Introduction to RTOS - Solution to Part 8 (Software Timers)

By ShawnHymel

Concepts

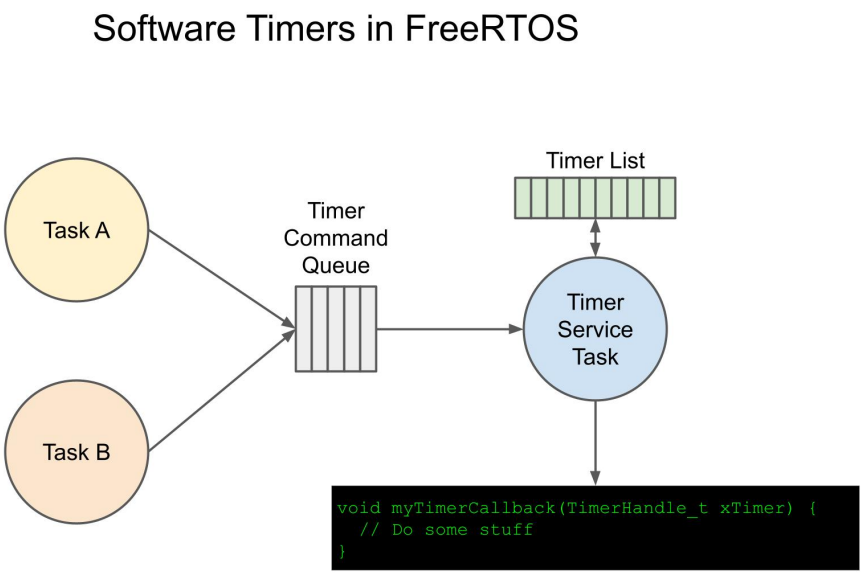
Timers (in embedded systems) allow us to delay the execution of some function or execute a function periodically. These can be hardware timers, which are unique to the architecture, or software timers that are based on some running code or the RTOS tick timer. Note that some hardware timers also allow us to control some hardware functions without software intervention at all.

In FreeRTOS, there are a few ways to delay the execution of a function:

- vTaskDelay() allows us block the currently running task for a set amount of time (given in ticks).
- You can also perform a non-blocking delay by comparing xTaskGetTickCount() with some known timestamp
- Many microcontrollers (and microprocessors) include one or more hardware timers. These can be configured (often by setting various registers) to count up or down and trigger an interrupt service routine (ISR) when they expire (or reach a particular number).
- Software timers exist in code and are not hardware dependent (except for the fact that the RTOS tick timer usually relies on a hardware timer).

FreeRTOS (and many other RTOSes) gives us software timers that we can use to delay calling a function or call a function periodically. FreeRTOS offers an API that makes managing these timers much easier (you can read the API documentation here).

When you include the FreeRTOS timer library, it will automatically run a timer service task (also known as a "timer daemon") separately from all your other tasks. This service task is in charge of managing all of the timers you set and calling the various callback functions. API function calls communicate with the timer service task through a queue to ensure that all commands are received.



When a timer is created, you assign a function (a “callback function”) that is called whenever the timer expires. Note that timers are dependent on the tick timer, which means you can never create a timer with less resolution than the tick (1 ms by default). Additionally, you can set the software timer to be “one-shot” (executes the callback function once after the timer expires) or “auto-reload” (executes the callback function periodically every time the timer expires).

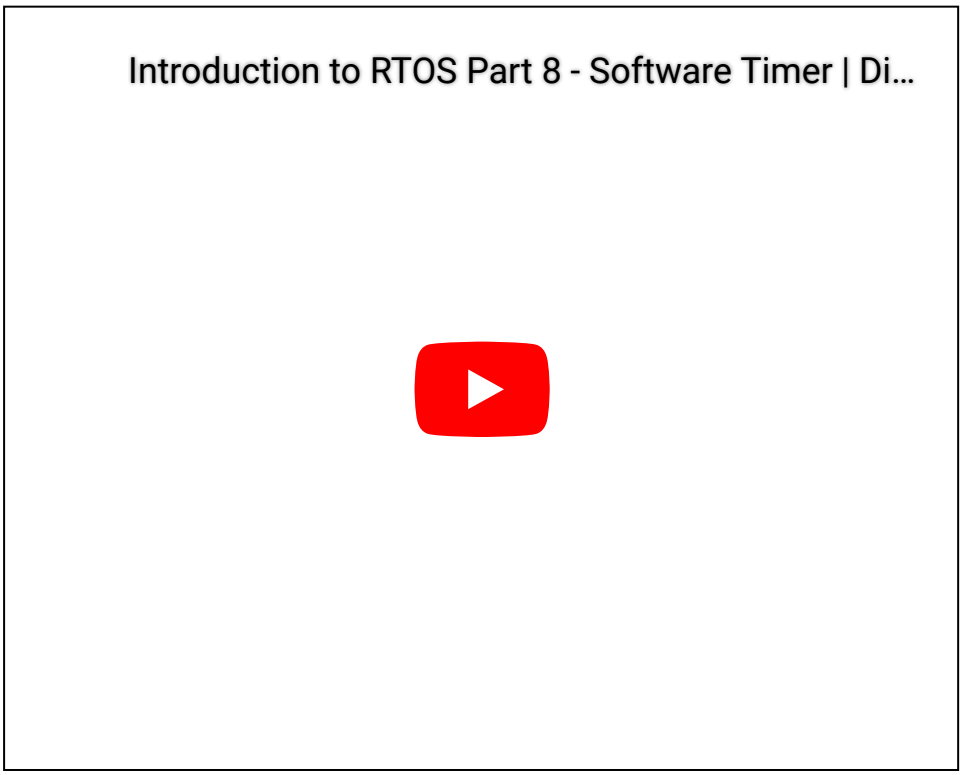
Required Hardware

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

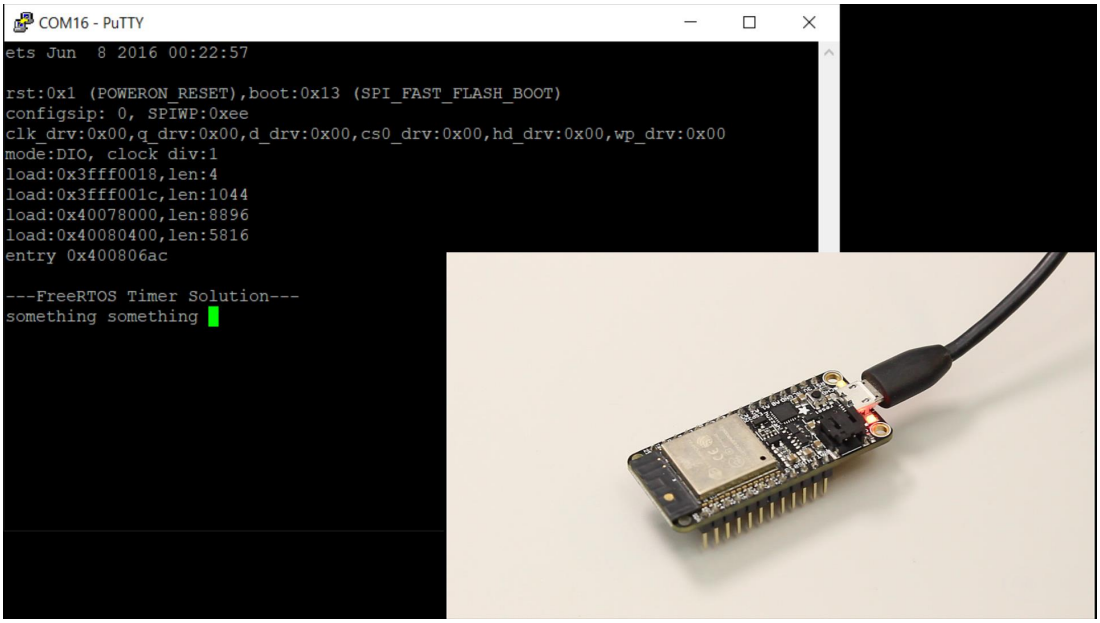
If you have not done so, please watch the following video, which provides the steps necessary to creating tasks and assigning priorities. It also demonstrates a working version of the challenge:



Challenge

Your challenge is to create an auto-dim feature using the onboard LED. We’ll pretend that the onboard LED (LED_BUILTIN) is the backlight to an LCD.

Create a task that echoes characters back to the serial terminal (as we’ve done in previous challenges). When the first character is entered, the onboard LED should turn on. It should stay on so long as characters are being entered.



Use a timer to determine how long it's been since the last character was entered (hint: you can use xTimerStart() to restart a timer's count, even if it's already running). When there has been 5 seconds of inactivity, your timer's callback function should turn off the LED.

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 Solution to LED Dimmer
 *
 * Turn on LED when entering serial commands. Turn it off if serial is inactive
 * for 5 seconds.
 *
 * Date: February 1, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

// You'll likely need this on vanilla FreeRTOS
//#include timers.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 TickType_t dim_delay = 5000 / portTICK_PERIOD_MS;

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

// Globals
static TimerHandle_t one_shot_timer = NULL;

//*****************************************************************************
// Callbacks

// Turn off LED when timer expires
void autoDimmerCallback(TimerHandle_t xTimer) {
  digitalWrite(led_pin, LOW);
}
```

```
//*****************************************************************************
// Tasks

// Echo things back to serial port, turn on LED when while entering input
void doCLI(void *parameters) {

  char c;

  // Configure LED pin
  pinMode(led_pin, OUTPUT);

  while (1) {

    // See if there are things in the input serial buffer
    if (Serial.available() > 0) {

      // If so, echo everything back to the serial port
      c = Serial.read();
      Serial.print(c);

      // Turn on the LED
      digitalWrite(led_pin, HIGH);

      // Start timer (if timer is already running, this will act as
      // xTimerReset() instead)
      xTimerStart(one_shot_timer, portMAX_DELAY);
    }
  }
}

//*****************************************************************************
// 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 Timer Solution---");

  // Create a one-shot timer
  one_shot_timer = xTimerCreate(
                     "One-shot timer",     // Name of timer
                     dim_delay,            // Period of timer (in ticks)
                     pdFALSE,              // Auto-reload
                     (void *)0,            // Timer ID
                     autoDimmerCallback);  // Callback function

  // Start command line interface (CLI) task
  xTaskCreatePinnedToCore(doCLI,
                          "Do CLI",
                          1024,
                          NULL,
                          1,
                          NULL,
                          app_cpu);

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

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

Explanation

To begin, we set our timer delay (note that it's given in ticks, so we must convert milliseconds to ticks), choose the LED pin, and create a handle to our software timer.

Copy Code

```
// Settings
static const TickType_t dim_delay = 5000 / portTICK_PERIOD_MS;

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

// Globals
static TimerHandle_t one_shot_timer = NULL;
```

This callback function gets called whenever the tick timer expires. Note that it is executed at whatever priority level the timer service task is set at (which is priority level 1 by default in the ESP32 Arduino package). In it, we simply turn off the LED.

Copy Code

```
// Turn off LED when timer expires
void autoDimmerCallback(TimerHandle_t xTimer) {
  digitalWrite(led_pin, LOW);
}
```

Our task is relatively simple. It just echoes anything received on the serial port back out the same serial port (I'm not handling return characters like I've done previously, but you're welcome to add that feature in).

Copy Code

```
// Echo things back to serial port, turn on LED when while entering input
void doCLI(void *parameters) {

  char c;

  // Configure LED pin
  pinMode(led_pin, OUTPUT);

  while (1) {

    // See if there are things in the input serial buffer
    if (Serial.available() > 0) {

      // If so, echo everything back to the serial port
      c = Serial.read();
      Serial.print(c);

      // Turn on the LED
      digitalWrite(led_pin, HIGH);

      // Start timer (if timer is already running, this will act as
      // xTimerReset() instead)
      xTimerStart(one_shot_timer, portMAX_DELAY);
    }
  }
}
```

The important part of the task is that each time a new character is received, the LED is turned on and we start the timer with xTimerStart(). If the timer is already running, xTimerStart() works like xTimerReset() and resets the counter. This allows the LED to stay on if we're still entering characters.

In setup(), we start the serial port (as we've previously done) and create the timer. Note that we're using a one-shot timer, as we do not need the function being called periodically. Once the LED is turned off, that function does not need to be called again until the LED has been turned on again. xTimerStart() (called in the task) allows us to restart a timer if it has already expired.

Copy Code

```
// Create a one-shot timer
one_shot_timer = xTimerCreate(
```

```
                "One-shot timer",     // Name of timer
                dim_delay,            // Period of timer (in ticks)
                pdFALSE,              // Auto-reload
                (void *)0,            // Timer ID
                autoDimmerCallback);  // Callback function
```

Next, we start the task, which is in charge of handling the command line interface. We pretend that the command line interface (in this case) is similar to an LCD where we enter commands through some kind of interface (say, some buttons). The backlight would stay on so long as buttons are being pressed.

Copy Code

```
// Start command line interface (CLI) task
xTaskCreatePinnedToCore(doCLI,
                        "Do CLI",
                        1024,
                        NULL,
                        1,
                        NULL,
                        app_cpu);
```

Finally, we delete the “setup and loop” task (as we’ve done in previous lectures) and ensure that loop() is empty.

Copy Code

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

Recommended Reading

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

If you would like to dig into these topics further, I recommend checking out the following excellent articles:

- Understanding and Using FreeRTOS Software Timers: https://dzone.com/articles/understanding-and-using-freertos-software-timers
- FreeRTOS Software Timer documentation page: https://www.freertos.org/RTOS-software-timer.html
- FreerTOS Software Timer API documentation: https://www.freertos.org/FreeRTOS-Software-Timer-API-Functions.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 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 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

TechForum

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

Visit TechForum

Arduino | 3D Printing | Raspberry Pi

Project Details

Platforms

Arduino

Development

C

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