

Introduction to RTOS - Solution to Part 3 (Task Scheduling)

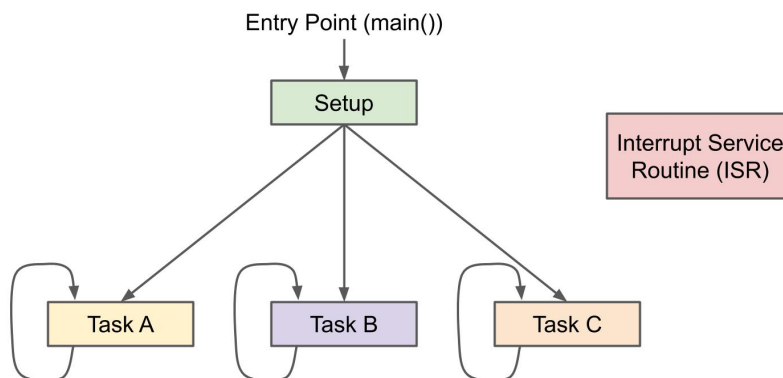
[By ShawnHymel](#)

FreeRTOS allows us to set priorities for tasks, which allows the scheduler to preempt lower priority tasks with higher priority tasks. The scheduler is a piece of software inside the operating system in charge of figuring out which task should run at each tick.

Concepts

Writing a multi-threaded (or multi-task) program looks something like this in code:

What our code looks like

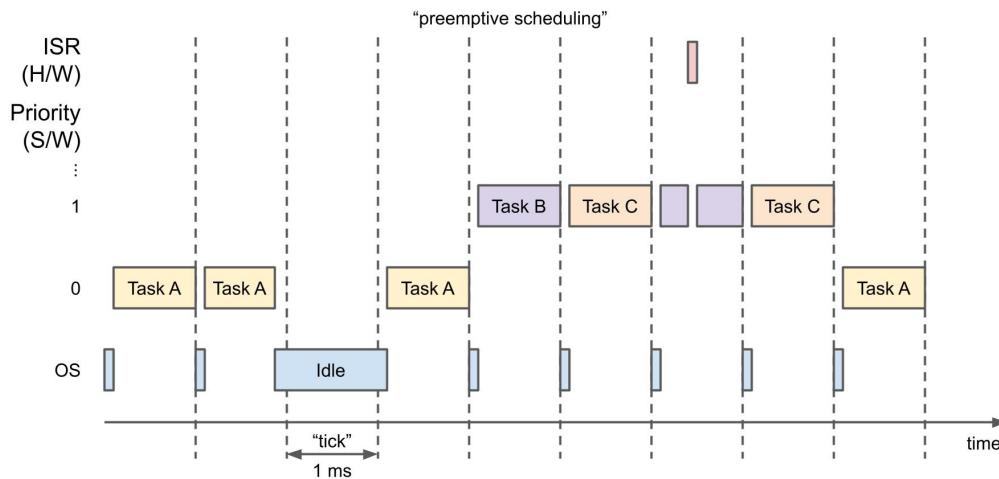


Each task appears to run concurrently in its own while loop (assuming we don't end a thread after a single execution). In microcontrollers, we can also set up independent interrupt service routines (ISRs) that can preempt any of the tasks to execute some code. An ISR is used to handle things like hardware timer overflows, pin state changes, or new communication on a bus.

In a single-core system, the CPU must divide up the tasks into time slices so that they can appear to run concurrently. The scheduler in an operating system is charged with figuring out which task to run each time slice.

What actually happens*

*assuming single-core processor



In FreeRTOS, the default time slice is 1 ms, and a time slice is known as a "tick." A hardware timer is configured to create an interrupt every 1 ms. The ISR for that timer runs the scheduler, which chooses the task to run next.

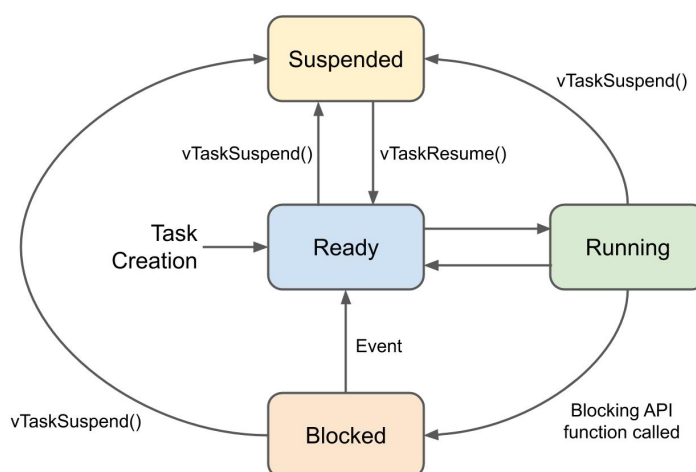
At each tick interrupt, the task with the highest priority is chosen to run. If the highest priority tasks have the same priority, they are executed in a round-robin fashion. If a task with a higher priority than the currently running task becomes available (e.g. in the "Ready" state), then it will immediately run. It does not wait for the next tick.

A hardware interrupt is always considered to have a higher priority than any task running in software. As a result, a hardware ISR can interrupt any task. Because of this, we recommend keeping any ISR code as short as possible to reduce interruptions to the running tasks.

When we create tasks, we can assign them priorities. In fact, we can even change priorities of tasks with `vTaskPrioritySet()` (read more about that function [here](#)).

Tasks in FreeRTOS can be in one of four states.

Task States



As soon as a task is created, it enters the Ready state. Here, it tells the scheduler that it's ready to run. Each tick, the scheduler chooses 1 task to run that's in the ready state (on a multi-core system, the scheduler can choose multiple tasks, but we won't cover multi-core systems here).

While running, a task is in the Running state and can be returned to the Ready state by the scheduler.

Functions that cause the task to wait, like `vTaskDelay()`, put the task in the Blocked state. Here, the task is waiting for some other event to occur, such as the timer on the `vTaskDelay()` to expire. The task may also be waiting for some resource, like a semaphore, to be released by another task. Tasks in the Blocked state allow other tasks to run instead.

Finally, an explicit call to `vTaskSuspend()` can put a task in the Suspended mode (much like putting that task to sleep). Any task may put any task (including itself) into the Suspended mode. A task may only be returned to the Ready state by an explicit call to `vTaskResume()` by another task.

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

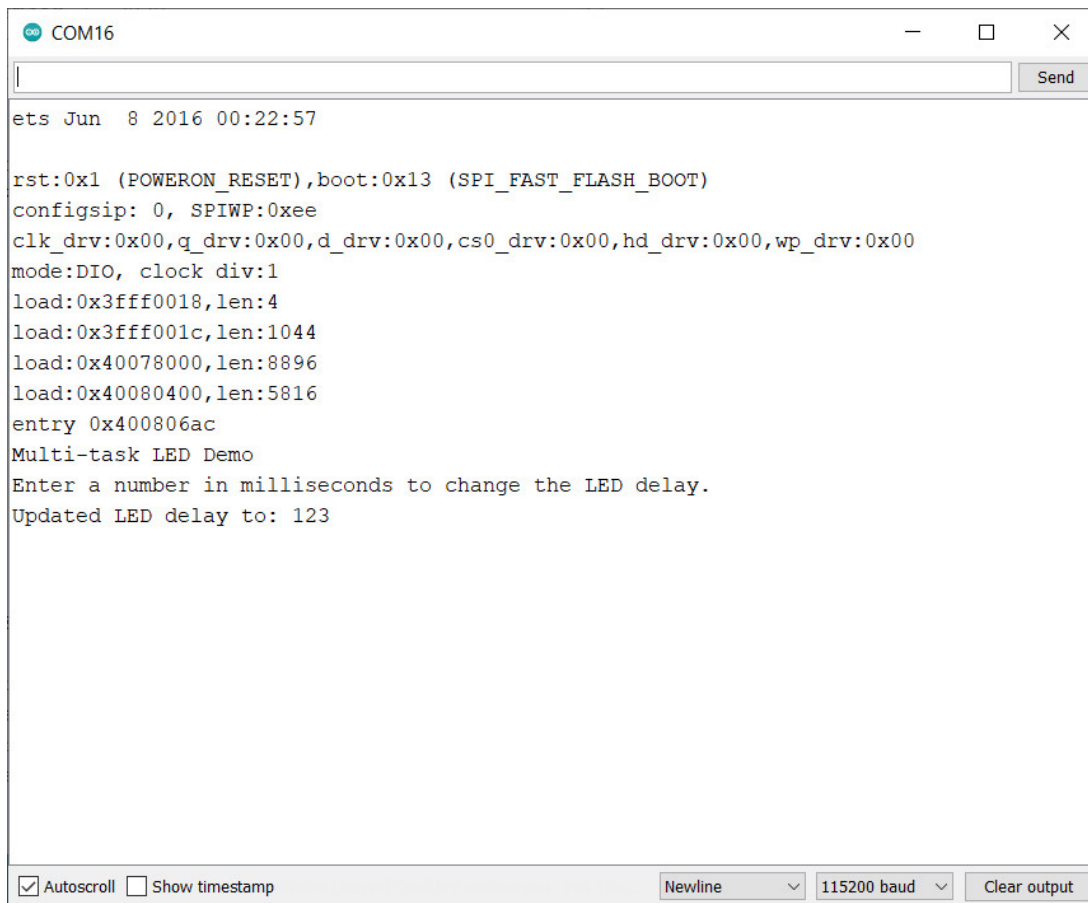
If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:

Introduction to RTOS Part 3 - Task Scheduling | Digi-Key Electronics



Challenge

Using FreeRTOS, create two separate tasks. One listens for an integer over UART (from the Serial Monitor) and sets a variable when it sees an integer. The other task blinks the onboard LED (or other connected LED) at a rate specified by that integer. In effect, you want to create a multi-threaded system that allows for the user interface to run concurrently with the control task (the blinking LED).



```
ets Jun 8 2016 00:22:57

rst:0x1 (POWERON_RESET),boot:0x13 (SPI_FAST_FLASH_BOOT)
configsip: 0, SPIWP:0xee
clk_drv:0x00,q_drv:0x00,d_drv:0x00,cs0_drv:0x00,hd_drv:0x00,wp_drv:0x00
mode:DIO, clock div:1
load:0x3fff0018,len:4
load:0x3fff001c,len:1044
load:0x40078000,len:8896
load:0x40080400,len:5816
entry 0x400806ac
Multi-task LED Demo
Enter a number in milliseconds to change the LED delay.
Updated LED delay to: 123
```

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

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

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

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

// Settings
static const uint8_t buf_len = 20;

// Pins
static const int led_pin = LED_BUILTIN;

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

//*****
// Tasks

// Task: Blink LED at rate set by global variable
```

```

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

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

    char c;
    char buf[buf_len];
    uint8_t idx = 0;

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

    // Loop forever
    while (1) {

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

            // Update delay variable and reset buffer if we get a newline character
            if (c == '\n') {
                led_delay = atoi(buf);
                Serial.print("Updated LED delay to: ");
                Serial.println(led_delay);
                memset(buf, 0, buf_len);
                idx = 0;
            } else {

                // Only append if index is not over message limit
                if (idx < buf_len - 1) {
                    buf[idx] = c;
                    idx++;
                }
            }
        }
    }
}

//*****
// Main

void setup() {

    // Configure pin
    pinMode(led_pin, OUTPUT);

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

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

```

```

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

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

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

```

Explanation

While you are welcome to use `Serial.parseInt()`, I'm going to show the solution with `atoi()`, which is a more generic (non-Arduino) way of calculating integers from strings. While I receive characters with `Serial.read()`, you can substitute your own UART code if you're using a non-Arduino board.

To use `atoi()`, we need `stdlib.h`, so we include that in our program.

We then set our core (for demo purposes) and define buffer length for storing our received string. Once again, you don't need this buffer if you're using `Serial.parseInt()`.

After that, we define our LED pin and starting delay (500 ms).

Copy Code

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

Note that this delay is assigned to a global variable (`led_delay`). When we parse the integer from the Serial Monitor, we store it into this global variable. The toggle task will continually read this global variable and use it as its delay for blinking the LED.

Our first task should look very similar to the previous challenge: we're just blinking an LED at a constant rate. The only difference is that the blink rate is set by the `led_delay` global variable.

Copy Code

```

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

```

The second task is a little more complicated. When it first starts, we create a local array (local to that task) that is 20 characters long. We then clear the array with the following:

Copy Code

```
memset(buf, 0, buf_len);
```

In the task's main while loop, we wait for any incoming characters over the UART port. If we get one, it's stored to a simple char variable:

Copy Code

```
c = Serial.read();
```

We check to see if that character is the newline character ('\n'). If it is, we convert everything in the buffer up to that newline character to an integer. The `led_delay` global variable is updated to that new integer, and we print something to the console letting the user know what happened. After, we clear the buffer again and reset the index counter.

Copy Code

```
// Update delay variable and reset buffer if we get a newline character
if (c == '\n') {
    led_delay = atoi(buf);
    Serial.print("Updated LED delay to: ");
    Serial.println(led_delay);
    memset(buf, 0, buf_len);
    idx = 0;
}
```

Finally, if the received character is not a newline character, it is saved to the buffer, and the index counter is incremented. Notice that we prevent the index counter from exceeding the buffer length to prevent a buffer overflow. If the buffer is already filled, the character is just ignored.

Copy Code

```
else {
    // Only append if index is not over message limit
    if (idx < buf_len - 1) {
        buf[idx] = c;
        idx++;
    }
}
```

In `setup()`, we initialize the Serial port and wait for 1 second. This allows the serial port to finish setting up before we print anything to it. After, we start our 2 tasks with the same priority (priority 1) and pinned to core 1 (ESP32 only).

Notice that when we're done creating the tasks, we delete the task containing the `setup()` and `loop()` functions. This will prevent `loop()` from running!

Copy Code

```
vTaskDelete(NULL);
```

Instead of `NULL`, you could put the handle of another task as the argument if you wanted to delete a different task.

Finally, we leave `loop()` blank. In fact, execution should never reach there, as we deleted its associated task.

Try playing around with the priorities of the tasks. What happens if you make the "Toggle LED" task priority 2? Why does everything still work?

What happens if you make the "Read Serial" task priority 2 instead (and leave the toggle task at priority 1)? Why does the LED task stop blinking?

Recommended Reading


Example code from the video and the solution can also be found in the following repository:
<https://github.com/ShawnHymel/introduction-to-rtos>.

If you'd like to dig deeper into FreeRTOS, how the scheduler works, and priorities, I recommend checking out these excellent articles:

- FreeRTOS Book and Reference Guide: https://www.freertos.org/Documentation/RTOS_book.html
- FreeRTOS Task Priorities: <https://www.freertos.org/RTOS-task-priority.html>
- Introduction to Basic RTOS Features using SAM4L-EK FreeRTOS Port: http://ww1.microchip.com/downloads/en/DeviceDoc/Atmel-42247-Introduction-to-Basic-RTOS-Features-using-SAM4L-EK-FreeRTOS-Port_Training-Manual.pdf
- [What is a Real-Time Operating System Part 1\(RTOS\)?](#)
- [Introduction to RTOS - Solution to Part 2 \(FreeRTOS\)](#)
- [Introduction to RTOS - Solution to Part 4 \(Memory Management\)](#)
- [Introduction to RTOS - Solution to Part 5 \(FreeRTOS Queue Example\)](#)
- [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 \(Multicore Systems\)](#)

Key Parts and Components

1 Items



Mfr Part # 3405

HUZZAH32 ESP32 FEATHER LOOSE HDR

Adafruit Industries LLC

\$19.95

Details



[Add all Digi-Key Parts to Cart](#)



Have questions or comments? Continue the conversation on [TechForum](#), DigiKey's online community and technical resource.

Visit TechForum

Project Details

License

Attribution

Get Involved

Like

Save



1-800-344-4539


218-681-6674



sales@digikikey.com



218-681-3380

 United States | Copyright © 1995-2023, DigiKey. | All Rights Reserved.
Local Support: 701 Brooks Avenue South, Thief River Falls, MN 56701 USA

Do Not Sell / Do Not Share My Personal Information