

FreeRTOS X

- [Critical Section](#)
- [FreeRTOS Producer Consumer Tasks](#)

Critical Section

Objective

To go over **Critical Sections** in an application as well as other kernel API calls that, which for the most part, you should refrain from using unless necessary.

What are Critical Sections

Critical sections (also called critical regions) are sections of code that makes sure that only one thread is accessing resource or section of memory at a time. In a way, you are making the critical code **atomic** in a sense that another task or thread will only run after you exit the critical section.

Implementations of a critical section are varied. Many systems create critical sections using semaphores, but that is not the only way to produce a critical section.

How to Define a Critical Section

```
/* Enter the critical section. In this example, this function is itself called
from within a critical section, so entering this critical section will result
in a nesting depth of 2. */
taskENTER_CRITICAL();

/* Perform the action that is being protected by the critical section here. */
/* Exit the critical section. In this example, this function is itself called
from a critical section, so this call to taskEXIT_CRITICAL() will decrement the
nesting count by one, but not result in interrupts becoming enabled. */
taskEXIT_CRITICAL();
```

Code Block 1. Entering and Exiting a Critical Section (FreeRTOS.org)

Using the two API calls `taskENTER_CRITICAL()` and `taskEXIT_CRITICAL()`, one is able to enter and exit a

critical section.

Implementation in FreeRTOS

Typically, when FreeRTOS is ported to a system, critical sections will **STOP/DISABLE** the **OS Tick interrupt** that calls the RTOS kernel. If the OS tick interrupt triggers during your critical section, the interrupt is in a **pending** state until you re-enable interrupts. It is not missed, but is delayed due to the interrupts that get disabled.

If your task takes too long to do its operation, RTOS can perform in a real time manner because it has been shutdown during your critical section. Which is why you need to be super selective about using a critical section.

The FreeRTOS implementation for Critical Sections by Espressive (ESP32 platform) does not use RTOS, but actually uses a mutex that is passed in instead. It becomes an abstraction to using semaphore take and give API calls.

Critical Section with interrupt enable/disable vs. Mutex

First of all, a mutex provides you the ability to guard critical section of code that you do not want to run in multiple tasks at the same time. For instance, you do not want SPI bus to be used simultaneously in multiple tasks. Choose a mutex whenever possible, but note that a critical section with interrupt disable and re-enable method is typically faster. If all you need to do is read or write to a few standard number data types atomically then a critical section can be utilized. But a better alternative would be to evaluate the structure of your tasks and see if there is really a need to use a mutex or critical section.

Use a mutex when using a peripheral that you must not use simultaneously, like SPI, UART, I2C etc. For example, disabling and re-enabling interrupts to guard your SPI from being accessed by another task is a poor choice. This is because during the entire SPI transaction, you will have your interrupts disabled and no other (higher) priority tasks can get scheduled and the OS could miss its ticks. In this case, a mutex is a better choice because you only want to guard the tasks from accessing this critical section from each other, and you do not need care if other tasks get scheduled if they will not use the SPI bus.

FreeRTOS Producer Consumer Tasks

Objective

- Learn how Tasks and Queues work
 - Assess how task priorities affect the RTOS Queue cooperative scheduling
-

Queues and Task Priorities

Tasks of equal priority that are both ready to run are scheduled by the RTOS in a round-robin fashion. This type of context switch is called **Preemptive Context Switch**.

Queues' API can also perform context switches, but this is a type of **Cooperative Context Switch**. What this means is that if `xQueueSend()` API is sending an item to a higher priority task that was waiting on the same queue using the `xQueueReceive()` API, then the sending task will switch context inside of the `xQueueSend()` function over to the other task. Therefore, task priorities matter when using the queue API.

Also note that when the cooperative context switch occurs, it does not wait for the next tick of preemptive scheduling to switch context. Typical RTOSes support both cooperative and preemptive scheduling, and in fact, you can turn off preemptive scheduling in `FreeRTOSConfig.h`

```
static QueueHandle_t switch_queue;
typedef enum {
    switch__off,
    switch__on
```

```

} switch_e;
// TODO: Create this task at PRIORITY_LOW
void producer(void *p) {
    while (1) {
        // This xQueueSend() will internally switch context to "consumer" task because it is higher priority
        // Then, when the consumer task sleeps, we will resume out of xQueueSend() and go over to the next task

        // TODO: Get some input value from your board
        const switch_e switch_value = get_switch_input_from_switch0();

        // TODO: Print a message before xQueueSend()
        // Note: Use printf() and not fprintf(stderr, ...) because stderr is a polling printf
        xQueueSend(switch_queue, &switch_value, 0);
        // TODO: Print a message after xQueueSend()

        vTaskDelay(1000);
    }
}
// TODO: Create this task at PRIORITY_HIGH
void consumer(void *p) {
    switch_e switch_value;
    while (1) {
        // TODO: Print a message before xQueueReceive()
        xQueueReceive(switch_queue, &switch_value, portMAX_DELAY);
        // TODO: Print a message after xQueueReceive()
    }
}
void main(void) {
    // TODO: Create your tasks
    xTaskCreate(producer, ...);
    xTaskCreate(consumer, ...);

    // TODO Queue handle is not valid until you create it
    switch_queue = xQueueCreate(<depth>, sizeof(switch_e)); // Choose depth of item being our enum (1 should be fine)

    vTaskStartScheduler();
}

```

```
}
```

Assignment

- Finish `producer task` that reads a switch value and sends it to the queue
 - Create an enumeration such as `typedef enum { switch__off, switch__on} switch_e;`
- Create a queue, and have the `producer task` send switch values every second to the queue
- Finish `consumer task` that is waiting on the enumeration sent by the `producer task`

After ensuring that the producer task is sending values to the consumer task, do the following:

- Ensure that the following is already setup:
 - Print a message `producer task` **before** and **after** sending the switch value to the queue
 - Print a message in the `consumer task` **before** and **after** receiving an item from the queue
 - You may use the following:

```
printf("%s(), line %d, sending message\n", __FUNCTION__, __LINE__);
```

Note down the **Observations** by doing the following:

- Use higher priority for `producer task`, and note down the order of the print-outs
- Use higher priority for `consumer task`, and note down the order of the print-outs
- Use same priority level for both tasks, and note down the order of the print-outs

Answer **Additional Questions**:

- What is the purpose of the block time during `xQueueReceive()`?
- What if you use ZERO block time during `xQueueReceive()`?

What to turn in

- Submit all relevant source code
- Relevant screenshots of serial terminal output
- Submit explanation to the questions as comments in your code at the top of your source code file
 - Explanation of the **Observations**
 - Explanation for the **Additional Questions**

Extra Credit

This extra credit will help you in future labs, so it is highly recommended that you achieve this. You will add a CLI handler to be able to:

- Suspend a task by name
- Resume a task by name

Please follow [this article](#) to add your CLI command. Here is reference code for your CLI:

```
app_cli_status_e cli__task_control(app_cli__argument_t argument, sl_string_t user_input_minus_command_name,
                                   app_cli__print_string_function cli_output) {
    sl_string_t s = user_input_minus_command_name;
    // If the user types 'taskcontrol suspend led0' then we need to suspend a task with the name of 'led0'
    // In this case, the user_input_minus_command_name will be set to 'suspend led0' with the command-name
    if (sl_string__begins_with_ignore_case(s, "suspend")) {
        // TODO0: Use sl_string API to remove the first word, such that variable 's' will equal to 'led0'
        // TODO0: Or you can do this: char name[16]; sl_string__scanf("%*s %16s", name);

        // Now try to query the tasks with the name 'led0'
        TaskHandle_t task_handle = xTaskGetHandle(s);
        if (NULL == task_handle) {
            // note: we cannot use 'sl_string__printf("Failed to find %s", s);' because that would print exit code
            sl_string__insert_at(s, "Could not find a task with name:");
            cli_output(NULL, s);
        } else {
            // TODO0: Use vTaskSuspend()
        }
    }
}
```



```
} else if (sl_string__begins_with_ignore_case(s, "resume")) {  
    // TODO  
} else {  
    cli_output(NULL, "Did you mean to say suspend or resume?\n");  
}  
return APP_CLI_STATUS__SUCCESS;}
```
