

Module 1:

Introduction to RTOS

& Task Scheduling

Learning Objectives

1. Understand the fundamentals of Real-Time Operating Systems (RTOS).
2. Explore the basics of task scheduling and prioritization.
3. Learn how to create and manage tasks using RTOS APIs.
4. Understand the role of tick interrupts in task scheduling.
5. Get introduced to basic task communication and synchronization mechanisms.

- [Module 1: Introduction to RTOS & Task Scheduling](#)
- [Code Sample](#)

Module 1: Introduction to RTOS & Task Scheduling

Understanding Real-Time Operating Systems (RTOS)

A Real-Time Operating System (RTOS) is a type of operating system designed to meet the time constraints of real-time applications. Real-time applications are those that have strict time limits for completing their tasks, such as controlling robots, processing sensor data, or streaming audio/video. An RTOS ensures that real-time application tasks are executed predictably and deterministically, without being affected by other tasks or external events.

RTOS differs from general-purpose operating systems (GPOS) such as Windows, Linux, or macOS in several ways. GPOS are optimized for user experience and functionality rather than timing performance. GPOS can use complex algorithms and data structures to manage memory, files, processes, and resources, which can introduce variability and non-determinism in task execution times. GPOS may also perform background activities such as software updates, file indexing, or virus scanning, which can interfere with the primary tasks of real-time applications.

In contrast, RTOS is optimized for simplicity and efficiency rather than functionality and user experience. RTOS typically uses fixed-size memory blocks, static data structures, and pre-allocated resources to minimize overhead and latency. RTOS also avoids performing background activities unrelated to real-time applications. RTOS may sacrifice some common features and services found in GPOS, such as graphical user interfaces, file systems, networking, or security, to achieve better timing performance.

An RTOS generally consists of the following components:

- **Kernel:** The core of the operating system that manages tasks, interrupts, memory, and other resources.
 - **Scheduler:** A part of the kernel that decides which task to run next based on their priorities and deadlines.
 - **Set of APIs:** Functions that allow applications to interact with the operating system and access its services.
 - **Set of device drivers:** Software modules that enable communication with hardware devices.
-

Importance of Task Scheduling in IoT

Task scheduling is one of the most important functions of an RTOS. Task scheduling is the process of deciding which task will be executed on the CPU at a given time. A task is a unit of work that performs a specific function in a real-time application. For example, a task might read data from a sensor, process an image, or send a message to another device.

Task scheduling is crucial in IoT (Internet of Things) applications, which involve numerous devices interacting with each other and their environment through sensors and actuators. IoT applications often have strict time constraints and require high reliability and responsiveness. For example, an IoT application might monitor temperature and humidity in a greenhouse and control the ventilation and irrigation systems accordingly. The application must ensure that sensor readings are accurate and timely, and that control actions are executed quickly and correctly.

Task scheduling in an RTOS aims to achieve two main objectives: feasibility and optimality. Feasibility means that all tasks can meet their deadlines without missing any. Optimality means that some performance criteria, such as CPU utilization, power consumption, or response time, are maximized or minimized. Achieving both objectives can be challenging, especially when there are many tasks with different priorities, periods, deadlines, execution times, and dependencies.

Task scheduling is the process of assigning tasks to processors and determining the order and timing of their execution. Task scheduling in IoT is important for the following reasons:

1. Ensuring that tasks meet deadlines and quality of service requirements, such as latency, throughput, reliability, and energy efficiency.
 2. Optimizing the use of resources, such as CPU, memory, bandwidth, and power.
 3. Balancing the workload among multiple processors or cores, especially in multicore systems.
 4. Adapting to dynamic changes in the environment, such as workload variations, network conditions, or user preferences.
-

Types of Task Scheduling Algorithms

There are many types of task scheduling algorithms that can be used in an RTOS. Each algorithm has its advantages and disadvantages, depending on the characteristics of the tasks and the system. Some of the most common types of task scheduling algorithms are:

1. **Run to Completion (RTC):** The RTC scheduler is very simple. It runs one task until it completes, then stops it. After that, it runs the next task in the same way. This continues until all tasks have been run, then the sequence starts again. The simplicity of this scheme has the drawback that the allocation of time for each task is entirely influenced by the others. The system will not be very deterministic.

2. **Round Robin (RR):** The RR scheduler is similar to the RTC scheduler, except that a task does not need to finish its work before releasing the CPU. When rescheduled, it resumes from where it left off. The RR scheduler gives each task an equal share of CPU time in a circular order. This scheme is more flexible than RTC but still depends on the behavior of each task and does not hold the processor for too long.
 3. **Time Slice (TS):** The TS scheduler is a type of preemptive scheduler that divides time into slots or ticks. Each slot might be 1 ms or less. At each tick interrupt, the scheduler selects one task to run from those ready to execute. The TS scheduler ensures that no task "starves" for CPU time but can introduce frequent context switches.
 4. **Fixed Priority (FP):** The FP scheduler assigns each task a static priority level to indicate its relative urgency. The scheduler always selects the highest priority task from those ready to execute. If tasks with the highest priority have the same priority, they are executed in a round-robin fashion. If a higher priority task than the currently running task becomes ready, the higher priority task will immediately replace the lower priority task. The FP scheduler is widely used in real-time systems because it is simple and effective.
 5. **Earliest Deadline First (EDF):** The EDF scheduler assigns each task a dynamic priority based on their deadlines. The scheduler always selects the task with the nearest deadline from those ready to execute. If two tasks have the same deadline, they are executed in a round-robin fashion. The EDF scheduler is optimal in the sense that it can schedule any feasible set of tasks. However, it can be difficult to implement and verify in practice.
-

Introduction to FreeRTOS

FreeRTOS is a widely-used open-source RTOS kernel in embedded systems, especially for IoT applications. FreeRTOS is designed to be simple, portable, and customizable. It supports various architectures, such as ARM, AVR, PIC, MSP430, and ESP32. It also supports various platforms, such as Arduino, Raspberry Pi, and AWS. FreeRTOS offers many features and services, such as:

1. **Tasks:** FreeRTOS allows the creation of multiple tasks that can run concurrently on one core or multiple cores. Each task has a priority level, stack size, and optional name. Tasks can be created, deleted, suspended, resumed, delayed, or synchronized using various API functions.
2. **Queues:** FreeRTOS provides queues for communication and synchronization between tasks. A queue is a data structure that can store a fixed number of items. Tasks can send and receive items to and from a queue using API functions. Queues can also be used to implement semaphores and mutexes.
3. **Timers:** FreeRTOS provides software timers for periodic or one-shot execution of callback functions. A timer is an object with a period, expiration time, and optional name. Timers can be created, deleted, started, stopped, or reset using API functions.
4. **Event Groups:** FreeRTOS provides event groups for signaling between tasks or between tasks and interrupts. An event group is a set of bits that can be set or cleared individually or collectively. Tasks can wait for one or more bits to be set in an event group using API functions.

5. **Notifications:** FreeRTOS provides notifications for lightweight and fast communication between tasks or between tasks and interrupts. A notification is a 32-bit value that can be sent to a task using API functions. Notifications can be used as binary or counting semaphores, mutexes, event flags, or data values.
-

ESP32: A Powerful Microcontroller for IoT

The ESP32 is a versatile, low-cost microcontroller with built-in WiFi and Bluetooth capabilities, making it ideal for IoT (Internet of Things) applications. Developed by Espressif Systems, it features a dual-core processor, allowing it to handle multiple tasks simultaneously. The ESP32 is equipped with a wide range of peripherals, including ADCs (Analog-to-Digital Converters), DACs (Digital-to-Analog Converters), GPIOs (General-Purpose Input/Outputs), PWM (Pulse Width Modulation), and more. It also supports various communication protocols, such as SPI, I2C, and UART, enabling seamless integration with other devices and sensors. With its power-efficient design, the ESP32 is well-suited for battery-powered projects, providing robust performance for real-time applications, wireless communication, and sensor data processing.

CP2102 vs. CH340: Differences in USB-to-Serial Converters

Both the CP2102 and CH340 are USB-to-serial converters commonly used in microcontroller development boards, including those based on the ESP32. They serve the same primary function of facilitating communication between the microcontroller and a computer via USB, but they differ in several key aspects:

- **Manufacturer:**
 - **CP2102:** Produced by Silicon Labs, the CP2102 is known for its stability and reliability. It's a popular choice in higher-end development boards.
 - **CH340:** Made by WCH (Nanjing QinHeng Corp), the CH340 is a more cost-effective alternative, commonly found in budget-friendly boards.
- **Driver Support:**
 - **CP2102:** Drivers for CP2102 are widely supported across various operating systems, including Windows, macOS, and Linux. The installation process is generally straightforward, with minimal compatibility issues.
 - **CH340:** While also well-supported, CH340 drivers sometimes require manual installation, especially on macOS and Linux. In some cases, users may encounter more difficulties with initial setup compared to CP2102.
- **Performance:**
 - **CP2102:** Offers slightly better performance with higher data rates and lower latency, making it suitable for applications where speed and responsiveness are critical.

- **CH340:** While capable of handling most typical tasks, CH340 may exhibit slightly higher latency and lower maximum data rates compared to CP2102. However, it is usually sufficient for standard applications.
- **Cost:**
 - **CP2102:** Tends to be more expensive due to its advanced features and brand reputation.
 - **CH340:** More affordable, which is why it is often found in low-cost development boards and devices.

In summary, the CP2102 is generally preferred for its robust performance and ease of use, especially in professional or high-demand scenarios. The CH340, on the other hand, is a budget-friendly option that is adequate for most hobbyist and educational purposes.

Setting Up FreeRTOS on ESP32

The ESP32 is an affordable, power-efficient microcontroller that supports WiFi and Bluetooth connectivity. It has two cores: the protocol core (CPU0) and the application core (CPU1). The protocol core runs wireless protocol stacks such as WiFi, Bluetooth, and BLE, while the application core runs user application code. Here's how to install and configure the Arduino Core for ESP32:

1. Download and Install the Latest Version of the Arduino IDE:

- Download the latest version of the Arduino IDE from the [Arduino website](#).
- Install the Arduino IDE following the provided instructions.

2. Configure the Arduino IDE for ESP32:

- Open the Arduino IDE.
- Navigate to `File > Preferences`.
- In the `Additional Boards Manager URLs` field, enter:

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

- Click `OK`.

3. Install the ESP32 Board Package:

- Go to `Tools > Board > Boards Manager`.
- Search for `esp32`.
- Install the latest version of the `ESP32 by Espressif Systems` package.

4. Select Your ESP32 Board Model:

- Navigate to `Tools > Board > ESP32 Arduino`.
- Select your specific ESP32 board model (e.g., `ESP32 Dev Module` or `ESP32 Wrover Module`).

After completing these steps, your Arduino IDE will be set up to program the ESP32 using the FreeRTOS kernel. You can now start writing and uploading code to your ESP32 board.

Creating and Managing Tasks in FreeRTOS on ESP32

To create and manage tasks in FreeRTOS on the ESP32, you can use the following API functions:

1. **xTaskCreate()**

- This function creates a new task and dynamically allocates the required memory. It returns a handle to the created task or NULL if the task creation fails. The syntax of this function is:

```
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:
 - **pvTaskCode**: A pointer to the function that implements the task. The function must have the prototype `void vTaskCode(void *pvParameters)`.
 - **pcName**: A descriptive name for the task, typically used for debugging purposes.
 - **usStackDepth**: The number of words (not bytes) to allocate for the task's stack.
 - **pvParameters**: A pointer to a variable that will be passed as a parameter to the task function.
 - **uxPriority**: The priority at which the task will run. Higher numbers indicate higher priority.
 - **pvCreatedTask**: A pointer to a variable that will receive the handle of the created task.
- Return values:
 - **pdPASS**: The task was successfully created and added to the ready list.
 - **errCOULD_NOT_ALLOCATE_REQUIRED_MEMORY**: The task could not be created due to insufficient available heap memory.

2. **xTaskCreatePinnedToCore()**

- This function is similar to `xTaskCreate()` but allows you to specify the core number on which the task will run. This can be useful for performance or affinity reasons. The syntax is:

```

 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 are the same as `xTaskCreate()`, except:
 - xCoreID**: The core number on which the task should run. This can be 0 or 1 for dual-core ESP targets or a valid core number for multi-core ESP targets.
- Return values are the same as `xTaskCreate()`.

3. **vTaskDelete()**

- This function deletes a task and frees the memory allocated by it. It can be used to delete the calling task or another task. The syntax is:

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

- Parameters:
 - xTask**: The handle of the task to be deleted. Passing NULL will cause the calling task to be deleted.
- Return value: None.

4. **vTaskDelay()**

- This function blocks the calling task for a specified number of ticks (milliseconds). It can be used to implement periodic tasks. The syntax is:

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

- Parameters:
 - xTicksToDelay**: The number of ticks to delay. One tick is a unit of time defined by the `configTICK_RATE_HZ` constant in `FreeRTOSConfig.h`.
- Return value: None.

5. **vTaskDelayUntil()**

- This function blocks the calling task until a specific time, relative to the time when the function was last called. It can be used to implement tasks with a fixed frequency. The syntax is:

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

- Parameters:
 - pxPreviousWakeTime**: A pointer to a 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 this function. The function will update the variable with

the current time after each call.

- **xTimeIncrement**: The cycle time period, in ticks. The task will be unblocked at times `(pxPreviousWakeTime + xTimeIncrement)`, `(pxPreviousWakeTime + xTimeIncrement * 2)`, and so on.

- Return value: None.

6. **vTaskSuspend()**

- This function suspends a task, preventing it from being scheduled until it is resumed by another task. The syntax is:

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

- Parameters:

- **xTaskToSuspend**: The handle of the task to be suspended. Passing NULL will cause the calling task to be suspended.

- Return value: None.

7. **vTaskResume()**

- This function resumes a task that was suspended by `vTaskSuspend()`. The syntax is:

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

- Parameters:

- **xTaskToResume**: The handle of the task to be resumed.

- Return value: None.

8. **vTaskPrioritySet()**

- This function changes the priority of a task. The syntax is:

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

- Parameters:

- **xTask**: The handle of the task whose priority will be changed. Passing NULL will cause the priority of the calling task to be changed.

- **uxNewPriority**: The new priority for the task.

- Return value: None.

9. **uxTaskPriorityGet()**

- This function returns the priority of a task. The syntax is:

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

- Parameters:

- **xTask**: The handle of the task whose priority will be obtained. Passing NULL will cause the priority of the calling task to be returned.

- Return value:

- The priority of the specified task.

10. **eTaskGetState()**

- This function returns the state of a task. The syntax is:

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

- Parameters:
 - **xTask**: The handle of the task whose state will be obtained.
- Return value:
 - The possible states of the task are:
 1. **eRunning**: The task is currently running.
 2. **eReady**: The task is ready to run.
 3. **eBlocked**: The task is blocked, waiting for an event.
 4. **eSuspended**: The task is suspended.
 5. **eDeleted**: The task has been deleted.
 6. **eInvalid**: The task handle is invalid.

Code Sample

```
/**
 * FreeRTOS LED Demo
 *
 * One task flashes an LED at a rate specified by a value set in another task.
 *
 * Date: December 4, 2020
 * Author: Shawn Hymel
 * License: 0BSD
 */

// Needed for atoi()
#include <stdlib.h>

// Use only core 1 for demo purposes
#if CONFIG_FREERTOS_UNICORE
static const BaseType_t app_cpu = 0;
#else
static const BaseType_t app_cpu = 1;
#endif

// Settings
static const uint8_t buf_len = 20;

// Pins
static const int led_pin = LED_BUILTIN;

// Globals
static int led_delay = 500; // ms

/*****
// Tasks

// Task: Blink LED at rate set by global variable
void toggleLED(void *parameter)
{
```

```

while (1)
{
    digitalWrite(led_pin, HIGH);
    vTaskDelay(led_delay / portTICK_PERIOD_MS);
    digitalWrite(led_pin, LOW);
    vTaskDelay(led_delay / portTICK_PERIOD_MS);
}
}

// Task: Read from serial terminal
// Feel free to use Serial.readString() or Serial.parseInt(). I'm going to show
// it with atoi() in case you're doing this in a non-Arduino environment. You'd
// also need to replace Serial with your own UART code for non-Arduino.
void readSerial(void *parameters)
{
    char c;
    char buf[buf_len];
    uint8_t idx = 0;

    // Clear whole buffer
    memset(buf, 0, buf_len);

    // Loop forever
    while (1)
    {
        // Read characters from serial
        if (Serial.available() > 0)
        {
            c = Serial.read();
            // Update delay variable and reset buffer if we get a newline character
            if (c == '\n')
            {
                led_delay = atoi(buf);
                Serial.print("Updated LED delay to: ");
                Serial.println(led_delay);
                memset(buf, 0, buf_len);
                idx = 0;
            }
            else
            {
                // Only append if index is not over message limit

```

```

        if (idx < buf_len - 1)
        {
            buf[idx] = c;
            idx++;
        }
    }
}

}

}

}

//*****
// Main

void setup()
{
    // Configure pin
    pinMode(led_pin, OUTPUT);

    // Configure serial and wait a second
    Serial.begin(115200);
    vTaskDelay(1000 / portTICK_PERIOD_MS);
    Serial.println("Multi-task LED Demo");
    Serial.println("Enter a number in milliseconds to change the LED delay.");

    // Start blink task
    xTaskCreatePinnedToCore(    // Use xTaskCreate() in vanilla FreeRTOS
        toggleLED,             // Function to be called
        "Toggle LED",          // Name of task
        1024,                  // Stack size (bytes in ESP32, words in FreeRTOS)
        NULL,                   // Parameter to pass
        1,                      // Task priority
        NULL,                   // Task handle
        app_cpu);               // Run on one core for demo purposes (ESP32 only)

    // Start serial read task
    xTaskCreatePinnedToCore(    // Use xTaskCreate() in vanilla FreeRTOS
        readSerial,            // Function to be called
        "Read Serial",         // Name of task
        1024,                  // Stack size (bytes in ESP32, words in FreeRTOS)
        NULL,                   // Parameter to pass
        1,                      // Task priority (must be same to prevent lockup)
        NULL,                   // Task handle

```

```
app_cpu);          // Run on one core for demo purposes (ESP32 only)

// Delete "setup and loop" task
vTaskDelete(NULL);
}

void loop()
{
    // Execution should never get here
}
```