" architecture. In this architecture, a single setup process is performed to initialize components before entering a loop that executes all tasks. During the loop, interrupts can be processed based on external events. For many use cases, this architectural structure is sufficient to complete the required tasks.

a5ac711c-328b-48f6-9d8c-f207e3abb184

Tasks in this architecture are executed sequentially. Consequently, if there are many tasks to run, there is a possibility of missing deadlines.

For example, if you create a device to read sensor data (like temperature or heart rate) and simultaneously upload it in real-time to a server via a WiFi connection, the superloop architecture on an Arduino Uno would struggle. This is because the process of communicating with the server (e.g., an HTTP request) takes a relatively long time and will block the main loop. As a result, sensor readings could be delayed or even missed entirely. This means that if sensor data needs to be read with precision (e.g., every few microseconds or milliseconds), this architecture cannot guarantee that timing.

## The Solution?

The ESP-32 can leverage parallelism to complete these tasks so they run concurrently.

GPOS systems are designed to run multiple processes at once, commonly supported by multi-tasking and multi-threading, so the user can execute several tasks simultaneously. In general, timing deadlines for tasks on a GPOS are not crucial, and delays can be tolerated as long as they are not visible to the user.

In this context, the RTOS acts as the operating system that manages resource allocation and scheduling for each task. [1]. 16adc64e-5857-4505-b27e-e66b375037a1

Based on the image above, an RTOS can divide program execution into several tasks. For instance, Task 1 is responsible for reading data from a sensor, Task 2 is responsible for uploading data to a server, while Task 3 can be used for a periodic process like logging.

Each task has its own priority, and through the RTOS API, we can configure it to run on a specific thread or schedule it as needed. In this way, the RTOS ensures that each task can be executed concurrently and on schedule, without interfering with each other.

This does not mean the number of tasks in an RTOS is limited to the two cores of the ESP-32. Rather, the RTOS can manage priorities and perform event scheduling and time-slicing to ensure the timeliness and deadlines of each task are met according to its priority.

Takeaway

Although an RTOS offers many advantages, it doesn't mean the bare-metal (super loop) architecture is always unsuitable. On 8-bit microcontrollers like the Arduino UNO (ATmega328p), the bare-metal approach is actually more efficient because the overhead of an RTOS scheduler is too large for the available resources. With bare metal, simple applications like reading a sensor, turning on an LED, or serial communication can run more lightly without additional overhead. Consequently, microcontrollers that utilize an RTOS tend to have higher minimum specifications.

cbf78b6c-3d84-4222-a636-2b619714ef66

When moving to more powerful microcontrollers like the ESP32, the use of an RTOS becomes increasingly relevant. Besides having dual-core capabilities, the ESP32 is designed for IoT applications, thus requiring multitasking capabilities so that the WiFi and Bluetooth stacks can run concurrently with the user's application, while also ensuring the deadlines of its operations are met.

# 1.4 FreeRTOS

So what is FreeRTOS? [4]

FreeRTOS is one of the most widely used RTOS implementations in the world of embedded systems and IoT. As its name implies, FreeRTOS is open-source and free to use.

Based on the previous explanation of RTOS, FreeRTOS acts as a lightweight OS that runs on a microcontroller (like the ESP32) to perform task scheduling, memory management, and inter-task communication. FreeRTOS provides an API that allows developers to:

- Create and manage multiple tasks that can run concurrently.
- Set priority scheduling so that important tasks (e.g., critical sensor readings) are always processed faster than low-priority tasks (e.g., logging).
- Use time-slicing and event-driven scheduling so that tasks outnumbering the available cores can still run according to their schedule.

# 1.5 Additional References

- What Is a Real-Time Operating System (RTOS)? – DigiKey Maker.io
- Real-Time Operating System (RTOS): Components, Types, Examples – Guru99
- RTOS Fundamentals – FreeRTOS Official Documentation