Introduction to RTOS - Solution to Part 5 (FreeRTOS Queue Example)

By ShawnHymel

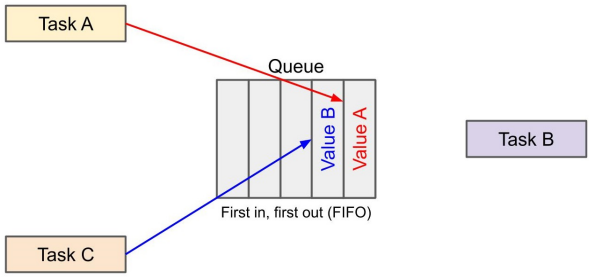
A queue in a real-time operating system (RTOS) is a kernel object that is capable of passing information between tasks without incurring overwrites from other tasks or entering into a race condition. A queue is a first in, first out (FIFO) system where items are removed from the queue once read.

Concepts

Most multi-threaded operating systems offer a number of kernel objects that assist in creating thread-safe communications between threads. Such objects include things like queues, mutexes, and semaphores (which we will cover in a later lecture).

A queue is a simple FIFO system with atomic reads and writes. “Atomic operations” are those that cannot be interrupted by other tasks during their execution. This ensures that another task cannot overwrite our data before it is read by the intended target.

In this example, Task A writes some data to a queue. No other thread can interrupt Task A during that writing process. After, Task B can write some other piece of data to the queue. Task B’s data will appear behind Task A’s data, as the queue is a FIFO system.



Note that in FreeRTOS, information is copied into a queue by value and not by reference. That means if you use the xQueueSend() function to send a piece of data to a queue, all of the data will be copied into the queue atomically.

This can be helpful if you believe your data will go out of scope prior to it being read, but it might require a long copying time if it’s a large piece of data, like a string. While you can copy pointers to a queue, you will want to be sure that the data that is being pointed to is read prior to changing it.

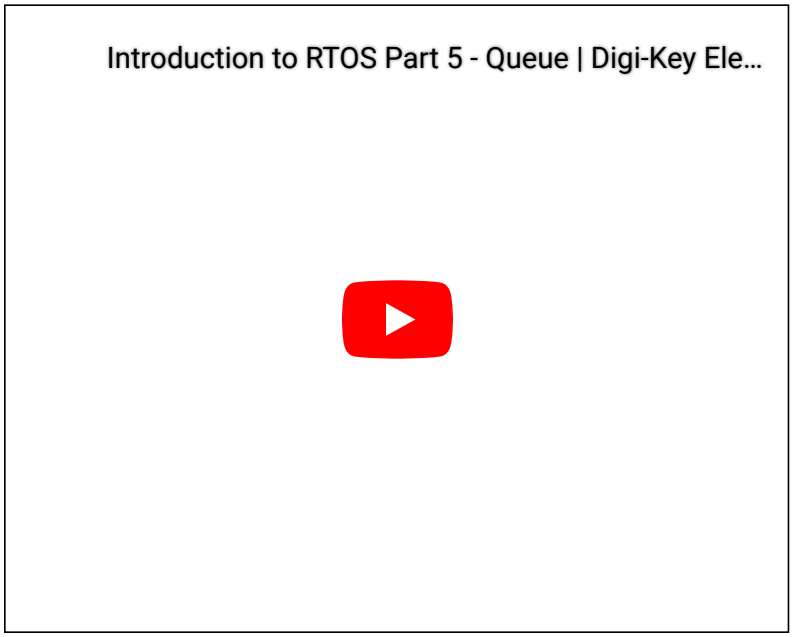
Required Hardware

Any ESP32 development board should work, so long as it’s supported in the Arduino IDE. See here for a list of supported ESP32 boards. You may also use any development board capable of running FreeRTOS, although my solution will likely vary some (as the ESP32 runs a modified version of FreeRTOS called ESP-IDF).

This solution uses the Adafruit Feather HUZZAH32.

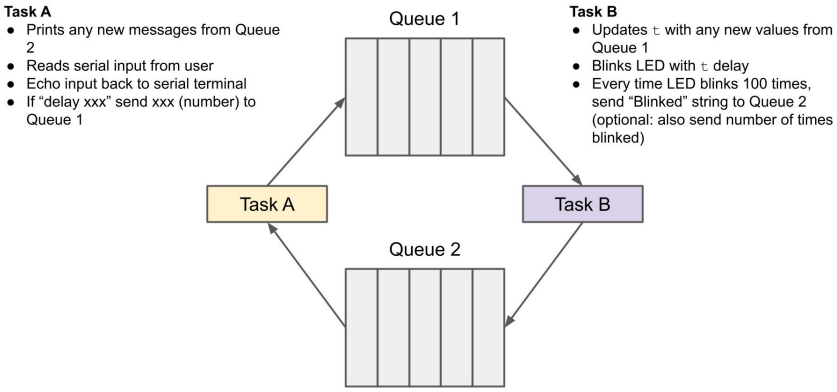
Video

If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:



Challenge

Use FreeRTOS to create two tasks and two queues.



Task A should print any new messages it receives from Queue 2. Additionally, it should read any Serial input from the user and echo back this input to the serial input. If the user enters "delay" followed by a space and a number, it should send that number to Queue 1.

Task B should read any messages from Queue 1. If it contains a number, it should update its delay rate to that number (milliseconds). It should also blink an LED at a rate specified by that delay. Additionally, every time the LED blinks 100 times, it should send the string "Blinked" to Queue 2. You can also optionally send the number of times the LED blinked (e.g. 100) as part of struct that encapsulates the string and this number.

Solution

Spoilers below! I highly encourage you to try the challenge on your own before comparing your answer to mine. Note that my solution may not be the only way to solve the challenge.

Copy Code

```
/**
 * Solution to 05 - Queue Challenge
 *
 * One task performs basic echo on Serial. If it sees "delay" followed by a
 * number, it sends the number (in a queue) to the second task. If it receives
 * a message in a second queue, it prints it to the console. The second task
 * blinks an LED. When it gets a message from the first queue (number), it
 * updates the blink delay to that number. Whenever the LED blinks 100 times,
 * the second task sends a message to the first task to be printed.
 *
 * Date: January 18, 2021
 * Author: Shawn Hymel
```

```
 * License: 0BSD
 */

// 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 = 255;     // Size of buffer to look for command
static const char command[] = "delay "; // Note the space!
static const int delay_queue_len = 5;   // Size of delay_queue
static const int msg_queue_len = 5;     // Size of msg_queue
static const uint8_t blink_max = 100;   // Num times to blink before message

// Pins (change this if your Arduino board does not have LED_BUILTIN defined)
static const int led_pin = LED_BUILTIN;

// Message struct: used to wrap strings (not necessary, but it's useful to see
// how to use structs here)
typedef struct Message {
  char body[20];
  int count;
} Message;

// Globals
static QueueHandle_t delay_queue;
static QueueHandle_t msg_queue;

//*****************************************************************************
// Tasks

// Task: command line interface (CLI)
void doCLI(void *parameters) {

  Message rcv_msg;
  char c;
  char buf[buf_len];
  uint8_t idx = 0;
  uint8_t cmd_len = strlen(command);
  int led_delay;

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

  // Loop forever
  while (1) {

    // See if there's a message in the queue (do not block)
    if (xQueueReceive(msg_queue, (void *)&rcv_msg, 0) == pdTRUE) {
      Serial.print(rcv_msg.body);
      Serial.println(rcv_msg.count);
    }

    // Read characters from serial
    if (Serial.available() > 0) {
      c = Serial.read();

      // Store received character to buffer if not over buffer limit
      if (idx < buf_len - 1) {
        buf[idx] = c;
        idx  ;
      }

      // Print newline and check input on 'enter'
      if ((c == '\n') || (c == '\r')) {

        // Print newline to terminal
        Serial.print("\r\n");

        // Check if the first 6 characters are "delay "
        if (memcmp(buf, command, cmd_len) == 0) {

          // Convert last part to positive integer (negative int crashes)
          char* tail = buf   cmd_len;
          led_delay = atoi(tail);
          led_delay = abs(led_delay);

          // Send integer to other task via queue
          if (xQueueSend(delay_queue, (void *)&led_delay, 10) != pdTRUE) {
            Serial.println("ERROR: Could not put item on delay queue.");
          }
        }

        // Reset receive buffer and index counter
        memset(buf, 0, buf_len);
        idx = 0;

      // Otherwise, echo character back to serial terminal
```

```
      } else {
        Serial.print(c);
      }
    }
  }
}

// Task: flash LED based on delay provided, notify other task every 100 blinks
void blinkLED(void *parameters) {

  Message msg;
  int led_delay = 500;
  uint8_t counter = 0;

  // Set up pin
  pinMode(LED_BUILTIN, OUTPUT);

  // Loop forever
  while (1) {

    // See if there's a message in the queue (do not block)
    if (xQueueReceive(delay_queue, (void *)&led_delay, 0) == pdTRUE) {

      // Best practice: use only one task to manage serial comms
      strcpy(msg.body, "Message received ");
      msg.count = 1;
      xQueueSend(msg_queue, (void *)&msg, 10);
    }

    // Blink
    digitalWrite(led_pin, HIGH);
    vTaskDelay(led_delay / portTICK_PERIOD_MS);
    digitalWrite(led_pin, LOW);
    vTaskDelay(led_delay / portTICK_PERIOD_MS);

    // If we've blinked 100 times, send a message to the other task
    counter  ;
    if (counter >= blink_max) {

      // Construct message and send
      strcpy(msg.body, "Blinked: ");
      msg.count = counter;
      xQueueSend(msg_queue, (void *)&msg, 10);

      // Reset counter
      counter = 0;
    }
  }
}

//*****************************************************************************
// Main (runs as its own task with priority 1 on core 1)

void setup() {

  // Configure Serial
  Serial.begin(115200);

  // Wait a moment to start (so we don't miss Serial output)
  vTaskDelay(1000 / portTICK_PERIOD_MS);
  Serial.println();
  Serial.println("---FreeRTOS Queue Solution---");
  Serial.println("Enter the command 'delay xxx' where xxx is your desired ");
  Serial.println("LED blink delay time in milliseconds");

  // Create queues
  delay_queue = xQueueCreate(delay_queue_len, sizeof(int));
  msg_queue = xQueueCreate(msg_queue_len, sizeof(Message));

  // Start CLI task
  xTaskCreatePinnedToCore(doCLI,
                          "CLI",
                          1024,
                          NULL,
                          1,
                          NULL,
                          app_cpu);

  // Start blink task
  xTaskCreatePinnedToCore(blinkLED,
                          "Blink LED",
                          1024,
                          NULL,
                          1,
                          NULL,
                          app_cpu);

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

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

Explanation

Let's look through the code. The settings and LED pin should look pretty familiar at this point. I use a struct as a way to encapsulate multiple, related pieces of data that need to be copied to Queue 2.

Copy Code

```
// Message struct: used to wrap strings (not necessary, but it's useful to see
// how to use structs here)
typedef struct Message {
  char body[20];
  int count;
} Message;
```

You'll also need two separate queues, so we create those as global variables.

Copy Code

```
// Globals
static QueueHandle_t delay_queue;
static QueueHandle_t msg_queue;
```

In Task A (which I call with the doCLI() function), I use a char buffer to record things from the Serial terminal. You're welcome to implement this in a number of different ways, but I examine serial input character-by-character.

The important thing here is that if we see something in Queue 2, it should be printed to the console. Notice that I do not block waiting for something in Queue 2. If nothing is there, the task simply moves on.

Copy Code

```
// See if there's a message in the queue (do not block)
if (xQueueReceive(msg_queue, (void *)&rcv_msg, 0) == pdTRUE) {
  Serial.print(rcv_msg.body);
  Serial.println(rcv_msg.count);
}
```

I check and echo any characters received on the serial port. If the first string is "delay," I parse it and then send the number after it to Queue 1.

Copy Code

```
// Check if the first 6 characters are "delay "
if (memcmp(buf, command, cmd_len) == 0) {

  // Convert last part to positive integer (negative int crashes)
  char* tail = buf   cmd_len;
  led_delay = atoi(tail);
  led_delay = abs(led_delay);

  // Send integer to other task via queue
  if (xQueueSend(delay_queue, (void *)&led_delay, 10) != pdTRUE) {
    Serial.println("ERROR: Could not put item on delay queue.");
  }
}
```

In Task B, I set up the LED pin as output. In the while loop, I first check to see if there are any messages in Queue 1. Once again, it is non-blocking to allow the rest of the task to execute if nothing is present in the queue.

Copy Code

```
// See if there's a message in the queue (do not block)
if (xQueueReceive(delay_queue, (void *)&led_delay, 0) == pdTRUE) {

  // Best practice: use only one task to manage serial comms
  strcpy(msg.body, "Message received ");
  msg.count = 1;
  xQueueSend(msg_queue, (void *)&msg, 10);
}
```

If something is in the queue, we read it, and it updates the led_delay variable. Note that if nothing is in the queue, led_delay is not changed.

Additionally, I send a message back to Task A using Queue 2. This is not required, but I did it as a test to show that a message was received in Queue 1. I send a simple string and set the count member to 1 (some arbitrary value).

The task blinks the LED once and updates a counter. When the counter hits 100, it is reset, and a message is sent back to Task A using Queue 2.

Copy Code

```
  // If we've blinked 100 times, send a message to the other task
  counter  ;
  if (counter >= blink_max) {

    // Construct message and send
    strcpy(msg.body, "Blinked: ");
    msg.count = counter;
    xQueueSend(msg_queue, (void *)&msg, 10);

    // Reset counter
    counter = 0;
  }
```

The setup() is straightforward: we create two queues, assign them to the global handles, and then start the two tasks.

There is one big flaw with this design: the blink section is blocking! Task B must wait for an entire blink cycle before sending or reading from the queues. This can be devastating if a user has to wait a long time before seeing their input take some action.

Perhaps, afterall, having a global, shared delay variable was the way to go. However, this was just an exercise in using queues, nothing more. Using shared resources is still a viable method, but we might need different ways to protect them from being overwritten by other threads. That's where mutexes and semaphores come in, which we'll discuss in future episodes.

Recommended Reading

All demonstrations and solutions for this course can be found in this GitHub repository.

If you'd like to dig deeper into RTOS queues (specifically, FreeRTOS queues), I recommend checking out these excellent articles:

- FreeRTOS Queues: https://www.freertos.org/Embedded-RTOS-Queues.html
- FreeRTOS Queue API: https://www.freertos.org/a00018.html
- Using queue with struct: https://www.freertos.org/FreeRTOS_Support_Forum_Archive/August_2015/freertos_Send_a_struct_through_Queue_64d28ac3j.html
- Atomic operations: https://stackoverflow.com/questions/52196678/what-are-atomic-operations-for-newbies
- Atomic operations in FreeRTOS: https://docs.aws.amazon.com/freertos/latest/userguide/atomic.html
- What is a Real-Time Operating System Part 1(RTOS)?
- Introduction to RTOS - Solution to Part 2 (FreeRTOS)
- Introduction to RTOS - Solution to Part 3 (Task Scheduling)
- Introduction to RTOS - Solution to Part 4 (Memory Management)
- Introduction to RTOS - Solution to Part 6 (FreeRTOS Mutex Example)
- Introduction to RTOS - Solution to Part 7 (FreeRTOS Semaphore Example)
- Introduction to RTOS - Solution to Part 8 (Software Timers)
- Introduction to RTOS - Solution to Part 9 (Hardware Interrupts)
- Introduction to RTOS - Solution to Part 10 (Deadlock and Starvation)
- Introduction to RTOS - Solution to Part 11 (Priority Inversion)
- Introduction to RTOS - Solution to Part 12 (Multicore Systems)

Project Details

Platforms

Arduino

Development

C++

Tags

Arduino

RTOS

License

Attribution

Get Involved

Like

Save

1-800-344-4539 218-681-6674 sales@digikey.com 218-681-3380

Local Support: 701 Brooks Avenue South, Thief River Falls, MN 56701 USA