Real-Time Embedded Multithreading: Using ThreadX® and ARM®

Edward L. Lamie

CMP*Books*
San Francisco, CA

CHAPTER 3

RTOS CONCEPTS AND DEFINITIONS

3.1 Introduction

The purpose of this chapter is to review some of the essential concepts and definitions used in embedded systems.[1] You have already encountered several of these terms in previous chapters, and you will read about several new concepts here.

3.2 Priorities

Most embedded real-time systems use a priority system as a means of establishing the relative importance of threads in the system. There are two classes of priorities: static and dynamic. A *static priority* is one that is assigned when a thread is created and remains constant throughout execution. A *dynamic priority* is one that is assigned when a thread is created, but can be changed at any time during execution. Furthermore, there is no limit on the number of priority changes that can occur.

ThreadX provides a flexible method of dynamic priority assignment. Although each thread must have a priority, ThreadX places no restrictions on how priorities may be used. As an extreme case, all threads could be assigned the same priority that would never change. However, in most cases, priority values are carefully assigned and modified only to reflect the change of importance in the processing of threads. As illustrated by Figure 3.1, ThreadX provides priority values from 0 to 31, inclusive, where the value 0 represents the highest priority and the value 31 represents the lowest priority.

[1] A relatively small number of terms and concepts are reviewed in this chapter. For a more complete listing, see the online Embedded Systems Glossary by Michael Barr at http://www.netrino.com/Publications/Glossary/.

Figure 3.1 Priority values.

Priority Value	Meaning
0	Highest priority
1	
:	
31	Lowest priority

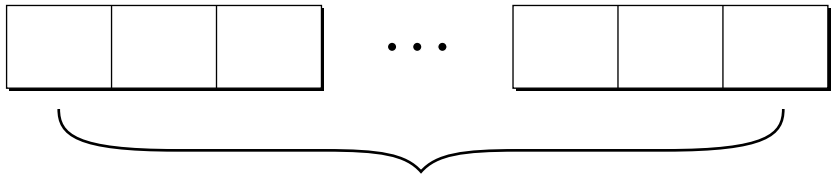
3.3 Ready Threads and Suspended Threads

ThreadX maintains several internal data structures to manage threads in their various states of execution. Among these data structures are the Suspended Thread List and the Ready Thread List. As implied by the nomenclature, threads on the Suspended Thread List have been *suspended*—temporarily stopped executing—for some reason. Threads on the Ready Thread List are not currently executing but are ready to run.

When a thread is placed in the Suspended Thread List, it is because of some event or circumstance, such as being forced to wait for an unavailable resource. Such a thread remains in that list until that event or circumstance has been resolved. When a thread is removed from the Suspended Thread List, one of two possible actions occurs: it is placed on the Ready Thread List, or it is terminated.

When a thread is ready for execution, it is placed on the Ready Thread List. When ThreadX schedules a thread for execution, it selects and removes the thread in that list that has the highest priority. If all the threads on the list have equal priority, ThreadX selects the thread that has been waiting the longest.[2] Figure 3.2 contains an illustration of how the Ready Thread List appears.

Figure 3.2 Ready Thread List.

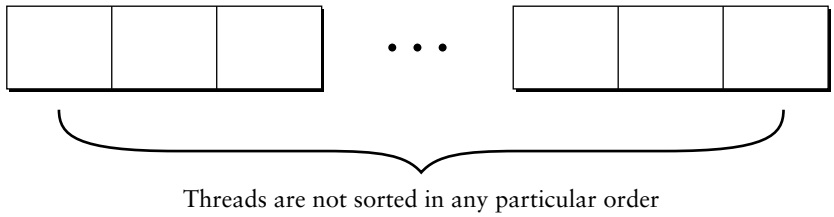


If for any reason a thread is not ready for execution, it is placed in the Suspended Thread List. For example, if a thread is waiting for a resource, if it is in "sleep" mode, if it was created with a TX_DONT_START option, or if it was explicitly suspended, then

[2] This latter selection algorithm is commonly known as First In First Out, or FIFO.

it will reside in the Suspended Thread List until that situation has cleared. Figure 3.3 contains a depiction of this list.

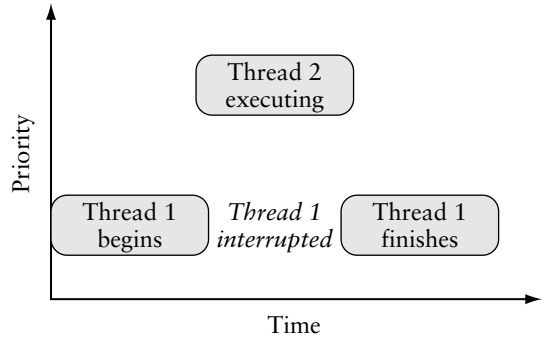
Figure 3.3 Suspended Thread List.



3.4 Preemptive, Priority-Based Scheduling

The term *preemptive, priority-based scheduling* refers to the type of scheduling in which a higher priority thread can interrupt and suspend a currently executing thread that has a lower priority. Figure 3.4 contains an example of how this scheduling might occur.

Figure 3.4 Thread preemption.



In this example, Thread 1 has control of the processor. However, Thread 2 has a higher priority and becomes ready for execution. ThreadX then interrupts Thread 1 and gives Thread 2 control of the processor. When Thread 2 completes its work, ThreadX returns control to Thread 1 at the point where it was interrupted. The developer does not have to be concerned about the details of the scheduling process. Thus, the developer is able to develop the threads in isolation from one another because the scheduler determines when to execute (or interrupt) each thread.

3.5 Round-Robin Scheduling

The term *round-robin scheduling* refers to a scheduling algorithm designed to provide processor sharing in the case in which multiple threads have the same priority. There are two primary ways to achieve this purpose, both of which are supported by ThreadX.

Figure 3.5 illustrates the first method of round-robin scheduling, in which Thread 1 is executed for a specified period of time, then Thread 2, then Thread 3, and so on to Thread n, after which the process repeats. See the section titled *Time-Slice* for more information about this method. The second method of round-robin scheduling is achieved by the use of a cooperative call made by the currently executing thread that temporarily relinquishes control of the processor, thus permitting the execution of other threads of the same or higher priority. This second method is sometimes called *cooperative multithreading*. Figure 3.6 illustrates this second method of round-robin scheduling.

Figure 3.5 Round-robin processing.

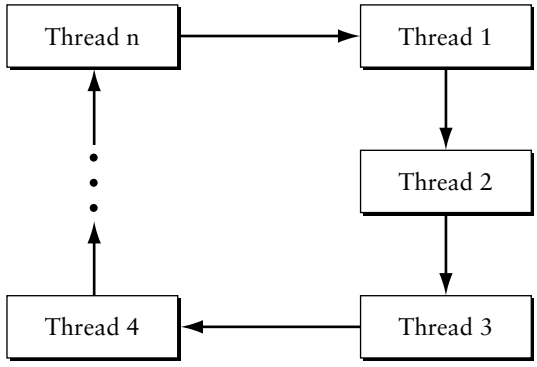
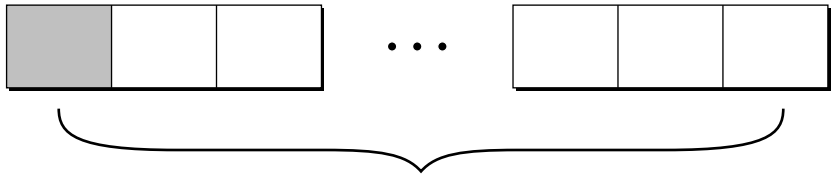


Figure 3.6 Example of cooperative multithreading.



Ready Thread List containing threads with the same priority. Currently executing thread (shaded) voluntarily relinquishes the processor and is placed on this list.

With cooperative multithreading, when an executing thread relinquishes control of the processor, it is placed at the end of the Ready Thread List, as indicated by the shaded thread in the figure. The thread at the front of the list is then executed, followed by the next thread on the list, and so on until the shaded thread is at the front of the list. For convenience, Figure 3.6 shows only ready threads with the same priority. However, the Ready Thread List can hold threads with several different priorities. In that case, the scheduler will restrict its attention to the threads that have the highest priority.

In summary, the cooperative multithreading feature permits the currently executing thread to voluntarily give up control of the processor. That thread is then placed on the

Ready Thread List and it will not gain access to the processor until after all other threads that have the same (or higher) priority have been processed.

3.6 Determinism

As noted in Chapter 1, an important feature of real-time embedded systems is the concept of determinism. The traditional definition of this term is based on the assumption that for each system state and each set of inputs, a unique set of outputs and next state of the system can be determined. However, we strengthen the definition of determinism for real-time embedded systems by requiring that the time necessary to process any task is predictable. In particular, we are less concerned with average response time than we are with worst-case response time. For example, we must be able to guarantee the worst-case response time for each system call in order for a real-time embedded system to be deterministic. In other words, simply obtaining the correct answer is not adequate. We must get the right answer within a specified time frame.

Many RTOS vendors claim their systems are deterministic and justify that assertion by publishing tables of minimum, average, and maximum number of clock cycles required for each system call. Thus, for a given application in a deterministic system, it is possible to calculate the timing for a given number of threads, and determine whether real-time performance is actually possible for that application.

3.7 Kernel

A *kernel* is a minimal implementation of an RTOS. It normally consists of at least a scheduler and a context switch handler. Most modern commercial RTOSes are actually kernels, rather than full-blown operating systems.

3.8 RTOS

An RTOS is an operating system that is dedicated to the control of hardware, and must operate within specified time constraints. Most RTOSes are used in embedded systems.

3.9 Context Switch

A *context* is the current execution state of a thread. Typically, it consists of such items as the program counter, registers, and stack pointer. The term *context switch* refers to the saving of one thread's context and restoring a different thread's context so that it can be executed. This normally occurs as a result of preemption, interrupt handling, time-slicing (see below), cooperative round-robin scheduling (see below), or suspension of a thread because it needs an unavailable resource. When a thread's context is restored, then the thread resumes execution at the point where it was stopped. The kernel performs the context switch operation. The actual code required to perform context switches is necessarily processor-specific.

3.10 Time-Slice

The length of time (i.e., number of timer-ticks) for which a thread executes before relinquishing the processor is called its *time-slice*. When a thread's (optional) time-slice expires in ThreadX, all other threads of the same or higher priority levels are given a chance to execute before the time-sliced thread executes again. Time-slicing provides another form of round-robin scheduling. ThreadX provides optional time-slicing on a per-thread basis. The thread's time-slice is assigned during creation and can be modified during execution. If the time-slice is too short, then the scheduler will waste too much processing time performing context switches. However, if the time-slice is too long then threads might not receive the attention they need.

3.11 Interrupt Handling

An essential requirement of real-time embedded applications is the ability to provide fast responses to asynchronous events, such as hardware or software interrupts. When an interrupt occurs, the context of the executing thread is saved and control is transferred to the appropriate interrupt vector. An *interrupt vector* is an address for an *interrupt service routine (ISR),* which is user-written software designed to handle or service the needs of a particular interrupt. There may be many ISRs, depending on the number of interrupts that needs to be handled. The actual code required to service interrupts is necessarily processor-specific.

3.12 Thread Starvation

One danger of preemptive, priority-based scheduling is *thread starvation*. This is a situation in which threads that have lower priorities rarely get to execute because the processor spends most of its time on higher-priority threads. One method to alleviate this problem is to make certain that higher-priority threads do not monopolize the processor. Another solution would be to gradually raise the priority of starved threads so that they do get an opportunity to execute.

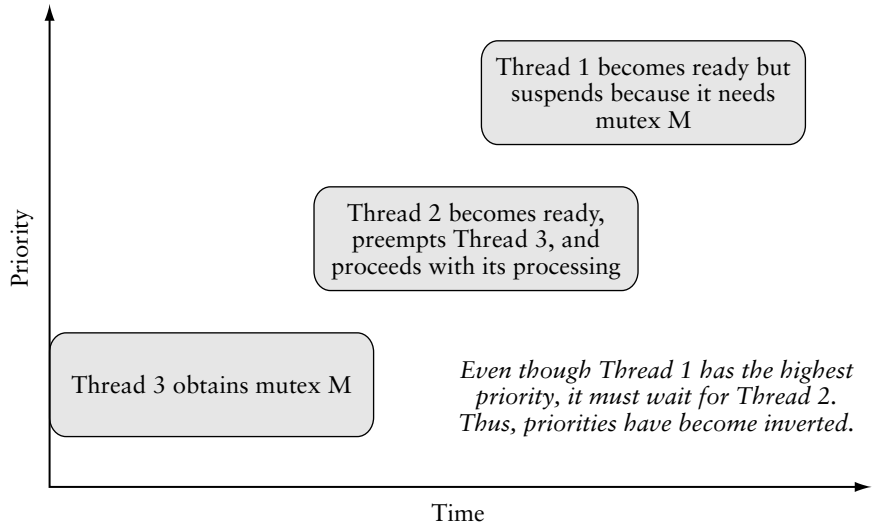
3.13 Priority Inversion

Undesirable situations can occur when two threads with different priorities share a common resource. *Priority inversion* is one such situation; it arises when a higher-priority thread is suspended because a lower-priority thread has acquired a resource needed by the higher-priority thread. The problem is compounded when the shared resource is not in use while the higher-priority thread is waiting. This phenomenon may cause priority inversion time to become nondeterministic and lead to application failure. Consider Figure 3.7, which shows an example of the priority inversion problem.

In this example, Thread 3 (with the lowest priority) becomes ready. It obtains mutex M and begins its execution. Some time later, Thread 2 (which has a higher priority) becomes ready, preempts Thread 3, and begins its execution. Then Thread 1 (which has the highest priority of all) becomes ready. However, it needs mutex M, which is owned

by Thread 3, so it is suspended until mutex M becomes available. Thus, the higher-priority thread (i.e., Thread 1) must wait for the lower-priority thread (i.e., Thread 2) before it can continue. During this wait, the resource protected by mutex M is not being used because Thread 3 has been preempted by Thread 2. The concept of priority inversion is discussed more thoroughly in a later chapter.

Figure 3.7 Example of priority inversion.



3.14 Priority Inheritance

Priority inheritance is an optional feature that is available with ThreadX for use only with the mutex services. (Mutexes are discussed in more detail in the next chapter.) Priority inheritance allows a lower-priority thread to temporarily assume the priority of a higher-priority thread that is waiting for a mutex owned by the lower-priority thread. This capability helps the application to avoid nondeterministic priority inversion by eliminating preemption of intermediate thread priorities. This concept is discussed more thoroughly in a later chapter.

3.15 Preemption-Threshold

Preemption-threshold[3] is a feature that is unique to ThreadX. When a thread is created, the developer has the option of specifying a priority ceiling for disabling preemption.

[3] Preemption-threshold is a trademark of Express Logic, Inc. There are several university research papers that analyze the use of preemption-threshold in real-time scheduling algorithms. A complete list of URLs for these papers can be found at http://www.expresslogic.com/research.html.

This means that threads with priorities greater than the specified ceiling are still allowed to preempt, but those with priorities equal to or less than the ceiling are not allowed to preempt that thread. The preemption-threshold value may be modified at any time during thread execution. Consider Figure 3.8, which illustrates the impact of preemption-threshold. In this example, a thread is created and is assigned a priority value of 20 and a preemption-threshold of 15. Thus, only threads with priorities higher than 15 (i.e., 0 through 14) will be permitted to preempt this thread. Even though priorities 15 through 19 are higher than the thread's priority of 20, threads with those priorities will not be allowed to preempt this thread. This concept is discussed more thoroughly in a later chapter.

Figure 3.8 Example of preemption-threshold.

Priority	Comment
0 : 14	Preemption allowed for threads with priorities from 0 to 14 (inclusive).
15 : 19	Thread is assigned preemption-threshold = 15 [This has the effect of disabling preemption for threads with priority values from 15 to 19 (inclusive).]
20 : 31	Thread is assigned Priority = 20.

3.16 Key Terms and Phrases

asynchronous event
context switch
cooperative multithreading
determinism
interrupt handling
kernel
preemption
preemption-threshold
priority
priority inheritance
priority inversion
ready thread
Ready Thread List
round-robin scheduling
RTOS
scheduling
sleep mode
suspended thread
Suspended Thread List
thread starvation
time-slice
timer-tick

3.17 Problems

1. When a thread is removed from the Suspended Thread List, either it is placed on the Ready Thread List or it is terminated. Explain why there is not an option for that thread to become the currently executing thread immediately after leaving the Suspended Thread List.
2. Suppose every thread is assigned the same priority. What impact would this have on the scheduling of threads? What impact would there be if every thread had the same priority and was assigned the same duration time-slice?
3. Explain how it might be possible for a preempted thread to preempt its preemptor? Hint: Think about priority inheritance.
4. Discuss the impact of assigning every thread a preemption-threshold value of 0 (the highest priority).