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 you 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 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.

Revision #3
Created 6 years ago by Admin
Updated 1 year ago by Preet Kang