Module 5 - Software Timer

5.1 Introduction to Real-Time Multitasking

What is an RTOS? Tasks and Scheduling

A **Real-Time Operating System (RTOS)** is a specialized operating system designed for embedded systems that must process data and events within a strict, predictable timeframe. Unlike a desktop OS (like Windows or macOS) which prioritizes throughput and fairness, an RTOS prioritizes **determinism,** the ability to guarantee that a task will be completed within a specific deadline.

The fundamental unit of execution in an RTOS is a **Task**. You can think of a task as an independent function that runs in its own context, with its own stack and priority. For example, in a smart device, you might have one task for managing the Wi-Fi connection, another for reading sensor data, and a third for updating a display.

The core component of the RTOS is the **Scheduler**. Its job is to manage which task gets to use the CPU at any given moment. By rapidly switching between tasks (a process called a "context switch"), the scheduler creates the illusion that all tasks are running simultaneously, a concept known as **multitasking**. In a priority-based scheduler like the one in FreeRTOS, the scheduler will always ensure that if multiple tasks are ready to run, the one with the highest priority is the one that executes.

The Problem with `delay()`: Blocking vs. Non-Blocking Operations

In simple microcontroller programming (like a basic Arduino sketch), it is common to use a `delay()` function to control timing. For example, to blink an LED every second, you might write:

```
void loop() {
  digitalWrite(LED_PIN, HIGH);
  delay(1000); // Wait for 1 second
  digitalWrite(LED_PIN, LOW);
  delay(1000); // Wait for 1 second
}
```

This works, but it is extremely inefficient. During the `delay(1000)` call, the CPU is completely occupied doing nothing, it is stuck in a busy-wait loop, unable to perform any other work. This is

known as a **blocking** operation. A blocking function halts the execution of its thread or task until a specific event occurs (in this case, the passage of time).

In a multitasking RTOS environment, blocking is the enemy of responsiveness. If one task calls `delay()`, it effectively puts the entire system on hold (unless a higher-priority task preempts it), preventing other, potentially critical, tasks from running. The goal in an RTOS is to use **non-blocking** operations. A task should perform its work and, if it needs to wait, it should inform the scheduler so that a lower-priority task can use the CPU in the meantime.

The Need for Asynchronous Events

The solution to the blocking problem is to design systems around **asynchronous events**. An asynchronous event is one that occurs independently of the main program flow, allowing the system to react to it without having to constantly wait and check for it.

An RTOS provides two primary mechanisms for handling asynchronous events:

1. **Hardware Interrupts:** These are signals sent directly from hardware peripherals to the CPU, demanding immediate attention. When an interrupt occurs, the CPU immediately pauses whatever it is doing, executes a special function called an **Interrupt Service Routine (ISR)**, and then resumes its previous work. This is ideal for high-priority, time-critical events triggered by the outside world, such as a button being pressed, data arriving on a communication bus, or a hardware timer reaching zero.
2. **Software Timers:** These are timers managed by the RTOS itself. You can ask the RTOS to execute a specific function (a "callback") at some point in the future, either once or repeatedly. This allows you to schedule application-level events without blocking. Instead of calling delay(), a task can start a software timer and then continue with other work or yield control of the CPU to other tasks. The RTOS will ensure the callback function is executed at the correct time.

By using timers and interrupts, you can build complex, responsive applications where tasks spend almost no time waiting and are instead driven by the occurrence of events.

5.2 An Overview of Asynchronous Tools in FreeRTOS

Software Timers: For Application-Scheduled Events

A **FreeRTOS Software Timer** is a tool used to schedule the execution of a function at a future time. It's like setting an alarm clock within your software. When the timer expires, the RTOS automatically calls a predefined function, known as a **callback function**.

Key Characteristics:

- **Managed by the RTOS:** Software timers are managed by a dedicated RTOS task (the "timer daemon"). This means they do not consume CPU time while they are waiting to expire.
- **Tied to the System Tick:** The resolution of a software timer is determined by the FreeRTOS system tick rate (`configTICK_RATE_HZ`). You cannot schedule a timer for a period shorter than one tick.
- **Use Case:** Ideal for repetitive, low-priority, or application-level timing. For example, you might use a software timer to:
  - Read a temperature sensor every five seconds.
  - Update a clock display once per minute.
  - Turn off an LED 500ms after it was turned on.

Types of Software Timers:

1. **One-Shot Timer:** Executes its callback function only once after it is started.
2. **Auto-Reload Timer:** Executes its callback function repeatedly at a fixed interval until it is explicitly stopped.

Hardware Interrupts: For Hardware-Triggered Events

A **Hardware Interrupt** is a mechanism for a hardware peripheral to signal the CPU that it needs immediate attention. Unlike a software timer, which is scheduled by your application, an interrupt is triggered by an external, physical event.

Key Characteristics:

- **High Priority:** An interrupt will immediately preempt the currently running code, regardless of the task's priority. The CPU will save its current state and jump to execute the ISR.
- **Hardware-Driven:** They are generated by peripherals like GPIO pins (e.g., a button press), hardware timers (for precise timing), or communication interfaces like UART/SPI (e.g., data has arrived).
- **Use Case:** Essential for time-critical operations and reacting to external events with minimal latency. For example, you would use a hardware interrupt to:
  - Count pulses from a motor encoder to measure its speed.
  - Immediately stop a machine when a safety limit switch is triggered.
  - Capture incoming data from a high-speed sensor before it is overwritten.

In summary, the choice between them is driven by the source of the event:

- Use a **Software Timer** when the event is driven by the logic of your application ("I need to do X in 500 milliseconds").
- Use a **Hardware Interrupt** when the event is driven by an external hardware signal that requires an immediate response ("The hardware needs attention NOW").

5.3 Deep Dive: FreeRTOS Software Timers

Creating, Starting, and Stopping Timers

Interacting with FreeRTOS software timers is done through a standard set of API functions. The core steps are to create a timer, start it, and, if needed, stop, reset, or delete it.

Creating a Timer

A software timer is created using the `xTimerCreate()` function. This function does not start the timer; it only allocates the necessary resources and returns a handle that you will use to reference the timer in other API calls.

The function signature is:

```
TimerHandle_t xTimerCreate( const char * const pcTimerName,
                            const TickType_t xTimerPeriodInTicks,
                            const UBaseType_t uxAutoReload,
                            void * const pvTimerID,
                            TimerCallbackFunction_t pxCallbackFunction );
```

Parameters:

1. `pcTimerName`: A descriptive name for the timer, used mainly for debugging.
2. `xTimerPeriodInTicks`: The timer's period in system ticks. You can use the `pdMS_TO_TICKS()` macro to easily convert milliseconds to ticks.
3. `uxAutoReload`: Set to `pdTRUE` for an auto-reload timer or `pdFALSE` for a one-shot timer.
4. `pvTimerID`: A unique identifier for the timer. This is an application-defined value that can be used within the callback function to determine which timer has expired.
5. `pxCallbackFunction`: A pointer to the function that will be executed when the timer expires.

It is crucial to **always check the return value** of `xTimerCreate()`. If it returns `NULL`, the timer could not be created, most likely due to insufficient FreeRTOS heap memory.

Controlling a Timer

Once you have a valid timer handle, you can control it with the following functions:

- **To start or restart a timer:** `xTimerStart(TimerHandle_t xTimer, TickType_t xBlockTime)`

This places the timer into the active state. If the timer was already running, it will be reset to its initial period.

- **To stop a timer:**
  `xTimerStop(TimerHandle_t xTimer, TickType_t xBlockTime)`
  This stops the timer from running.
- **To reset a timer:**
  `xTimerReset(TimerHandle_t xTimer, TickType_t xBlockTime)`
  This is equivalent to calling xTimerStart() on a running timer. It resets the timer's period back to its starting value.
- **To delete a timer:**
  `xTimerDelete(TimerHandle_t xTimer, TickType_t xBlockTime)`
  This frees the memory allocated when the timer was created. Once deleted, the handle is no longer valid.

The xBlockTime parameter in these functions specifies how long the calling task should wait if the command cannot be sent to the timer daemon task immediately (because its command queue is full). Using `portMAX_DELAY` will cause the task to wait indefinitely, which is a safe option in most cases.

3.2 One-Shot vs. Auto-Reload Timers

FreeRTOS offers two types of software timers, defined at creation time by the uxAutoReload parameter.

One-Shot Timer (`uxAutoReload` = `pdFALSE`)

A one-shot timer will execute its callback function **only once** after its period expires. It is useful for performing a single, delayed action.

- **Example Use Case:** You want to turn off a motor 10 seconds after it has been started.

Example Creation:

```
TimerHandle_t xOneShotTimer;

void vOneShotCallback(TimerHandle_t xTimer); // Forward declaration

void setup() {
  xOneShotTimer = xTimerCreate(
      "OneShot",                // Timer name
      pdMS_TO_TICKS(2000),      // 2000ms period
      pdFALSE,                  // Set as a one-shot timer
      (void *) 0,               // Timer ID = 0
      vOneShotCallback          // Callback function
```

```
  );

  if (xOneShotTimer != NULL) {
    xTimerStart(xOneShotTimer, 0);
  }
}
```

Auto-Reload Timer (`uxAutoReload` = `pdTRUE`)

An auto-reload timer will execute its callback function **repeatedly** at a fixed interval. After the callback is executed, the timer automatically resets and starts counting down again.

- **Example Use Case:** You need to read a sensor and print its value every 1000 milliseconds.

Example Creation:

```
TimerHandle_t xAutoReloadTimer;

void vAutoReloadCallback(TimerHandle_t xTimer); // Forward declaration

void setup() {
  xAutoReloadTimer = xTimerCreate(
      "AutoReload",             // Timer name
      pdMS_TO_TICKS(1000),      // 1000ms period
      pdTRUE,                   // Set as an auto-reload timer
      (void *) 1,               // Timer ID = 1
      vAutoReloadCallback       // Callback function
  );

  if (xAutoReloadTimer != NULL) {
    xTimerStart(xAutoReloadTimer, 0);
  }
}
```

3.3 Writing Effective Timer Callback Functions

The callback function is the heart of the software timer. It's the code that runs when the timer expires. To ensure system stability, it must be written carefully.

The function must have the following signature:

```
void YourCallbackName(TimerHandle_t xTimer);
```

The single parameter, `xTimer`, is the handle of the timer that just expired. This is very useful when a single callback function is used for multiple timers. You can retrieve the Timer ID you assigned during creation to identify which timer it was.

Example Callback Implementation:

```
void vTimerCallback(TimerHandle_t xTimer) {
  // Get the ID of the timer that expired
  uint32_t ulTimerID = (uint32_t) pvTimerGetTimerID(xTimer);

  // Check which timer it was and perform an action
  if (ulTimerID == 0) {
    // This was the one-shot timer
    Serial.println("One-shot timer expired.");
  } else if (ulTimerID == 1) {
    // This was the auto-reload timer
    Serial.println("Auto-reload timer expired.");
  }
}
```

Rules for Writing Callback Functions

Timer callbacks execute in the context of the FreeRTOS timer daemon task, not an ISR. However, they share similar restrictions because multiple callbacks may need to execute in sequence.

1. **Keep them short and fast.** A long-running callback will delay the execution of other pending timer callbacks.
2. **Never block.** Do not call any function that could block, such as `vTaskDelay()` or waiting on a semaphore or queue with a long timeout. Doing so will halt the timer daemon task, preventing any other software timers in the system from running.

5.4 Deep Dive: ESP32 Hardware Interrupts

Configuring Hardware Timers on the ESP32

The ESP32 microcontroller comes with four general-purpose 64-bit hardware timers. These timers are highly precise and can be used to generate interrupts at specific intervals, independent of the RTOS scheduler.

The configuration involves four main steps:

1. **Initialize the Timer:** `timerBegin(uint8_t num, uint16_t prescaler, bool countUp)`
   - `num`: The timer you want to use (0 to 3).
   - `prescaler`: A value used to divide the base clock (usually 80 MHz). A prescaler of 80 will make the timer count up every 1 microsecond (80,000,000 Hz / 80 = 1,000,000 Hz).
   - `countUp`: true for counting up, false for counting down.
2. **Attach the ISR:** `timerAttachInterrupt(hw_timer_t *timer, void (*fn)(void), bool edge)`
   - This function links your ISR function to the hardware timer.
3. **Set the Alarm Value:** `timerAlarmWrite(hw_timer_t *timer, uint64_t alarm_value, bool autoreload)`
   - This sets the counter value at which the interrupt will be generated. For a 1 MHz timer clock, an alarm_value of 1,000,000 will trigger an interrupt every second.
   - If autoreload is true, the timer will automatically restart after the interrupt, making it periodic.
4. **Enable the Alarm:** `timerAlarmEnable(hw_timer_t *timer)`
   - This starts the timer and enables the interrupt generation.

Writing an Interrupt Service Routine (ISR)

An ISR is a special function that the CPU executes in response to a hardware interrupt.

On the ESP32, it is critical to use the `IRAM_ATTR` attribute in the function definition:

```
void IRAM_ATTR onTimer() {
  // Your interrupt code here...
}
```

`IRAM_ATTR` tells the compiler to place the ISR code into the ESP32's Internal RAM (IRAM). This is essential for performance and reliability. If an ISR is in flash memory, the CPU may have to wait for

the flash to be read, which can introduce unacceptable latency and jitter into the interrupt response time.

Example of a complete hardware timer setup:

```
// Timer handle
hw_timer_t *timer = NULL;

// The ISR function to be called
void IRAM_ATTR onTimer() {
  // Toggle an LED or perform a quick action
  digitalWrite(LED_PIN, !digitalRead(LED_PIN));
}

void setup() {
  pinMode(LED_PIN, OUTPUT);

  // 1. Initialize timer 0 with a prescaler of 80
  timer = timerBegin(0, 80, true);

  // 2. Attach the ISR to our timer
  timerAttachInterrupt(timer, &onTimer, true);

  // 3. Set the alarm to trigger every 1,000,000 counts (1 second)
  timerAlarmWrite(timer, 1000000, true);

  // 4. Enable the alarm
  timerAlarmEnable(timer);
}
```

The Golden Rules of ISRs

Interrupt Service Routines are powerful but dangerous if used incorrectly. Because they can interrupt any part of your code at any time, they must follow strict rules to avoid crashing the system.

1. **Keep it Fast:** An ISR must execute as quickly as possible. The longer an ISR runs, the more it delays the main program and other interrupts. The best ISRs do the absolute minimum work required, such as setting a flag or sending data to a queue, and then exit. Defer all complex data processing to a regular FreeRTOS task.

2. **Never Block:** An ISR **must never, ever block**. This means you cannot call functions like `vTaskDelay()`, `delay()`, or wait for a semaphore, mutex, or queue. The system is in a special interrupt context, and attempting to block will lead to a system crash.
3. **Use ISR-Safe API Functions:** When you need to interact with FreeRTOS from an ISR (for example, to signal a task), you **must** use the special ISR-safe version of the API function. These functions are specially designed to be non-blocking and safe to call from an interrupt context. They are easily recognizable as they all end with the suffix **`...FromISR()`**.
   - **Correct:** `xSemaphoreGiveFromISR()`
   - **Incorrect:** `xSemaphoreGive()`
   - **Correct:** `xQueueSendFromISR()`
   - **Incorrect:** `xQueueSend()`
4. **Be Careful with Global Variables:** If an ISR modifies a global variable that is also accessed by a task, you must protect that variable to prevent data corruption (a "race condition"). The primary method for this is using a **critical section**, which will be discussed in the next part.

5.5 The Core Challenge: ISRs and Tasks Synchronization

Understanding the Problem: Shared Data and Race Conditions

When a hardware interrupt occurs, the CPU immediately stops executing the current task and jumps to the ISR. This can happen at any time, even in the middle of a single line of C code that takes multiple machine instructions to execute. If the ISR and the task both access the same global variable, the system is vulnerable to a **race condition**.

A race condition is an undesirable situation that occurs when the outcome of a process depends on the uncontrollable sequence of events. In our case, it's a bug that occurs when the timing of the interrupt corrupts shared data.

A Classic Example of a Race Condition:
Imagine a global variable `volatile int counter = 0;`.

- An ISR, triggered by a timer, is programmed to increment the counter: `counter++;`.
- A task in the main application is programmed to decrement it: `counter--;`.

Let's trace a potential failure scenario. Assume `counter` is currently `10`.

1. **Task Executes:** The task reads the value of `counter` (10) into a CPU register.
2. **Task Calculates:** The CPU calculates the new value, `10 - 1 = 9`.
3. **CONTEXT SWITCH (INTERRUPT):** Before the task can write the value `9` back to the `counter` variable in memory, a hardware interrupt occurs!
4. **ISR Executes:** The ISR runs. It reads the value of `counter` from memory (which is still 10).
5. **ISR Calculates:** The ISR calculates `10 + 1 = 11`.
6. **ISR Writes:** The ISR writes the value `11` back to the counter variable in memory.
7. **ISR Finishes:** The interrupt is complete, and the CPU returns control to the task, restoring its state exactly where it left off.
8. **Task Resumes:** The task is completely unaware that it was interrupted. Its next step is to write its calculated value (`9`) back to the `counter` variable.
9. **Corruption:** The value `11` that the ISR correctly calculated is now overwritten with `9`. The increment operation has been completely lost.

This is the fundamental problem of concurrency: protecting shared resources from uncontrolled, simultaneous access.

The ISR Context and `...FromISR()` Functions

To solve the synchronization problem, we need tools to manage access to shared data. However, as we learned in Part 2, ISRs operate in a special **interrupt context**. They are not tasks and are not managed by the RTOS scheduler. This leads to a critical rule: **an ISR can never block**.

If an ISR tried to wait for a resource (like calling `xQueueSend()` and the queue was full), it would effectively block. But since the ISR is not a task, the scheduler has no other context to switch to. The entire system would freeze, leading to a crash.

To solve this, FreeRTOS provides a special set of ISR-safe functions that are non-blocking. You can recognize them by their `...FromISR()` suffix.

- `xQueueSend()` -> `xQueueSendFromISR()`
- `xSemaphoreGive()` -> `xSemaphoreGiveFromISR()`

These functions include an optional parameter, `pxHigherPriorityTaskWoken`. An ISR uses this parameter to inform the RTOS if its action (e.g., giving a semaphore) has unblocked a task that has a higher priority than the task that was originally interrupted. If so, the RTOS can perform an immediate context switch to the higher-priority task as soon as the ISR finishes, ensuring the system remains as responsive as possible.

5.6 Synchronization Mechanisms: A Comparative Guide

FreeRTOS provides three primary mechanisms for safely managing shared resources between tasks and ISRs.

Critical Sections: The "Big Hammer" for Protection

A **critical section** is a section of code that is guaranteed to run to completion without being preempted by an interrupt or another task. It is the most direct and forceful way to prevent a race condition.

How it Works: It works by temporarily disabling all interrupts system-wide.

- In a task, you wrap the critical code with `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()`.
- In an ISR, you use `portENTER_CRITICAL_ISR()` and `portEXIT_CRITICAL_ISR()`.

Example:

```
volatile int counter = 0;

void IRAM_ATTR onTimer() {
    portENTER_CRITICAL_ISR(&timerMux);
    counter++; // This is now safe
    portEXIT_CRITICAL_ISR(&timerMux);
}

void printValues(void * parameter) {
    while (true) {
        taskENTER_CRITICAL();
        counter--; // This is now safe
        Serial.println(counter);
        taskEXIT_CRITICAL();
        vTaskDelay(pdMS_TO_TICKS(2000));
```

```
    }
}
```

- **When to Use:** Only for protecting very short, fast operations on shared variables where other mechanisms are too slow or complex.
- **Risk:** While a critical section is active, all interrupts are disabled. This can severely impact the real-time responsiveness of the system. **Keep critical sections as short as humanly possible.**

Semaphores: The Best Tool for Pure Signaling

A **semaphore** is a signaling mechanism. It does not transfer data. It is used to signal that an event has occurred or to control access to a resource. For ISR-to-task communication, a **binary semaphore** is typically used.

How it Works: Think of it as a flag.

1. A task tries to "take" the semaphore using `xSemaphoreTake()`. If the semaphore is not available, the task enters the Blocked state, consuming no CPU time.
2. An ISR, responding to a hardware event, "gives" the semaphore using `xSemaphoreGiveFromISR()`.
3. Giving the semaphore unblocks the waiting task, moving it to the Ready state. The scheduler will then run the task when it is its turn.

Example:

```
SemaphoreHandle_t binSemaphore = NULL;

void IRAM_ATTR onTimer() {
    xSemaphoreGiveFromISR(binSemaphore, NULL);
}

void processValues(void * parameter) {
    while (true) {
        // Wait here until the ISR gives the semaphore
        if (xSemaphoreTake(binSemaphore, portMAX_DELAY) == pdTRUE) {
            // The event occurred. Process the data.
            Serial.println("Processing data now...");
        }
    }
}
```

- **When to Use:** When an ISR needs to notify a task that an event has happened (e.g., "ADC conversion is complete, the data is ready to be read"). It's a pure synchronization primitive.

Queues: The Best Tool for Transferring Data

A **queue** is the most powerful and often the best mechanism for ISR-to-task communication. It provides a thread-safe First-In, First-Out (FIFO) buffer to not only signal an event but to also safely transfer data from the ISR to the task.

How it Works:

1. A task waits to "receive" from a queue using `xQueueReceive()`. If the queue is empty, the task enters the Blocked state.
2. An ISR generates some data (e.g., reads a sensor). It then "sends" this data to the queue using `xQueueSendFromISR()`. This action copies the data into the queue's buffer.
3. The act of sending data to the queue unblocks the waiting task. The task then receives the copy of the data from the queue for safe processing.

This approach is superior because the ISR and the task never access the same variable directly. They only interact via the RTOS-managed queue, which eliminates race conditions by design.

Example:

```
QueueHandle_t sensorQueue = NULL;

void IRAM_ATTR onTimer() {
    // Read sensor and package data into a struct
    sensorData_t data;
    data.adcValue1 = analogRead(34);

    // Send a COPY of the data to the queue
    xQueueSendFromISR(sensorQueue, &data, NULL);
}

void processValues(void * parameter) {
    sensorData_t receivedData;
    while (true) {
        // Wait here until data arrives in the queue
        if (xQueueReceive(sensorQueue, &receivedData, portMAX_DELAY) == pdTRUE) {
            // Safely process the received data
            Serial.println(receivedData.adcValue1);
```

```
        }
    }
}
```

- **When to Use:** Whenever an ISR needs to pass data to a task for processing. This is the preferred method in almost all data-generating ISR scenarios.

5.7 Choosing the Right Tool: A Practical Comparison

Deciding which synchronization mechanism to use is a key skill in embedded programming. Use the following table and questions as a guide.

Mechanism	Purpose	Transfers Data?	When to Use	Primary Risk
Critical Section	Mutual Exclusion	No	Protecting a few lines of code that modify a shared variable. Must be extremely fast.	Halts system responsiveness by disabling all interrupts. Can easily break real-time deadlines.
Semaphore	Signaling	No	Notifying a task that a specific event has occurred. Deferring ISR work to a task.	Does not help with transferring the actual data associated with the event.
Queue	Data Transfer	Yes	Sending data of any type from an ISR to a task for processing.	Minor overhead for copying data into the queue. May not be suitable for very large data structures.

Decision-Making Guide

When designing an interaction between an ISR and a task, ask yourself these questions:

1. **Do I need to pass data from the ISR to the task?**
   - **Yes:** Use a **Queue**. This is the safest and most robust solution for transferring data.
   - **No:** Go to question 2.
2. **Is my goal simply to wake up a task to do some work when an interrupt occurs?**
   - **Yes:** Use a **Semaphore**. It is a lightweight and highly efficient signaling mechanism.
   - **No:** Go to question 3.
3. **Do I only need to protect a single, simple variable (like a counter or flag) during a very quick read-modify-write operation?**
   - **Yes:** A **Critical Section** is an option, but only if the operation is genuinely just a few lines of code. Be aware of the impact on system latency.
   - **No:** Re-evaluate your design. You likely need a semaphore or a queue.

In modern RTOS development, **Queues and Semaphores are almost always preferred over Critical Sections** for managing ISR-task interactions. They provide cleaner, safer, and more scalable solutions that have less impact on the overall real-time performance of your system.

5.8 Advanced Project: A Multi-Sensor Data Logger

In this chapter, we will build a complete data logging application that utilizes all the core concepts we have learned: hardware interrupts for precise data acquisition, queues for safe data transfer, multiple tasks with different priorities for processing and logging, and a software timer for periodic status checks.

Project Goal

We will create a system that performs the following actions:

1. A **hardware timer** will generate an interrupt every 200 milliseconds.
2. The **Interrupt Service Routine (ISR)** will simulate reading data from two sensors (e.g., temperature and humidity) and will send this data package to a queue.
3. A high-priority **"Processing Task"** will wait for data to arrive in the queue. When it does, it will perform a simple calculation (e.g., calculate an average) and place the result into a second queue.
4. A low-priority **"Logging Task"** will wait for results to arrive in the second queue and print them to the Serial Monitor.
5. A **software timer** will fire every 5 seconds to print a "System OK" status message, demonstrating a non-critical, periodic background action.

This architecture is a common and robust pattern in embedded systems. It decouples the time-critical data acquisition (in the ISR) from the less critical data processing and logging (in the tasks), ensuring the system remains responsive.

You Will Need

- An ESP32 development board.
- The Arduino IDE with the ESP32 board package installed.

```
#include <Arduino.h>


// Define task priorities
#define PROCESSING_TASK_PRIORITY 2
#define LOGGING_TASK_PRIORITY    1
```

```
// Define handles for RTOS objects
QueueHandle_t sensorDataQueue;
QueueHandle_t resultQueue;
TimerHandle_t systemHealthTimer;
hw_timer_t *hardwareTimer = NULL;

// Data structure to hold raw sensor readings
typedef struct {
  int temperature;
  int humidity;
} SensorData;

// Data structure for the processed result
typedef struct {
  float averageValue;
} ProcessedResult;

// --- Interrupt Service Routine ---
// This function runs every time the hardware timer fires.
// It must be fast and non-blocking.
void IRAM_ATTR onTimer() {
  // Simulate reading sensor data
  SensorData data;
  data.temperature = random(20, 30); // Simulate temp between 20-29°C
  data.humidity = random(40, 60);    // Simulate humidity between 40-59%

  // Send a COPY of the data to the queue.
  // Use the ISR-safe version of the function.
  xQueueSendFromISR(sensorDataQueue, &data, NULL);
}

// --- Software Timer Callback ---
// This function runs every time the software timer expires.
void systemHealthCallback(TimerHandle_t xTimer) {
  Serial.println("[HEALTH] System OK");
}

// --- High-Priority Task: Processing ---
void processingTask(void *parameter) {
```

```
  SensorData receivedData;
  ProcessedResult result;

  while (true) {
    // Wait indefinitely until an item arrives in the sensorDataQueue
    if (xQueueReceive(sensorDataQueue, &receivedData, portMAX_DELAY) == pdPASS) {
      // We have data, now process it.
      Serial.println("[PROCESS] Data received. Calculating average...");

      result.averageValue = (receivedData.temperature + receivedData.humidity) / 2.0;

      // Send the result to the logging task via the resultQueue
      xQueueSend(resultQueue, &result, portMAX_DELAY);
    }
  }
}

// --- Low-Priority Task: Logging ---
void loggingTask(void *parameter) {
  ProcessedResult receivedResult;

  while (true) {
    // Wait indefinitely until a result arrives in the resultQueue
    if (xQueueReceive(resultQueue, &receivedResult, portMAX_DELAY) == pdPASS) {
      // We have a result, now log it to the console.
      Serial.print("[LOG] Processed Average: ");
      Serial.println(receivedResult.averageValue);
      Serial.println("--------------------");
    }
  }
}

void setup() {
  Serial.begin(115200);
  delay(1000);
  Serial.println("--- Multi-Sensor Data Logger ---");

  // 1. Create the queues
  // Queue to hold 10 SensorData structs
```

```
sensorDataQueue = xQueueCreate(10, sizeof(SensorData));
// Queue to hold 5 ProcessedResult structs
resultQueue = xQueueCreate(5, sizeof(ProcessedResult));

// 2. Create the software timer for system health checks
systemHealthTimer = xTimerCreate(
    "HealthTimer",          // Name
    pdMS_TO_TICKS(5000),    // 5000ms period
    pdTRUE,                 // Auto-reload
    (void *) 0,             // Timer ID
    systemHealthCallback    // Callback function
);

// 3. Create the tasks
xTaskCreate(
    processingTask,         // Function to implement the task
    "Processing Task",      // Name of the task
    2048,                   // Stack size in words
    NULL,                   // Task input parameter
    PROCESSING_TASK_PRIORITY, // Priority of the task
    NULL                    // Task handle
);

xTaskCreate(
    loggingTask,
    "Logging Task",
    2048,
    NULL,
    LOGGING_TASK_PRIORITY,
    NULL
);

// 4. Configure the hardware timer
// Use a prescaler of 80 to get a 1MHz clock (80MHz / 80)
hardwareTimer = timerBegin(0, 80, true);
timerAttachInterrupt(hardwareTimer, &onTimer, true);
// Set alarm for 200,000 counts (200ms at 1MHz)
timerAlarmWrite(hardwareTimer, 200000, true);
timerAlarmEnable(hardwareTimer);
```

```
  // 5. Start the software timer
  if (systemHealthTimer != NULL) {
    xTimerStart(systemHealthTimer, 0);
  }

  Serial.println("System initialized. Starting data acquisition...");
}

void loop() {
  // The main loop is empty. All work is done by RTOS tasks and timers.
  vTaskDelete(NULL); // Delete the loop task to save resources
}
```

Code Walkthrough

1. **Data Structures:** We define two `structs`, `SensorData` and `ProcessedResult`, to create organized data packages that can be sent to queues. This is much cleaner than passing raw variables.
2. **Hardware Interrupt (`onTimer`)**: This ISR is the "producer." It runs at a precise interval, generates data, and immediately sends it to the `sensorDataQueue`. It does no processing and exits quickly, as a good ISR should.
3. **Processing Task (`processingTask`)**: This task is a "consumer-producer." It has a higher priority because we want to process data as soon as it's available. It waits on `sensorDataQueue`. When data arrives, it performs a quick calculation and sends the result to `resultQueue`.
4. **Logging Task (`loggingTask`)**: This is the final "consumer." It has a lower priority because logging is generally not a time-critical operation. It waits for fully processed data on `resultQueue` and prints it. Because of its lower priority, it will only run when the `processingTask` is blocked (waiting for data).
5. **Software Timer (`systemHealthCallback`)**: This runs completely independently of the main data flow. Every 5 seconds, it prints a status message, demonstrating how you can easily add periodic, non-critical background functions to your application without interfering with the main logic.
6. **`setup()` Function**: This is where we initialize all our RTOS objects. We create the queues, create the tasks, configure the hardware timer, and start the software timer.
7. **`loop()` Function**: In a complex RTOS application, the main `loop()` function is often no longer needed. All continuous work is handled by tasks. We can delete the loop task with `vTaskDelete(NULL)` to free up its stack memory.

How to Test It

1. Upload the code to your ESP32.
2. Open the Arduino Serial Monitor at 115200 baud.
3. You should see the following pattern:
    - Every 200 milliseconds, the "[PROCESS]" message will appear, indicating the high-priority task has received data from the ISR.
    - Immediately after, the "[LOG]" message will print the calculated average, followed by a separator.
    - Every 5 seconds, a "[HEALTH] System OK" message will appear, independently of the other messages.