Chapter 5

Sleep modes and Multitasking

Dr. Zhang Jianwen
elezhan@nus.edu.sg
Office: E4-05-21

Power consumption different

- IoT applications should be power–efficient.
- Power consumptions of ESP32-S3 in different modes

Active mode	Modem Sleep	Light Sleep	Deep Sleep	Hibernation
Wi-Fi and CPU are active	CPU active, Wi-Fi off (light sleep)	CPU pause, Peripherals power down. Wi-Fi and RTC are operational.	CPU and peripherals power down. RTC are operational.	All power down except RTC IO and a small amount of RTC RAM.
160-260 mA	3-15 mA	800 μA=1.2 mA	10-20 μA	2.5 μA

Active modes

- Active mode
  - Normal operation mode where the CPU is running, and all peripherals (such as Wi-Fi, Bluetooth, etc.) can be active. The power consumption in this mode is relatively high.
  - when the device needs to perform tasks such as data processing, communication, or sensor reading.

Sleep modes

- Modem sleep mode
  - CPU remains active, but Wi-Fi and Bluetooth radios are turned off or are in a low-power state when not actively transmitting or receiving data. The CPU can operate at different frequencies depending on the application’s needs, and peripherals can remain active.
  - When the CPU needs to continue processing data but does not require constant network connectivity. An example would be a device that periodically sends data over Wi-Fi.

Sleep modes

- Light sleep mode
  - CPU can pause its operation while still keeping Wi-Fi and some other peripherals active. The chip reduces the clock speed and disables certain functions to conserve power, but the system can wake up quickly to handle tasks like maintaining Wi-Fi connectivity or responding to an interrupt.
  - When the device needs to maintain network connectivity with minimal power usage, such as waiting for incoming data or notifications.

Sleep modes

- Deep sleep mode
  - CPU and most peripherals are powered down, including Wi-Fi and Bluetooth. Only the RTC (Real-Time Clock) and RTC memory remain active, allowing the device to wake up based on a timer, an external signal (e.g., GPIO interrupt), or a sensor event. This mode is very power-efficient and is designed for long-term sleep with periodic wake-ups.
  - When the device needs to wake up periodically to perform a task, such as data logging at intervals or responding to an external trigger.

Sleep modes

- Hibernation mode (Hibernation-Like Deep Sleep for ESP32-S3)
  - Both the CPU and most peripherals, including RTC memory, are powered down. The only components that remain active are the RTC IO (for wake-up triggers) and a small amount of RTC RAM (for storing the wake-up reason and some minimal data). The device can only wake up from an external signal (like a GPIO interrupt) or a timer.
  - When the device needs to stay in a very low-power state for extended periods and wake up only occasionally, such as in a deep hibernation scenario where the device waits for a user interaction or a critical event.

Sleep mode APIs

- Sleep mode programming
  - Configure the device to enter different sleep modes, control wake-up sources, and manage power consumption.
  - APIs provided by the Espressif framework. Library “esp_sleep.h”.
  - Enter sleep modes

Sleep Mode	How to Start	Power Consumption	Power down or inactive
Modem Sleep	`WiFi.setSleep(true);`	Moderate (CPU active, modem off)	Modem (Wi-Fi, Bluetooth)
Light Sleep	`esp_light_sleep_start();`	Lower (CPU paused, peripherals off)	Pause CPU or clock down
Deep Sleep	`esp_deep_sleep_start();`	Low (CPU off, RTC peripherals on)	Only RTC and RTC memory
Hibernation	`esp_sleep_pd_config()` + `esp_deep_sleep_start();`	Very Low (Most components off)	Only RTC IO and a few RTC RAM.

Sleep mode APIs

- Wake-up and Post-wake-up
  - Timer for Light Sleep, Deep Sleep and Hibernation
  - GPIO for Light Sleep, Deep Sleep and Hibernation
  - Touchpad for Light Sleep and Deep Sleep

Mode	Wake-Up Sources	Post-Wake-Up Behavior	Wake-Up Time
Active	N/A	Already running, no wake-up needed	N/A
Modem Sleep	Data transmission, network events	Modem reactivates, CPU remains active	Instant for modem
Light Sleep	Timer, GPIO(EXT0, EXT1), Touchpad, UART, etc.	Resumes from where it left off; fast wake-up	Microseconds to ms
Deep Sleep	Timer, GPIO(EXT0, EXT1), Touchpad, ULP	Full reboot, starts from `setup()`	Milliseconds
Hibernation	GPIO (EXT0, EXT1), Timer (limited)	Full reboot, very limited retention	Milliseconds

Sleep mode Wake-Up APIs

- `esp_sleep_enable_timer_wakeup(uint64_t time_in_us);`

Configures the ESP32-S3 to wake up after a specified time in microseconds. It is for periodic tasks, such as waking up every few minutes to take a sensor reading.

time_in_us: time period to go into sleep mode.
Return: esp_err_t value. It is ESP_OK or ESP_ERR_INVALID_ARG.

```
esp_sleep_enable_timer_wakeup(10 * 1000000); // Wake up after 10 seconds
esp_deep_sleep_start();
```

Sleep mode Wake-Up APIs

- `esp_sleep_enable_ext0_wakeup(gpio_num_t gpio_num, int level);`

Enables wake-up from Deep Sleep or Light Sleep using an external GPIO (EXT0) on a specified pin. It is for applications that need to wake up when a button is pressed or a specific GPIO pin changes state.

gpio_num: GPIO pin number used as wake-up source
level: The logic level triggering the wake-up (LOW or HIGH).
Return: An esp_err_t value indicating the result of the operation. It is ESP_OK or ESP_ERR_INVALID_ARG.

```
esp_sleep_enable_ext0_wakeup(GPIO_NUM_0, 0); // Wake up on a LOW signal on GPIO 0
esp_deep_sleep_start();
```

Sleep mode Wake-Up APIs

- `esp_sleep_enable_ext1_wakeup(uint64_t mask, esp_sleep_ext1_wakeup_mode_t mode);`

Enables wake-up from Deep Sleep or Light Sleep using multiple GPIOs (EXT1), specified by a bitmask, with a combination of high or low levels. It is for multiple GPIOs can trigger a wake-up, such as multiple sensors.

mask: bitmask representing the GPIO pins that will be used as wake-up sources.
mode: specifies the condition under which the wake-up will occur. It can be ESP_EXT1_WAKEUP_ALL_LOW or ESP_EXT1_WAKEUP_ANY_HIGH.
Return: An esp_err_t value indicating the result of the operation. It is ESP_OK or ESP_ERR_INVALID_ARG.

```
uint64_t mask = (1ULL << GPIO_NUM_0) | (1ULL << GPIO_NUM_4);
// Wake up on HIGH signal on GPIO 0 or 4
esp_sleep_enable_ext1_wakeup(mask, ESP_EXT1_WAKEUP_ANY_HIGH);
esp_deep_sleep_start();
```

Sleep mode Wake-Up APIs

- esp_sleep_enable_touchpad_wakeup();

Enables wake-up from Deep Sleep using a touchpad. It is for touch-sensitive applications where the device wakes up on user interaction.
Return: esp_err_t value, which indicates whether the operation was successful(ESP_OK) or failed(ESP_ERR_INVALID_STATE).

```
esp_sleep_enable_touchpad_wakeup();
esp_light_sleep_start(); // Enter Light Sleep
```

- esp_sleep_get_wakeup_cause() ;

Retrieves the cause of the last wake-up from sleep mode.
Return: esp_sleep_wakeup_cause_t type, which is an enumeration that indicates the wake-up cause.

```
esp_sleep_wakeup_cause_t wakeup_reason = esp_sleep_get_wakeup_cause();
if (wakeup_reason == ESP_SLEEP_WAKEUP_TIMER) {
    Serial.println("Wakeup caused by timer");
}
```

Start sleep mode APIs

- `esp_light_sleep_start();`

Puts the ESP32-S3 into Light Sleep mode. The device can wake up from a timer, external interrupt, or other configured wake-up sources. It is used when the device needs to maintain network connectivity or other peripherals while conserving power

Return: An esp_err_t value indicating the result of the operation. It is ESP_OK or other value indicating what might be wrong.

```
esp_light_sleep_start(); // Enter Light Sleep
```

- `esp_deep_sleep_start();`

Puts the ESP32-S3 into Deep Sleep mode. The device powers down most peripherals and wakes up only from the configured wake-up sources. It is used for long periods of inactivity, such as when the device only needs to wake up periodically or based on a specific event.

Return: An esp_err_t value indicating the result of the operation. It is ESP_OK or other value indicating what might be wrong.

```
esp_deep_sleep_start(); // Enter Deep Sleep
```

Sleep mode configure APIs

- `esp_sleep_pd_config(esp_sleep_pd_domain_t domain, esp_sleep_pd_option_t option)`

Configures the power domains (like RTC peripherals or RTC memory) to be powered down or retained during sleep modes. It is for optimizing power consumption by selectively powering down unused components.

domain: specifies the power domain you want to configure. The power domain represents different sections of the ESP32-S3 chip that can be selectively powered down.
option: specifies the power-down option for the selected domain.
Return: esp_err_t indicating success(ESP_OK) or failure(ESP_ERR_INVALID_ARG).

```
// Keep RTC fast memory on during Deep Sleep
sp_sleep_pd_config(ESP_PD_DOMAIN_RTC_FAST_MEM, ESP_PD_OPTION_ON);
// Power down RTC peripherals during sleep
esp_sleep_pd_config(ESP_PD_DOMAIN_RTC_PERIPH, ESP_PD_OPTION_OFF);
esp_deep_sleep_start(); // Enter Deep Sleep

// Let the system decide the CPU power state
esp_sleep_pd_config(ESP_PD_DOMAIN_CPU, ESP_PD_OPTION_AUTO);
```

UART wakes-up from light sleep

- Light Sleep woken up by UART.

```
#include "esp_sleep.h"

void setup() {
  Serial.begin(115200);

  // Set up a callback to be triggered when UART receives data
  Serial.onReceive([](int bytesAvailable) {
    Serial.println("UART wake-up: Data received");
    // Handle the received data here
  });

  Serial.println("Entering light sleep. Send data to UART to wake up...");

  // Enable UART wake-up
  esp_sleep_enable_uart_wakeup(ESP_SLEEP_WAKEUP_UART0);

  // Enter Light Sleep mode
  esp_light_sleep_start();

  // This code resumes execution after wake-up
  Serial.println("Woke up from light sleep!");
}

void loop() {
  // Main loop code, if needed
}
```

Sleep mode use cases

- Periodic Sensor Reading (Deep Sleep + Timer Wake-Up)
  - Wakes up periodically to read sensor data, store it, and then go back to sleep.

```
esp_sleep_enable_timer_wakeup(15 * 60 * 1000000); // Wake up every 15 minutes
esp_deep_sleep_start();
```

- Button-Activated Device (Deep Sleep + GPIO Wake-Up)
  - Remains in Deep Sleep and wakes up when a button is pressed.

```
// Wake up on a LOW signal on GPIO 0
esp_sleep_enable_ext0_wakeup(GPIO_NUM_0, 0);
esp_deep_sleep_start();
```

Sleep mode use cases

- Low-Power Network Device (Light Sleep + Wi-Fi Wake-Up)
  - Maintains Wi-Fi connectivity in Light Sleep, waking up when there's incoming data.

```
esp_light_sleep_start(); // Enter Light Sleep, Wi-Fi stays connected
```

- Multi-Sensor Wake-Up (Deep Sleep + Multiple GPIOs)
  - Wakes up when any of several connected sensors detect an event.

```
uint64_t mask = (1ULL << GPIO_NUM_0) | (1ULL << GPIO_NUM_2);

// Wake up on HIGH signal on GPIO 0 or 2
esp_sleep_enable_ext1_wakeup(mask, ESP_EXT1_WAKEUP_ANY_HIGH);
esp_deep_sleep_start();
```

Sleep mode use cases

- Touch-Activated Device (Deep Sleep + Touchpad Wake-Up)
  - Wakes up from Deep Sleep when a touchpad is touched.

```
esp_sleep_enable_touchpad_wakeup();
esp_deep_sleep_start();
```

RTC-Related Programming

- Wake-Up from Deep-Sleep with Timer

```
void setup() {
  Serial.begin(115200);

  // Configure the wake-up timer (e.g., wake up after 10 seconds)
  esp_sleep_enable_timer_wakeup(10 * 1000000);

  Serial.println("Going to sleep now...");

  delay(1000);
  esp_deep_sleep_start(); // Enter deep sleep
}

void loop() {
  // This code will not be executed after deep sleep
}
```

RTC-Related Programming

- Keeping Data Across Deep-Sleep

```
RTC_DATA_ATTR int bootCount = 0; // Declare a variable in RTC slow memory

void setup() {
  Serial.begin(115200);
  bootCount++;
  Serial.printf("Boot count: %d\n", bootCount);

  // Set up a wake-up timer or GPIO wake-up
  esp_sleep_enable_timer_wakeup(10 * 1000000); // 10 seconds in microseconds

  Serial.println("Going to sleep now...");
  delay(1000);

  esp_deep_sleep_start(); // Enter deep sleep
}

void loop() {
  // This code will not be executed after deep sleep
}
```

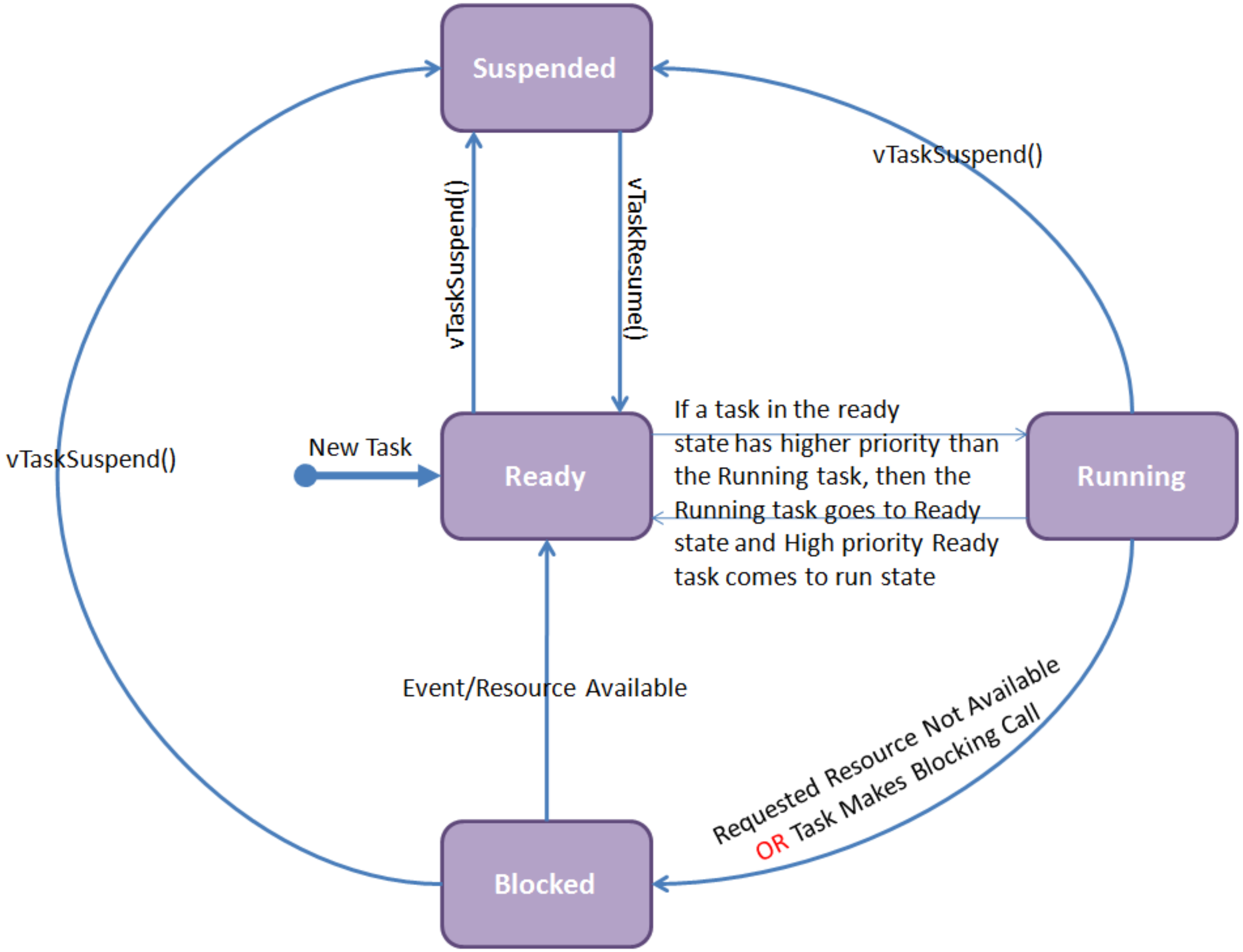
Multitasking

- Multitasking: Different parts of a program run simultaneously or appear to run simultaneously, sharing the processor's time and resources.
  - Tasks or Threads: Basic unit.
  - Task Scheduler: Schedule task to run based on various factors such as priority, task state, and resource availability.
  - Context Switching: Saving state of current task and restoring next task state.
  - Preemptive Multitasking: scheduler can interrupt a currently running task to start a higher-priority task

Multitasking

- Cooperative Multitasking: Tasks voluntarily yield CPU.
- Time-Slicing: A scheduling strategy where each task is given a fixed time slice to run.
- Task Creation: Defining tasks and task scheduling.
- Task states
  - Running: The task is currently being executed by the processor.
  - Ready: The task is ready to run and waiting for processor time.
  - Blocked: The task is waiting for an event or resource (e.g., waiting for input, delay, semaphore).
  - Suspended: The task is inactive and not considered for scheduling until explicitly resumed.

Task states and transitions



Multitasking implementation

- FreeRTOS
  - Light weight, open source real-time OS for embedded systems.
  - Arduino IDE includes supporting for FreeRTOS.
    - Task scheduler is running and provide a multitasking environment.
    - Setup and loop functions are two tasks under the scheduling.
  - APIs: Implement task manage for applications

APIs for task management

- xTaskCreate():Creates a new task and adds it to the list of tasks that the scheduler manages.

```
BaseType_t xTaskCreate(
    TaskFunction_t pvTaskCode,    // Pointer to the function to run as a task
    const char * const pcName,    // A descriptive name for the task
    uint16_t usStackDepth,        // Stack size in words
    void * const pvParameters,    // Parameters to pass to the task function
    UBaseType_t uxPriority,       // Task priority
    TaskHandle_t *pxCreatedTask   // Task handle to manage the task
);
```

- ❖ pvTaskCode: function return void type and take parameters from the void type list with pointer pvParameters.
- ❖ typedef unsigned int UBaseType_t; // 32 bits unsigned int
- ❖ TaskHandle_t: the task handle, a pointer to a task control block(structure).

APIs for task management

```
#include <Arduino.h>
#include <FreeRTOS.h>

// Task function prototype
void blinkTask(void *pvParameters);

void setup() {
    Serial.begin(115200);

    // Create a task
    xTaskCreate(
        blinkTask,       // Task function
        "Blink Task",    // Task name
        1000,            // Stack size in words
        NULL,            // No parameter passed to the task
        1,               // Task priority
        NULL             // Task handle (not used in this example)
    );
}

void loop() {
    // The loop is empty as tasks are managed by FreeRTOS
}

void blinkTask(void *pvParameters) {
    // Configure the LED pin as output
    pinMode(LED_BUILTIN, OUTPUT);

    // Task's main loop
    while (1) {
        digitalWrite(LED_BUILTIN, HIGH); // Turn LED on
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay for 500 milliseconds
        digitalWrite(LED_BUILTIN, LOW);  // Turn LED off
        vTaskDelay(500 / portTICK_PERIOD_MS); // Delay for 500 milliseconds
    }
}
```

APIs for task management

- xTaskCreatePinnedToCore(): Similar to xTaskCreate() but allows the task to be pinned to a specific core on dual-core processors.

```
BaseType_t xTaskCreatePinnedToCore(
    TaskFunction_t pvTaskCode,   // Function that implements the task.
    const char * const pcName,   // Name of the task.
    const uint32_t usStackDepth, // Stack size in words, not bytes.
    void * const pvParameters,   // Parameter passed to the task function.
    UBaseType_t uxPriority,      // Priority of the task.
    TaskHandle_t *pxCreatedTask, // Task handle (can be NULL if not used).
    const BaseType_t xCoreID     // Core to pin the task (0 or 1).
);
```

❖ There are two cores in ESP32-S3, 0 and 1.

❖ void * const pvParameters: pvParameters is a void type pointer. It points to a the same addr after it is initialized. The value in this addr can vary.

APIs for task management

```
#include <Arduino.h>

// Task function declaration
void TaskFunction(void *pvParameters);

void setup() {
    Serial.begin(115200);

    // Create a task pinned to core 1
    xTaskCreatePinnedToCore(
        TaskFunction,   // Task function
        "Task1",        // Name of the task
        1024,           // Stack size in words
        NULL,           // No parameters passed to the task function
        1,              // Task priority
        NULL,           // No task handle
        1               // Pin to core 1
    );
}

void loop() {
    // Main loop can be used for other tasks
    delay(1000);
}

// Define the task function
void TaskFunction(void *pvParameters) {
    while (true) {
        Serial.println("Running on Core 1");
        delay(1000); // Delay to simulate workload
    }
}
```

APIs for task management

- vTaskDelete(): Deletes a task and frees the memory allocated for its stack.

```
void vTaskDelete(TaskHandle_t xTaskToDelete);
```

❖ TaskHandle_t is needed if other task to be deleted. It can be NULL if deleting the current task.

APIs for task management

```
#include <Arduino.h>

// Task function declarations
void Task1(void *pvParameters);
void Task2(void *pvParameters);

TaskHandle_t task1Handle = NULL;

void setup() {
    Serial.begin(115200);

    // Create Task1
    xTaskCreate(
        Task1,              // Task function
        "Task1",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters passed to the task function
        1,                  // Task priority
        &task1Handle        // Task handle to reference Task1
    );

    // Create Task2
    xTaskCreate(
        Task2,              // Task function
        "Task2",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters passed to the task function
        1,                  // Task priority
        NULL                // No task handle needed for Task2
    );
}
```

```
void loop() {
    // Main loop remains empty or used for other tasks
    delay(1000);
}

// Define Task1 function
void Task1(void *pvParameters) {
    while (true) {
        Serial.println("Task1 is running.");
        delay(1000); // Delay to simulate workload
    }
}

// Define Task2 function
void Task2(void *pvParameters) {
    Serial.println("Task2 is deleting Task1.");
    vTaskDelete(task1Handle); // Delete Task1
    Serial.println("Task1 deleted.");

    // Delete itself after completing
    vTaskDelete(NULL); // Delete Task2 (the current task)
}
```

APIs for task management

- vTaskDelay(): Suspends the task for a specified number of ticks.

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

- ❖ TickType_t is an unsigned int type to represent time in terms of number of ticks.
- ❖ The argument of vTaskDelay() is often provided in the form of time(in ms)/portTICK_PERIOD_MS, where portTICK_PERIOD_MS is a macro represent the time(in ms) for one tick.
- ❖ This function is used to control the timing of tasks. Unlike delay() blocking the CPU for the task, this function yields the CPU for other tasks to use while blocking the current task.

APIs for task management

```
#include <Arduino.h>

// Task function declarations
void Task1(void *pvParameters);

void setup() {
    Serial.begin(115200);

    // Create Task1
    xTaskCreate(
        Task1,              // Task function
        "Task1",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters passed to the task function
        1,                  // Task priority
        NULL                // No task handle needed
    );
}

void loop() {
    // Main loop can be used for other tasks
    delay(1000);
}

// Define Task1 function
void Task1(void *pvParameters) {
    while (true) {
        Serial.println("Task1 is running.");

        // Delay the task for 1000 milliseconds (1 second). So that this task run periodically.
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Task Control APIs

- vTaskPrioritySet(): Dynamically change the priority of a specific task.

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

❖ uxNewPriority is an integer value between 0 to configMAX_PRIORITIES-1, where configMAX_PRIORITIES is a constant defined in FreeRTOS configuration file.

❖ High-priority tasks can pre-empt low-priority tasks.

Task Control APIs

```cpp
#include <Arduino.h>

// Task function declarations
void Task1(void *pvParameters);
void Task2(void *pvParameters);

TaskHandle_t task1Handle = NULL;

void setup() {
    Serial.begin(115200);

    // Create Task1 with a lower priority
    xTaskCreate(
        Task1,          // Task function
        "Task1",        // Name of the task
        1024,           // Stack size in words
        NULL,           // No parameters to pass
        1,              // Task priority
        &task1Handle    // Task handle for Task1
    );

    // Create Task2 with a higher priority
    xTaskCreate(
        Task2,          // Task function
        "Task2",        // Name of the task
        1024,           // Stack size in words
        NULL,           // No parameters to pass
        2,              // Higher task priority
        NULL            // No task handle needed
    );
}
```

```cpp
void loop() {
    // Main loop can be used for other tasks
    delay(1000);
}

// Define Task1 function
void Task1(void *pvParameters) {
    while (true) {
        Serial.println("Task1 is running at priority 1.");

       // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

// Define Task2 function
void Task2(void *pvParameters) {
    // Task2 will change the priority of Task1
    Serial.println("Task2 is running and changing Task1 priority to 
3.");

    vTaskPrioritySet(task1Handle, 3); // Change Task1 priority to 3

    while (true) {
        Serial.println("Task2 is running at priority 2.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Task Control APIs

- uxTaskPriorityGet(): Gets the priority of a specific task.

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

❖ If xTask is NULL, the calling task priority is returned.

Task Control APIs

```
#include <Arduino.h>

// Task function declarations
void Task1(void *pvParameters);
void Task2(void *pvParameters);

TaskHandle_t task1Handle = NULL;

void setup() {
    Serial.begin(115200);

    // Create Task1
    xTaskCreate(
        Task1,              // Task function
        "Task1",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters to pass
        2,                  // Task priority
        &task1Handle        // Task handle for Task1
    );

    // Create Task2
    xTaskCreate(
        Task2,              // Task function
        "Task2",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters to pass
        1,                  // Task priority
        NULL                // No task handle needed
    );
}

void loop() {
    // Main loop can be used for other tasks
    delay(1000);
}
```

```
// Define Task1 function
void Task1(void *pvParameters) {
    while (true) {
        Serial.println("Task1 is running.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}


// Define Task2 function
void Task2(void *pvParameters) {
    // Get the priority of Task1
    UBaseType_t task1Priority = uxTaskPriorityGet(task1Handle);

    Serial.print("Task2: Task1 current priority is ");
    Serial.println(task1Priority);

    // Do other tasks
    while (true) {
        Serial.println("Task2 is running.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Task Control APIs

- vTaskSuspend(): Suspends a specific task. The task will not be scheduled until it is resumed.

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

- ❖ Suspending a task prevents it from being scheduled to run, effectively pausing its execution until it is resumed.
- ❖ It can be useful for managing task execution flow, conserving CPU resources, and synchronizing tasks.
- ❖ If NULL is passed as argument, the calling task will be suspended.
- ❖ Use vTaskResume() to resume the task.

Task Control APIs

```
#include <Arduino.h>

// Task function declarations
void Task1(void *pvParameters);
void Task2(void *pvParameters);

TaskHandle_t task1Handle = NULL;

void setup() {
    Serial.begin(115200);

    // Create Task1
    xTaskCreate(
        Task1,              // Task function
        "Task1",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters to pass
        2,                  // Task priority
        &task1Handle        // Task handle for Task1
    );

    // Create Task2
    xTaskCreate(
        Task2,              // Task function
        "Task2",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters to pass
        1,                  // Task priority
        NULL                // No task handle needed
    );
}

void loop() {
    // Main loop can be used for other tasks
    delay(1000);
}
```

```
// Define Task1 function
void Task1(void *pvParameters) {
    while (true) {
        Serial.println("Task1 is running.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}


// Define Task2 function
void Task2(void *pvParameters) {
    // Get the priority of Task1
    UBaseType_t task1Priority = uxTaskPriorityGet(task1Handle);

    Serial.print("Task2: Task1 current priority is ");
    Serial.println(task1Priority);

    // Do other tasks
    while (true) {
        Serial.println("Task2 is running.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Task Control APIs

- vTaskResume(): Resumes a previously suspended task.

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

- ❖ xTaskToResume must be the handle of a task previously suspended using vTaskSuspend(). Passing an invalid task handle or a handle to a task that was not suspended may result in undefined behavior.
- ❖ It is useful for controlling the flow of execution in a FreeRTOS-based system, such as managing tasks that need to wait for specific events or conditions to proceed..

Task Control APIs

```
#include <Arduino.h>

// Task function declarations
void Task1(void *pvParameters);
void Task2(void *pvParameters);

TaskHandle_t task1Handle = NULL;

void setup() {
    Serial.begin(115200);

    // Create Task1
    xTaskCreate(
        Task1,              // Task function
        "Task1",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters to pass
        1,                  // Task priority
        &task1Handle        // Task handle for Task1
    );

    // Create Task2
    xTaskCreate(
        Task2,              // Task function
        "Task2",            // Name of the task
        1024,               // Stack size in words
        NULL,               // No parameters to pass
        2,                  // Higher task priority
        NULL                // No task handle needed
    );
}

void loop() {
    // Main loop can be used for other tasks
    delay(1000);
}
```

```
// Define Task1 function
void Task1(void *pvParameters) {
    while (true) {
        Serial.println("Task1 is running.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}


// Define Task2 function
void Task2(void *pvParameters) {
    // Suspend Task1 to simulate some condition
    Serial.println("Task2 is suspending Task1.");
    vTaskSuspend(task1Handle); // Suspend Task1
    Serial.println("Task1 is suspended.");

    // Simulate some work in Task2
    // Wait for 5 seconds
    vTaskDelay(5000 / portTICK_PERIOD_MS);

    // Resume Task1 after completing work
    Serial.println("Task2 is resuming Task1.");
    vTaskResume(task1Handle); // Resume Task1
    Serial.println("Task1 is resumed.");

    // Do other tasks
    while (true) {
        Serial.println("Task2 is running.");

        // Delay to simulate workload
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}
```

Inter-Task Communication APIs - Queue

- `xQueueCreate()`: Creates a queue to hold a specified number of items.

```
QueueHandle_t xQueueCreate(UBaseType_t uxQueueLength, UBaseType_t uxItemSize);
```

- **uxQueueLength**: the maximum number of items that the queue can hold.
- **uxItemSize**: the size, in bytes, of each item that the queue will hold.
- **Return:** a handle to the created queue (of type QueueHandle_t) if the queue was successfully created. Otherwise NULL is returned if the queue could not be created.

Inter-Task Communication APIs-Queue

- xQueueSend(): Sends an item to the back of the queue.

```
BaseType_t xQueueSend(QueueHandle_t xQueue, const void *pvItemToQueue,
                                              TickType_t xTicksToWait);
```

- **xQueue:** the handle to the queue to which the data is being sent.
- **pvItemToQueue**: a pointer to the item that is to be placed on the queue.
- **xTicksToWait**: the maximum amount of time the task should block (wait) if the queue is full. It is in the range of 0 to portMAX_DELAY.
- **Return:** pdPASS if the item was successfully sent to the queue or errQUEUE_FULL (0) if the item could not be sent due to the queue being full.

Inter-Task Communication APIs - Queue

- xQueueReceive(): Receives an item from the queue.

```
BaseType_t xQueueReceive(QueueHandle_t xQueue, void *pvBuffer,
                                                    TickType_t xTicksToWait);
```

- **xQueue**: the handle to the queue from which the data will be received.
- **pvBuffer:** a pointer to the memory location where the received item will be stored.
- **xTicksToWait:** the maximum amount of time the task should block (wait) for an item to be available if the queue is empty. It is in the range of 0 to portMAX_DELAY.
- **Return**: pdTRUE (1) if success or pdFALSE (0) if fail.

Inter-Task Communication APIs-Queue

```
#include <Arduino.h>
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"

QueueHandle_t myQueue;

void producerTask(void *pvParameters) {
    int count = 0;
    while (1) {
        count++;
        // Send count to the queue
        if (xQueueSend(myQueue, &count, portMAX_DELAY) == pdPASS) {
            Serial.println("Sent: " + String(count));
        }
        vTaskDelay(1000 / portTICK_PERIOD_MS); // 1-second delay
    }
}

void consumerTask(void *pvParameters) {

    int receivedValue;

    while (1) {
        // Receive value from the queue
        if (xQueueReceive(myQueue, &receivedValue, portMAX_DELAY) == pdTRUE) {
            Serial.println("Received: " + String(receivedValue));
        }
    }
}
```

```
void setup() {
    Serial.begin(115200);

    // Create a queue that can hold up to 5 integers
    myQueue = xQueueCreate(5, sizeof(int));

    if (myQueue == NULL) {
        Serial.println("Failed to create queue.");
        while (1);
    }

    // Create tasks
    xTaskCreate(producerTask, "Producer Task", 2048, NULL, 1, NULL);
    xTaskCreate(consumerTask, "Consumer Task", 2048, NULL, 1, NULL);
}

void loop() {
    // Do nothing here. The tasks will handle the execution.
}
```

Inter-Task Communication APIs - Semaphore

- `xSemaphoreCreateBinary()`: Creates a binary semaphore.

```
SemaphoreHandle_t xSemaphoreCreateBinary(void);
```

- ❖ xSemaphoreCreateBinary() returns the handle of the semaphore created. The handle is the type of SemaphoreHandle_t which is a pointer to the semaphore. If there is insufficient heap memory available to create the semaphore, the function returns NULL.
- ❖ In FreeRTOS, a semaphore can be seen as a token that tasks or interrupts can take or give.
- ❖ A binary semaphore can either be in a taken state (0) or an available state (1).

Inter-Task Communication APIs - Semaphore

- xSemaphoreGive(): Gives the semaphore, incrementing its count.

```
BaseType_t xSemaphoreGive(SemaphoreHandle_t xSemaphore);
```

- **xSemaphore:** the handle to the semaphore being given. It must be created with one of the semaphore creation functions like xSemaphoreCreateBinary().
- **Return**: returns pdTRUE if the semaphore was successfully given, otherwise, it returns pdFALSE.

Inter-Task Communication APIs - Semaphore

- xSemaphoreTake(): Takes the semaphore, decrementing its count, and waits if the count is zero.

```
BaseType_t xSemaphoreTake(SemaphoreHandle_t xSemaphore, TickType_t xTicksToWait);
```

- **xSemaphore**: the handle to the semaphore being taken. This is the semaphore created with functions like xSemaphoreCreateBinary().
- **xTicksToWait**: the maximum amount of time, in tick periods, that the task should wait for the semaphore to become available. Use portMAX_DELAY to wait indefinitely.
- **Return**: pdTRUE if the semaphore was successfully taken, otherwise pdFALSE is returned.

Inter-Task Communication APIs - Semaphore

```
#include <Arduino.h>
#include <freertos/FreeRTOS.h>
#include <freertos/semphr.h>

SemaphoreHandle_t xBinarySemaphore;
int sharedData = 0;  // Shared resource

void producerTask(void *parameter) {
    while (true) {
   // Simulate data production by generating a random number
        sharedData = random(1, 100);
        Serial.print("Producer: Generated data = ");
        Serial.println(sharedData);

        // Signal the consumer that new data is ready
        xSemaphoreGive(xBinarySemaphore);

        // Simulate some production delay
        vTaskDelay(1000 / portTICK_PERIOD_MS);
    }
}

void consumerTask(void *parameter) {
    while (true) {
        // Wait for the signal from the producer
        if (xSemaphoreTake(xBinarySemaphore, portMAX_DELAY) == pdTRUE) {
            // Process the data (print it in this case)
            Serial.print("Consumer: Consumed data = ");
            Serial.println(sharedData);
        }
    }
}
```

```
void setup() {
    Serial.begin(115200);
    randomSeed(analogRead(0)); // Initialize random seed

    // Create the binary semaphore
    xBinarySemaphore = xSemaphoreCreateBinary();

    if (xBinarySemaphore != NULL) {
        // Create the producer and consumer tasks
        xTaskCreate(producerTask, "Producer Task", 1000, NULL, 1, NULL);
        xTaskCreate(consumerTask, "Consumer Task", 1000, NULL, 1, NULL);
    } else {
        Serial.println("Failed to create semaphore");
    }
}

void loop() {
    // Do nothing in the loop
}
```

Inter-Task Communication APIs - Mutexes

- xSemaphoreCreateMutex(): Creates a mutex.

```
SemaphoreHandle_t xSemaphoreCreateMutex(void);
```

- xSemaphoreCreateMutex() creates a mutex type semaphore. A mutex (Mutual Exclusion) is a specialized type of semaphore used to manage access to a shared resource.
- xSemaphoreCreateMutex(), xSemaphoreGive(), and xSemaphoreTake() are often used together to manage access to shared resources in a multitasking environment.
- **Return**: returns a handle to the mutex (SemaphoreHandle_t) if successful, or NULL if there is insufficient heap memory available to create the mutex.

Inter-Task Communication APIs - Mutexes

```
#include <Arduino.h>
#include <freertos/FreeRTOS.h>
#include <freertos/semphr.h>

SemaphoreHandle_t xMutex;

void task1(void *parameter) {
    while (true) {
        // Attempt to take the mutex
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            Serial.println("Task 1: Acquired the mutex");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            xSemaphoreGive(xMutex);
            Serial.println("Task 1: Released the mutex");
        }
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}

void task2(void *parameter) {
    while (true) {
        // Attempt to take the mutex
        if (xSemaphoreTake(xMutex, portMAX_DELAY) == pdTRUE) {
            Serial.println("Task 2: Acquired the mutex");
            vTaskDelay(1000 / portTICK_PERIOD_MS);
            xSemaphoreGive(xMutex);
            Serial.println("Task 2: Released the mutex");
        }
        vTaskDelay(500 / portTICK_PERIOD_MS);
    }
}
```

```
void setup() {
    Serial.begin(115200);

    // Create a mutex
    xMutex = xSemaphoreCreateMutex();

    if (xMutex != NULL) {
        xTaskCreate(task1, "Task 1", 1000, NULL, 1, NULL);
        xTaskCreate(task2, "Task 2", 1000, NULL, 1, NULL);
        Serial.println("Mutex created and tasks started");
    } else {
        Serial.println("Failed to create mutex");
    }
}

void loop() {
    // Do nothing in loop
}
```

Inter-Task Communication APIs - Timer Management

- xTimerCreate(): Creates a software timer.

```
TimerHandle_t xTimerCreate(
    const char *pcTimerName,        // Text name for the timer

    TickType_t xTimerPeriodInTicks,
       // Timer period in ticks (1 tick = 1/portTICK_PERIOD_MS seconds)

    UBaseType_t uxAutoReload,
       // Auto-reload flag (1 for auto-reload, 0 for one-shot)

    void *pvTimerID,
       // Timer ID (optional, can be used to store data)

    TimerCallbackFunction_t pxCallbackFunction
      // Callback function to execute when the timer expires
);
```

Inter-Task Communication APIs - Timer Management

- **pcTimerName**: an optional name for the timer.
- **xTimerPeriodInTicks**: The period of the timer in ticks.
- **uxAutoReload**: pdTRUE (1) for the timer to automatically reload and pdFALSE (0) for a one-shot timer. When the timer expires, the callback function is called.
- **pvTimerID**: A unique identifier that can be associated with the timer.
- **pxCallbackFunction**: A pointer to the callback function that will be called when the timer expires.
- `Return:` Returns a handle to the created timer, TimerHandle_t. Otherwise, NULL is returned.

Inter-Task Communication APIs - Timer Management

- `xTimerStart()`: Starts a created timer.

```
BaseType_t xTimerStart(
    TimerHandle_t xTimer, // Handle to the timer to be started

    // Time to wait for the command to be successfully sent to the timer
    // command queue (usually set to 0)
    TickType_t xTicksToWait
);
```

- **xTimer**: the handle of the timer to be started. This handle is returned when the timer is created using xTimerCreate().
- **xTicksToWait**: the maximum time the calling task should block waiting for the timer command to be sent to the timer service task. If set to 0, the function will not block; it will return immediately if the command cannot be sent.
- **Return**: pdPASS is successful or pdFAIL if not.

Inter-Task Communication APIs - Timer Management

```
#include <Arduino.h>
#include <FreeRTOS.h>

TimerHandle_t myTimer;

// Callback function for the timer
void myTimerCallback(TimerHandle_t xTimer) {
    Serial.println("Timer callback function executed.");
}

void setup() {
    Serial.begin(115200);

    // Create a timer with a period of 1000 milliseconds (1 second)
    myTimer = xTimerCreate(
        "MyTimer",              // Name of the timer
        pdMS_TO_TICKS(1000),    // Timer period in ticks (1000 ms converted to ticks)
        pdTRUE,                 // Auto-reload flag (pdTRUE for auto-reload, pdFALSE for one-shot)
        (void *)0,              // Timer ID (not used here)
        myTimerCallback         // Callback function to execute when the timer expires
    );

    // Check if the timer was created successfully
    if (myTimer == NULL) {
        Serial.println("Failed to create timer.");
    } else {
        // Start the timer
        if (xTimerStart(myTimer, 0) != pdPASS) {
            Serial.println("Failed to start timer.");
        }
    }
}
```

```
void loop() {
    // Your loop code here (if any)
}
```

Inter-Task Communication APIs - Memory Management

- pvPortMalloc(): Allocates memory dynamically from the FreeRTOS heap.

```
void *pvPortMalloc(size_t xSize);
```

- vPortFree(): Frees memory allocated by pvPortMalloc().

```
void vPortFree(void *pv);
```

- **xSize**: The number of bytes to allocate. This specifies the size of the memory block you wish to allocate from the heap.
- **Return**: pvPortMalloc() returns a void type pointer if success or NULL if fail.
- **pv**: A pointer to the memory block that needs to be freed.

Inter-Task Communication APIs - Memory Management

```
#include <Arduino.h>
#include <FreeRTOS.h>

void setup() {
    Serial.begin(115200);

    // Request 100 bytes of memory
    void *pMemory = pvPortMalloc(100);

    if (pMemory != NULL) {
        Serial.println("Memory allocated successfully.");

        // Example usage: cast and store a value
        int *pInt = (int *)pMemory;
        *pInt = 42;  // Store an integer value at the allocated memory
        Serial.print("Stored value: ");
        Serial.println(*pInt);

        // Free the memory once done
        vPortFree(pMemory);
        Serial.println("Memory freed.");
    } else {
        Serial.println("Memory allocation failed.");
    }
}

void loop() {
    // Your main loop code here
}
```

Inter-Task Communication APIs - Event Groups

- `xEventGroupCreate()`: Creates an event group.

```
EventGroupHandle_t xEventGroupCreate(void);
```

- `xEventGroupSetBits()`: Sets bits in the event group

```
EventBits_t xEventGroupSetBits(EventGroupHandle_t xEventGroup,
                               const EventBits_t uxBitsToSet);
```

- `xEventGroupWaitBits()`: Waits for specified bits to be set.

```
EventBits_t xEventGroupWaitBits(
    EventGroupHandle_t xEventGroup,
    const EventBits_t uxBitsToWaitFor,
    const BaseType_t xClearOnExit,
    const BaseType_t xWaitForAllBits,
    TickType_t xTicksToWait
    );
```

Inter-Task Communication APIs - Event Groups

- xEventGroupCreate() returns a handle (EventGroupHandle_t) to the created event group, or NULL if the creation fails.
- **uxBitsToSet:** the bit or bits you want to set. Each bit in this parameter corresponds to a bit in the event group.
- xEventGroupSetBits() returns the value of the event group's bits after the specified bits have been set.
- **uxBitsToWaitFor**: a bitmask that indicates which bits the task is waiting for. Multiple bits can be ORed together if the task is waiting for any or all of those bits.
- **xClearOnExit**: If this is set to pdTRUE, the bits specified in uxBitsToWaitFor will be cleared when the task exits the function. If set to pdFALSE, the bits remain set.
- **xWaitForAllBits**: If this is set to pdTRUE, the function will block until all the bits specified in uxBitsToWaitFor are set. If set to pdFALSE, it will return as soon as any one of the bits is set.
- **xTicksToWait**: The maximum time to wait in ticks. Use portMAX_DELAY to wait indefinitely.
- xEventGroupWaitBits() returns the value of the event group bits at the time it exits or beforethey were cleared .

Inter-Task Communication APIs - Event Groups

```
#include <Arduino.h>
#include <FreeRTOS.h>
#include <event_groups.h>

// Declare an event group handle
EventGroupHandle_t xCreatedEventGroup;

// Define event bit masks
const int BIT_0 = (1 << 0); // Event bit 0

void task1(void *pvParameters) {
    for (;;) {
        // Wait for BIT_0 to be set within the event group
        EventBits_t uxBits = xEventGroupWaitBits(
            xCreatedEventGroup,   // Event group handle
            BIT_0,                // Bits to wait for
            pdTRUE,               // Clear bits on exit
            pdFALSE,              // Wait for any bit (not all bits)
            portMAX_DELAY         // Block indefinitely until bits are set
        );

        if ((uxBits & BIT_0) != 0) {
            Serial.println("Task 1: BIT_0 is set, doing some work...");
            // Simulate work
            vTaskDelay(pdMS_TO_TICKS(500));
        }
    }
}

void task2(void *pvParameters) {
    for (;;) {
        Serial.println("Task 2: Setting BIT_0 in event group.");
        xEventGroupSetBits(xCreatedEventGroup, BIT_0);

        // Delay to allow task1 to process the event
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
```

```
void setup() {
    Serial.begin(115200);

    // Create the event group
    xCreatedEventGroup = xEventGroupCreate();

    if (xCreatedEventGroup == NULL) {
        Serial.println("Event group creation failed. Not enough heap memory.");
        return;
    }

    // Create two tasks
    xTaskCreate(task1, "Task 1", 1024, NULL, 1, NULL);
    xTaskCreate(task2, "Task 2", 1024, NULL, 1, NULL);
}

void loop() {
    // Main loop does nothing in this example
}
```

Use Cases for Multitasking

- Sensor Data Collection and Processing
  - Task 1: Reads data from a sensor (e.g., temperature, humidity) periodically.
  - Task 2: Processes the data and sends it over a communication channel (e.g., Wi-Fi, Bluetooth).
- Web Server and Client Communication
  - Task 1: Handles incoming HTTP requests and serves web pages.
  - Task 2: Periodically sends data to a cloud server or listens for commands.

Use Cases for Multitasking

- Bluetooth Communication and Actuator Control
  - Task 1: Listens for commands via Bluetooth (BLE).
  - Task 2: Controls an actuator (e.g., motor, LED) based on received commands.
- Data Logging and Display
  - Task 1: Logs sensor data to an SD card.
  - Task 2: Updates a display (e.g., OLED) with the latest sensor readings.

Best Practices for Multitasking

- Avoid Blocking Calls: Avoid using functions like delay() that block the entire task. Instead, use vTaskDelay() or non-blocking alternatives.
- Task Prioritization: Assign priorities to tasks based on their importance and time sensitivity.
- Core Affinity: Use xTaskCreatePinnedToCore() to balance the load across the two cores of the ESP32-S3.
- Resource Management: Use semaphores and mutexes to manage shared resources between tasks to avoid race conditions.

Troubleshooting Common Issues

- Task Crashes: If tasks crash, check for stack overflows. Increase the stack size allocated to the task if necessary.
- Memory Leaks: Monitor the available heap memory using xPortGetFreeHeapSize() to prevent memory exhaustion.
- Watchdog Timers: Ensure tasks are not taking too long to execute without yielding, which can trigger the watchdog timer.