Extending Semaphores with Non-Preemptive Critical Sections in FreeRTOS

by

Carlos Salazar

CIS 721 Final Project Report

Spring 2014

KANSAS STATE UNIVERSITY
Manhattan, Kansas

Chapter 1

Introduction

The following paper describes the design and implementation of non-preemptive critical sections (NPCS) for semaphores in the FreeRTOS real-time operating system. Non-preemptive critical sections prevent deadlock from occurring in a preemptive operating system by ensuring that a task that has taken a semaphore may not be preempted until it gives that semaphore. This implementation was tested using the Texas Instruments Tiva(TM) C Series LaunchPad EK-TM4C123GXL microcontroller board[1]. The remainder of this section contains background information on the EK-TM4C123GXL microcontroller and a description of non-preemptive critical sections. Following the introduction, chapter 2 provides an overview of FreeRTOS. Chapter 3 describes the modifications made to FreeRTOS in order to implement NPCS. Chapter 4 presents a test application which exhibits deadlock, which is resolved using our NPCS implementation.

1.1 TI Tiva(TM) C Series LaunchPad EK-TM4C123GXL

The EK-TM4C123GXL 1.1 is an inexpensive microcontroller created by Texas Instruments. From 1.1 we see that the EK-TM4C123GXL is a low power device with a very small amount

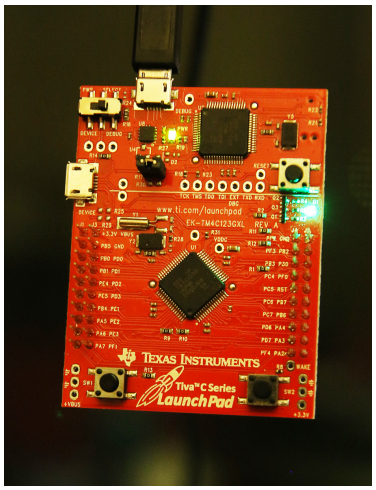

Figure 1.1: *EK-TM4C123GXL*

Feature Name	Specs
CPU	Cortex M4 32 bit 80 MHz
Storage	256 KB Flash
RAM	32 KB
EEPROM	2 KB
ROM	On-chip, with drivers and boot loaders
Other features	3 buttons RGB LED USB Host interface

Table 1.1: *EK-TM4C123GXL specifications*[1]

of memory and storage available. This makes it an ideal platform for the FreeRTOS operating system. All implementation work described in this paper was done using the EK-TM4C123GXL port of FreeRTOS.

1.2 Non-Preemptive Critical Sections (NPCS)

In a real-time preemptive operating system, it is possible to enter a deadlocked state in which no task in the system is able to proceed[2]. Such a scenario can occur with a set of n tasks and n mutexes. Each task gains access to one mutex and requests access to another, however no task may relinquish the mutexes it holds until it gets another mutex. Each task successfully gets access to one of the mutexes it needs, and is then immediately preempted by another task until all mutexes are held, at which point each task attempts to get its second mutex. Because all mutexes are held by tasks that can not relinquish their mutexes until they obtain another mutex, no task can be executed. We consider the section of code between the take and give operations for a mutex to be a critical section. Deadlock is defined by the circular-wait scenario described above, thus our goal is to prevent circular-waits on

mutexes. The simplest solution is to make it impossible to preempt a task once it has entered a critical section. This guarantees that only one task at a time is in a critical section, thus no other mutexes are held by other tasks. This solution is known as non-preemptive critical sections (NPCS). Using NPCS prevents us from ever entering a circular wait/deadlock[2]. In chapters 3 and 4 we describe the implementation of NPCS in FreeRTOS and an application that demonstrates how deadlock may be prevented with NPCS, respectively.

Chapter 2

FreeRTOS

FreeRTOS[3] is a free and open-source real-time operating system for embedded systems. It is maintained by Real Time Engineers Ltd and distributed under the GPL. It is quite popular, having been ported to many different microcontrollers[4]. FreeRTOS is implemented in C and assembly, with a highly compact codebase. The common parts of the operating system are written in C, while a handful of functions have port-specific implementations in assembly.

2.1 Tasks

FreeRTOS operates using tasks. Tasks are sequential units of execution which are used the way one might use threads or processes in a traditional operating system. Only one task may be executed at a time in FreeRTOS, so concurrency is handled through multitasking[5] - FreeRTOS runs each task for some period and then switches to the next one as determined by the scheduler. Each task is assigned a priority, by the developer of the application.

2.2 Scheduling

The scheduler[6] in FreeRTOS determines the order in which tasks should be interleaved and executed. When the scheduler pauses the execution of one task and begins executing another, this is known as a context switch[7]. The scheduler determines when context switches occur, except when a task itself triggers a context switch by sleeping, or blocking on some resource.

2.2.1 Interrupts/ISRs

Operations such as clock tick increments are conducted using interrupts (ISR in FreeRTOS nomenclature)[8]. ISRs are like tasks in that they are series of instructions which are scheduled and executed, but are different in that they are meant to handle something urgent like a signal[9] from the hardware, and may not themselves be preempted during their execution. A task which is interrupted may not necessarily be scheduled again immediately once the interrupt returns. If a higher priority task is ready it will be scheduled instead.

2.2.2 Critical Sections in FreeRTOS

FreeRTOS has something called critical sections which are distinct from the definition used earlier in 1.2. A critical section[10] in FreeRTOS is a region of code which falls between calls to the taskENTER_CRITICAL and taskEXIT_CRITICAL macros within this region, no context switches may occur whatsoever. This also means that ISRs are not executed within the critical section.

2.3 API

Furthermore, FreeRTOS has three APIs[11]: full featured, alternative, and lightweight. For this project we have focused on the full featured API. The alternative API implements everything that exist in the full featured API except that it trades off interrupt responsiveness for raw execution time by making extensive use of critical sections. However, this API is deprecated and use of it is discouraged for new projects. The lightweight API is usable within ISRs and is even faster than the alternative API, however it is far more difficult to use because it it is not interrupt safe, and does not automatically context switch when a higher priority task unblocks due to a queue operation.

2.4 Queues

In FreeRTOS communication and synchronization between tasks or interrupts is carried out using queues[12]. Generally these queues are used as FIFO buffers. Queues usually pass data by value, but can also be used to pass references. Queues may be blocked upon - a task will block if it tries to read data from an empty queue. When multiple tasks block on the same queue, the highest priority task will unblock first.

2.5 Semaphores

In this section, we describe semaphores as they exist in FreeRTOS. Semaphores in FreeRTOS are implemented using the queue mechanism described in 2.3. There are four types of semaphores in FreeRTOS: binary, counting, mutex, and recursive mutex. Binary semaphores[13] are similar to mutex semaphores, except that they lack priority inheritance. The FreeRTOS documentation describes a binary semaphore as a queue which can only hold one item. Users of this queue do not care what is in the queue, so much as whether or not there is something

in the queue. A counting semaphore[14], using the queue description is just a queue with length greater than one. This may be used to track events that have occurred but have not been processed, or to manage access to a limited pool of resources. Mutex semaphores[15] are binary semaphores which also use priority inheritance. Priority inheritance is meant to reduce priority inversion, but does not resolve it. The documentation recommends mutexes for enforcing mutual exclusion, and binary semaphores for task synchronization. The final type of semaphore in FreeRTOS, the recursive mutex is the same as a mutex except that it may be taken repeatedly by a single task, however the recursive mutex semaphore will not unblock until it has been given exactly the same amount of times as it has been taken.

The NPCS implementation described in this paper is applied to the first three types of semaphores (binary, counting, and mutex).

Chapter 3

NPCS Implementation

In this section we describe the changes made to FreeRTOS in order to implement non-preemptive critical sections(NPCS) for semaphores. As mentioned in 2.5, there are four types of semaphores in FreeRTOS, the implementation of NPCS described here is only applied to the binary, counting, and mutex semaphores. Furthermore, the evaluation of these changes was made with a demo application which uses only mutex semaphores 4.

We attempted to minimally alter FreeRTOS during our implementation, as can be seen in figure 3.1, only small subset of source files were ever changed. The changes within those files were kept to a minimum as well.

The functions called when using a semaphore[16], are defined within semphr.h and implemented using C macros. Specifically, in we altered two functions in semphr.h: xSemaphoreTake() and xSemaphoreGive(). The macros for these functions originally called xQueueGenericReceive() and xQueueGenericSend() respectively. In order to implement NPCS, we replaced calls to these functions with calls to two new functions named npcsQueueTakeMutex() and npcsQueueGiveMutex().

The functions used to implement semaphores are all actually contained within queue.c. In queue.c, we created the npcsQueueTakeMutex() and npcsQueueGiveMutex() functions.

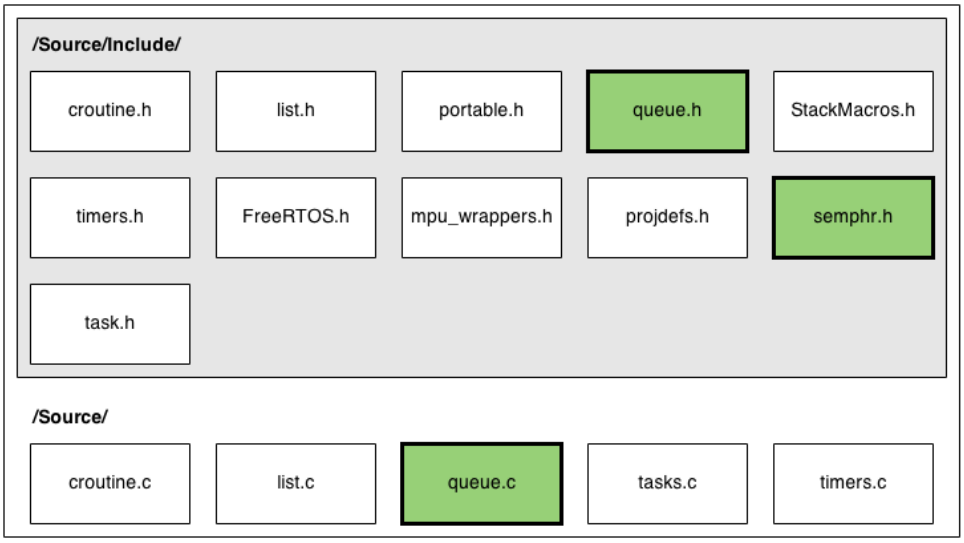

Figure 3.1: *Changes made to the FreeRTOS source code. Green indicates a source file which was altered during the NPCS implementation.*

These functions ultimately end up using the same queue operations that the original, non-NPCS semaphores also used. However, some additional things happen in these functions, as we describe in the below sections 3.1 and 3.2.

Finally, queue.h was modified only as necessary to add function prototypes for the new functions we added to queue.c.

3.1 npcsQueueTakeMutex()

In npcsQueueTakeMutex(), when a task takes a mutex, its handle is stored. At this point, its priority is elevated to the maximum priority, while it's original priority is stored. A counter is incremented to keep track of how many mutexes this task has given and taken so that we may later restore the task back to its original priority. After all of this has been done, xQueueGenericReceive() is called. Everything described here takes place within a critical section. However, when this function is first entered, we check to see if the task which called the function is the indicated mutex holder. If they are not, they are forced to block until

the mutex counter returns to 0 (meaning the other task is done with all mutexes).

3.1.1 How this varies from standard NPCS

This is a slight variation on strict NPCS because we made the decision to allow preemption by interrupts so that things such as clock ticks may continue. In this configuration, it is possible for other tasks to get scheduled while a mutex is held if the mutex holder sleeps or yields. It is still impossible for a different task to take a mutex until the task in the NPCS returns its mutex, making deadlock impossible. Furthermore, as soon as the sleep finishes in the mutex holding task, it will be immediately rescheduled- preempting any other task due to it holding the maximum priority (which we assume no other task will ever be assigned).

3.2 npcsQueueGiveMutex()

This function is somewhat simpler than npcsQueueTakeMutex() 3.1. In this function, we make sure that the caller of the function is actually the holder of the mutex they are attempting to give (as a sanity check). If the task calling this function holds the, then we call xQueueGenericSend(), allowing the mutex to be taken again. Next, we decrement the mutex counter. If the counter is 0 after the decrement, we restore the current task's original priority. All of the operations that happen following the sanity check are conducted within a critical section 2.5, thus we can avoid any issues that may occur if a task were to take the next mutex before the mutex counter is decremented. The use of critical sections in this function and in 3.1 ensure that this NPCS implementation is thread safe.

Chapter 4

Demo Application

In order to evaluate the NPCS implementation described in chapter 3, we created a demo application which exhibits deadlock as described in section 1.2. This application is a modified version of the FreeRTOS demo application originally distributed by Texas Instruments with their FreeRTOS port for the EK-TM4C123GXL. First, we created a model of the deadlock application using the UPPAAL model checking tool, verified that deadlock does in fact occur in that model, and then verified that NPCS would resolve the deadlock. We provide details of this model in section 4.1. In section 3.2, we describe the implementation of this demo application.

4.1 UPPAAL Model

We modeled a simple deadlock application using the UPPAAL model checker[17]. The model consists of four templates, which can be seen as the different diagrams in figure 4.1. On the left side, we have TaskA and TaskB. On the right, we have SemaphoreA and SemaphoreB. The scenario being modeled is this: TaskA needs to take SemaphoreA and then take SemaphoreB; TaskB needs to take SemaphoreB and then take SemaphoreA. Neither task

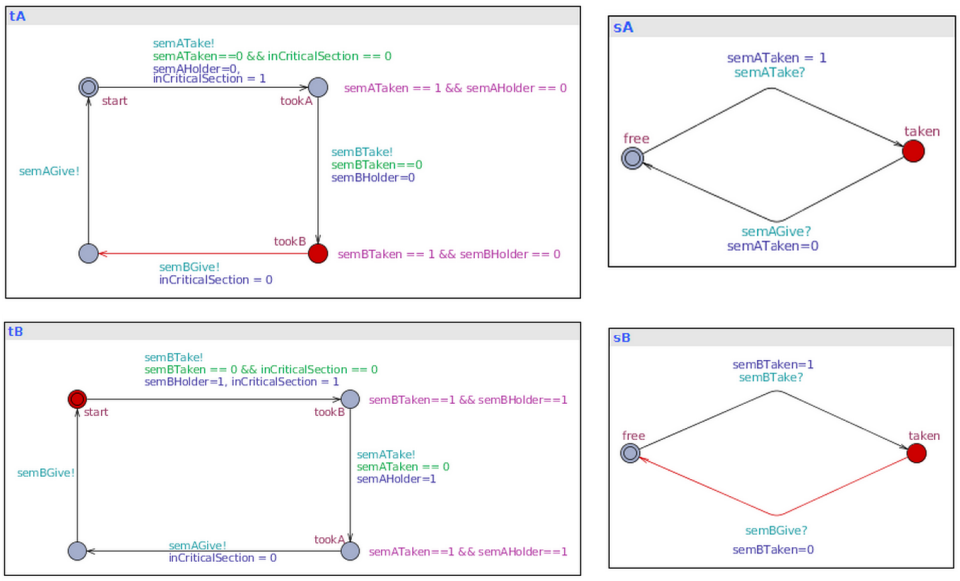

Figure 4.1: *UPPAAL model of deadlock demo. This version of the model has NPCS.*

will will give any semaphores back until it has taken both SemaphoreA and SemaphoreB. Thus, without further additions to this model, we may encounter a deadlock.

The tasks are modelled as having four states and four transitions. In each transition, tasks are taking or giving semaphores. Some of the states enforce invariants, but mostly serve in this model as a way of enforcing that a specific order of transitions are taken in order to model the given scenario.

Semaphores are modeled as having two states and two transitions. A semaphore in this case is either taken or free. Each semaphore begins in the free state

Two versions of the model exist: deadlock.xml and deadlock.q exhibit the deadlock problem, while npcsDeadlock.xml and npcsDeadlock.q demonstrate how use of non-preemptive critical sections solves the deadlock problem.

A non-preemptive critical section is modeled by adding an "inCriticalSection" boolean variable. Whenever a task takes the first of the two semaphores that it needs, it sets the variable to true, and once it has returned a semaphore it sets the variable back to false. The model does not allow a task to take its first mutex if inCriticalSection is true, because this

indicates that the other task is in the critical section.

4.2 Demo Implementation

As we mentioned at the beginning of this chapter, we created a demo application which is based on the FreeRTOS demo application made by Texas Instruments. In the original demo application, there are three LED tasks and a switch task. Each of the LED tasks causes the LED on the EK-TM4C123GXL board to flash either red, green or blue. Pressing the button in the lower left corner of the board toggles between LED tasks, causing the led to change colors. The button in the lower right corner changes the frequency at which the LED blinks for the selected task.

Our application only requires two tasks, so we removed one of the LED tasks. We designated one of these tasks as TaskA and the other as TaskB. We added a main task, which allows us to orchestrate the other tasks so that we may ensure that deadlock will occur. Within the main task, we have created two semaphores - SemaphoreA and SemaphoreB.

The left button's functionality is unchanged - when it is pressed, the active task will toggle between A and B, causing the light to blink either red or green. The right button however, causes a message to be sent to both TaskA and TaskB, causing them to grab mutexes in a way that causes deadlock. When TaskA receives the right button press message, it immediately takes SemaphoreA and then waits for ten seconds before trying to take SemaphoreB. TaskB sleeps for five seconds, takes SemaphoreB and then waits for ten seconds before attempting to take SemaphoreA.

Without non-preemptive critical sections, this will result in a deadlock. However, using the NPCS implementation described in chapter 3, deadlock does not occur. This is because as soon as TaskA takes SemaphoreA, it becomes impossible for TaskB to take a semaphore until TaskA has given back all semaphores that it holds.

Bibliography

[1] Texas Instruments. Tiva c series tm4c123g (ek-tm4c123gxl), 2014. URL http://www.ti.com/ww/en/launchpad/launchpads-connected-ek-tm4c123gxl.html.

[2] Jane W. S. Liu. *Real-Time Systems*, volume 1st Ed. New York: Prentice-Hall, 2000.

[3] FreeRTOS.org. Freertos - market leading rtos (real time operating system) for embedded systems with internet of things extensions, 2014. URL http://www.freertos.org.

[4] FreeRTOS.org. Freertos ports, 2014. URL http://www.freertos.org/a00090.html.

[5] FreeRTOS.org. Multitasking, 2014. URL http://www.freertos.org/implementation/a00004.html.

[6] FreeRTOS.org. Scheduling, 2014. URL http://www.freertos.org/implementation/a00005.html.

[7] FreeRTOS.org. Context switching, 2014. URL http://www.freertos.org/implementation/a00006.html.

[8] FreeRTOS.org. The rtos tick, 2014. URL http://www.freertos.org/implementation/a00011.html.

[9] FreeRTOS.org. Gcc signal attribute, 2014. URL http://www.freertos.org/implementation/a00012.html.

[10] FreeRTOS.org. Kernel control, 2014. URL http://www.freertos.org/a00020.html#taskENTER_CRITICAL.

[11] FreeRTOS.org. Intertask communcations, 2014. URL http://www.freertos.org/message_passing_performance.html.

[12] FreeRTOS.org. Queues, 2014. URL http://www.freertos.org/Embedded-RTOS-Queues.html.

[13] FreeRTOS.org. Freertos binary semaphores, 2014. URL http://www.freertos.org/Embedded-RTOS-Binary-Semaphores.html.

[14] FreeRTOS.org. Freertos counting semaphores, 2014. URL http://www.freertos.org/Real-time-embedded-RTOS-Counting-Semaphores.html.

[15] FreeRTOS.org. Freertos mutexes, 2014. URL http://www.freertos.org/Real-time-embedded-RTOS-mutexes.html.

[16] FreeRTOS.org. Semaphores, 2014. URL http://www.freertos.org/a00113.html.

[17] Uppsala University and Aalborg University. Uppaal, 2014. URL http://www.uppaal.org.