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.

Revision #1
Created 2025-08-28 13:06:22 UTC by GI
Updated 2025-08-28 13:09:00 UTC by GI