

Introduction to RTOS - Solution to Part 6 (FreeRTOS Mutex Example)

[By ShawnHymel](#)

Concepts

A mutex (short for MUTual EXclusion) is a flag or lock used to allow only one thread to access a section of code at a time. It blocks (or locks out) all other threads from accessing the code or resource. This ensures that anything executed in that critical section is thread-safe and information will not be corrupted by other threads.

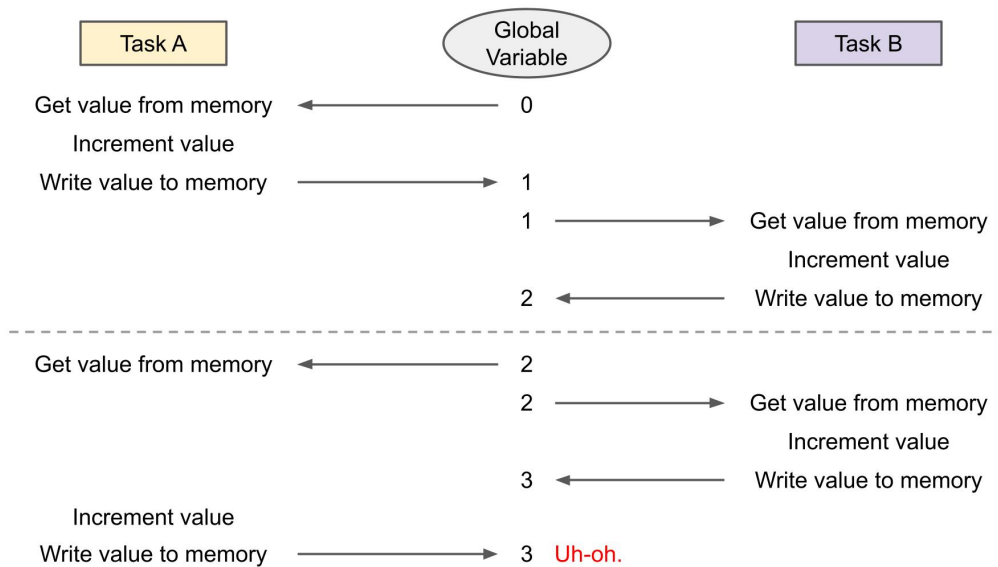
A mutex is like a single key sitting in a basket at a coffee shop. The key can be used to unlock a shared public restroom.



A person takes the key when they wish to use the shared resource (restroom) and returns it when they are done. While they are in the restroom, no one else may enter. Other people (analogous to threads) must wait for the key. When it is returned, another person may take the key to use the restroom.

Without such a locking mechanism, we are liable to end up with a [race condition](#) in our code as multiple threads attempt to read and modify a common resource (such as a global variable, serial port, etc.).

For example, let's look at two tasks attempting to increment a global variable. Assuming the increment cannot be done [atomically](#), each task must read the global variable from memory, increment it, and write it back to memory in separate instruction cycles. As a result, we do not have a way to control when other threads might interrupt this process.



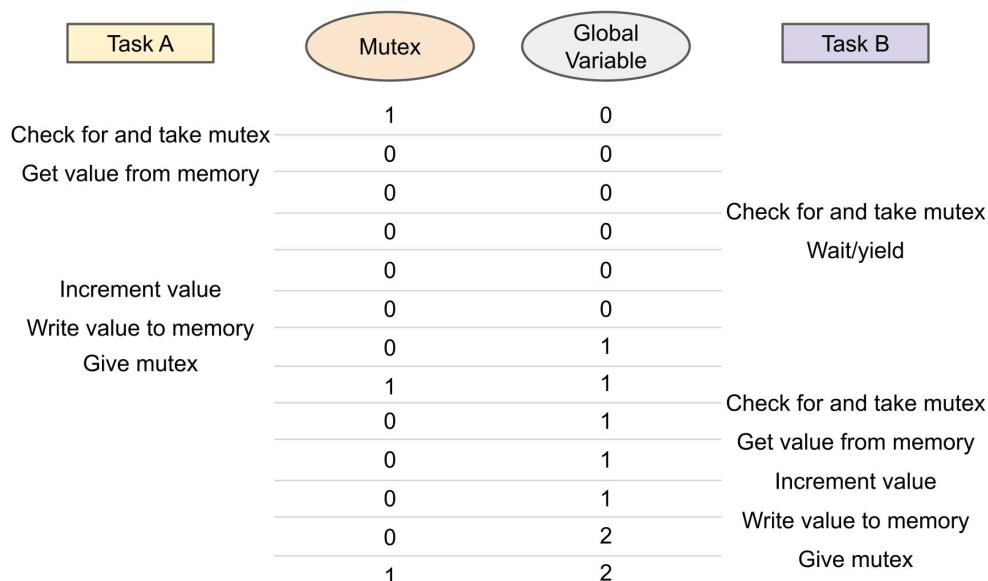
We can end up with a situation where Task A reads the value, Task B interrupts to read the same value, increment it, and write it back. When execution returns to Task A, it still has the original value in local, working memory. It increments that value and writes it back to memory, overwriting the work of Task B. This results in the same value being written to global/shared memory, even though two increment commands were executed.

This is known as a “race condition,” in that the exact timing of the sequence of events to perform an action can change the outcome. To eliminate the race condition, we need some kind of locking mechanism to prevent more than one thread from executing the read-modify-write sequence at a time.

We can use a mutex to help, as it allows mutual exclusion of thread execution in a [critical section](#). In an RTOS, a mutex is simply a global (or shared) binary value that can be accessed atomically. That means if a thread takes the mutex, it can read and decrement the value without being interrupted by other threads. Giving the mutex (incrementing the value by one) is also atomic.

What makes mutexes work, much like queues, is that a thread is forced to wait if a mutex is not available. If a thread sees that a mutex cannot be taken, it enters into the block state or it can do some other work before checking the mutex again.

Here is an example of how execution might work with two tasks attempting to increment a global variable protected by a mutex.



Each line denotes one (or a group) of instructions executed on the processor. Notice that Task A can only increment the global variable while it has the mutex. Task B can interrupt it, but it cannot enter the critical

section (as there is no mutex available). Only when Task A is done and returns the mutex can Task B enter the critical section to increment the global variable.

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 6 - Mutex | Digi-Key Ele...



Challenge

Starting with the code given below, modify it to protect the task parameter (delay_arg) with a mutex. With the mutex in place, the task should be able to read the parameter (parameters) into the local variable (num) before the calling function's stack memory goes out of scope (the value given by delay_arg).

Copy Code

```
/**
 * FreeRTOS Mutex Challenge
 *
 * Pass a parameter to a task using a mutex.
 *
 * Date: January 20, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

// You'll likely need this on vanilla FreeRTOS
#include semphr.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

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

//*****
// Tasks

// Blink LED based on rate passed by parameter
void blinkLED(void *parameters) {

    // Copy the parameter into a local variable
    int num = *(int *)parameters;

    // Print the parameter
    Serial.print("Received: ");
    Serial.println(num);

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

    // Blink forever and ever
    while (1) {
        digitalWrite(led_pin, HIGH);
        vTaskDelay(num / portTICK_PERIOD_MS);
        digitalWrite(led_pin, LOW);
        vTaskDelay(num / portTICK_PERIOD_MS);
    }
}

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

void setup() {

    Long int delay_arg;

    // 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 Mutex Challenge---");
    Serial.println("Enter a number for delay (milliseconds)");

    // Wait for input from Serial
    while (Serial.available() <= 0);

    // Read integer value
    delay_arg = Serial.parseInt();
    Serial.print("Sending: ");
    Serial.println(delay_arg);

    // Start task 1
    xTaskCreatePinnedToCore(blinkLED,
                           "Blink LED",
                           1024,
                           (void *)&delay_arg,
                           1,
                           NULL,
                           app_cpu);

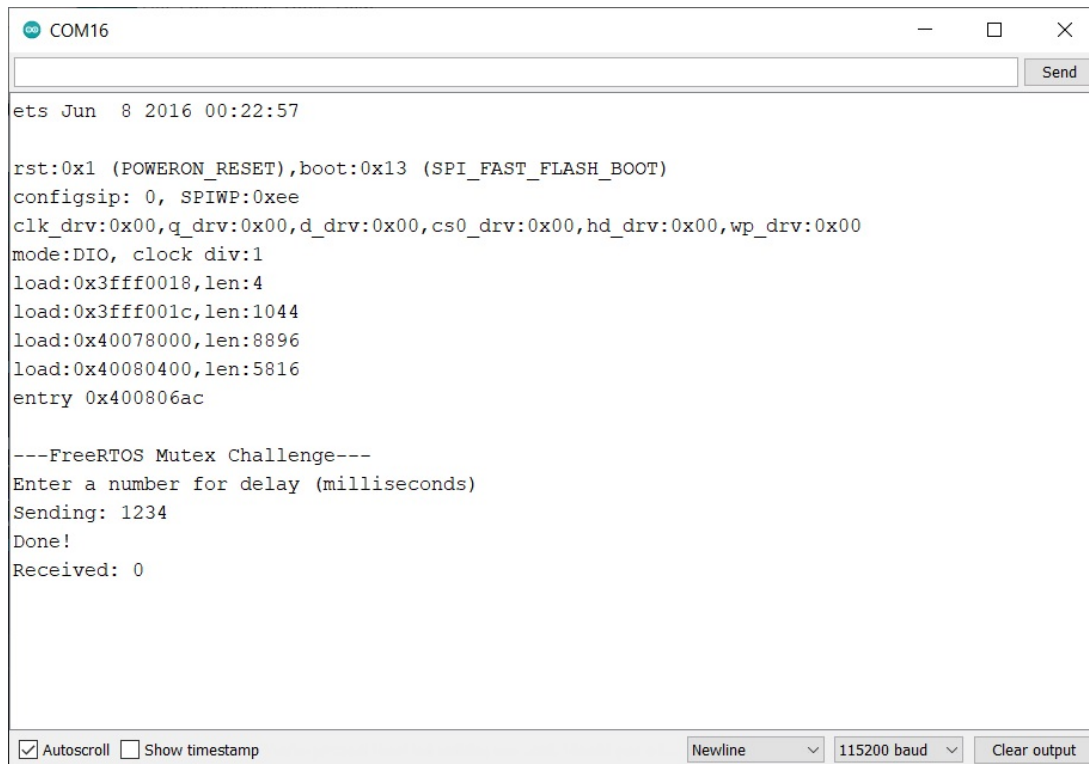
    // Show that we accomplished our task of passing the stack-based argument
    Serial.println("Done!");
}

void loop() {

    // Do nothing but allow yielding to lower-priority tasks
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}

```

When you run the code above as-is, you should see that regardless of what you enter for the number, the task will always read 0. That's because the local variable (`delay_arg`) goes out of scope before the task can read the value given by the parameters pointer.



```
COM16

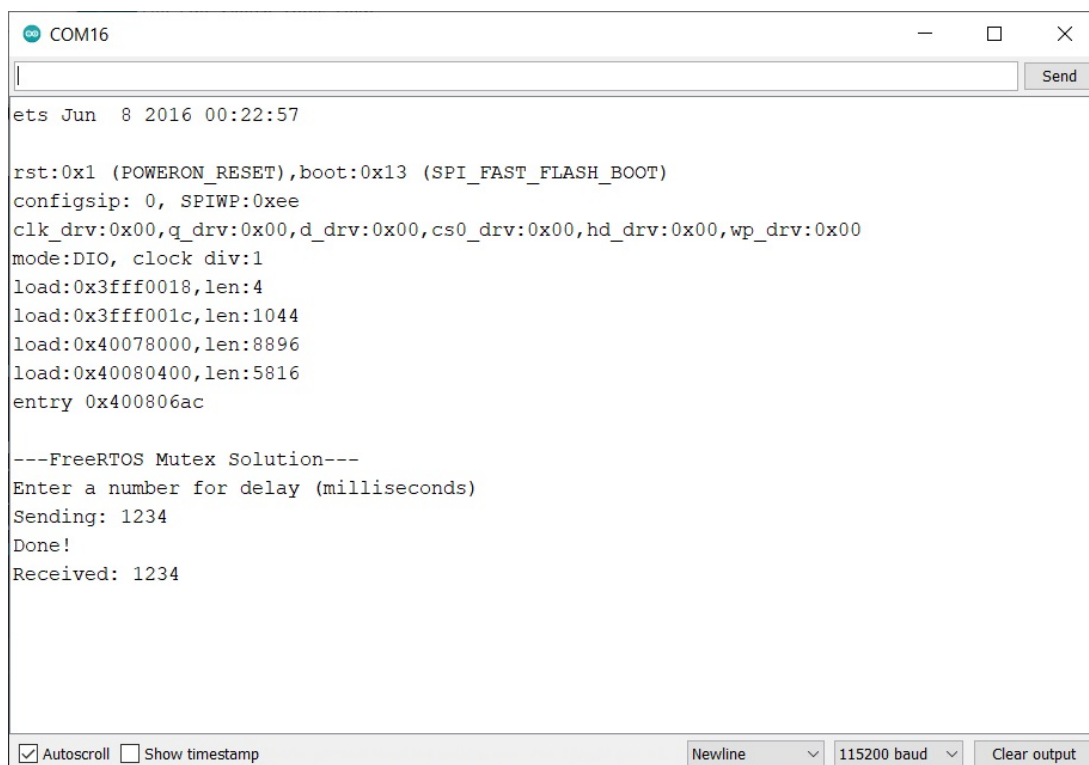
ets Jun  8 2016 00:22:57

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

---FreeRTOS Mutex Challenge---
Enter a number for delay (milliseconds)
Sending: 1234
Done!
Received: 0

Autoscroll Show timestamp Newline 115200 baud Clear output
```

After adding a mutex, the task should be able to read the parameters variable before the memory location goes out of scope. This is what you should see in the Serial terminal if you did it right:



```
COM16

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

---FreeRTOS Mutex Solution---
Enter a number for delay (milliseconds)
Sending: 1234
Done!
Received: 1234

Autoscroll Show timestamp Newline 115200 baud Clear output
```

Warning! This is a hack for two reasons:

1. The FreeRTOS documentation strongly discourages using stack memory to pass arguments to the task creation process, which is exactly what we're doing here (even if it does work). See the note on `pvParameters` in [this API reference for xTaskCreate](#) for more information.
2. This is an inappropriate use of a mutex. In fact, it's closer to how we should use a semaphore (to signal to another task that some information is ready to be consumed). However, that's perfect, as we will cover semaphores in the next lecture.

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 Mutex Solution
 *
 * Pass a parameter to a task using a mutex.
 *
 * Date: January 20, 2021
 * Author: Shawn Hymel
 * License: 0BSD
 */

// You'll likely need this on vanilla FreeRTOS
#include semphr.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

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

// Globals
static SemaphoreHandle_t mutex;

//*****
// Tasks

// Blink LED based on rate passed by parameter
void blinkLED(void *parameters) {

    // Copy the parameter into a local variable
    int num = *(int *)parameters;

    // Release the mutex so that the creating function can finish
    xSemaphoreGive(mutex);

    // Print the parameter
    Serial.print("Received: ");
    Serial.println(num);

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

    // Blink forever and ever
    while (1) {
        digitalWrite(led_pin, HIGH);
        vTaskDelay(num / portTICK_PERIOD_MS);
        digitalWrite(led_pin, LOW);
        vTaskDelay(num / portTICK_PERIOD_MS);
    }
}

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

void setup() {

    long int delay_arg;

    // 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 Mutex Solution---");
Serial.println("Enter a number for delay (milliseconds)");

// Wait for input from Serial
while (Serial.available() <= 0);

// Read integer value
delay_arg = Serial.parseInt();
Serial.print("Sending: ");
Serial.println(delay_arg);

// Create mutex before starting tasks
mutex = xSemaphoreCreateMutex();

// Take the mutex
xSemaphoreTake(mutex, portMAX_DELAY);

// Start task 1
xTaskCreatePinnedToCore(blinkLED,
                        "Blink LED",
                        1024,
                        (void *)&delay_arg,
                        1,
                        NULL,
                        app_cpu);

// Do nothing until mutex has been returned (maximum delay)
xSemaphoreTake(mutex, portMAX_DELAY);

// Show that we accomplished our task of passing the stack-based argument
Serial.println("Done!");
}

void loop() {

    // Do nothing but allow yielding to lower-priority tasks
    vTaskDelay(1000 / portTICK_PERIOD_MS);
}

```

Explanation

There may be more than one way to accomplish this task with mutexes. My solution added 5 lines of code. The first is to create a global mutex handle at the top.

Copy Code

```

// Globals
static SemaphoreHandle_t mutex;

```

In the task function, we “give” the mutex back just after copying in the value from the parameters pointer.

Copy Code

```

// Release the mutex so that the creating function can finish
xSemaphoreGive(mutex);

```

Because a mutex will initialize at 1, it means we need to “take” the mutex in our setup code, which we do just after creating it (before we start the task).

Copy Code

```

// Create mutex before starting tasks
mutex = xSemaphoreCreateMutex();

// Take the mutex
xSemaphoreTake(mutex, portMAX_DELAY);

```

After we start the task, we block the “setup and loop” task until the mutex is given back (which is done in the task). We do this by trying to “take” the mutex and delaying (blocking) for the maximum amount of time.

Copy Code

```
// Do nothing until mutex has been returned (maximum delay)
xSemaphoreTake(mutex, portMAX_DELAY);
```

That’s it! We create a mutex, “take” it to set it to 0 and then wait for the task to set it back to 1 after it’s read the parameters. This is a roundabout way to read a parameter, but it gets the job done and sets us up to learn more about semaphores.

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:

- Difference between “lock,” “mutex,” and “semaphore:” <https://stackoverflow.com/questions/2332765/lock-mutex-semaphore-whats-the-difference>
- Mutexes and Semaphores Demystified: <https://barrgroup.com/embedded-systems/how-to/rtos-mutex-semaphore>
- FreeRTOS Semaphores (and Mutexes) API reference: <https://www.freertos.org/a00113.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 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

[Arduino](#) | [3D Printing](#) | [Raspberry Pi](#)

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



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