THREADX®

the high-performance embedded kernel

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

http://www.expresslogic.com

Part Number: 000-1001
Revision 4.0c

Contents

Figures

About This Guide

This guide provides comprehensive information about ThreadX, the high-performance real-time kernel from Express Logic, Inc.

It is intended for the embedded real-time software developer. The developer should be familiar with standard real-time operating system functions and the C programming language.

Organization

Chapter 1	Provides a basic overview of ThreadX and its relationship to real-time embedded development.
Chapter 2	Gives the basic steps to install and use ThreadX in your application right *out of the box*.
Chapter 3	Describes in detail the functional operation of ThreadX, the high-performance real-time kernel.
Chapter 4	Details the application's interface to ThreadX.
Chapter 5	Describes writing I/O drivers for ThreadX applications.
Chapter 6	Describes the demonstration application that is supplied with every ThreadX processor support package.

Chapter 7	Details the internal construction of ThreadX.
Appendix A	ThreadX API
Appendix B	ThreadX constants
Appendix C	ThreadX data types
Appendix D	ThreadX source files
Appendix E	ASCII chart
Index	Topic cross reference

Guide Conventions

Italics	typeface denotes book titles, emphasizes important words, and indicates variables.
Boldface	typeface denotes file names, key words, and further emphasizes important words and variables.
i	Information symbols draw attention to important or additional information that could affect performance or function.
!	Warning symbols draw attention to situations in which developers should take care to avoid because they could cause fatal errors.

ThreadX Data Types

In addition to the custom ThreadX control structure data types, there are a series of special data types that are used in ThreadX service call interfaces. These special data types map directly to data types of the underlying C compiler. This is done to insure portability between different C compilers. The exact implementation can be found in the ***tx_port.h*** file included on the distribution disk.

The following is a list of ThreadX service call data types and their associated meanings:

UINT	Basic unsigned integer. This type must support 8-bit unsigned data; however, it is mapped to the most convenient unsigned data type, which may support 16- or 32-bit signed data.
ULONG	Unsigned long type. This type must support 32-bit unsigned data.
VOID	Almost always equivalent to the compiler's void type.
CHAR	Most often a standard 8-bit character type.

Additional data types are used within the ThreadX source. They are also located in the ***tx_port.h*** file.

Customer Support Center

Support engineers	858.613.6640
Support fax	858.521.4259
Support email	support@expresslogic.com
Web page	http://www.expresslogic.com

Where to Send Comments

The staff at Express Logic is always striving to provide you with better products. To help us achieve this goal, email any comments and suggestions to the Customer Support Center at

comments@expresslogic.com

Please type "technical publication" in the subject line.

CHAPTER 1

Introduction to ThreadX

ThreadX is a high-performance real-time kernel designed specifically for embedded applications. This chapter contains an introduction to the product and a description of its applications and benefits.

ThreadX Unique Features

Unlike other real-time kernels, ThreadX is designed to be versatile—easily scaling among small micro-controller-based applications through those that use powerful RISC and DSP processors.

What makes ThreadX so scalable? The reason is based on its underlying architecture. Because ThreadX services are implemented as a C library, only those services actually used by the application are brought into the run-time image. Hence, the actual size of ThreadX is completely determined by the application. For most applications, the instruction image of ThreadX ranges between 2 KBytes and 15 KBytes in size.

picokernel™ Architecture

What about performance? Instead of layering kernel functions on top of each other like traditional *microkernel* architectures, ThreadX services plug directly into its core. This results in the fastest possible context switching and service call performance. We call this non-layering design a *picokernel* architecture.

ANSI C Source Code

ThreadX is written primarily in ANSI C. A small amount of assembly language is needed to tailor the kernel to the underlying target processor. This design makes it possible to port ThreadX to a new processor family in a very short time—usually within weeks!

Not A Black Box

Most distributions of ThreadX include the complete C source code as well as the processor-specific assembly language. This eliminates the “black-box” problems that occur with many commercial kernels. By using ThreadX, application developers can see exactly what the kernel is doing—there are no mysteries!

The source code also allows for application specific modifications. Although not recommended, it is certainly beneficial to have the ability to modify the kernel if it is absolutely required.

These features are especially comforting to developers accustomed to working with their own *in-house kernels*. They expect to have source code and the ability to modify the kernel. ThreadX is the ultimate kernel for such developers.

A Potential Standard

Because of its versatility, high-performance *picokernel* architecture, and great portability, ThreadX has the potential to become an industry standard for embedded applications.

Embedded Applications

What is an embedded application? Embedded applications are applications that execute on microprocessors buried inside of products like cellular phones, communication equipment, automobile engines, laser printers, medical devices, etc. Another distinction of embedded applications is that their software and hardware have a dedicated purpose.

Real-time Software

When time constraints are imposed on the application software, it is given the *real-time* label. Basically, software that must perform its processing within an exact period of time is called *real-time* software. Embedded applications are almost always real-time because of their inherent interaction with the external world.

Multitasking

As mentioned, embedded applications have a dedicated purpose. In order to fulfill this purpose, the software must perform a variety of duties or *tasks*. A task is a semi-independent portion of the application that carries out a specific duty. It is also the case that some tasks or duties are more important than others. One of the major difficulties in an embedded application is the allocation of the processor between the various application tasks. This allocation of processing between competing tasks is the primary purpose of ThreadX.

Tasks vs. Threads

Another distinction about tasks must be made. The term task is used in a variety of ways. It sometimes means a separately loadable program. In other instances, it might refer to an internal program segment.

In contemporary operating system discussion, there are two terms that more or less replace the use of task, namely *process* and *thread*. A *process* is a completely independent program that has its own address space, while a thread is a semi-independent program segment that executes within a process. Threads share the same process address space. The overhead associated with thread management is minimal.

Most embedded applications cannot afford the overhead (both memory and performance) associated with a full-blown process-oriented operating system. In addition, smaller microprocessors don’t have the hardware architecture to support a true process-oriented operating system. For these reasons, ThreadX implements a thread model, which is both extremely efficient and practical for most real-time embedded applications.

To avoid confusion, ThreadX does not use the term *task*. Instead, the more descriptive and contemporary name *thread* is used.

ThreadX Benefits

Using ThreadX provides many benefits to embedded applications. Of course, the primary benefit rests in how embedded application threads are allocated processing time.

Improved Responsiveness

Prior to real-time kernels like ThreadX, most embedded applications allocated processing time with a simple control loop, usually from within the C *main* function. This approach is still used in very small or simple applications. However, in large or complex applications it is not practical because the response time to any event is a function of the worst-case processing time of one pass through the control loop.

Making matters worse, the timing characteristics of the application change whenever modifications are made to the control loop. This makes the application inherently unstable and very difficult to maintain and improve on.

ThreadX provides fast and deterministic response times to important external events. ThreadX accomplishes this through its preemptive, priority-based scheduling algorithm, which allows a higher-priority thread to preempt an executing lower-priority thread. As a result, the worst-case response time approaches the time required to perform a context switch. This is not only deterministic, but it is also extremely fast.

Software Maintenance

The ThreadX kernel enables application developers to concentrate on specific requirements of their application threads without having to worry about changing the timing of other areas of the application. This feature also makes it much easier to repair or enhance an application that utilizes ThreadX.

Increased Throughput

A possible work-around to the control loop response time problem is to add more polling. This improves the responsiveness, but still doesn't guarantee a constant worst-case response time and does nothing to enhance future modification of the application. Also, the processor is now performing even more unnecessary processing because of the extra polling. All of this unnecessary processing reduces the overall throughput of the system.

An interesting point regarding overhead is that many developers assume that multi-threaded environments like ThreadX increase overhead and have a negative impact on total system throughput. But in some cases, multi-threading actually reduces overhead by eliminating all of the redundant polling that occurs in control loop environments. The overhead associated with multi-threaded kernels is typically a function of the time required for context switching. If the context switch time is less than the polling process, ThreadX provides a solution with the potential of less overhead and more throughput. This makes ThreadX an obvious choice for applications that have any degree of complexity or size.

Processor Isolation

ThreadX provides a robust processor-independent interface between the application and the underlying processor. This allows developers to concentrate on the application rather than spending a significant amount of time learning hardware details.

Dividing the Application

In control loop-based applications, each developer must have an intimate knowledge of the entire application's run-time behavior and requirements. This is because the processor allocation logic is dispersed throughout the entire application. As an application increases in size or complexity, it becomes impossible for all developers to remember the precise processing requirements of the entire application.

ThreadX frees each developer from the worries associated with processor allocation and allows them to concentrate on their specific piece of the embedded application. In addition, ThreadX forces the application to be divided into clearly defined threads. By itself, this division of the application into threads makes development much simpler.

Ease of Use

ThreadX is designed with the application developer in mind. The ThreadX architecture and service call interface are designed to be easily understood. As a result, ThreadX developers can quickly use its advanced features.

Improve Time-to-market

All of the benefits of ThreadX accelerate the software development process. ThreadX takes care of most processor issues, thereby removing this effort from the development schedule. All of this results in a faster time to market!

Protecting the Software Investment

Because of its architecture, ThreadX is easily ported to new processor environments. This, coupled with the fact ThreadX insulates applications from details of the underlying processors, makes ThreadX applications highly portable. As a result, the application's migration path is guaranteed and the original development investment is protected.

CHAPTER 2

Installation and Use of ThreadX

This chapter contains a description of various issues related to installation, setup, and usage of the high-performance ThreadX kernel.

Host Considerations

Embedded development is usually performed on IBM-PC or Unix host computers. After the application is compiled, linked, and located on the host, it is downloaded to the target hardware for execution.

Usually the target download is done over an RS-232 serial interface; however, parallel interfaces and ethernet are becoming more popular. See the development tool documentation for available options.

Debugging is typically done over the same link as the program image download. A variety of debuggers exist, ranging from small monitor programs running on the target through Background Debug Monitor (BDM) and In-Circuit Emulator (ICE) tools. Of course, the ICE tools provide the most robust debugging of actual target hardware.

As for resources used on the host, the source code for ThreadX is delivered in ASCII format and requires approximately 1 MBytes of space on the host computer's hard disk.

*Please review the supplied **readme.txt** file for additional host system considerations and options.*

Target Considerations

ThreadX requires between 2 KBytes and 20 KBytes of Read Only Memory (ROM) on the target. Another 1 to 2 KBytes of the target's Random Access Memory (RAM) are required for the ThreadX system stack and other global data structures.

For timer-related functions like service call time-outs, time-slicing, and application timers to function, the underlying target hardware must provide a periodic interrupt source. If the processor has this capability, it is utilized by ThreadX. Otherwise, if the target processor does not have the ability to generate a periodic interrupt, the user's hardware must provide it. Setup and configuration of the timer interrupt is typically located in the ***tx_ill*** assembly file in the ThreadX distribution.

*ThreadX is still functional even if no periodic timer interrupt source is available. However, none of the timer-related services are functional. Please review the supplied **readme.txt** file for any additional host system considerations and/or options.*

Product Distribution

ThreadX is shipped on a single CD-ROM compatible disk. Two types of ThreadX packages are available—*standard* and *premium*. The *standard* package includes minimal source code; while the *premium* package contains complete ThreadX source code.

The exact contents of the distribution disk depends on the target processor, development tools, and the ThreadX package purchased. However, the following is a list of several important files that are common to most product distributions:

readme.txt	This file contains specific information about the ThreadX port, including information about the target processor and the development tools.
tx_api.h	This C header file contains all system equates, data structures, and service prototypes.

tx_port.h	This C header file contains all development tool specific data definitions and structures.
demo.c	This C file contains a small demo application.
build_ap.bat	This file is an MS-DOS batch file that tells how to build the ThreadX demonstration.
build_tx.bat	This file is an MS-DOS batch file that tells how to build the ThreadX C library. It is distributed with the *premium* package
tx.lib	This is the binary version of the ThreadX C library. It is distributed with the *standard* package.

i *All files and batch file commands are in lower-case. This naming convention makes it easier to convert the commands to Unix development platforms.*

ThreadX Installation

Installation of ThreadX is straightforward. The following instructions apply to virtually any installation. However, examine the ***readme.txt*** file for changes specific to the actual development tool environment.

Step 1: Backup the ThreadX distribution disk and store it in a safe location.

Step 2: On the host hard drive, make a directory called “threadx” or something similar. The ThreadX kernel files will reside in this directory.

Step 3: Copy all files from the ThreadX distribution CD-ROM into the directory created in step 2.

Step 4: If the standard package was purchased, installation of ThreadX is now complete.

If the premium package was purchased, execute the ***build_tx.bat*** batch file to build the ThreadX run-time library.

i *Application software needs access to the ThreadX library file (usually called* **tx.lib***) and the C include files* **tx_api.h** *and* **tx_port.h***. This is accomplished either by setting the appropriate path for the development tools or by copying these files into the application development area.*

Using ThreadX

Using ThreadX is easy. Basically, the application code must include ***tx_api.h*** during compilation and link with the ThreadX run-time library ***tx.lib***.

There are four steps required to build a ThreadX application:

Step 1: Include the ***tx_api.h*** file in all application files that use ThreadX services or data structures.

Step 2: Create the standard C ***main*** function. This function must eventually call ***tx_kernel_enter*** to start ThreadX. Application-specific initialization that does not involve ThreadX may be added prior to entering the kernel.

i *The ThreadX entry function* **tx_kernel_enter** *does not return. So be sure not to place any processing or function calls after it.*

Step 3: Create the ***tx_application_define*** function. This is where the initial system resources are created. Examples of system resources include threads, queues, memory pools, event flag groups, mutexes, and semaphores.

Step 4: Compile application source and link with the ThreadX run-time library ***tx.lib***. The resulting image can be downloaded to the target and executed!

Small Example System

The small example system in Figure 1 on page 35 shows the creation of a single thread with a priority of 3. The thread executes, increments a counter, then sleeps for one clock tick. This process continues forever.

```
#include            "tx_api.h"

unsigned long       my_thread_counter = 0;
TX_THREAD           my_thread;

main( )
{
      /* Enter the ThreadX kernel.  */
      tx_kernel_enter( );
}

void  tx_application_define(void *first_unused_memory)
{

      /* Create my_thread!  */
      tx_thread_create(&my_thread, "My Thread",
          my_thread_entry, 0x1234, first_unused_memory, 1024,
             3, 3, TX_NO_TIME_SLICE, TX_AUTO_START);
}

void   my_thread_entry(ULONG  thread_input)
{
      /* Enter into a forever loop.  */
      while(1)
      {

             /* Increment thread counter.  */
             my_thread_counter++;

             /* Sleep for 1 tick.  */
             tx_thread_sleep(1);
      }
}
```

FIGURE 1. Template for Application Development

Although this is a simple example, it provides a good template for real application development. Once again, please see the ***readme.txt*** file for additional details.

Troubleshooting

Each ThreadX port is delivered with a demonstration application. It is always a good idea to first get the demonstration system running—either on actual target hardware or the specific demonstration environment.

Build the demonstration by executing the ***build_ap.bat*** batch file. This file builds an application image (with ThreadX included) that is ready for download to the target.

*See the **readme.txt** file supplied with the distribution for more specific details regarding the demonstration system.*

If the demonstration system does not work, try the following things to narrow the problem:

1. Determine how much of the demonstration is running.
2. Increase stack sizes (this is more important in actual application code than it is for the demonstration).
3. Disable the timer interrupt and all others that might cause a problem.

Configuration Options

There is really only one generic configuration option for ThreadX, and it is used to bypass service call error checking. If the condition compilation flag **TX_DISABLE_ERROR_CHECKING** is defined within an application C file, all basic parameter error checking is disabled. This option is used to improve performance (by as much as 30%). However, this should be done only after the application is thoroughly debugged.

*ThreadX API return values **NOT** affected by disabling error checking are listed in **bold** in the “Return Values” section of the API description in Chapter 4. The non-bold return values are void if error checking is disabled by the **TX_DISABLE_ERROR_CHECKING** option.*

Additional development tool options are described in the ***readme.txt*** supplied on the distribution disk.

ThreadX Version ID

The current version of ThreadX is available to both the user and the application software during run-time. The programmer can find the ThreadX version in the ***readme.txt*** file. This file also contains a version history of the corresponding port. Application software can obtain the ThreadX version by examining the global string ***_tx_version_id***.

CHAPTER 3

Functional Components of ThreadX

This chapter contains a description of the high-performance ThreadX kernel from a functional perspective. Each functional component is presented in an easy-to-understand manner.

Execution Overview

There are four types of program execution within a ThreadX application: Initialization, Thread Execution, Interrupt Service Routines (ISRs), and Application Timers.

Figure 2 on page 43 shows each different type of program execution. More detailed information about each of these types is found in subsequent sections of this chapter.

Initialization

As the name implies, this is the first type of program execution in a ThreadX application. Initialization includes all program execution between processor reset and the entry point of the *thread scheduling loop.*

Thread Execution

After initialization is complete, ThreadX enters its thread scheduling loop. The scheduling loop looks for an application thread ready for execution. When a ready thread is found, ThreadX transfers control to it. Once the thread is finished (or another higher-priority thread becomes ready), execution transfers back to the thread scheduling loop in order to find the next highest priority ready thread.

This process of continually executing and scheduling threads is the most common type of program execution in ThreadX applications.

Interrupt Service Routines (ISR)

Interrupts are the cornerstone of real-time systems. Without interrupts it would be extremely difficult to respond to changes in the external world in a timely manner. What happens when an interrupt occurs? Upon detection of an interrupt, the processor saves key information about the current program execution

Execution Overview

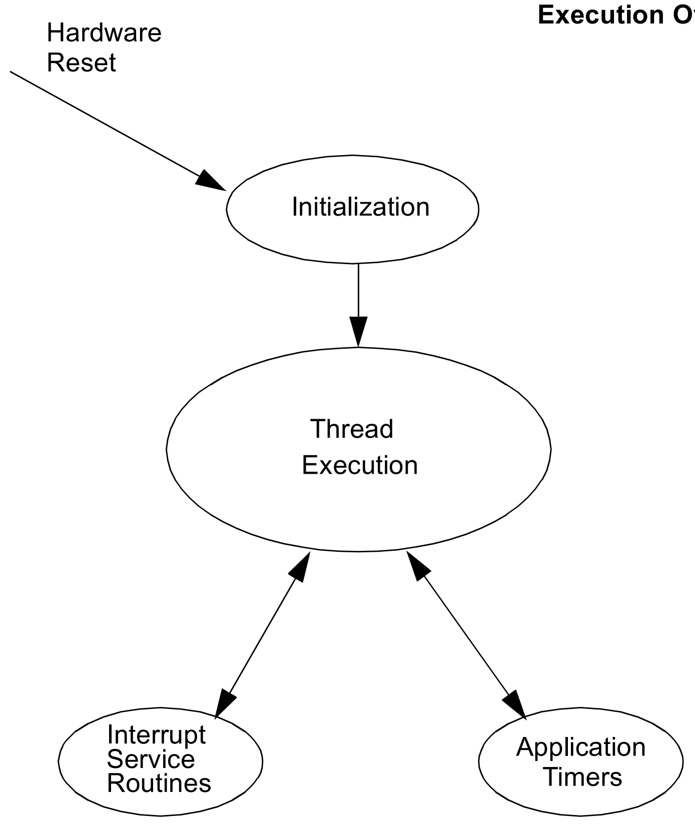

FIGURE 2. Types of Program Execution

(usually on the stack), then transfers control to a predefined program area. This predefined program area is commonly called an Interrupt Service Routine.

What type of program execution was interrupted? In most cases, interrupts occur during thread execution (or in the thread scheduling loop). However, interrupts may also occur inside of an executing ISR or an Application Timer.

Application Timers

Application timers are very similar to ISRs, except the actual hardware implementation (usually a single periodic hardware interrupt is used) is hidden from the application. Such timers are used by applications to perform time-outs, periodics, and/or watchdog services. Just like ISRs, application timers most often interrupt thread execution. Unlike ISRs, however, Application Timers cannot interrupt each other.

Memory Usage

ThreadX resides along with the application program. As a result, the static memory (or fixed memory) usage of ThreadX is determined by the development tools; e.g., the compiler, linker, and locator. Dynamic memory (or run-time memory) usage is under direct control of the application.

Static Memory Usage

Most of the development tools divide the application program image into five basic areas: *instruction*, *constant*, *initialized data*, *uninitialized data*, and *system stack*. Figure 3 on page 45 shows an example of these memory areas.

It is important to realize that this is only an example. The actual static memory layout is specific to the processor, development tools, and the underlying hardware.

The instruction area contains all of the program’s processor instructions. This area is typically the largest and is often located in ROM.

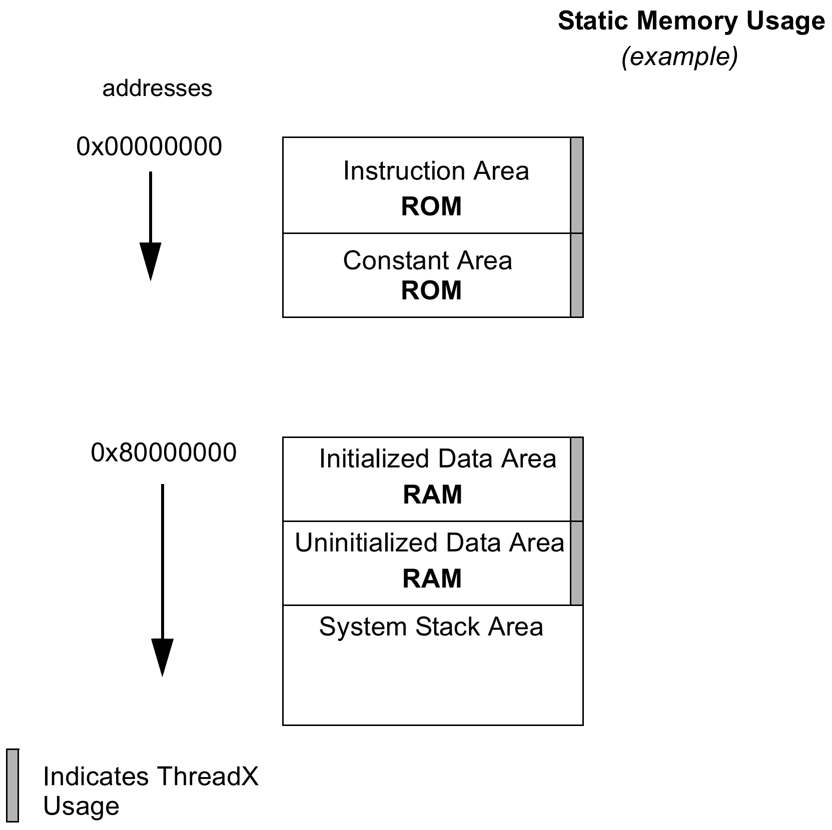

FIGURE 3. Memory Area Example

The constant area contains various compiled constants, including strings defined or referenced within the program. In addition, this area contains the “initial copy” of the initialized data area. During the compiler’s initialization process, this portion of the constant area is used to setup the initialized data area in RAM. The constant area usually follows the instruction area and is often located in ROM.

The initialized data and uninitialized data areas contain all of the global and static variables. These areas are always located in RAM.

The system stack is generally setup immediately following the initialized and uninitialized data areas. The system stack is used by the compiler during initialization and then by ThreadX during initialization and subsequently in ISR processing.

Dynamic Memory Usage

As mentioned before, dynamic memory usage is under direct control of the application. Control blocks and memory areas associated with stacks, queues, and memory pools can be placed anywhere in the target's memory space. This is an important feature because it facilitates easy utilization of different types of physical memory.

For example, suppose a target hardware environment has both fast memory and slow memory. If the application needs extra performance for a high-priority thread, its control block (TX_THREAD) and stack can be placed in the fast memory area, which might greatly enhance its performance.

Initialization

Understanding the initialization process is very important. The initial hardware environment is setup here. In addition, this is where the application is given its initial personality.

ThreadX attempts to utilize (whenever possible) the complete development tool's initialization process. This makes it easier to upgrade to new versions of the development tools in the future.

System Reset

All microprocessors have reset logic. When a reset occurs (either hardware or software), the address of the application's entry point is retrieved from a specific memory location. After the entry point is retrieved, the processor transfers control to that location.

The application entry point is quite often written in the native assembly language and is usually supplied by the development tools (at least in template form). In some cases, a special version of the entry program is supplied with ThreadX.

Development Tool Initialization

After the low-level initialization is complete, control transfers to the development tool's high-level initialization. This is usually the place where initialized global and static C variables are setup. Remember that their initial values are retrieved from the constant area. Exact initialization processing is development tool specific.

main

When the development tool initialization is complete, control transfers to the user-supplied *main* function. At this point, the application controls what happens next. For most applications, the main function simply calls *tx_kernel_enter*, which is the entry into ThreadX. However, applications can perform preliminary processing (usually for hardware initialization) prior to entering ThreadX.

The call to tx_kernel_enter does not return, so don't place any processing after it!

tx_kernel_enter

The entry function coordinates initialization of various internal ThreadX data structures and then calls the application's definition function *tx_application_define*.

When *tx_application_define* returns, control is transferred to the thread scheduling loop. This marks the end of initialization!

Application Definition Function

The *tx_application_define* function defines all of the initial application threads, queues, semaphores, mutexes, event flags, memory pools, and timers. It is also possible to create and delete system resources from threads during the normal operation of the application. However, all initial application resources are defined here.

The *tx_application_define* function has a single input parameter and it is certainly worth mentioning. The *first-available* RAM address is the sole input parameter to this function. It is typically used as a starting point for initial run-time memory allocations of thread stacks, queues, and memory pools.

After initialization is complete, only an executing thread can create and delete system resources—including other threads. Therefore, at least one thread must be created during initialization.

Interrupts

Interrupts are left disabled during the entire initialization process. If the application somehow enables interrupts, unpredictable behavior may occur. Figure 4 on page 49 shows the entire initialization process, from system reset through application-specific initialization.

Thread Execution

Scheduling and executing application threads is the most important activity of ThreadX. What exactly is a thread? A thread is typically defined as semi-

Initialization Process

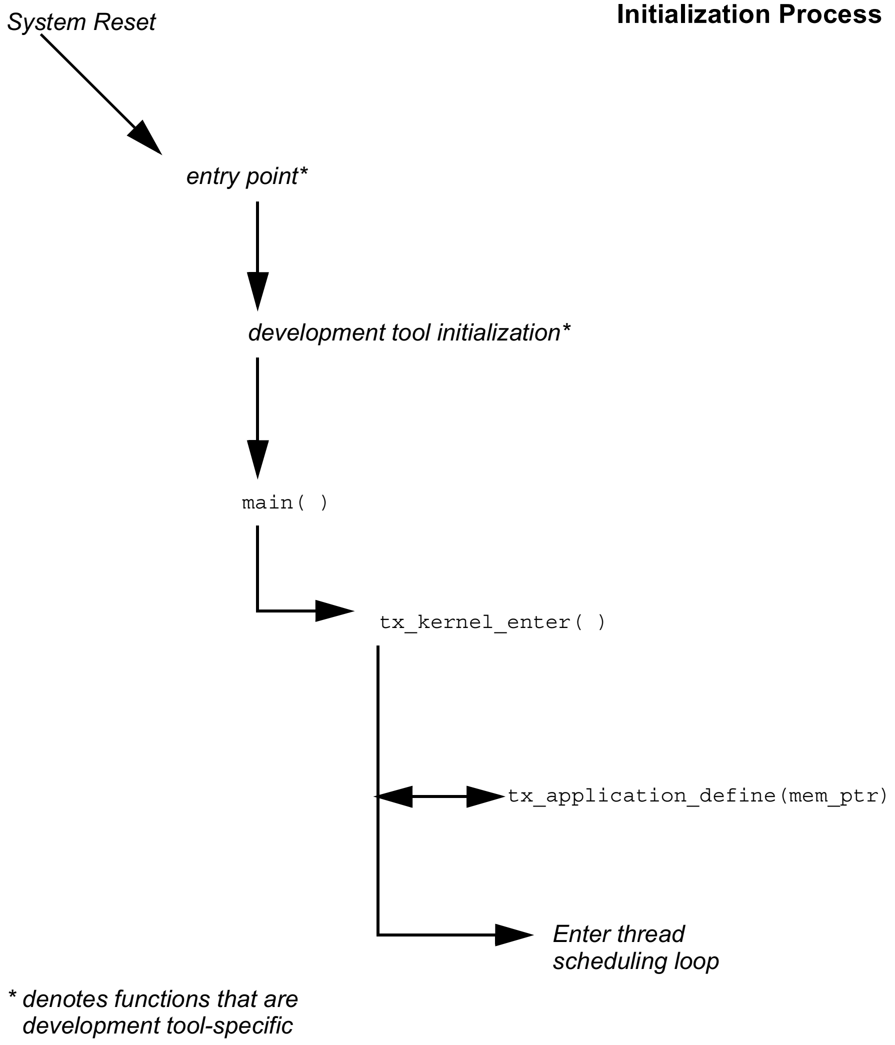
System Reset
entry point*
development tool initialization*
main( )
tx_kernel_enter( )
tx_application_define(mem_ptr)
Enter thread scheduling loop
* denotes functions that are development tool-specific

FIGURE 4. Initialization Process

independent program segment with a dedicated purpose. The combined processing of all threads makes an application.

How are threads created? Threads are created dynamically by calling *tx_thread_create* during initialization or during thread execution. Threads are created in either a *ready* or *suspended* state.

Thread Execution States

Understanding the different processing states of threads is a key ingredient to understanding the entire multi-threaded environment. In ThreadX there are five distinct thread states, namely *ready*, *suspended*, *executing*, *terminated*, and *completed*. Figure 5 on page 51 shows the thread state transition diagram for ThreadX.

A thread is in a *ready* state when it is ready for execution. A ready thread is not executed until it is the highest priority thread ready. When this happens, ThreadX executes the thread, which changes its state to *executing*.

If a higher-priority thread becomes ready, the executing thread reverts back to a *ready* state. The newly ready high-priority thread is then executed, which changes its logical state to *executing*. This transition between *ready* and *executing* states occurs every time thread preemption occurs.

It is important to point out that at any given moment only one thread is in an *executing* state. This is because a thread in the *executing* state actually has control of the underlying processor.

Threads that are in a *suspended* state are not eligible for execution. Reasons for being in a *suspended* state include suspension for time, queue messages, semaphores, mutexes, event flags, memory, and basic thread suspension. Once the cause for suspension is removed, the thread is placed back in a *ready* state.

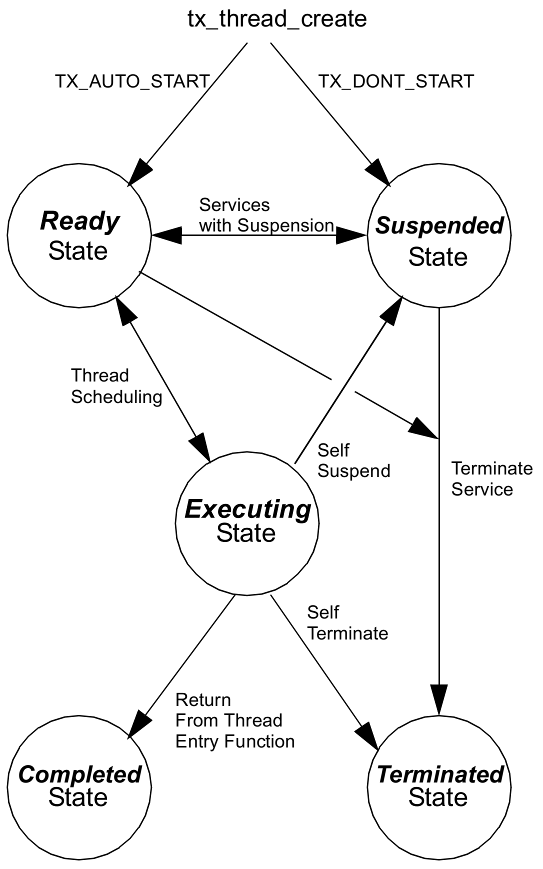

FIGURE 5. Thread State Transition

A thread in a *completed* state indicates the thread completed its processing and returned from its entry function. Remember that the entry function is specified during thread creation. A thread in a *completed* state cannot execute again.

A thread is in a *terminated* state because another thread or itself called the *tx_thread_terminate* service. A thread in a *terminated* state cannot execute again.

If re-starting a completed or terminated thread is desired, the application must first delete the thread. It can then be re-created and re-started.

Thread Priorities

As mentioned before, a thread is defined as a semi-independent program segment with a dedicated purpose. However, all threads are not created equal! The dedicated purpose of some threads is much more important than others. This heterogeneous type of thread importance is a hallmark of embedded real-time applications.

How does ThreadX determine a thread's importance? When a thread is created, it is assigned a numerical value representing its importance or *priority*. Valid numerical priorities range between 0 and 31, where a value of 0 indicates the highest thread priority and a value of 31 represents the lowest thread priority.

Threads can have the same priority as others in the application. In addition, thread priorities can be changed during run-time.

Thread Scheduling

ThreadX schedules threads based upon their priority. The ready thread with the highest priority is executed first. If multiple threads of the same priority are ready, they are executed in a *first-in-first-out* (FIFO) manner.

Round-Robin Scheduling

Round-robin scheduling of multiple threads having the same priority is supported by ThreadX. This is accomplished through cooperative calls to *tx_thread_relinquish*. Calling this service gives all other ready threads at the same priority a chance to execute before the *tx_thread_relinquish* caller executes again.

Time-Slicing

Time-slicing provides another form of round-robin scheduling. In ThreadX, time-slicing is available on a per-thread basis. The thread's time-slice is assigned during creation and can be modified during run-time.

What exactly is a time-slice? A time-slice specifies the maximum number of timer ticks (timer interrupts) that a thread can execute without giving up the processor. When a time-slice expires, all other ready threads of the same priority level are given a chance to execute before the time-sliced thread executes again.

A fresh thread time-slice is given to a thread after it suspends, relinquishes, makes a ThreadX service call that causes preemption, or is itself time-sliced.

When a time-sliced thread is preempted, it will resume before other ready threads of equal priority for the remainder of its time-slice.

Using time-slicing results in a slight amount of system overhead. Since time-slicing is only useful in cases where multiple threads share the same priority, threads having a unique priority should not be assigned a time-slice.

Preemption

Preemption is the process of temporarily interrupting an executing thread in favor of a higher-priority thread. This process is invisible to the executing thread. When the higher-priority thread is finished, control is transferred back to the exact place where the preemption took place.

This is a very important feature in real-time systems because it facilitates fast response to important application events. Although a very important feature, preemption can also be a source of a variety of problems, including starvation, excessive overhead, and priority inversion.

Preemption-Threshold™

In order to ease some of the inherent problems of preemption, ThreadX provides a unique and advanced feature called *preemption-threshold*.

What is a preemption-threshold? A preemption-threshold allows a thread to specify a priority *ceiling* for disabling preemption. Threads that have higher priorities than the ceiling are still allowed to preempt, while those less than the ceiling are not allowed to preempt.

For example, suppose a thread of priority 20 only interacts with a group of threads that have priorities between 15 and 20. During its critical sections, the thread of priority 20 can set its preemption-threshold to 15, thereby preventing preemption from all of the threads that it interacts with. This still permits really important threads (priorities between 0 and 14) to preempt this thread during its critical section processing, which results in much more responsive processing.

Of course, it is still possible for a thread to disable all preemption by setting its preemption-threshold to 0. In addition, preemption-thresholds can be changed during run-time.

Note that using preemption-threshold disables time-slicing for the specified thread.

Priority Inheritance

ThreadX also supports optional priority inheritance within its mutex services described later in this chapter. Priority inheritance allows a lower priority thread to temporarily assume the priority of a high priority thread that is waiting for a mutex owned by the lower priority thread. This capability helps the application to avoid un-deterministic priority inversion by eliminating preemption of intermediate thread priorities. Of course, *preemption-threshold* may be used to achieve a similar result.

Thread Creation

Application threads are created during initialization or during the execution of other application threads. There are no limits on the number of threads that can be created by an application.

Thread Control Block TX_THREAD

The characteristics of each thread are contained in its control block. This structure is defined in the ***tx_api.h*** file.

A thread's control block can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Locating the control block in other areas requires a bit more care, just like all dynamically allocated memory. If a control block is allocated within a C function, the memory associated with it is part of the calling thread's stack. In general, using local storage for control blocks should be avoided because once the function returns, then all of its local variable stack space is released—regardless of whether another thread is using it for a control block!

In most cases, the application is oblivious to the contents of the thread's control block. However, there are some situations, especially in debug, where looking at certain members is quite useful. The following are a few of the more useful control block members:

tx_run_count This member contains a counter of how many times the thread has been scheduled. An increasing counter indicates the thread is being scheduled and executed.

tx_state This member contains the state of the associated thread. The following list represents the possible thread states:

`TX_READY`	`(0x00)`
`TX_COMPLETED`	`(0x01)`
`TX_TERMINATED`	`(0x02)`
`TX_SUSPENDED`	`(0x03)`
`TX_SLEEP`	`(0x04)`
`TX_QUEUE_SUSP`	`(0x05)`
`TX_SEMAPHORE_SUSP`	`(0x06)`
`TX_EVENT_FLAG`	`(0x07)`
`TX_BLOCK_MEMORY`	`(0x08)`
`TX_BYTE_MEMORY`	`(0x09)`
`TX_MUTEX_SUSP`	`(0x0D)`
`TX_IO_DRIVER`	`(0x0A)`

Of course there are many other interesting fields in the thread control block, including the stack pointer, time-slice value, priorities, etc. The user is welcome to review any and all of the control block members, but modification is strictly prohibited!

*There is no equate for the "executing" state mentioned earlier in this section. It is not necessary since there is only one executing thread at a given time. The state of an executing thread is also **TX_READY**.*

Currently Executing Thread

As mentioned before, there is only one thread executing at any given time. There are several ways to identify the executing thread, depending on who is making the request.

A program segment can get the control block address of the executing thread by calling ***tx_thread_identify***. This is useful in shared portions of application code that are executed from multiple threads.

In debug sessions, users can examine the internal ThreadX pointer ***_tx_thread_current_ptr***. It contains the control block address of the currently executing thread. If this pointer is NULL, no application thread is executing; i.e., ThreadX is waiting in its scheduling loop for a thread to become ready.

Thread Stack Area

Each thread must have its own stack for saving the context of its last execution and compiler use. Most C compilers use the stack for making function calls and for temporarily allocating local variables. Figure 6 shows a typical thread's stack.

Where is a thread stack located? This is really up to the application. The stack area is specified during thread creation and can be located anywhere in the target's address space. This is a very important feature because it allows applications to improve performance of important threads by placing their stack in high-speed RAM.

How big should a stack be? This is one of the most frequently asked questions about threads. A thread's stack area must be large enough to accommodate worst-case function call nesting, local variable allocation, and saving its last execution context.

The minimum stack size, **TX_MINIMUM_STACK**, is defined by ThreadX. A stack of this size supports

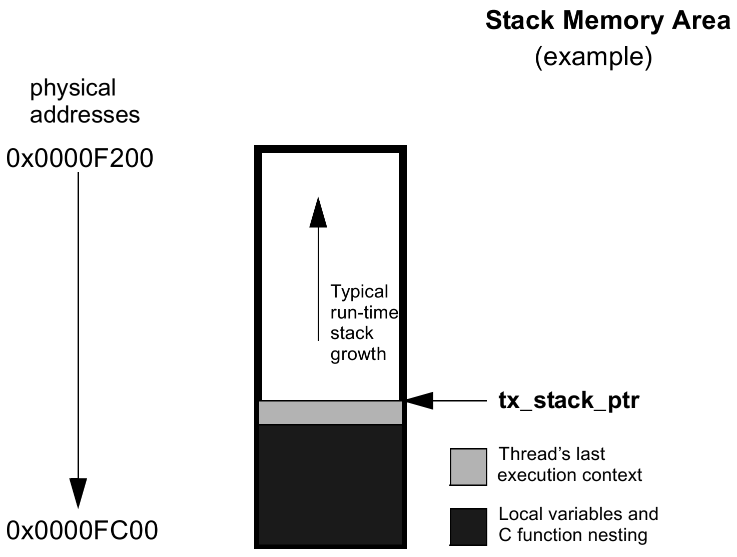
Stack Memory Area
(example)
physical addresses
0x0000F200
0x0000FC00
Typical run-time stack growth
tx_stack_ptr
Thread's last execution context
Local variables and C function nesting

FIGURE 6. Typical Thread Stack

saving a thread's context and minimum amount of function calls and local variable allocation.

For most threads, the minimum stack size is simply too small. The user must come up with the worst-case size requirement by examining function-call nesting and local variable allocation. Of course, it is always better to error towards a larger stack area.

After the application is debugged, it is possible to go back and tune the thread stacks sizes if memory is scarce. A favorite trick is to preset all stack areas with an easily identifiable data pattern like (0xEFEF) prior to creating the threads. After the application has been thoroughly put through its paces, the stack areas can be examined to see how much was actually used by finding the area of the stack where the preset pattern is still intact. Figure 7 on page 59 shows a stack preset to 0xEFEF after thorough thread execution.

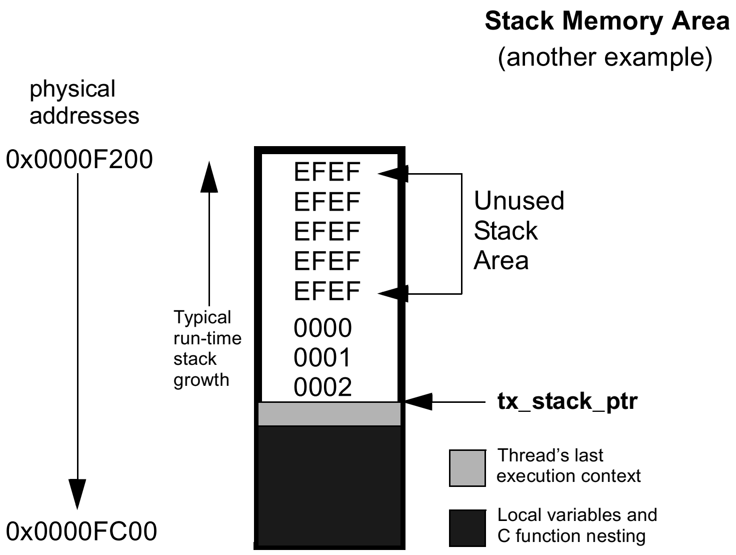

FIGURE 7. Stack Preset to 0xEFEF

Memory Pitfalls

The stack requirements for threads can be quite large. Therefore, it is important to design the application to have a reasonable number of threads. Furthermore, some care must be taken to avoid excessive stack usage within threads. Recursive algorithms and large local data structures should generally be avoided.

What happens when a stack area is too small? In most cases, the run-time environment simply assumes there is enough stack space. This causes thread execution to corrupt memory adjacent (usually before) its stack area. The results are very unpredictable, but most often result in an un-natural change in the program counter. This is often called "jumping into the weeds." Of course, the only way to prevent this is to ensure that all thread stacks are large enough.

Reentrancy

One of the real beauties of multi-threading is that the same C function can be called from multiple threads. This provides great power and also helps reduce code space. However, it does require that C functions called from multiple threads are *reentrant*.

What does reentrant mean? Basically, a reentrant function stores the caller's return address on the current stack and does not rely on global or static C variables that it previously setup. Most compilers place the return address on the stack. Hence, application developers must only worry about the use of *globals* and *statics*.

An example of a non-reentrant function is the string token function "strtok" found in the standard C library. This function remembers the previous string pointer on subsequent calls. It does this with a static string pointer. If this function is called from multiple threads, it would most likely return an invalid pointer.

Thread Priority Pitfalls

Selecting thread priorities is one of the most important aspects of multi-threading. It is sometimes very tempting to assign priorities based on a perceived notion of thread importance rather than determining what is exactly required during run-time. Misuse of thread priorities can starve other threads, create priority inversion, reduce processing bandwidth, and make the application's run-time behavior difficult to understand.

As mentioned before, ThreadX provides a priority-based, preemptive scheduling algorithm. Lower priority threads do not execute until there are no higher-priority threads ready for execution. If a higher-priority thread is always ready, the lower-priority threads never execute. This condition is called *thread starvation*.

Most starvation problems are detected early in debug and can be solved by ensuring that higher priority threads don't execute continuously. Alternatively, logic can be added to the application that gradually raises the priority of starved threads until they get a chance to execute.

Another unpleasant pitfall associated with thread priorities is *priority inversion*. Priority inversion takes place when a higher-priority thread is suspended because a lower-priority thread has a needed resource. Of course, in some instances it is necessary for two threads of different priority to share a common resource. If these threads are the only ones active, the priority inversion time is bounded by the time the lower-priority thread holds the resource. This condition is both deterministic and quite normal. However, if threads of intermediate priority become active during this priority inversion condition, the priority inversion time is no longer deterministic and could cause an application failure.

There are principally three distinct methods of preventing un-deterministic priority inversion in ThreadX. First, the application priority selections and run-time behavior can be designed in a manner that prevents the priority inversion problem. Second, lower-priority threads can utilize *preemption-threshold* to block preemption from intermediate threads while they share resources with higher-priority threads. Finally, threads using ThreadX mutex objects to protect system resources may utilize the optional mutex *priority inheritance* to eliminate un-deterministic priority inversion.

Priority Overhead

One of the most overlooked ways to reduce overhead in multi-threading is to reduce the number of context switches. As previously mentioned, a context switch occurs when execution of a higher-priority thread is favored over that of the executing thread. It is worthwhile to mention that higher-priority threads can become ready as a result of both external events (like interrupts) and from service calls made by the executing thread.

To illustrate the effects thread priorities have on context switch overhead, assume a three thread environment with threads named *thread_1*, *thread_2*, and *thread_3*. Assume further that all of the threads are in a state of suspension waiting for a message. When thread_1 receives a message, it immediately forwards it to thread_2. Thread_2 then forwards the message to thread_3. Thread_3 just discards the message. After each thread processes its message, they go back and wait for another.

The processing required to execute these three threads varies greatly depending on their priorities. If all of the threads have the same priority, a single context switch occurs between their execution. The context switch occurs when each thread suspends on an empty message queue.

However, if thread_2 is higher-priority than thread_1 and thread_3 is higher-priority than thread_2, the number of context switches doubles. This is because another context switch occurs inside of the *tx_queue_send* service when it detects that a higher-priority thread is now ready.

The ThreadX preemption-threshold mechanism can avoid these extra context switches and still allow the previously mentioned priority selections. This is a really important feature because it allows several thread priorities during scheduling, while at the same time eliminating some of the unwanted context switching between them during thread execution.

Debugging Pitfalls

Debugging multi-threaded applications is a little more difficult because the same program code can be executed from multiple threads. In such cases, a break-point alone may not be enough. The debugger must also view the current thread pointer ***_tx_thread_current_ptr*** to see if the calling thread is the one to debug.

Much of this is being handled in multi-threading support packages offered through various development tool vendors. Because of its simple design, integrating ThreadX with different development tools is relatively easy.

Stack size is always an important debug topic in multi-threading. Whenever totally strange behavior is seen, it is usually a good first guess to increase stack sizes for all threads—especially the stack size of the last executing thread!

Message Queues

Message queues are the primary means of inter-thread communication in ThreadX. One or more messages can reside in a message queue. A message queue that holds a single message is commonly called a *mailbox*.

Messages are copied to a queue by *tx_queue_send* and are copied from a queue by *tx_queue_receive*. The only exception to this is when a thread is suspended while waiting for a message on an empty queue. In this case, the next message sent to the queue is placed directly into the thread's destination area.

Each message queue is a public resource. ThreadX places no constraints on how message queues are used.

Creating Message Queues

Message queues are created either during initialization or during run-time by application threads. There are no limits on the number of message queues in an application.

Message Size

Each message queue supports a number of fixed-sized messages. The available message sizes are 1, 2, 4, 8, and 16 32-bit words. The message size is specified when the queue is created.

Application messages greater than 16 words must be passed by pointer. This is accomplished by creating a queue with a message size of 1 word (enough to hold a pointer) and then sending and receiving message pointers instead of the entire message.

Message Queue Capacity

The number of messages a queue can hold is a function of its message size and the size of the memory area supplied during creation. The total message capacity of the queue is calculated by dividing the number of bytes in each message into the total number of bytes in the supplied memory area.

For example, if a message queue that supports a message size of 1 32-bit word (4 bytes) is created with a 100-byte memory area, its capacity is 25 messages.

Queue Memory Area

As mentioned before, the memory area for buffering messages is specified during queue creation. Like other memory areas in ThreadX, it can be located anywhere in the target's address space.

This is an important feature because it gives the application considerable flexibility. For example, an application might locate the memory area of a very important queue in high-speed RAM in order to improve performance.

Thread Suspension

Application threads can suspend while attempting to send or receive a message from a queue. Typically, thread suspension involves waiting for a message from an empty queue. However, it is also possible for a thread to suspend trying to send a message to a full queue.

After the condition for suspension is resolved, the service requested is completed and the waiting thread is resumed. If multiple threads are suspended on the same queue, they are resumed in the order they were suspended (FIFO).

However, priority resumption is also possible if the application calls ***tx_queue_prioritize*** prior to the queue service that lifts thread suspension. The queue prioritize service places the highest priority thread at the front of the suspension list, while leaving all other suspended threads in the same FIFO order.

Time-outs are also available for all queue suspensions. Basically, a time-out specifies the maximum number of timer ticks the thread will stay suspended. If a time-out occurs, the thread is resumed and the service returns with the appropriate error code.

Queue Control Block TX_QUEUE

The characteristics of each message queue are found in its control block. It contains interesting information such as the number of messages in the queue. This structure is defined in the ***tx_api.h*** file.

Message queue control blocks can also be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Message Destination Pitfall

As mentioned previously, messages are copied between the queue area and application data areas. It is very important to insure that the destination for a received message is large enough to hold the entire message. If not, the memory following the message destination will likely be corrupted.

This is especially lethal when a too-small message destination is on the stack—nothing like corrupting the return address of a function!

Counting Semaphores

ThreadX provides 32-bit counting semaphores that range in value between 0 and 4,294,967,295. There are two operations for counting semaphores: *tx_semaphore_get* and *tx_semaphore_put*. The get operation decreases the semaphore by one. If the semaphore is 0, the get operation is not successful. The inverse of the get operation is the put operation. It increases the semaphore by one.

Each counting semaphore is a public resource. ThreadX places no constraints on how counting semaphores are used.

Counting semaphores are typically used for *mutual exclusion*. However, counting semaphores can also be used as a method for event notification.

Mutual Exclusion

Mutual exclusion pertains to controlling the access of threads to certain application areas (also called *critical sections* or *application resources*). When used for mutual exclusion, the “current count” of a semaphore represents the total number of threads that are allowed access. In most cases, counting semaphores used for mutual exclusion will have an initial value of 1, meaning that only one thread can access the associated resource at a time. Counting semaphores that only have values of 0 or 1 are commonly called *binary semaphores*.

If a binary semaphore is being used, the user must prevent the same thread from performing a get operation on a semaphore it already owns. A second get would be unsuccessful and could cause indefinite suspension of the calling thread and permanent unavailability of the resource.

Event Notification

It is also possible to use counting semaphores as event notification, in a producer-consumer fashion. The consumer attempts to get the counting semaphore while the producer increases the semaphore whenever something is available. Such semaphores usually have an initial value of 0 and won't increase until the producer has something ready for the consumer.

Creating Counting Semaphores

Counting semaphores are created either during initialization or during run-time by application threads. The initial count of the semaphore is specified during creation. There are no limits on the number of counting semaphores in an application.

Thread Suspension

Application threads can suspend while attempting to perform a get operation on a semaphore with a current count of 0.

Once a put operation is performed, the suspended thread's get operation is performed and the thread is resumed. If multiple threads are suspended on the same counting semaphore, they are resumed in the same order they were suspended (FIFO).

However, priority resumption is also possible if the application calls ***tx_semaphore_prioritize*** prior to the semaphore put call that lifts thread suspension. The semaphore prioritize service places the highest priority thread at the front of the suspension list, while leaving all other suspended threads in the same FIFO order.

Semaphore Control Block TX_SEMAPHORE

The characteristics of each counting semaphore are found in its control block. It contains interesting information such as the current semaphore count. This structure is defined in the ***tx_api.h*** file.

Semaphore control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Deadly Embrace

One of the most interesting and dangerous pitfalls associated with semaphores used for mutual exclusion is the *deadly embrace*. A deadly embrace, or *deadlock*, is a condition where two or more threads are suspended indefinitely while attempting to get semaphores already owned by other threads.

This condition is best illustrated by a two thread, two semaphore example. Suppose the first thread owns the first semaphore and the second thread owns the second semaphore. If the first thread attempts to get the second semaphore and at the same time the second thread attempts to get the first semaphore, both threads enter a deadlock condition. In addition, if these threads stay suspended forever, their associated resources are locked-out forever as well. Figure 8 on page 69 illustrates this example.

Deadly Embrace
(example)

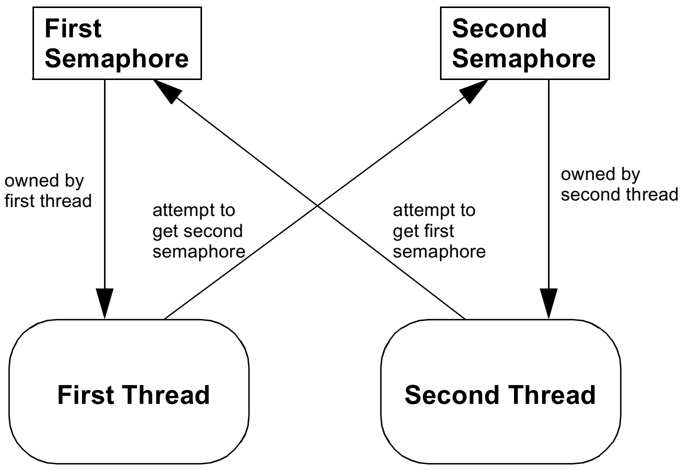

FIGURE 8. Example of Suspended Threads

How are deadly embraces avoided? Prevention in the application is the best method for real-time systems. This amounts to placing certain restrictions on how threads obtain semaphores. Deadly embraces are avoided if threads can only have one semaphore at a time. Alternatively, threads can own multiple semaphores if they all gather them in the same order. In the previous example, if the first and second thread obtain the first and second semaphore in order, the deadly embrace is prevented.

It is also possible to use the suspension time-out associated with the get operation to recover from a deadly embrace.

Priority Inversion

Another pitfall associated with mutual exclusion semaphores is priority inversion. This topic is discussed more fully in “Thread Priority Pitfalls” on page 60.

The basic problem results from a situation where a lower-priority thread has a semaphore that a higher-priority thread needs. This in itself is normal. However, threads with priorities in between them may cause the priority inversion to last a non-deterministic amount of time. This can be handled through careful selection of thread priorities, using preemption- thresholds, and temporarily raising the priority of the thread that owns the resource to that of the high-priority thread.

Mutexes

In addition to semaphores, ThreadX also provides a mutex object. A mutex is basically a binary semaphore, which means that only one thread can own a mutex at a time. In addition, the same thread may perform a successful mutex get operation on an owned mutex multiple times, 4,294,967,295 to be exact. There are two operations on the mutex object, namely ***tx_mutex_get*** and ***tx_mutex_put***. The get operation obtains a mutex not owned by another thread, while the put operation releases a previously obtained mutex. In order for a thread to release a mutex, the number of put operations must equal the number of prior get operations.

Each mutex is a public resource. ThreadX places no constraints on how mutexes are used.

ThreadX mutexes are used solely for *mutual exclusion*. Unlike counting semaphores, mutexes have no use as a method for event notification.

Mutex Mutual Exclusion

Similar to the discussion in the counting semaphore section, mutual exclusion pertains to controlling the access of threads to certain application areas (also called *critical sections* or *application resources*). When available, a ThreadX mutex will have an ownership count of 0. Once the mutex is obtained by a thread, the ownership count is incremented once for every get operation performed on the mutex and decremented for every put operation.

Creating Mutexes

ThreadX mutexes are created either during initialization or during run-time by application threads. The initial condition of a mutex is always "available." Mutex creation is also where the determination is made as to whether or not the mutex implements *priority inheritance.*

Thread Suspension

Application threads can suspend while attempting to perform a get operation on a mutex already owned by another thread.

Once the same number of put operations are performed by the owning thread, the suspended thread's get operation is performed, giving it ownership of the mutex, and the thread is resumed. If multiple threads are suspended on the same mutex, they are resumed in the same order they were suspended (FIFO).

However, priority resumption is done automatically if the mutex priority inheritance was selected during creation. In addition, priority resumption is also possible if the application calls ***tx_mutex_prioritize*** prior to the mutex put call that lifts thread suspension. The mutex prioritize service places the highest priority thread at the front of the suspension list, while leaving all other suspended threads in the same FIFO order.

Mutex Control Block TX_MUTEX

The characteristics of each mutex are found in its control block. It contains interesting information such as the current mutex ownership count along with the pointer of the thread that owns the mutex. This structure is defined in the ***tx_api.h*** file.

Mutex control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Deadly Embrace

One of the most interesting and dangerous pitfalls associated with mutex ownership is the *deadly embrace*. A deadly embrace, or *deadlock*, is a condition where two or more threads are suspended indefinitely while attempting to get a mutex already owned by the other threads. The discussion of *deadly embrace* and its remedies found in the previous semaphore discussion is completely valid for the mutex object as well.

Priority Inversion

As mentioned previously, a major pitfall associated with mutual exclusion is priority inversion. This topic is discussed more fully in “Thread Priority Pitfalls” on page 60.

The basic problem results from a situation where a lower-priority thread has a semaphore that a higher-priority thread needs. This in itself is normal. However, threads with priorities in between them may cause the priority inversion to last a non-deterministic amount of time. Unlike semaphores discussed previously, the ThreadX mutex object has optional *priority inheritance*. The basic idea behind priority inheritance is that a lower priority thread has its priority raised temporarily to the priority of a high priority thread that wants the same mutex owned by the lower priority thread. When the lower priority thread releases the mutex, its original priority is then restored and the higher priority thread is given ownership of the mutex. This feature eliminates un-deterministic priority inversion by bounding the amount of inversion to the time the lower priority thread holds the mutex. Of course, the techniques discussed earlier in this chapter to handle un-deterministic priority inversion are also valid with mutexes as well.

Event Flags

Event flags provide a powerful tool for thread synchronization. Each event flag is represented by a single bit. Event flags are arranged in groups of 32.

Threads can operate on all 32 event flags in a group at the same time. Events are set by *tx_event_flags_set* and are retrieved by *tx_event_flags_get*.

Setting event flags is done with a logical AND/OR operation between the current event flags and the new event flags. The type of logical operation (either an AND or OR) is specified in the *tx_event_flags_set* call.

There are similar logical options for retrieval of event flags. A get request can specify that all specified event flags are required (a logical AND). Alternatively, a get request can specify that any of the specified event flags will satisfy the request (a logical OR). The type of logical operation associated with event flag retrieval is specified in the *tx_event_flags_get* call.

i *Event flags that satisfy a get request are consumed, i.e. set to zero, if* ***TX_OR_CLEAR*** *or* ***TX_AND_CLEAR*** *are specified by the request.*

Each event flag group is a public resource. ThreadX places no constraints on how event flag groups are used.

Creating Event Flag Groups

Event flag groups are created either during initialization or during run-time by application threads. At time of their creation, all event flags in the group are set to zero. There are no limits on the number of event flag groups in an application.

Thread Suspension

Application threads can suspend while attempting to get any logical combination of event flags from a group. Once an event flag is set, the get requests of all suspended threads are reviewed. All the threads that now have the required event flags are resumed.

It is important to emphasize that all suspended threads on an event flag group are reviewed when its event flags are set. This, of course, introduces additional overhead. Therefore, it is generally good practice to limit the number of threads using the same event flag group to a reasonable number.

Event Flag Group Control Block

TX_EVENT_FLAGS_GROUP

The characteristics of each event flag group are found in its control block. It contains information such as the current event flag settings and the number of threads suspended for events. This structure is defined in the **tx_api.h** file.

Event group control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Memory Block Pools

Allocating memory in a fast and deterministic manner is always a challenge in real-time applications. With this in mind, ThreadX provides the ability to create and manage multiple pools of fixed-size memory blocks.

Since memory block pools consist of fixed-size blocks, there are never any fragmentation problems. Of course, fragmentation causes behavior that is inherently un-deterministic. In addition, the time required to allocate and free a fixed-size memory is comparable to that of simple linked-list manipulation. Furthermore, memory block allocation and de-allocation is done at the head of the available list. This provides the fastest possible linked list processing and might help keep the actual memory block in cache.

Lack of flexibility is the main drawback of fixed-size memory pools. The block size of a pool must be large enough to handle the worst case memory requirements of its users. Of course, memory may be wasted if many different size memory requests are made to the same pool. A possible solution is to make several different memory block pools that contain different sized memory blocks.

Each memory block pool is a public resource. ThreadX places no constraints on how pools are used.

Creating Memory Block Pools

Memory block pools are created either during initialization or during run-time by application threads. There are no limits on the number of memory block pools in an application.

Memory Block Size

As mentioned earlier, memory block pools contain a number of fixed-size blocks. The block size, in bytes, is specified during creation of the pool.

ThreadX adds a small amount of overhead—the size of a C pointer—to each memory block in the pool. In addition, ThreadX might have to pad the block size in order to keep the beginning of each memory block on proper alignment.

Pool Capacity

The number of memory blocks in a pool is a function of the block size and the total number of bytes in the memory area supplied during creation. The capacity of a pool is calculated by dividing the block size (including padding and the pointer overhead bytes) into the total number of bytes in the supplied memory area.

Pool’s Memory Area

As mentioned before, the memory area for the block pool is specified during creation. Like other memory areas in ThreadX, it can be located anywhere in the target’s address space.

This is an important feature because of the considerable flexibility it gives the application. For example, suppose that a communication product has a high-speed memory area for I/O. This memory area is easily managed by making it into a ThreadX memory block pool.

Thread Suspension

Application threads can suspend while waiting for a memory block from an empty pool. When a block is returned to the pool, the suspended thread is given this block and resumed.

If multiple threads are suspended on the same memory block pool, they are resumed in the order they were suspended (FIFO).

However, priority resumption is also possible if the application calls ***tx_block_pool_prioritize*** prior to the block release call that lifts thread suspension. The block pool prioritize service places the highest priority thread at the front of the suspension list, while leaving all other suspended threads in the same FIFO order.

Memory Block Pool Control Block TX_BLOCK_POOL

The characteristics of each memory block pool are found in its control block. It contains information such as the number of memory blocks left and their size. This structure is defined in the ***tx_api.h*** file.

Pool control blocks can also be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Overwriting Memory Blocks

It is very important to ensure that the user of an allocated memory block does not write outside its boundaries. If this happens, corruption occurs in an adjacent (usually subsequent) memory area. The results are unpredictable and quite often fatal!

Memory Byte Pools

ThreadX memory byte pools are similar to a standard C heap. Unlike the standard C heap, it is possible to have multiple memory byte pools. In addition, threads can suspend on a pool until the requested memory is available.

Allocations from memory byte pools are similar to traditional *malloc* calls, which include the amount of memory desired (in bytes). Memory is allocated from the pool in a *first-fit* manner, i.e., the first free memory block that satisfies the request is used. Excess memory from this block is converted into a new block and placed back in the free memory list. This process is called *fragmentation*.

Adjacent free memory blocks are *merged* together during a subsequent allocation search for a large enough free memory block. This process is called *de-fragmentation*.

Each memory byte pool is a public resource. ThreadX places no constraints on how pools are used, except that memory byte services can not be called from ISRs.

Creating Memory Byte Pools

Memory byte pools are created either during initialization or during run-time by application threads.There are no limits on the number of memory byte pools in an application.

Pool Capacity

The number of allocatable bytes in a memory byte pool is slightly less than what was specified during creation. This is because management of the free memory area introduces some overhead. Each free memory block in the pool requires the equivalent of two C pointers of overhead. In addition, the pool is created with two blocks, a large free block and a small permanently allocated block at the end of the memory area. This allocated block is used to improve performance of the allocation algorithm. It eliminates the need to continuously check for the end of the pool area during merging.

During run-time, the amount of overhead in the pool typically increases. Allocations of an odd number of bytes are padded to insure proper alignment of the next memory block. In addition, overhead increases as the pool becomes more fragmented.

Pool's Memory Area

The memory area for a memory byte pool is specified during creation. Like other memory areas in ThreadX, it can be located anywhere in the target's address space.

This is an important feature because of the considerable flexibility it gives the application. For example, if the target hardware has a high-speed memory area and a low-speed memory area, the user can manage memory allocation for both areas by creating a pool in each of them.

Thread Suspension

Application threads can suspend while waiting for memory bytes from a pool. When sufficient contiguous memory becomes available, the suspended threads are given their requested memory and resumed.

If multiple threads are suspended on the same memory byte pool, they are given memory (resumed) in the order they were suspended (FIFO).

However, priority resumption is also possible if the application calls ***tx_byte_pool_prioritize*** prior to the byte release call that lifts thread suspension. The byte pool prioritize service places the highest priority thread at the front of the suspension list, while leaving all other suspended threads in the same FIFO order.

Memory Byte Pool Control Block TX_BYTE_POOL

The characteristics of each memory byte pool are found in its control block. It contains useful information such as the number of available bytes in the pool. This structure is defined in the ***tx_api.h*** file.

Pool control blocks can also be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Un-deterministic Behavior

Although memory byte pools provide the most flexible memory allocation, they also suffer from somewhat un-deterministic behavior. For example, a memory byte pool may have 2,000 bytes of memory available but may not be able to satisfy an allocation request of 1,000 bytes. This is because there are no guarantees on how many of the free bytes are contiguous. Even if a 1,000 byte free block exits, there are no guarantees on how long it might take to find the block. It is completely possible that the entire memory pool would need to be searched in order to find the 1,000 byte block.

Because of this, it is generally good practice to avoid using memory byte services in areas where deterministic, real-time behavior is required. Many applications pre-allocate their required memory during initialization or run-time configuration.

Overwriting Memory Blocks

It is very important to insure that the user of allocated memory does not write outside its boundaries. If this happens, corruption occurs in an adjacent (usually subsequent) memory area. The results are unpredictable and quite often fatal!

Application Timers

Fast response to asynchronous external events is the most important function of real-time, embedded applications. However, many of these applications must also perform certain activities at pre-determined intervals of time.

ThreadX application timers provide applications with the ability to execute application C functions at specific intervals of time. It is also possible for an application timer to expire only once. This type of timer is called a *one-shot timer*, while repeating interval timers are called *periodic timers*.

Each application timer is a public resource. ThreadX places no constraints on how application timers are used.

Timer Intervals

In ThreadX time intervals are measured by periodic timer interrupts. Each timer interrupt is called a timer *tick*. The actual time between timer ticks is specified by the application, but 10ms is the norm for most implementations. The periodic timer setup is typically found in the ***tx_ill*** assembly file.

It is worth mentioning that the underlying hardware must have the ability to generate periodic interrupts in order for application timers to function. In some cases, the processor has a built-in periodic interrupt capability. If the processor doesn't have this ability, the user's board must have a peripheral device that can generate periodic interrupts.

ThreadX can still function even without a periodic interrupt source. However, all timer-related processing is then disabled. This includes time-slicing, suspension time-outs, and timer services.

Timer Accuracy

Timer expirations are specified in terms of ticks. The specified expiration value is decreased by one on each timer tick. Since an application timer could be enabled just prior to a timer interrupt (or timer tick), the actual expiration time could be up to one tick early.

If the timer tick rate is 10ms, application timers may expire up to 10ms early. This is more significant for 10ms timers than 1 second timers. Of course, increasing the timer interrupt frequency decreases this margin of error.

Timer Execution

Application timers execute in the order they become active. For example, if three timers are created with the same expiration value and activated, their corresponding expiration functions are guaranteed to execute in order they were activated.

Creating Application Timers

Application timers are created either during initialization or during run-time by application threads. There are no limits on the number of application timers in an application.

Application Timer Control Block TX_TIMER

The characteristics of each application timer are found in its control block. It contains useful information such as the 32-bit expiration identification value. This structure is defined in the ***tx_api.h*** file.

Application timer control blocks can be located anywhere in memory, but it is most common to make the control block a global structure by defining it outside the scope of any function.

Excessive Timers

By default, application timers execute from within a hidden system thread that runs at priority zero, which is higher than any application thread. Because of this, processing inside application timers should be kept to a minimum.

It is also important to avoid, whenever possible, timers that expire every timer tick. Such a situation might induce excessive overhead in the application.

As mentioned previously, application timers are executed from a hidden system thread. It is, therefore, very important not to select suspension on any ThreadX service calls made from within the application timer's expiration function.

Relative Time

In addition to the application timers mentioned previously, ThreadX provides a single continuously incrementing 32-bit tick counter. The tick counter or *time* is increased by one on each timer interrupt.

The application can read or set this 32-bit counter through calls to *tx_time_get* and *tx_time_set*, respectively. The use of this tick counter is determined completely by the application. It is not used internally by ThreadX.

Interrupts

Fast response to asynchronous events is the principal function of real-time, embedded applications. How does the application know such an event is present? Typically, this is accomplished through hardware interrupts.

An interrupt is an asynchronous change in processor execution. Typically, when an interrupt occurs, the processor saves a small portion of the current execution on the stack and transfers control to the appropriate interrupt vector. The interrupt vector is basically just the address of the routine responsible for handling the specific type interrupt. The exact interrupt handling procedure is processor specific.

Interrupt Control

The *tx_interrupt_control* service allows applications to enable and disable interrupts. The previous interrupt enable/disable posture is returned by this service. It is important to mention that interrupt control only affects the currently executing program segment. For example, if a thread disables interrupts, they only remain disabled during execution of that thread.

A Non-Maskable Interrupt (NMI) is defined as an interrupt that the cannot be disabled by the hardware. Such an interrupt may be used by ThreadX applications. However, the application's NMI handling routine is not allowed to use ThreadX context management or any API services.

ThreadX Managed Interrupts

ThreadX provides applications with complete interrupt management. This management includes saving and restoring the context of the interrupted execution. In addition, ThreadX allows certain services to be called from within Interrupt Service Routines (ISRs). The following is a list of ThreadX services allowed from application ISRs:

```
tx_block_allocate
tx_block_pool_info_get
tx_block_pool_prioritize
tx_block_release
tx_byte_pool_info_get
tx_byte_pool_prioritize
tx_event_flags_info_get
tx_event_flags_get
```

```
tx_event_flags_set
tx_interrupt_control
tx_queue_front_send
tx_queue_info_get
tx_queue_prioritize
tx_queue_receive
tx_queue_send
tx_semaphore_get
tx_semaphore_info_get
tx_semaphore_prioritize
tx_semaphore_put
tx_thread_identify
tx_thread_info_get
tx_thread_resume
tx_thread_wait_abort
tx_time_get
tx_time_set
tx_timer_activate
tx_timer_change
tx_timer_deactivate
tx_timer_info_get
```

Suspension is not allowed from ISRs. Therefore, special care must be made not to specify suspension in service calls made from ISRs.

ISR Template

In order to manage application interrupts, several ThreadX utilities must be called in the beginning and end of application ISRs. The exact format for interrupt handling varies between ports. Please review the ***readme.txt*** file on the distribution disk for specific instructions on managing ISRs.

The following small code segment is typical of most ThreadX managed ISRs. In most cases, this processing is in assembly language.

```
_application_ISR_entry:
; Save context and prepare for
; ThreadX use by calling the ISR
; entry function.
CALL __tx_thread_context_save

; The ISR can now call ThreadX
; services and its own C functions

; When the ISR is finished, context
; is restored (or thread preemption)
; by calling the context restore
; function. Control does not return!
JUMP __tx_thread_context_restore
```

High-Frequency Interrupts

Some interrupts occur at such a high-frequency that saving and restoring full context upon each interrupt would consume excessive processing bandwidth. In such cases, it is common for the application to have a small assembly language ISR that does a limited amount of processing for a majority of these high-frequency interrupts.

After a certain point in time, the small ISR may need to interact with ThreadX. This is accomplished by simply calling the entry and exit functions described in the above template.

Interrupt Latency

ThreadX locks out interrupts over brief periods of time. The maximum amount of time interrupts are disabled is on the order of the time required to save or restore a thread's context.

CHAPTER **4**

Description of ThreadX Services

This chapter contains a description of all ThreadX services (listed below) in alphabetic order. Their names are designed so that you will find all similar services grouped together. For example, all memory block services are found at the beginning of this chapter.

In the “Return Values” section in the following API descriptions, values in **BOLD** are not affected by the **TX_DISABLE_ERROR_CHECKNG** define that is used to disable API error checking; while non-bold values are completely disabled.

tx_block_allocate

Allocate a fixed-size block of memory

Prototype

```
UINT tx_block_allocate(TX_BLOCK_POOL *pool_ptr, VOID **block_ptr,
                           ULONG wait_option)
```

Description

This service allocates a fixed-size memory block from the specified memory pool. The actual size of the memory block is determined during memory pool creation.

Input Parameters

pool_ptr	Pointer to a previously created memory block pool.
block_ptr	Pointer to a destination block pointer. On successful allocation, the address of the allocated memory block is placed where this parameter points to.
wait_option	Defines how the service behaves if there are no memory blocks available. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. *This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR*.

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a memory block is available.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for a memory block.

Return Values

TX_SUCCESS	(0x00)	Successful memory block allocation.
TX_DELETED	(0x01)	Memory block pool was deleted while thread was suspended.
TX_NO_MEMORY	(0x10)	Service was unable to allocate a block of memory.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer or ISR.
TX_POOL_ERROR	(0x02)	Invalid memory block pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer to destination pointer.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_BLOCK_POOL my_pool;
unsigned char *memory_ptr;
UINT status;

/* Allocate a memory block from my_pool. Assume that the
   pool has already been created with a call to
   tx_block_pool_create.  */
status = tx_block_allocate(&my_pool, (VOID **) &memory_ptr,
                               TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the
   address of the allocated block of memory.  */
```

See Also

tx_block_pool_create, tx_block_pool_delete, tx_block_pool_info_get, tx_block_pool_prioritize, tx_block_release

tx_block_pool_create

Create a pool of fixed-size memory blocks

Prototype

```
UINT tx_block_pool_create(TX_BLOCK_POOL *pool_ptr,
                           CHAR *name_ptr, ULONG block_size,
                           VOID *pool_start, ULONG pool_size)
```

Description

This service creates a pool of fixed-size memory blocks. The memory area specified is divided into as many fixed-size memory blocks as possible using the formula:

total blocks = (**total bytes**) / (**block size** + sizeof(void *))

*Each memory block contains one pointer of overhead that is invisible to the user and is represented by the "sizeof(void *)" in the preceding formula.*

Input Parameters

pool_ptr	Pointer to a memory block pool control block.
name_ptr	Pointer to the name of the memory block pool.
block_size	Number of bytes in each memory block.
pool_start	Starting address of the memory block pool.
pool_size	Total number of bytes available for the memory block pool.

Return Values

TX_SUCCESS	(0x00)	Successful memory block pool creation.
TX_POOL_ERROR	(0x02)	Invalid memory block pool pointer. Either the pointer is NULL or the pool is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the pool.
TX_SIZE_ERROR	(0x05)	Size of pool is invalid.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_BLOCK_POOL my_pool;
UINT status;

/* Create a memory pool whose total size is 1000 bytes
   starting at address 0x100000. Each block in this
   pool is defined to be 50 bytes long.  */

status =  tx_block_pool_create(&my_pool, "my_pool_name",
                                      50, (VOID *) 0x100000, 1000);

/* If status equals TX_SUCCESS, my_pool contains 18
   memory blocks of 50 bytes each. The reason
   there are not 20 blocks in the pool is
   because of the one overhead pointer associated with each
   block.  */
```

See Also

tx_block_allocate, tx_block_pool_delete, tx_block_pool_info_get, tx_block_pool_prioritize, tx_block_release

tx_block_pool_delete

Delete fixed-size block of memory pool

Prototype

```
UINT tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr)
```

Description

This service deletes the specified block-memory pool. All threads suspended waiting for a memory block from this pool are resumed and given a TX_DELETED return status.

It is the application's responsibility to manage the memory area associated with the pool, which is available after this service completes. In addition, the application must prevent use of a deleted pool or its former memory blocks.

Input Parameters

pool_ptr	Pointer to a previously created memory block pool.

Return Values

TX_SUCCESS	(0x00)	Successful memory block pool deletion.
TX_POOL_ERROR	(0x02)	Invalid memory block pool pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_BLOCK_POOL my_pool;
UINT status;

   /* Delete entire memory block pool. Assume that the pool
      has already been created with a call to
      tx_block_pool_create.  */
   status =  tx_block_pool_delete(&my_pool);

   /* If status equals TX_SUCCESS, the memory block pool is
      deleted.  */
```

See Also

tx_block_allocate, tx_block_pool_create, tx_block_pool_info_get, tx_block_pool_prioritize, tx_block_release

tx_block_pool_info_get

Retrieve information about block pool

Prototype

```
UINT tx_block_pool_info_get(TX_BLOCK_POOL *pool_ptr, CHAR **name,
                            ULONG *available, ULONG *total_blocks,
                            TX_THREAD **first_suspended,
                            ULONG *suspended_count,
                            TX_BLOCK_POOL **next_pool)
```

Description

This service retrieves information about the specified block memory pool.

Input Parameters

pool_ptr	Pointer to previously created memory block pool.
name	Pointer to destination for the pointer to the block pool's name.
available	Pointer to destination for the number of available blocks in the block pool.
total_blocks	Pointer to destination for the total number of blocks in the block pool.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this block pool.
suspended_count	Pointer to destination for the number of threads currently suspended on this block pool.
next_pool	Pointer to destination for the pointer of the next created block pool.

Return Values

TX_SUCCESS	(0x00)	Successful block pool information retrieve.
TX_POOL_ERROR	(0x02)	Invalid memory block pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BLOCK_POOL my_pool;
CHAR *name;
ULONG available;
ULONG total_blocks;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_BLOCK_POOL *next_pool;
UINT status;

/* Retrieve information about a the previously created
   block pool "my_pool." */
status =  tx_block_pool_info_get(&my_pool, &name,
               &available,&total_packets,
               &first_suspended, &suspended_count,
               &next_pool);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_block_pool_allocate, tx_block_pool_create, tx_block_pool_delete, tx_block_pool_prioritize, tx_block_release

tx_block_pool_prioritize

Prioritize block pool suspension list

Prototype

```
UINT tx_block_pool_prioritize(TX_BLOCK_POOL *pool_ptr)
```

Description

This service places the highest priority thread suspended for a block of memory on this pool at the front of the suspension list. All other threads remain in the same FIFO order they were suspended in.

Input Parameters

pool_ptr	Pointer to a memory block pool control block.

Return Values

TX_SUCCESS	(0x00)	Successful block pool prioritize.
TX_POOL_ERROR	(0x02)	Invalid memory block pool pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BLOCK_POOL my_pool;
UINT status;

   /* Ensure that the highest priority thread will receive
      the next free block in this pool.  */
      status =  tx_block_pool_prioritize(&my_pool);

   /* If status equals TX_SUCCESS, the highest priority
      suspended thread is at the front of the list. The
      next tx_block_release call will wake up this thread. */
```

See Also

tx_block_allocate, tx_block_pool_create, tx_block_pool_delete, tx_block_pool_info_get, tx_block_release

tx_block_release

Release a fixed-size block of memory

Prototype

```
UINT tx_block_release(VOID *block_ptr)
```

Description

This service releases a previously allocated block back to its associated memory pool. If there are one or more threads suspended waiting for memory block from this pool, the first thread suspended is given this memory block and resumed.

The application must prevent using a memory block area after it has been released back to the pool.

Input Parameters

block_ptr	Pointer to the previously allocated memory block.

Return Values

TX_SUCCESS	(0x00)	Successful memory block release.
TX_PTR_ERROR	(0x03)	Invalid pointer to memory block.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_BLOCK_POOL my_pool;
unsigned char *memory_ptr;
UINT status;


/* Release a memory block back to my_pool. Assume that the
   pool has been created and the memory block has been
   allocated.  */
status =  tx_block_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the block of memory pointed
   to by memory_ptr has been returned to the pool.  */
```

See Also

tx_block_allocate, tx_block_pool_create, tx_block_pool_delete, tx_block_pool_info_get, tx_block_pool_prioritize

tx_byte_allocate

Allocate bytes of memory

Prototype

```
UINT tx_byte_allocate(TX_BYTE_POOL *pool_ptr,
                          VOID **memory_ptr, ULONG memory_size,
                          ULONG wait_option)
```

Description

This service allocates the specified number of bytes from the specified byte-memory pool.

The performance of this service is a function of the block size and the amount of fragmentation in the pool. Hence, this service should not be used during time-critical threads of execution.

Input Parameters

pool_ptr	Pointer to a previously created memory pool.
memory_ptr	Pointer to a destination memory pointer. On successful allocation, the address of the allocated memory area is placed where this parameter points to.
memory_size	Number of bytes requested.
wait_option	Defines how the service behaves if there is not enough memory available. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. *This is the only valid option if the service is called from initialization.*

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until enough memory is available.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the memory.

Return Values

TX_SUCCESS	(0x00)	Successful memory allocation.
TX_DELETED	(0x01)	Memory pool was deleted while thread was suspended.
TX_NO_MEMORY	(0x10)	Service was unable to allocate the memory.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer to destination pointer.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
TX_BYTE_POOL my_pool;
unsigned char *memory_ptr;
UINT status;

/* Allocate a 112 byte memory area from my_pool. Assume
   that the pool has already been created with a call to
   tx_byte_pool_create.  */
status =  tx_byte_allocate(&my_pool, (VOID **) &memory_ptr,
                       112, TX_NO_WAIT);

/* If status equals TX_SUCCESS, memory_ptr contains the
   address of the allocated memory area.  */
```

See Also

tx_byte_pool_create, tx_byte_pool_delete, tx_byte_pool_info_get, tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_create

Create a memory pool of bytes

Prototype

```
UINT tx_byte_pool_create(TX_BYTE_POOL *pool_ptr,
                          CHAR *name_ptr, VOID *pool_start,
                          ULONG pool_size)
```

Description

This service creates a memory pool in the area specified. Initially the pool consists of basically one very large free block. However, the pool is broken into smaller blocks as allocations are made.

Input Parameters

pool_ptr	Pointer to a memory pool control block.
name_ptr	Pointer to the name of the memory pool.
pool_start	Starting address of the memory pool.
pool_size	Total number of bytes available for the memory pool.

Return Values

TX_SUCCESS	(0x00)	Successful memory pool creation.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer. Either the pointer is NULL or the pool is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the pool.
TX_SIZE_ERROR	(0x05)	Size of pool is invalid.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_BYTE_POOL my_pool;
UINT status;

/* Create a memory pool whose total size is 2000 bytes
   starting at address 0x500000.  */
status =  tx_byte_pool_create(&my_pool, "my_pool_name",
             (VOID *) 0x500000, 2000);

/* If status equals TX_SUCCESS, my_pool is available for
   allocating memory.  */
```

See Also

tx_byte_allocate, tx_byte_pool_delete, tx_byte_pool_info_get, tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_delete

Delete a memory pool of bytes

Prototype

```
UINT tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr)
```

Description

This service deletes the specified memory pool. All threads suspended waiting for memory from this pool are resumed and given a TX_DELETED return status.

It is the application's responsibility to manage the memory area associated with the pool, which is available after this service completes. In addition, the application must prevent use of a deleted pool or memory previously allocated from it.

Input Parameters

pool_ptr	Pointer to a previously created memory pool.

Return Values

TX_SUCCESS	(0x00)	Successful memory pool deletion.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_BYTE_POOL my_pool;
UINT status;

/* Delete entire memory pool. Assume that the pool has already
   been created with a call to tx_byte_pool_create.  */
status =  tx_byte_pool_delete(&my_pool);

/* If status equals TX_SUCCESS, memory pool is deleted. */
```

See Also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_info_get, tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_info_get

Retrieve information about byte pool

Prototype

```
UINT tx_byte_pool_info_get(TX_BYTE_POOL *pool_ptr, CHAR **name,
                           ULONG *available, ULONG *fragments,
                           TX_THREAD **first_suspended,
                           ULONG *suspended_count,
                           TX_BYTE_POOL **next_pool)
```

Description

This service retrieves information about the specified memory byte pool.

Input Parameters

pool_ptr	Pointer to previously created memory pool.
name	Pointer to destination for the pointer to the byte pool's name.
available	Pointer to destination for the number of available bytes in the pool.
fragments	Pointer to destination for the total number of memory fragments in the byte pool.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this byte pool.
suspended_count	Pointer to destination for the number of threads currently suspended on this byte pool.
next_pool	Pointer to destination for the pointer of the next created byte pool.

Return Values

TX_SUCCESS	(0x00)	Successful pool information retrieve.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BYTE_POOL my_pool;
CHAR *name;
ULONG available;
ULONG fragments;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_BYTE_POOL *next_pool;
UINT status;

/* Retrieve information about a the previously created
   block pool "my_pool." */
status =  tx_byte_pool_info_get(&my_pool, &name,
                    &available, &fragments,
                    &first_suspended, &suspended_count,
                    &next_pool);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete, tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_prioritize

Prioritize the byte pool suspension list

Prototype

```
UINT tx_byte_pool_prioritize(TX_BYTE_POOL *pool_ptr)
```

Description

This service places the highest priority thread suspended for memory on this pool at the front of the suspension list. All other threads remain in the same FIFO order they were suspended in.

Input Parameters

pool_ptr	Pointer to a memory pool control block.

Return Values

TX_SUCCESS	(0x00)	Successful memory pool prioritize.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BYTE_POOL my_pool;
UINT status;

/* Ensure that the highest priority thread will receive
   the next free memory from this pool.  */
status =  tx_byte_pool_prioritize(&my_pool);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_byte_release call will wake up this thread,
   if there is enough memory to satisfy its request. */
```

See Also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete, tx_byte_pool_info_get, tx_byte_release

tx_byte_release

Release bytes back to memory pool

Prototype

```
UINT tx_byte_release(VOID *memory_ptr)
```

Description

This service releases a previously allocated memory area back to its associated pool. If there are one or more threads suspended waiting for memory from this pool, each suspended thread is given memory and resumed until the memory is exhausted or until there are no more suspended threads. This process of allocating memory to suspended threads always begins with the first thread suspended.

The application must prevent using the memory area after it is released.

Input Parameters

memory_ptr	Pointer to the previously allocated memory area.

Return Values

TX_SUCCESS	(0x00)	Successful memory release.
TX_PTR_ERROR	(0x03)	Invalid memory area pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
unsigned char *memory_ptr;
UINT status;

/* Release a memory back to my_pool. Assume that the memory
   area was previously allocated from my_pool.  */
status =  tx_byte_release((VOID *) memory_ptr);

/* If status equals TX_SUCCESS, the memory pointed to by
   memory_ptr has been returned to the pool.  */
```

See Also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete, tx_byte_pool_info_get, tx_byte_pool_prioritize

tx_event_flags_create

Create an event flag group

Prototype

```
UINT tx_event_flags_create(TX_EVENT_FLAGS_GROUP *group_ptr,
                           CHAR *name_ptr)
```

Description

This service creates a group of 32 event flags. All 32 event flags in the group are initialized to zero. Each event flag is represented by a single bit.

Input Parameters

group_ptr	Pointer to an event flags group control block.
name_ptr	Pointer to the name of the event flags group.

Return Values

TX_SUCCESS	(0x00)	Successful event group creation.
TX_GROUP_ERROR	(0x06)	Invalid event group pointer. Either the pointer is NULL or the event group is already created.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_EVENT_FLAGS_GROUP my_event_group;
UINT status;

/* Create an event flag group.  */
status =  tx_event_flags_create(&my_event_group,
                   "my_event_group_name");

/* If status equals TX_SUCCESS, my_event_flag_group is ready
   for get and set services. */
```

See Also

tx_event_flags_delete, tx_event_flags_get, tx_event_flags_info_get, tx_event_flags_set

tx_event_flags_delete

Delete an event flag group

Prototype

```
UINT tx_event_flags_delete(TX_EVENT_FLAGS_GROUP *group_ptr)
```

Description

This service deletes the specified event flag group. All threads suspended waiting for events from this group are resumed and given a TX_DELETED return status.

The application must prevent use of a deleted event flag group.

Input Parameters

group_ptr	Pointer to a previously created event flags group.

Return Values

TX_SUCCESS	(0x00)	Successful event flag group deletion.
TX_GROUP_ERROR	(0x06)	Invalid event flag group pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_EVENT_FLAGS_GROUP my_event_flag_group;
UINT status;

/* Delete event flag group. Assume that the group has
   already been created with a call to
   tx_event_flags_create.  */
status =  tx_event_flags_delete(&my_event_flags_group);

/* If status equals TX_SUCCESS, the event flags group is
   deleted.  */
```

See Also

tx_event_flags_create, tx_event_flags_get, tx_event_flags_info_get, tx_event_flags_set

tx_event_flags_get

Get event flags from event flag group

Prototype

```
UINT tx_event_flags_get(TX_EVENT_FLAGS_GROUP *group_ptr,
                           ULONG requested_flags, UINT get_option,
                           ULONG *actual_flags_ptr, ULONG wait_option)
```

Description

This service retrieves event flags from the specified event flag group. Each event flag group contains 32 event flags. Each flag is represented by a single bit. This service can retrieve a variety of event flag combinations, as selected by the input parameters.

Input Parameters

group_ptr Pointer to a previously created event flag group.

requested_flags 32-bit unsigned variable that represents the requested event flags.

get_option Specifies whether all or any of the requested event flagsare required. The following are valid selections:

TX_AND	(0x02)
TX_AND_CLEAR	(0x03)
TX_OR	(0x00)
TX_OR_CLEAR	(0x01)

Selecting TX_AND or TX_AND_CLEAR specifies that all event flags must be present in the group. Selecting TX_OR or TX_OR_CLEAR specifies that any event flag is satisfactory. Event flags that satisfy the request are cleared (set to zero) if TX_AND_CLEAR or TX_OR_CLEAR are specified.

actual_flags_ptr Pointer to destination of where the retrieved event flags are placed. Note that the actual flags obtained may contain flags that were not requested.

wait_option	Defines how the service behaves if the selected event flags are not set. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the event flags are available.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the event flags.

Return Values

TX_SUCCESS	(0x00)	Successful event flags get.
TX_DELETED	(0x01)	Event flag group was deleted while thread was suspended.
TX_NO_EVENTS	(0x07)	Service was unable to get the specified events.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_GROUP_ERROR	(0x06)	Invalid event flags group pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer for actual event flags.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.
TX_OPTION_ERROR	(0x08)	Invalid get-option was specified.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
ULONG actual_events;
UINT status;


/* Request that event flags 0, 4, and 8 are all set. Also,
   if they are set they should be cleared. If the event
   flags are not set, this service suspends for a maximum of
   20 timer-ticks.  */
status =  tx_event_flags_get(&my_event_flags_group, 0x111,
                          TX_AND_CLEAR, &actual_events, 20);

/* If status equals TX_SUCCESS, actual_events contains the
   actual events obtained.  */
```

See Also

tx_event_flags_create, tx_event_flags_delete, tx_event_flags_info_get, tx_event_flags_set

tx_event_flags_info_get

Retrieve information about event flags group

Prototype

```
UINT tx_event_flags_info_get(TX_EVENT_FLAGS_GROUP *group_ptr,
                          CHAR **name, ULONG *current_flags,
                          TX_THREAD **first_suspended,
                          ULONG *suspended_count,
                          TX_EVENT_FLAGS_GROUP **next_group)
```

Description

This service retrieves information about the specified event flags group.

Input Parameters

group_ptr	Pointer to an event flags group control block.
name	Pointer to destination for the pointer to the event flag group’s name.
current_flags	Pointer to destination for the current set flags in the event flag group.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this event flag group.
suspended_count	Pointer to destination for the number of threads currently suspended on this event flag group.
next_group	Pointer to destination for the pointer of the next created event flag group.

Return Values

TX_SUCCESS	(0x00)	Successful event group information retrieval.
TX_GROUP_ERROR	(0x06)	Invalid event group pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_EVENT_FLAGS_GROUP my_event_group;
CHAR *name;
ULONG current_flags;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_EVENT_FLAGS_GROUP *next_group;
UINT status;

/* Retrieve information about a the previously created
   event flag group "my_event_group." */
status =  tx_event_flags_info_get(&my_event_group, &name,
                          &current_flags,
                          &first_suspended, &suspended_count,
                          &next_group);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_event_flags_create, tx_event_flags_delete, tx_event_flags_get, tx_event_flags_set

tx_event_flags_set

Set event flags in an event flag group

Prototype

```
UINT tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr,
                          ULONG  flags_to_set,UINT  set_option)
```

Description

This service sets or clears event flags in an event flag group, depending upon the specified set-option. All suspended threads whose event flag request is now satisfied are resumed.

Input Parameters

group_ptr	Pointer to the previously created event flag group control block.
flags_to_set	Specifies the event flags to set or clear based upon the set option selected.
set_option	Specifies whether the event flags specified are ANDed or ORed into the current event flags of the group. The following are valid selections:

TX_AND	(0x02)
TX_OR	(0x00)

Selecting TX_AND specifies that the specified event flags are **AND**ed into the current event flags in the group. This option is often used to clear event flags in a group. Otherwise, if TX_OR is specified, the specified event flags are **OR**ed with the current event in the group.

Return Values

TX_SUCCESS	(0x00)	Successful event flag set.
TX_GROUP_ERROR	(0x06)	Invalid pointer to event flags group.
TX_OPTION_ERROR	(0x08)	Invalid set-option specified.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT status;

/* Set event flags 0, 4, and 8.  */
status =  tx_event_flags_set(&my_event_flags_group,
                             0x111, TX_OR);

/* If status equals TX_SUCCESS, the event flags have been
   set and any suspended thread whose request was satisfied
   has been resumed.  */
```

See Also

tx_event_flags_create, tx_event_flags_delete, tx_event_flags_get, tx_event_flags_info_get

tx_interrupt_control

Enables and disables interrupts

Prototype

```
UINT tx_interrupt_control(UINT new_posture)
```

Description

This service enables or disables interrupts as specified by the input parameter **new_posture**.

If this service is called from an application thread, the interrupt posture remains part of that thread's context. For example, if the thread calls this routine to disable interrupts and then suspends, when it is resumed, interrupts are disabled again.

This service should not be used to enable interrupts during initialization! Doing so could cause unpredictable results.

Input Parameters

new_posture	This parameter specifies whether interrupts are disabled or enabled. Legal values include **TX_INT_DISABLE** and **TX_INT_ENABLE**. The actual values for these parameters are port specific. In addition, some processing architectures might support additional interrupt disable postures. Please see the ***readme.txt*** information supplied on the distribution disk for more details.

Return Values

previous posture	This service returns the previous interrupt posture to the caller. This allows users of the service to restore the previous posture after interrupts are disabled.

Allowed From

Threads, timers, and ISRs

Preemption Possible

No

Example

```
UINT my_old_posture;

/* Lockout interrupts */
my_old_posture =  tx_interrupt_control(TX_INT_DISABLE);

/* Perform critical operations that need interrupts
   locked-out....  */

/* Restore previous interrupt lockout posture.  */
tx_interrupt_control(my_old_posture);
```

See Also

None

tx_mutex_create

Create a mutual exclusion mutex

Prototype

```
UINT tx_mutex_create(TX_MUTEX *mutex_ptr,
                           CHAR *name_ptr, UINT priority_inherit)
```

Description

This service creates a mutex for inter-thread mutual exclusion for resource protection.

Input Parameters

mutex_ptr	Pointer to a mutex control block.
name_ptr	Pointer to the name of the mutex.
priority_inherit	Specifies whether or not this mutex supports priority inheritance. If this value is TX_INHERIT, then priority inheritance is supported. However, if TX_NO_INHERIT is specified, priority inheritance is not supported by this mutex.

Return Values

TX_SUCCESS	(0x00)	Successful mutex creation.
TX_MUTEX_ERROR	(0x1C)	Invalid mutex pointer. Either the pointer is NULL or the mutex is already created.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.
TX_INHERIT_ERROR	(0x1F)	Invalid priority inherit parameter.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Create a mutex to provide protection over a
   common resource.  */
status =  tx_mutex_create(&my_mutex,“my_mutex_name”,
                          TX_NO_INHERIT);

/* If status equals TX_SUCCESS, my_mutex is ready for
   use.  */
```

See Also

tx_mutex_delete, tx_mutex_get, tx_mutex_info_get, tx_mutex_prioritize, tx_mutex_put

tx_mutex_delete

Delete a mutual exclusion mutex

Prototype

```
UINT tx_mutex_delete(TX_MUTEX *mutex_ptr)
```

Description

This service deletes the specified mutex. All threads suspended waiting for the mutex are resumed and given a TX_DELETED return status.

It is the application's responsibility to prevent use of a deleted mutex.

Input Parameters

mutex_ptr	Pointer to a previously created mutex.

Return Values

TX_SUCCESS	(0x00)	Successful mutex deletion.
TX_MUTEX_ERROR	(0x1C)	Invalid mutex pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Delete a mutex. Assume that the mutex
   has already been created.  */
status =  tx_mutex_delete(&my_mutex);

/* If status equals TX_SUCCESS, the mutex is
   deleted.  */
```

See Also

tx_mutex_create, tx_mutex_get, tx_mutex_info_get, tx_mutex_prioritize, tx_mutex_put

tx_mutex_get

Obtain ownership of a mutex

Prototype

```
UINT tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG wait_option)
```

Description

This service attempts to obtain exclusive ownership of the specified mutex. If the calling thread already owns the mutex, an internal counter is incremented and a successful status is returned.

If the mutex is owned by another thread and this thread is higher priority and priority inheritance was specified at mutex create, the lower priority thread's priority will be temporarily raised to that of the calling thread.

Note that the priority of the lower-priority thread owning a mutex with priority-inheritance should never be modified by an external thread during mutex ownership.

Input Parameters

mutex_ptr	Pointer to a previously created mutex.
wait_option	Defines how the service behaves if the mutex is already owned by another thread. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. *This is the only valid option if the service is called from Initialization.*

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the mutex is available.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the mutex.

Return Values

TX_SUCCESS	(0x00)	Successful mutex get operation.
TX_DELETED	(0x01)	Mutex was deleted while thread was suspended.
TX_NOT_AVAILABLE	(0x1D)	Service was unable to get ownership of the mutex.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_MUTEX_ERROR	(0x1C)	Invalid mutex pointer.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization, threads, and timers

Preemption Possible

Yes

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Obtain exclusive ownership of the mutex "my_mutex".
   If the mutex "my_mutex" is not available, suspend until it
   becomes available. */
status =  tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);
```

See Also

tx_mutex_create, tx_mutex_delete, tx_mutex_info_get, tx_mutex_prioritize, tx_mutex_put

tx_mutex_info_get

Retrieve information about a mutex

Prototype

```
UINT tx_mutex_info_get(TX_MUTEX *mutex_ptr, CHAR **name,
                          ULONG *count, TX_THREAD **owner,
                          TX_THREAD **first_suspended,
                          ULONG *suspended_count, TX_MUTEX **next_mutex)
```

Description

This service retrieves information from the specified mutex.

Input Parameters

mutex_ptr	Pointer to mutex control block.
name	Pointer to destination for the pointer to the mutex's name.
count	Pointer to destination for the ownership count of the mutex.
owner	Pointer to destination for the owning thread's pointer.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this mutex.
suspended_count	Pointer to destination for the number of threads currently suspended on this mutex.
next_mutex	Pointer to destination for the pointer of the next created mutex.

Return Values

TX_SUCCESS	(0x00)	Successful mutex information retrieval.
TX_MUTEX_ERROR	(0x1C)	Invalid mutex pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_MUTEX my_mutex;
CHAR *name;
ULONG count;
TX_THREAD *owner;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_MUTEX *next_mutex;
UINT status;


/* Retrieve information about a the previously created
   mutex "my_mutex." */
status =  tx_mutex_info_get(&my_mutex, &name,
                            &count, &owner,
                            &first_suspended, &suspended_count,
                            &next_mutex);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_prioritize, tx_mutex_put

tx_mutex_prioritize

Prioritize mutex suspension list

Prototype

```
UINT tx_mutex_prioritize(TX_MUTEX *mutex_ptr)
```

Description

This service places the highest priority thread suspended for ownership of the mutex at the front of the suspension list. All other threads remain in the same FIFO order they were suspended in.

Input Parameters

mutex_ptr	Pointer to the previously created mutex.

Return Values

TX_SUCCESS	(0x00)	Successful mutex prioritize.
TX_MUTEX_ERROR	(0x1C)	Invalid mutex pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Ensure that the highest priority thread will receive
   ownership of the mutex when it becomes available.  */
status =  tx_mutex_prioritize(&my_mutex);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_mutex_put call that releases ownership of the
   mutex will give ownership to this thread and wake it
   up. */
```

See Also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_info_get, tx_mutex_put

tx_mutex_put

Release ownership of mutex

Prototype

```
UINT tx_mutex_put(TX_MUTEX *mutex_ptr)
```

Description

This service decrements the ownership count of the specified mutex. If the ownership count is zero, the mutex is made available.

If priority inheritance was selected during mutex creation, the priority of the releasing thread will be restored to the priority it had when it originally obtained ownership of the mutex. Any other priority changes made to the releasing thread during ownership of the mutex may be undone.

Input Parameters

mutex_ptr	Pointer to the previously created mutex.

Return Values

TX_SUCCESS	(0x00)	Successful mutex release.
TX_NOT_OWNED	(0x1E)	Mutex is not owned by caller.
TX_MUTEX_ERROR	(0x1C)	Invalid pointer to mutex.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
TX_MUTEX my_mutex;
UINT status;


/* Release ownership of "my_mutex." */
   status =  tx_mutex_put(&my_mutex);

/* If status equals TX_SUCCESS, the mutex ownership
   count has been decremented and if zero, released. */
```

See Also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_info_get, tx_mutex_prioritize

tx_queue_create

Create a message queue

Prototype

```
UINT tx_queue_create(TX_QUEUE *queue_ptr, CHAR *name_ptr,
                          UINT message_size,
                          VOID *queue_start, ULONG queue_size)
```

Description

This service creates a message queue that is typically used for inter-thread communication. The total number of messages is calculated from the specified message size and the total number of bytes in the queue.

If the total number of bytes specified in the queue's memory area is not evenly divisible by the specified message size, the remaining bytes in the memory area are not used.

Input Parameters

queue_ptr	Pointer to a message queue control block.
name_ptr	Pointer to the name of the message queue.
message_size	Specifies the size of each message in the queue. Message sizes range from 1 32-bit word to 16 32-bit words. Valid message size options are defined as follows:

TX_1_ULONG	(0x01)
TX_2_ULONG	(0x02)
TX_4_ULONG	(0x04)
TX_8_ULONG	(0x08)
TX_16_ULONG	(0x10)

queue_start	Starting address of the message queue.
queue_size	Total number of bytes available for the message queue.

Return Values

TX_SUCCESS	(0x00)	Successful message queue creation.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer. Either the pointer is NULL or the queue is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the message queue.
TX_SIZE_ERROR	(0x05)	Size of message queue is invalid.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_QUEUE my_queue;
UINT status;

/* Create a message queue whose total size is 2000 bytes
   starting at address 0x300000. Each message in this
   queue is defined to be 4 32-bit words long.  */
status =  tx_queue_create(&my_queue, "my_queue_name",
                          TX_4_ULONG, (VOID *) 0x300000, 2000);

/* If status equals TX_SUCCESS, my_queue contains room
   for storing 125 messages (2000 bytes/ 16 bytes per
   message).  */
```

See Also

tx_queue_delete, tx_queue_flush, tx_queue_front_send, tx_queue_info_get, tx_queue_prioritize, tx_queue_receive, tx_queue_send

tx_queue_delete

Delete a message queue

Prototype

```
UINT tx_queue_delete(TX_QUEUE *queue_ptr)
```

Description

This service deletes the specified message queue. All threads suspended waiting for a message from this queue are resumed and given a TX_DELETED return status.

It is the application's responsibility to manage the memory area associated with the queue, which is available after this service completes. In addition, the application must prevent use of a deleted queue.

Input Parameters

queue_ptr	Pointer to a previously created message queue.

Return Values

TX_SUCCESS	(0x00)	Successful message queue deletion.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_QUEUE my_queue;
UINT status;

/* Delete entire message queue. Assume that the queue
   has already been created with a call to
   tx_queue_create.  */
status =  tx_queue_delete(&my_queue);

/* If status equals TX_SUCCESS, the message queue is
   deleted.  */
```

See Also

tx_queue_create, tx_queue_flush, tx_queue_front_send, tx_queue_info_get, tx_queue_prioritize, tx_queue_receive, tx_queue_send

tx_queue_flush

Empty messages in a message queue

Prototype

```
UINT tx_queue_flush(TX_QUEUE *queue_ptr)
```

Description

This service deletes all messages stored in the specified message queue. If the queue is full, messages of all suspended threads are discarded. Each suspended thread is then resumed with a return status that indicates the message send was successful. If the queue is empty, this service does nothing.

Input Parameters

queue_ptr	Pointer to a previously created message queue.

Return Values

TX_SUCCESS	(0x00)	Successful message queue flush.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_QUEUE my_queue;
UINT status;

/* Flush out all pending messages in the specified message
   queue. Assume that the queue has already been created
   with a call to tx_queue_create.  */
status =  tx_queue_flush(&my_queue);

/* If status equals TX_SUCCESS, the message queue is
   empty.  */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_front_send, tx_queue_info_get, tx_queue_prioritize, tx_queue_receive, tx_queue_send

tx_queue_front_send

Send a message to the front of queue

Prototype

```
UINT tx_queue_front_send(TX_QUEUE *queue_ptr,
                          VOID *source_ptr, ULONG wait_option)
```

Description

This service sends a message to the front location of the specified message queue. The message is **copied** to the front of the queue from the memory area specified by the source pointer.

Input Parameters

queue_ptr	Pointer to a message queue control block.
source_ptr	Pointer to the message.
wait_option	Defines how the service behaves if the message queue is full. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. *This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.*

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until there is room in the queue.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

Return Values

TX_SUCCESS	(0x00)	Successful sending of message.
TX_DELETED	(0x01)	Message queue was deleted while thread was suspended.
TX_QUEUE_FULL	(0x0B)	Service was unable to send message because the queue was full.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.
TX_PTR_ERROR	(0x03)	Invalid source pointer for message.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Send a message to the front of "my_queue." Return
   immediately, regardless of success. This wait
   option is used for calls from initialization, timers,
   and ISRs.  */
status =  tx_queue_front_send(&my_queue, my_message,
                              TX_NO_WAIT);

/* If status equals TX_SUCCESS, the message is at the front
   of the specified queue. */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush, tx_queue_info_get, tx_queue_prioritize, tx_queue_receive, tx_queue_send

tx_queue_info_get

Retrieve information about a queue

Prototype

```
UINT tx_queue_info_get(TX_QUEUE *queue_ptr, CHAR **name,
        ULONG *enqueued, ULONG *available_storage
        TX_THREAD **first_suspended, ULONG *suspended_count,
        TX_QUEUE **next_queue)
```

Description

This service retrieves information about the specified message queue.

Input Parameters

queue_ptr	Pointer to a previously created message queue.
name	Pointer to destination for the pointer to the queue's name.
enqueued	Pointer to destination for the number of messages currently in the queue.
available_storage	Pointer to destination for the number of messages the queue currently has space for.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this queue.
suspended_count	Pointer to destination for the number of threads currently suspended on this queue.
next_queue	Pointer to destination for the pointer of the next created queue.

Return Values

TX_SUCCESS	(0x00)	Successful queue information get.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_QUEUE my_queue;
CHAR *name;
ULONG enqueued;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_QUEUE *next_queue;
UINT status;

/* Retrieve information about a the previously created
   message queue "my_queue." */
status =  tx_queue_info_get(&my_queue, &name,
                           &enqueued,
                           &first_suspended, &suspended_count,
                           &next_queue);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush, tx_queue_front_send, tx_queue_prioritize, tx_queue_receive, tx_queue_send

tx_queue_prioritize

Prioritize queue suspension list

Prototype

```
UINT tx_queue_prioritize(TX_QUEUE *queue_ptr)
```

Description

This service places the highest priority thread suspended for a message (or to place a message) on this queue at the front of the suspension list. All other threads remain in the same FIFO order they were suspended in.

Input Parameters

queue_ptr	Pointer to a previously created message queue.

Return Values

TX_SUCCESS	(0x00)	Successful queue prioritize.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_QUEUE my_queue;
UINT status;

/* Ensure that the highest priority thread will receive
   the next message placed on this queue.  */
status =  tx_queue_prioritize(&my_queue);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_queue_send or tx_queue_front_send call made
   to this queue will wake up this thread. */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush, tx_queue_front_send, tx_queue_info_get, tx_queue_receive, tx_queue_send

tx_queue_receive

Get a message from message queue

Prototype

```
UINT tx_queue_receive(TX_QUEUE *queue_ptr,
                          VOID *destination_ptr, ULONG wait_option)
```

Description

This service retrieves a message from the specified message queue. The retrieved message is **copied** from the queue into the memory area specified by the destination pointer. That message is then removed from the queue.

*The specified destination memory area must be large enough to hold the message; i.e., the message destination pointed to by **destination_ptr** must be at least as large as the message size for this queue. Otherwise, if the destination is not large enough, memory corruption occurs in the following memory area.*

Input Parameters

queue_ptr	Pointer to a previously created message queue.
destination_ptr	Location of where to copy the message.
wait_option	Defines how the service behaves if the message queue is empty. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a message is available.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for a message.

Return Values

TX_SUCCESS	(0x00)	Successful retrieval of message.
TX_DELETED	(0x01)	Message queue was deleted while thread was suspended.
TX_QUEUE_EMPTY	(0x0A)	Service was unable to retrieve a message because the queue was empty.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.
TX_PTR_ERROR	(0x03)	Invalid destination pointer for message.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Retrieve a message from "my_queue." If the queue is
   empty, suspend until a message is present. Note that
   this suspension is only possible from application
   threads.  */
status =  tx_queue_receive(&my_queue, my_message,
                           TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the message is in
   "my_message."  */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush, tx_queue_front_send, tx_queue_info_get, tx_queue_prioritize, tx_queue_send

tx_queue_send

Send a message to message queue

Prototype

```
UINT tx_queue_send(TX_QUEUE *queue_ptr,
                           VOID *source_ptr, ULONG wait_option)
```

Description

This service sends a message to the specified message queue. The sent message is **copied** to the queue from the memory area specified by the source pointer.

Input Parameters

queue_ptr	Pointer to a previously created message queue.
source_ptr	Pointer to the message.
wait_option	Defines how the service behaves if the message queue is full. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. *This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR.*

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until there is room in the queue.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

Return Values

TX_SUCCESS	(0x00)	Successful sending of message.
TX_DELETED	(0x01)	Message queue was deleted while thread was suspended.
TX_QUEUE_FULL	(0x0B)	Service was unable to send message because the queue was full.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_QUEUE_ERROR	(0x09)	Invalid message queue pointer.
TX_PTR_ERROR	(0x03)	Invalid source pointer for message.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_QUEUE my_queue;
UINT status;
ULONG my_message[4];

/* Send a message to "my_queue." Return immediately,
   regardless of success. This wait option is used for
   calls from initialization, timers, and ISRs.  */
status =  tx_queue_send(&my_queue, my_message, TX_NO_WAIT);

/* If status equals TX_SUCCESS, the message is in the
   queue.  */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush, tx_queue_front_send, tx_queue_info_get, tx_queue_prioritize, tx_queue_receive

tx_semaphore_create

Create a counting semaphore

Prototype

```
UINT tx_semaphore_create(TX_SEMAPHORE *semaphore_ptr,
                          CHAR *name_ptr, ULONG initial_count)
```

Description

This service creates a counting semaphore for inter-thread synchronization. The initial semaphore count is specified as an input parameter.

Input Parameters

semaphore_ptr	Pointer to a semaphore control block.
name_ptr	Pointer to the name of the semaphore.
initial_count	Specifies the initial count for this semaphore. Legal values range from 0x00000000 through 0xFFFFFFFF.

Return Values

TX_SUCCESS	(0x00)	Successful semaphore creation.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid semaphore pointer. Either the pointer is NULL or the semaphore is already created.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Create a counting semaphore whose initial value is 1.
   This is typically the technique used to make a binary
   semaphore. Binary semaphores are used to provide
   protection over a common resource.  */
status =  tx_semaphore_create(&my_semaphore,
                              "my_semaphore_name", 1);

/* If status equals TX_SUCCESS, my_semaphore is ready for
   use.  */
```

See Also

tx_semaphore_delete, tx_semaphore_get, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_delete

Delete a counting semaphore

Prototype

```
UINT tx_semaphore_delete(TX_SEMAPHORE *semaphore_ptr)
```

Description

This service deletes the specified counting semaphore. All threads suspended waiting for a semaphore instance are resumed and given a TX_DELETED return status.

It is the application's responsibility to prevent use of a deleted semaphore.

Input Parameters

semaphore_ptr	Pointer to a previously created semaphore.

Return Values

TX_SUCCESS	(0x00)	Successful counting semaphore deletion.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid counting semaphore pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Delete counting semaphore. Assume that the counting
   semaphore has already been created.  */
status =  tx_semaphore_delete(&my_semaphore);

/* If status equals TX_SUCCESS, the counting semaphore is
   deleted.  */
```

See Also

tx_semaphore_create, tx_semaphore_get, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_get

Get instance from counting semaphore

Prototype

```
UINT tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr,
                          ULONG wait_option)
```

Description

This service retrieves an instance (a single count) from the specified counting semaphore. As a result, the specified semaphore's count is decreased by one.

Input Parameters

semaphore_ptr Pointer to a previously created counting semaphore.

wait_option Defines how the service behaves if there are no instances of the semaphore available; i.e., the semaphore count is zero. The wait options are defined as follows:

TX_NO_WAIT	(0x00000000)
TX_WAIT_FOREVER	(0xFFFFFFFF)
timeout value	(0x00000001 through 0xFFFFFFFE)

Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. *This is the only valid option if the service is called from a non-thread; e.g., initialization, timer, or ISR.*

Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until a semaphore instance is available.

Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for a semaphore instance.

Return Values

TX_SUCCESS	(0x00)	Successful retrieval of a semaphore instance.
TX_DELETED	(0x01)	Counting semaphore was deleted while thread was suspended.
TX_NO_INSTANCE	(0x0D)	Service was unable to retrieve an instance of the counting semaphore (semaphore count is zero).
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid counting semaphore pointer.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Get a semaphore instance from the semaphore
   "my_semaphore." If the semaphore count is zero,
   suspend until an instance becomes available.
   Note that this suspension is only possible from
   application threads.  */
status =  tx_semaphore_get(&my_semaphore, TX_WAIT_FOREVER);

/* If status equals TX_SUCCESS, the thread has obtained
   an instance of the semaphore.  */
```

See Also

tx_semaphore_create, tx_semahore_delete, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_info_get

Retrieve information about a semaphore

Prototype

```
UINT tx_semaphore_info_get(TX_SEMAPHORE *semaphore_ptr,
                           CHAR **name, ULONG *current_value,
                           TX_THREAD **first_suspended,
                           ULONG *suspended_count,
                           TX_SEMAPHORE **next_semaphore)
```

Description

This service retrieves information about the specified semaphore.

Input Parameters

semaphore_ptr	Pointer to semaphore control block.
name	Pointer to destination for the pointer to the semaphore's name.
current_value	Pointer to destination for the current semaphore's count.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this semaphore.
suspended_count	Pointer to destination for the number of threads currently suspended on this semaphore.
next_semaphore	Pointer to destination for the pointer of the next created semaphore.

Return Values

TX_SUCCESS	(0x00)	Successful semaphore information retrieval.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid semaphore pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_SEMAPHORE my_semaphore;
CHAR *name;
ULONG current_value;
TX_THREAD *first_suspended;
ULONG suspended_count;
TX_SEMAPHORE *next_semaphore;
UINT status;

/* Retrieve information about a the previously created
   semaphore "my_semaphore." */
status =  tx_semaphore_info_get(&my_semaphore, &name,
                           &current_value,
                           &first_suspended, &suspended_count,
                           &next_semaphore);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_prioritize

Prioritize semaphore suspension list

Prototype

```
UINT tx_semaphore_prioritize(TX_SEMAPHORE *semaphore_ptr)
```

Description

This service places the highest priority thread suspended for an instance of the semaphore at the front of the suspension list. All other threads remain in the same FIFO order they were suspended in.

Input Parameters

semaphore_ptr	Pointer to a previously created semaphore.

Return Values

TX_SUCCESS	(0x00)	Successful semaphore prioritize.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid counting semaphore pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Ensure that the highest priority thread will receive
   the next instance of this semaphore.  */
status =  tx_semaphore_prioritize(&my_semaphore);

/* If status equals TX_SUCCESS, the highest priority
   suspended thread is at the front of the list. The
   next tx_semaphore_put call made to this queue will
   wake up this thread. */
```

See Also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_get, tx_semaphore_info_get, tx_semaphore_put

tx_semaphore_put

Place an instance in counting semaphore

Prototype

```
UINT tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr)
```

Description

This service puts an instance into the specified counting semaphore, which in reality increments the counting semaphore by one.

If this service is called when the semaphore is all ones (OxFFFFFFFF), the new put operation will cause the semaphore to be reset to zero.

Input Parameters

semaphore_ptr	Pointer to the previously created counting semaphore control block.

Return Values

TX_SUCCESS	(0x00)	Successful semaphore put.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid pointer to counting semaphore.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_SEMAPHORE my_semaphore;
UINT status;


/* Increment the counting semaphore "my_semaphore." */
status =  tx_semaphore_put(&my_semaphore);

/* If status equals TX_SUCCESS, the semaphore count has
   been incremented. Of course, if a thread was waiting,
   it was given the semaphore instance and resumed.  */
```

See Also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_get

tx_thread_create

Create an application thread

Prototype

```
UINT tx_thread_create(TX_THREAD *thread_ptr,
                          CHAR *name_ptr, VOID (*entry_function)(ULONG),
                          ULONG entry_input, VOID *stack_start,
                          ULONG stack_size, UINT priority,
                          UINT preempt_threshold, ULONG time_slice,
                          UINT auto_start)
```

Description

This service creates an application thread that starts execution at the specified task entry function. The stack, priority, preemption-threshold, and time-slice are among the attributes specified by the input parameters. In addition, the initial execution state of the thread is also specified.

Input Parameters

thread_ptr	Pointer to a thread control block.
name_ptr	Pointer to the name of the thread.
entry_function	Specifies the initial C function for thread execution. When a thread returns from this entry function, it is placed in a *completed* state and suspended indefinitely.
entry_input	A 32-bit value that is passed to the thread's entry function when it first executes. The use for this input is determined exclusively by the application.
stack_start	Starting address of the stack's memory area.
stack_size	Number bytes in the stack memory area. The thread's stack area must be large enough to handle its worst-case function call nesting and local variable usage.
priority	Numerical priority of thread. Legal values range from 0 through 31, where a value of 0 represents the highest priority.

preempt_threshold	Highest priority level (0-31) of disabled preemption. Only priorities higher than this level are allowed to preempt this thread. This value must be less than or equal to the specified priority. A value equal to the thread priority disables preemption-threshold.
time_slice	Number of timer-ticks this thread is allowed to run before other ready threads of the same priority are given a chance to run. Note that using preemption-threshold disables time-slicing. Legal time-slices selections range from 1 through 0xFFFFFFFF. A value of **TX_NO_TIME_SLICE** (a value of 0) disables time-slicing of this thread.
	Using time-slicing results in a slight amount of system overhead. Since time-slicing is only useful in cases where multiple threads share the same priority, threads having a unique priority should not be assigned a time-slice.
auto_start	Specifies whether the thread starts immediately or is placed in a suspended state. Legal options are **TX_AUTO_START** (0x01) and **TX_DONT_START** (0x00). If TX_DONT_START is specified, the application must later call tx_thread_resume in order for the thread to run.

Return Values

TX_SUCCESS	(0x00)	Successful thread creation.
TX_THREAD_ERROR	(0x0E)	Invalid thread control pointer. Either the pointer is NULL or the thread is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the entry point or the stack area is invalid, usually NULL.
TX_SIZE_ERROR	(0x05)	Size of stack area is invalid. Threads must have at least **TX_MINIMUM_STACK** bytes to execute.
TX_PRIORITY_ERROR	(0x0F)	Invalid thread priority, which is a value outside the range of 0-31.
TX_THRESH_ERROR	(0x18)	Invalid preemption-threshold specified. This value must be a valid priority less than or equal to the initial priority of the thread.
TX_START_ERROR	(0x10)	Invalid auto-start selection.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

/* Create a thread of priority 15 whose entry point is
   "my_thread_entry". This thread’s stack area is 1000
   bytes in size, starting at address 0x400000. The
   preemption-threshold is setup to allow preemption at
   priorities above 15. Time-slicing is disabled. This
   thread is automatically put into a ready condition.  */
status =  tx_thread_create(&my_thread, "my_thread_name",
                          my_thread_entry, 0x1234,
                          (VOID *) 0x400000, 1000,
                          15, 15, TX_NO_TIME_SLICE,
                          TX_AUTO_START);

/* If status equals TX_SUCCESS, my_thread is ready
   for execution!  */

...

/* Thread’s entry function. When "my_thread" actually
   begins execution, control is transferred to this
   function. */
VOID my_thread_entry (ULONG initial_input)
{

    /* When we get here, the value of initial_input is
       0x1234. See how this was specified during
       creation.  */

    /* The real work of the thread, including calls to
       other function should be called from here!  */

    /* When the this function returns, the corresponding
       thread is placed into a “completed” state and
       suspended.  */
}
```

See Also

tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_delete

Delete an application thread

Prototype

```
UINT tx_thread_delete(TX_THREAD *thread_ptr)
```

Description

This service deletes the specified application thread. Since the specified thread must be in a terminated or completed state, this service cannot be called from a thread attempting to delete itself.

It is the application's responsibility to manage the memory area associated with the thread's stack, which is available after this service completes. In addition, the application must prevent use of a deleted thread.

Input Parameters

thread_ptr	Pointer to a previously created application thread.

Return Values

TX_SUCCESS	(0x00)	Successful thread deletion.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_DELETE_ERROR	(0x11)	Specified thread is not in a terminated or completed state.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

No

Example

```
TX_THREAD my_thread;
UINT status;

/* Delete an application thread whose control block is
   "my_thread". Assume that the thread has already been
   created with a call to tx_thread_create.  */
status =  tx_thread_delete(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
   deleted.  */
```

See Also

tx_thread_create, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_identify

Retrieves pointer to currently executing thread

Prototype

```
TX_THREAD* tx_thread_identify(VOID)
```

Description

This service returns a pointer to the currently executing thread. If no thread is executing, this service returns a null pointer.

If this service is called from an ISR, the return value represents the thread running prior to the executing interrupt handler.

Input Parameters

None

Return Values

thread pointer	Pointer to the currently executing thread. If no thread is executing, the return value is TX_NULL.

Allowed From

Threads and ISRs

Preemption Possible

No

Example

```
TX_THREAD *my_thread_ptr;

/* Find out who we are!  */
my_thread_ptr =  tx_thread_identify();

/* If my_thread_ptr is non-null, we are currently executing
   from that thread or an ISR that interrupted that thread.
  Otherwise, this service was called
   from an ISR when no thread was running when the
   interrupt occurred.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_info_get

Retrieve information about a thread

Prototype

```
UINT tx_thread_info_get(TX_THREAD *thread_ptr, CHAR **name,
                          UINT *state, ULONG *run_count,
                          UINT *priority,
                          UINT *preemption_threshold,
                          ULONG *time_slice,
                          TX_THREAD **next_thread,
                          TX_THREAD **suspended_thread)
```

Description

This service retrieves information about the specified thread.

Input Parameters

thread_ptr	Pointer to thread control block.
name	Pointer to destination for the pointer to the thread's name.
state	Pointer to destination for the thread's current execution state. Possible values are as follows:

TX_READY	(0x00)
TX_COMPLETED	(0x01)
TX_TERMINATED	(0x02)
TX_SUSPENDED	(0x03)
TX_SLEEP	(0x04)
TX_QUEUE_SUSP	(0x05)
TX_SEMAPHORE_SUSP	(0x06)
TX_EVENT_FLAG	(0x07)
TX_BLOCK_MEMORY	(0x08)
TX_BYTE_MEMORY	(0x09)
TX_MUTEX_SUSP	(0x0D)
TX_IO_DRIVER	(0x0A)

run_count	Pointer to destination for the thread's run count.
priority	Pointer to destination for the thread's priority.
preemption_threshold	Pointer to destination for the thread's preemption-threshold.
time_slice	Pointer to destination for the thread's time-slice.

next_thread	Pointer to destination for next created thread pointer.
suspended_thread	Pointer to destination for pointer to next thread in suspension list.

Return Values

TX_SUCCESS	(0x00)	Successful thread information retrieval.
TX_THREAD_ERROR	(0x0E)	Invalid thread control pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_THREAD my_thread;
CHAR *name;
UINT state;
ULONG run_count;
UINT priority;
UINT preemption_threshold;
UINT time_slice;
TX_THREAD *next_thread;
TX_THREAD *suspended_thread;
UINT status;

/* Retrieve information about a the previously created
   thread "my_thread." */
status =  tx_thread_info_get(&my_thread, &name,
                   &state, &run_count,
                   &priority, &preemption_threshold,
                   &time_slice, &next_thread,&suspended_thread);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_preemption_change

Change preemption-threshold of application thread

Prototype

```
UINT tx_thread_preemption_change(TX_THREAD *thread_ptr,
                           UINT new_threshold, UINT *old_threshold)
```

Description

This service changes the preemption-threshold of the specified thread. The preemption-threshold prevents preemption of the specified thread by threads equal to or less than the preemption-threshold value.

Note that using preemption-threshold disables time-slicing for the specified thread.

Input Parameters

thread_ptr	Pointer to a previously created application thread.
new_threshold	New preemption-threshold priority level (0-31).
old_threshold	Pointer to a location to return the previous preemption-threshold.

Return Values

TX_SUCCESS	(0x00)	Successful preemption-threshold change.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_THRESH_ERROR	(0x18)	Specified new preemption-threshold is not a valid thread priority (a value other than 0-31) or is greater than (lower priority) than the current thread priority.
TX_PTR_ERROR	(0x03)	Invalid pointer to previous preemption-threshold storage location.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT my_old_threshold;
UINT status;

/* Disable all preemption of the specified thread. The
   current preemption-threshold is returned in
   "my_old_threshold". Assume that "my_thread" has
   already been created.  */
status =  tx_thread_preemption_change(&my_thread,
                           0, &my_old_threshold);

/* If status equals TX_SUCCESS, the application thread is
   non-preemptable by another thread. Note that ISRs are
   not prevented by preemption disabling.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_priority_change

Change priority of an application thread

Prototype

```
UINT tx_thread_priority_change(TX_THREAD *thread_ptr,
                               UINT new_priority, UINT *old_priority)
```

Description

This service changes the priority of the specified thread. Valid priorities range from 0 through 31, where 0 represents the highest priority level.

The preemption-threshold of the specified thread is automatically set to the new priority. If a new threshold is desired, the ***tx_thread_preemption_change*** *service must be used after this call.*

Input Parameters

thread_ptr	Pointer to a previously created application thread.
new_priority	New thread priority level (0-31).
old_priority	Pointer to a location to return the thread's previous priority.

Return Values

TX_SUCCESS	(0x00)	Successful priority change.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_PRIORITY_ERROR	(0x0F)	Specified new priority is not valid (a value other than 0-31).
TX_PTR_ERROR	(0x03)	Invalid pointer to previous priority storage location.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT my_old_priority;
UINT status;

/* Change the thread represented by "my_thread" to priority
   0.  */
status =  tx_thread_priority_change(&my_thread,
                           0, &my_old_priority);

/* If status equals TX_SUCCESS, the application thread is
   now at the highest priority level in the system.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_relinquish

Relinquish control to other application threads

Prototype

```
VOID  tx_thread_relinquish(VOID)
```

Description

This service relinquishes processor control to other ready-to-run threads at the same or higher priority.

Input Parameters

VOID

Return Values

VOID

Allowed From

Threads

Preemption Possible

Yes

Example

```
ULONG run_counter_1 =  0;
ULONG run_counter_2 =  0;

/* Example of two threads relinquishing control to
   each other in an infinite loop. Assume that
   both of these threads are ready and have the same
   priority. The run counters will always stay within one
   of each other.  */

VOID  my_first_thread(ULONG thread_input)
{

    /* Endless loop of relinquish.  */
    while(1)
    {

        /* Increment the run counter.  */
        run_counter_1++;

        /* Relinquish control to other thread.  */
        tx_thread_relinquish();
    }
}

VOID  my_second_thread(ULONG thread_input)
{

    /* Endless loop of relinquish.  */
    while(1)
    {

        /* Increment the run counter.  */
        run_counter_2++;

        /* Relinquish control to other thread.  */
        tx_thread_relinquish();
    }
}
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_resume

Resume suspended application thread

Prototype

```
UINT tx_thread_resume(TX_THREAD *thread_ptr)
```

Description

This service resumes or prepares for execution a thread that was previously suspended by a ***tx_thread_suspend*** call. In addition, this service resumes threads that were created without an automatic start.

Input Parameters

thread_ptr	Pointer to a suspended application thread.

Return Values

TX_SUCCESS	(0x00)	Successful thread resume.
TX_SUSPEND_LIFTED	(0x19)	Previously set delayed suspension was lifted.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_RESUME_ERROR	(0x12)	Specified thread is not suspended or was previously suspended by a service other than ***tx_thread_suspend***.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

/* Resume the thread represented by "my_thread".  */
status =  tx_thread_resume(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
   now ready to execute.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_sleep

Suspended current thread for specified time

Prototype

```
UINT tx_thread_sleep(ULONG timer_ticks)
```

Description

This service causes the calling thread to suspend for the specified number of timer ticks. The amount of physical time associated with a timer tick is application specific. This service can only be called only from an application thread.

Input Parameters

timer_ticks	The number of timer ticks to suspend the calling application thread, ranging from 0 through 0xFFFFFFFF. If 0 is specified, the service returns immediately.

Return Values

TX_SUCCESS	(0x00)	Successful thread sleep.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_CALLER_ERROR	(0x13)	Service called from a non-thread.

Allowed From

Threads

Preemption Possible

Yes

Example

```
UINT status;

/* Make the calling thread sleep for 100
   timer-ticks.  */
status =  tx_thread_sleep(100);

/* If status equals TX_SUCCESS, the currently running
   application thread slept for the specified number of
   timer-ticks.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_suspend

Suspend an application thread

Prototype

```
UINT tx_thread_suspend(TX_THREAD *thread_ptr)
```

Description

This service suspends the specified application thread. A thread may call this service to suspend itself.

If the specified thread is already suspended for another reason, this suspension is held internally until the prior suspension is lifted. When that happens, this unconditional suspension of the specified thread is performed. Further unconditional suspension requests have no effect.

Once suspended, the thread must be resumed by ***tx_thread_resume*** in order to execute again.

Input Parameters

thread_ptr	Pointer to an application thread.

Return Values

TX_SUCCESS	(0x00)	Successful thread suspend.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_SUSPEND_ERROR	(0x14)	Specified thread is in a terminated or completed state.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

/* Suspend the thread represented by "my_thread".  */
status =  tx_thread_suspend(&my_thread);

/* If status equals TX_SUCCESS, the application thread is
   unconditionally suspended.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_terminate

Terminates an application thread

Prototype

```
UINT tx_thread_terminate(TX_THREAD *thread_ptr)
```

Description

This service terminates the specified application thread regardless of whether the thread is suspended or not. A thread may call this service to terminate itself.

Once terminated, the thread must be deleted and re-created in order for it to execute again.

Note that time-slicing is disabled when using preemption-threshold to prevent preemption of higher-priority threads.

Input Parameters

thread_ptr	Pointer to application thread.

Return Values

TX_SUCCESS	(0x00)	Successful thread terminate.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

/* Terminate the thread represented by "my_thread".  */
status =  tx_thread_terminate(&my_thread);

/* If status equals TX_SUCCESS, the thread is terminated
   and cannot execute again until it is deleted and
   re-created.  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_time_slice_change

Changes time-slice of application thread

Prototype

```
UINT tx_thread_time_slice_change(TX_THREAD *thread_ptr,
                                 ULONG new_time_slice, ULONG *old_time_slice)
```

Description

This service changes the time-slice of the specified application thread. Selecting a time-slice for a thread insures that it won’t execute more than the specified number of timer ticks before other threads of the same or higher priorities have a chance to execute.

Note that using preemption-threshold disables time-slicing for the specified thread.

Input Parameters

thread_ptr	Pointer to application thread.
new_time_slice	New time slice value. Legal values include TX_NO_TIME_SLICE and numeric values from 1 through 0xFFFFFFFF.
old_time_slice	Pointer to location for storing the previous time-slice value of the specified thread.

Return Values

TX_SUCCESS	(0x00)	Successful time-slice chance.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer to previous time-slice storage location.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

No

Example

```
TX_THREAD    my_thread;
ULONG        my_old_time_slice;
UINT         status;

/* Change the time-slice of the thread associated with
   "my_thread" to 20. This will mean that "my_thread"
   can only run for 20 timer-ticks consecutively before
   other threads of equal or higher priority get a chance
   to run.  */
status =  tx_thread_time_slice_change(&my_thread, 20,
                           &my_old_time_slice);

/* If status equals TX_SUCCESS, the thread’s time-slice
   has been changed to 20 and the previous time-slice is
   in “my_old_time_slice.”  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_wait_abort

tx_thread_wait_abort

Abort suspension of specified thread

Prototype

```
UINT tx_thread_wait_abort(TX_THREAD *thread_ptr)
```

Description

This service aborts sleep or any other object suspension of the specified thread. If the wait is aborted, a TX_WAIT_ABORTED value is returned from the service that the thread was waiting on.

Note that this service does not release pure suspension that is made by the tx_thread_suspend service.

Input Parameters

thread_ptr	Pointer to a previously created application thread.

Return Values

TX_SUCCESS	(0x00)	Successful thread wait abort.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_WAIT_ABORT_ERROR	(0x1B)	Specified thread is not in a waiting state.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

/* Abort the suspension condition of "my_thread." */
status =  tx_thread_wait_abort(&my_thread);

/* If status equals TX_SUCCESS, the thread is now ready
   again, with a return value showing its suspension
   was aborted (TX_WAIT_ABORTED).  */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_identify, tx_thread_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change

tx_time_get

Retrieves the current time

Prototype

```
ULONG tx_time_get(VOID)
```

Description

This service returns the contents of the internal system clock. Each timer-tick increases the internal system clock by one. The system clock is set to zero during initialization and can be changed to a specific value by the service ***tx_time_set***.

i *The actual time each timer-tick represents is application specific.*

Input Parameters

None

Return Values

system clock ticks	Value of the internal, free running, system clock.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
ULONG current_time;

/* Pickup the current system time, in timer-ticks.  */
current_time =  tx_time_get();


/* Current time now contains a copy of the internal system
   clock.  */
```

See Also

tx_time_set

tx_time_set

Sets the current time

Prototype

```
VOID tx_time_set(ULONG new_time)
```

Description

This service sets the internal system clock to the specified value. Each timer-tick increases the internal system clock by one.

The actual time each timer-tick represents is application specific.

Input Parameters

new_time	New time to put in the system clock, legal values range from 0 through 0xFFFFFFFF.

Return Values

None

Allowed From

Threads, timers, and ISRs

Preemption Possible

No

Example

```
/* Set the internal system time to 0x1234.  */
tx_time_set(0x1234);

/* Current time now contains 0x1234 until the next timer
   interrupt. */
```

See Also

tx_time_get

tx_timer_activate

Activate an application timer

Prototype

```
UINT tx_timer_activate(TX_TIMER *timer_ptr)
```

Description

This service activates the specified application timer. The expiration routines of timers that expire at the same time are executed in the order they were activated.

Input Parameters

timer_ptr	Pointer to a previously created application timer.

Return Values

TX_SUCCESS	(0x00)	Successful application timer activation.
TX_TIMER_ERROR	(0x15)	Invalid application timer pointer.
TX_ACTIVATE_ERROR	(0x17)	Timer was already active.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER my_timer;
UINT status;

/* Activate an application timer. Assume that the
   application timer has already been created.  */
status =  tx_timer_activate(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
   now active.  */
```

See Also

tx_timer_change, tx_timer_create, tx_timer_deactivate, tx_timer_delete, tx_timer_info_get

tx_timer_change

Change an application timer

Prototype

```
UINT tx_timer_change(TX_TIMER *timer_ptr,
                          ULONG initial_ticks, ULONG reschedule_ticks)
```

Description

This service changes the expiration characteristics of the specified application timer. The timer must be deactivated prior to calling this service.

*A call to the **tx_timer_activate** service is required after this service in order to start the timer again.*

Input Parameters

timer_ptr	Pointer to a timer control block.
initial_ticks	Specifies the initial number of ticks for timer expiration. Legal values range from 1 through 0xFFFFFFFF.
reschedule_ticks	Specifies the number of ticks for all timer expirations after the first. A zero for this parameter makes the timer a *one-shot* timer. Otherwise, for periodic timers, legal values range from 1 through 0xFFFFFFFF.

Return Values

TX_SUCCESS	(0x00)	Successful application timer change.
TX_TIMER_ERROR	(0x15)	Invalid application timer pointer.
TX_TICK_ERROR	(0x16)	Invalid value (a zero) supplied for initial ticks.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER my_timer;
UINT status;

/* Change a previously created and now deactivated timer
   to expire every 50 timer ticks, including the initial
   expiration.  */
status =  tx_timer_change(&my_timer,50, 50);

/* If status equals TX_SUCCESS, the specified timer is
   changed to expire every 50 ticks.  */

/* Activate the specified timer to get it started again.  */
   status =  tx_timer_activate(&my_timer);
```

See Also

tx_timer_activate, tx_timer_create, tx_timer_deactivate, tx_timer_delete, tx_timer_info_get

tx_timer_create

Create an application timer

Prototype

```
UINT tx_timer_create(TX_TIMER *timer_ptr, CHAR *name_ptr,
                          VOID (*expiration_function)(ULONG),
                          ULONG expiration_input, ULONG initial_ticks,
                          ULONG reschedule_ticks, UINT auto_activate)
```

Description

This service creates an application timer with the specified expiration function and periodic.

Input Parameters

timer_ptr	Pointer to a timer control block
name_ptr	Pointer to the name of the timer.
expiration_function	Application function to call when the timer expires.
expiration_input	Input to pass to expiration function when timer expires.
initial_ticks	Specifies the initial number of ticks for timer expiration. Legal values range from 1 through 0xFFFFFFFF.
reschedule_ticks	Specifies the number of ticks for all timer expirations after the first. A zero for this parameter makes the timer a *one-shot* timer. Otherwise, for periodic timers, legal values range from 1 through 0xFFFFFFFF.
auto_activate	Determines if the timer is automatically activated during creation. If this value is **TX_AUTO_ACTIVATE** (0x01) the timer is made active. Otherwise, if the value **TX_NO_ACTIVATE** (0x00) is selected, the timer is created in a non-active state. In this case, a subsequent ***tx_timer_activate*** service call is necessary to get the timer actually started.

Return Values

TX_SUCCESS	(0x00)	Successful application timer creation.
TX_TIMER_ERROR	(0x15)	Invalid application timer pointer. Either the pointer is NULL or the timer is already created.
TX_TICK_ERROR	(0x16)	Invalid value (a zero) supplied for initial ticks.
TX_ACTIVATE_ERROR	(0x17)	Invalid activation selected.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_TIMER my_timer;
UINT status;

/* Create an application timer that executes
   "my_timer_function" after 100 ticks initially and then
   after every 25 ticks. This timer is specified to start
   immediately!  */
status =  tx_timer_create(&my_timer,"my_timer_name",
                          my_timer_function, 0x1234, 100, 25,
                          TX_AUTO_ACTIVATE);

/* If status equals TX_SUCCESS, my_timer_function will
   be called 100 timer ticks later and then called every
   25 timer ticks. Note that the value 0x1234 is passed to
   my_timer_function every time it is called.  */
```

See Also

tx_timer_activate, tx_timer_change, tx_timer_deactivate, tx_timer_delete, tx_timer_info_get

tx_timer_deactivate

Deactivate an application timer

Prototype

```
UINT tx_timer_deactivate(TX_TIMER *timer_ptr)
```

Description

This service deactivates the specified application timer. If the timer is already deactivated, this service has no effect.

Input Parameters

timer_ptr	Pointer to a previously created application timer.

Return Values

TX_SUCCESS	(0x00)	Successful application timer deactivation.
TX_TIMER_ERROR	(0x15)	Invalid application timer pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER my_timer;
UINT status;

/* Deactivate an application timer. Assume that the
   application timer has already been created.  */
status =  tx_timer_deactivate(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
   now deactivated.  */
```

See Also

tx_timer_activate, tx_timer_change, tx_timer_create, tx_timer_delete, tx_timer_info_get

tx_timer_delete

Delete an application timer

Prototype

```
UINT tx_timer_delete(TX_TIMER *timer_ptr)
```

Description

This service deletes the specified application timer.

It is the application's responsibility to prevent use of a deleted timer.

Input Parameters

timer_ptr	Pointer to a previously created application timer.

Return Values

TX_SUCCESS	(0x00)	Successful application timer deletion.
TX_TIMER_ERROR	(0x15)	Invalid application timer pointer.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads

Preemption Possible

No

Example

```
TX_TIMER my_timer;
UINT status;

/* Delete application timer. Assume that the application
   timer has already been created.  */
status =  tx_timer_delete(&my_timer);

/* If status equals TX_SUCCESS, the application timer is
   deleted.  */
```

See Also

tx_timer_activate, tx_timer_change, tx_timer_create, tx_timer_deactivate, tx_timer_info_get

tx_timer_info_get

Retrieve information about an application timer

Prototype

```
UINT tx_timer_info_get(TX_TIMER *timer_ptr, CHAR **name,
                          UINT *active, ULONG *remaining_ticks,
                          ULONG *reschedule_ticks,
                          TX_TIMER **next_timer)
```

Description

This service retrieves information about the specified application timer.

Input Parameters

timer_ptr	Pointer to a previously created application timer.
name	Pointer to destination for the pointer to the timer's name.
active	Pointer to destination for the timer active indication. If the timer is inactive or this service is called from the timer itself, a TX_FALSE value is returned. Otherwise, if the timer is active, a TX_TRUE value is returned.
remaining_ticks	Pointer to destination for the number of timer ticks left before the timer expires.
reschedule_ticks	Pointer to destination for the number of timer ticks that will be used to automatically reschedule this timer. If the value is zero, then the timer is a one-shot and won't be rescheduled.
next_timer	Pointer to destination for the pointer of the next created application timer.

Return Values

TX_SUCCESS	(0x00)	Successful timer information retrieval.
TX_TIMER_ERROR	(0x15)	Invalid application timer pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer (NULL) for any destination pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER my_timer;
CHAR *name;
UINT active;
ULONG remaining_ticks;
ULONG reschedule_ticks;
TX_TIMER *next_timer;
UINT status;

/* Retrieve information about a the previously created
   application timer "my_timer." */
status =  tx_timer_info_get(&my_timer, &name,
                       &active,&remaining_ticks,
                       &reschedule_ticks,
                       &next_timer);

/* If status equals TX_SUCCESS, the information requested is
   valid.  */
```

See Also

tx_timer_activate, tx_timer_change, tx_timer_create, tx_timer_deactivate, tx_timer_delete, tx_timer_info_get

CHAPTER 5

I/O Drivers for ThreadX

This chapter contains a description of I/O drivers for ThreadX. The information presented in this chapter is designed to help developers write application specific drivers. The following lists the I/O driver topics covered in this chapter:

I/O Driver Introduction

Communication with the external environment is an important component of most embedded applications. This communication is accomplished through hardware devices that are accessible to the embedded application software. The software components responsible for managing such devices are commonly called *I/O Drivers*.

I/O drivers in embedded, real-time systems are inherently application dependent. This is true for two principal reasons: the vast diversity of target hardware and the equally vast performance requirements imposed on real-time applications. Because of this, it is virtually impossible to provide a common set of drivers that will meet the requirements of every application. For these reasons, the information in this chapter is designed to help users customize *off-the-shelf* ThreadX I/O drivers and write their own specific drivers.

Driver Functions

ThreadX I/O drivers are composed of eight basic functional areas, as follows:

Driver Initialization
Driver Control
Driver Access
Driver Input
Driver Output
Driver Interrupts
Driver Status
Driver Termination

With the exception of initialization, each driver functional area is optional. Furthermore, the exact processing in each area is specific to the I/O driver.

Driver Initialization

This functional area is responsible for initialization of the actual hardware device and the internal data structures of the driver. Calling other driver services is not allowed until initialization is complete.

*The driver's initialization function component is typically called from the **tx_application_define** function or from an initialization thread.*

Driver Control

After the driver is initialized and ready for operation, this functional area is responsible for run-time control. Typically, run-time control consists of making changes to the underlying hardware device. Examples include changing the baud rate of a serial device or seeking a new sector on a disk.

Driver Access

Some I/O drivers are called only from a single application thread. In such cases, this functional area is not needed. However, in applications where multiple threads need simultaneous driver access, their interaction must be controlled by adding assign/ release facilities in the I/O driver. Alternatively, the application may use a semaphore to control driver access and avoid extra overhead and complication inside the driver.

Driver Input

This functional area is responsible for all device input. The principle issues associated with driver input usually involve how the input is buffered and how threads wait for such input.

Driver Output

This functional area is responsible for all device output. The principle issues associated with driver output usually involve how the output is buffered and how threads wait to perform output.

Driver Interrupts

Most real-time systems rely on hardware interrupts to notify the driver of device input, output, control, and error events. Interrupts provide a guaranteed response time to such external events. Instead of interrupts, the driver software may periodically check the external hardware for such events. This technique is called *polling*. It is less real-time than interrupts, but polling may make sense for some less real-time applications.

Driver Status

This function area is responsible for providing run-time status and statistics associated with the driver operation. Information managed by this function area typically includes the following:

Current device status
Input bytes
Output bytes
I/O error counts

Driver Termination

This functional area is optional. It is only required if the driver and/or the physical hardware device need to be shut down. After terminated, the driver must not be called again until it is re-initialized.

Simple Driver Example

An example is the best way to describe an I/O driver. In this example, the driver assumes a simple serial hardware device with a configuration register, an input register, and an output register. This simple driver example illustrates the initialization, input, output, and interrupt functional areas.

Simple Driver Initialization

The ***tx_sdriver_initialize*** function of the simple driver creates two counting semaphores that are used to manage the driver's input and output operation. The input semaphore is set by the input ISR when a character is received by the serial hardware device. Because of this, the input semaphore is created with an initial count of zero.

Conversely, the output semaphore indicates the availability of the serial hardware transmit register. It is created with a value of one to indicate the transmit register is initially available.

The initialization function is also responsible for installing the low-level interrupt vector handlers for input and output notifications. Like other ThreadX interrupt service routines, the low-level handler must call ***_tx_thread_context_save*** before calling the simple driver ISR. After the driver ISR returns, the low-level handler must call ***_tx_thread_context_restore***.

It is important that initialization is called before any of the other driver functions. Typically, driver initialization is called from ***tx_application_define****.*

See Figure 9 on page 226 for the initialization source code of the simple driver.

```
VOID    tx_sdriver_initialize(VOID)
{

    /* Initialize the two counting semaphores used to control
       the simple driver I/O.  */
    tx_semaphore_create(&tx_sdriver_input_semaphore,
                        "simple driver input semaphore", 0);
    tx_semaphore_create(&tx_sdriver_output_semaphore,
                        "simple driver output semaphore", 1);

    /* Setup interrupt vectors for input and output ISRs.
       The initial vector handling should call the ISRs
       defined in this file.  */

    /* Configure serial device hardware for RX/TX interrupt
       generation, baud rate, stop bits, etc. */
}
```

FIGURE 9. Simple Driver Initialization

Simple Driver Input

Input for the simple driver centers around the input semaphore. When a serial device input interrupt is received, the input semaphore is set. If one or more threads are waiting for a character from the driver, the thread waiting the longest is resumed. If no threads are waiting, the semaphore simply remains set until a thread calls the drive input function.

There are several limitations to the simple driver input handling. The most significant is the potential for dropping input characters. This is possible because there is no ability to buffer input characters that arrive before the previous character is processed. This is easily handled by adding an input character buffer.

Only threads are allowed to call the ***tx_sdriver_input*** *function.*

Figure 10 shows the source code associated with simple driver input.

```
UCHAR    tx_sdriver_input(VOID)
{

    /* Determine if there is a character waiting. If not,
        suspend.  */
    tx_semaphore_get(&tx_sdriver_input_semaphore,
                                            TX_WAIT_FOREVER;
    /* Return character from serial RX hardware register.  */
    return(*serial_hardware_input_ptr);
}


VOID    tx_sdriver_input_ISR(VOID)
{
    /* See if an input character notification is pending.  */
    if (!tx_sdriver_input_semaphore.tx_semaphore_count)
    {
        /* If not, notify thread of an input character.  */
        tx_semaphore_put(&tx_sdriver_input_semaphore);
    }
}
```

FIGURE 10. Simple Driver Input

Simple Driver Output

Output processing utilizes the output semaphore to signal when the serial device’s transmit register is free. Before an output character is actually written to the device, the output semaphore is obtained. If it is not available, the previous transmit is not yet complete.

The output ISR is responsible for handling the transmit complete interrupt. Processing of the output ISR amounts to setting the output semaphore, thereby allowing output of another character.

Only threads are allowed to call the ***tx_sdriver_output*** *function.*

Figure 11 shows the source code associated with simple driver output.

```
VOID    tx_sdriver_output(UCHAR alpha)
{

    /* Determine if the hardware is ready to transmit a
       character.  If not, suspend until the previous output
        completes.  */
    tx_semaphore_get(&tx_sdriver_output_semaphore,
                                            TX_WAIT_FOREVER);
    /* Send the character through the hardware.  */
    *serial_hardware_output_ptr =  alpha;
}


VOID    tx_sdriver_output_ISR(VOID)
{
    /* Notify thread last character transmit is
        complete.  */
    tx_semaphore_put(&tx_sdriver_output_semaphore);
}
```

FIGURE 11. Simple Driver Output

Simple Driver Shortcomings

This simple I/O driver example illustrates the basic idea of a ThreadX device driver. However, because the simple I/O driver does not address data buffering or any overhead issues, it does not fully represent real-world ThreadX drivers. The following section describes some of the more advanced issues associated with I/O drivers.

Advanced Driver Issues

As mentioned previously, I/O drivers have requirements as unique as their applications. Some applications may require an enormous amount of data buffering while another application may require optimized driver ISRs because of high-frequency device interrupts.

I/O Buffering

Data buffering in real-time embedded applications requires considerable planning. Some of the design is dictated by the underlying hardware device. If the device provides basic byte I/O, a simple circular buffer is probably in order. However, if the device provides block, DMA, or packet I/O, a buffer management scheme is probably warranted.

Circular Byte Buffers

Circular byte buffers are typically used in drivers that manage a simple serial hardware device like a UART. Two circular buffers are most often used in such situations—one for input and one for output.

Each circular byte buffer is comprised of a byte memory area (typically an array of UCHARs), a read pointer, and a write pointer. A buffer is considered empty when the read pointer and the write pointers reference the same memory location in the buffer. Driver initialization sets both the read and write buffer pointers to the beginning address of the buffer.

Circular Buffer Input

The input buffer is used to hold characters that arrive before the application is ready for them. When an input character is received (usually in an interrupt service routine), the new character is retrieved from the hardware device and placed into the input buffer at the location pointed to by the write pointer. The write pointer is then advanced to the next position in the buffer. If the next position is past the end of the buffer, the write pointer is set to the beginning of the buffer. The queue full condition is handled by cancelling the write pointer advancement if the new write pointer is the same as the read pointer.

Application input byte requests to the driver first examine the read and write pointers of the input buffer. If the read and write pointers are identical, the buffer is empty. Otherwise, if the read pointer is not the same, the byte pointed to by the read pointer is copied from the input buffer and the read pointer is advanced to the next buffer location. If the new read pointer is past the end of the buffer, it is reset to the beginning. Figure 12 shows the logic for the circular input buffer.

```
UCHAR    tx_input_buffer[MAX_SIZE];
UCHAR    tx_input_write_ptr;
UCHAR    tx_input_read_ptr;

/* Initialization.  */
tx_input_write_ptr =  &tx_input_buffer[0];
tx_input_read_ptr =    &tx_input_buffer[0];

/* Input byte ISR... UCHAR alpha has character from device.  */
save_ptr =  tx_input_write_ptr;
*tx_input_write_ptr++ =  alpha;
if (tx_input_write_ptr > &tx_input_buffer[MAX_SIZE-1])
    tx_input_write_ptr =  &tx_input_buffer[0];  /* Wrap */
if (tx_input_write_ptr == tx_input_read_ptr)
    tx_input_write_ptr =  save_ptr;  /* Buffer full */

/* Retrieve input byte from buffer...  */
if (tx_input_read_ptr != tx_input_write_ptr)
{
    alpha =  *tx_input_read_ptr++;
    if (tx_input_read_ptr > &tx_input_buffer[MAX_SIZE-1])
        tx_input_read_ptr =  &tx_input_buffer[0];
}
```

FIGURE 12. Logic for Circular Input Buffer

For reliable operation, it may be necessary to lockout interrupts when manipulating the read and write pointers of both the input and output circular buffers.

Circular Output Buffer

The output buffer is used to hold characters that have arrived for output before the hardware device finished sending the previous byte. Output buffer processing is similar to input buffer processing, except the transmit complete interrupt processing manipulates the output read pointer, while the application output request utilizes the output write pointer. Otherwise, the output buffer processing is the same. Figure 13 shows the logic for the circular output buffer.

```
UCHAR    tx_output_buffer[MAX_SIZE];
UCHAR    tx_output_write_ptr;
UCHAR    tx_output_read_ptr;

/* Initialization.  */
tx_output_write_ptr =  &tx_output_buffer[0];
tx_output_read_ptr =   &tx_output_buffer[0];

/* Transmit complete ISR... Device ready to send.  */
if (tx_output_read_ptr != tx_output_write_ptr)
{
    *device_reg =  *tx_output_read_ptr++;
    if (tx_output_read_reg > &tx_output_buffer[MAX_SIZE-1])
        tx_output_read_ptr =  &tx_output_buffer[0];
}

/* Output byte driver service.  If device busy, buffer!  */
save_ptr =  tx_output_write_ptr;
*tx_output_write_ptr++ =  alpha;
if (tx_output_write_ptr > &tx_output_buffer[MAX_SIZE-1])
    tx_output_write_ptr =  &tx_output_buffer[0];  /* Wrap */
if (tx_output_write_ptr == tx_output_read_ptr)
    tx_output_write_ptr =  save_ptr;  /* Buffer full!  */
```

FIGURE 13. Logic for Circular Output Buffer

Buffer I/O Management

To improve the performance of embedded microprocessors, many peripheral I/O devices transmit and receive data with buffers supplied by software. In some implementations, multiple buffers may be used to transmit or receive individual packets of data.

The size and location of I/O buffers is determined by the application and/or driver software. Typically, buffers are fixed in size and managed within a ThreadX block memory pool. Figure 14 describes a typical I/O buffer and a ThreadX block memory pool that manages their allocation.

```
typedef struct TX_IO_BUFFER_STRUCT
{
    struct TX_IO_BUFFER_STRUCT *tx_next_packet;
    struct TX_IO_BUFFER_STRUCT *tx_next_buffer;
    UCHAR  tx_buffer_area[TX_MAX_BUFFER_SIZE];
} TX_IO_BUFFER;


TX_BLOCK_POOL tx_io_block_pool;


/* Create a pool of I/O buffers.  Assume that the pointer
“free_memory_ptr” points to an available memory area that
is 64KBytes in size.  */
tx_block_pool_create(&tx_io_block_pool,
                  "Sample IO Driver Buffer Pool",
                  free_memory_ptr, 0x10000,
                  sizeof(TX_IO_BUFFER));
```

FIGURE 14. I/O Buffer

TX_IO_BUFFER

The typedef TX_IO_BUFFER consists of two pointers. The ***tx_next_packet*** pointer is used to link multiple packets on either the input or output list. The

tx_next_buffer pointer is used to link together buffers that make up an individual packet of data from the device. Both of these pointers are set to NULL when the buffer is allocated from the pool. In addition, some devices may require another field to indicate how much of the buffer area actually contains data.

Buffered I/O Advantage

What are the advantages of a buffer I/O scheme? The biggest advantage is that data is not copied between the device registers and the application's memory. Instead, the driver provides the device with a series of buffer pointers. Physical device I/O utilizes the supplied buffer memory directly.

Using the processor to copy input or output packets of information is extremely costly and should be avoided in any high throughput I/O situation.

Another advantage to the buffered I/O approach is that the input and output lists do not have full conditions. All of the available buffers can be on either list at any one time. This contrasts with the simple byte circular buffers presented earlier in the chapter. Each had a fixed size determined at compilation.

Buffered Driver Responsibilities

Buffered I/O drivers are only concerned with managing linked lists of I/O buffers. An input buffer list is maintained for packets that are received before the application software is ready. Conversely, an output buffer list is maintained for packets being sent faster than the hardware device can handle them. Figure 15 on page 234 shows simple input and output linked lists of data packets and the buffer(s) that make up each packet.

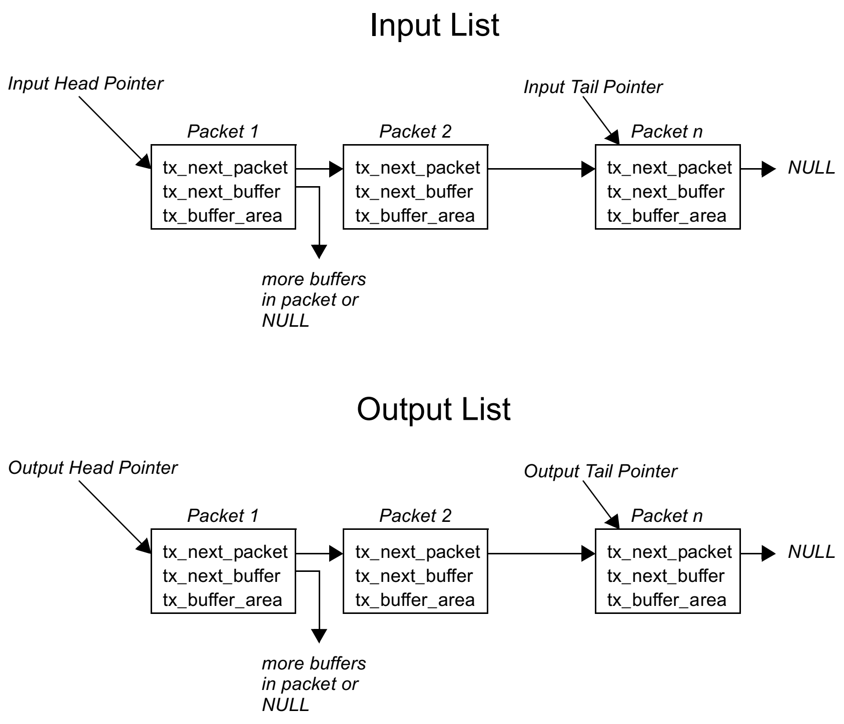

FIGURE 15. Input-Output Lists

Applications interface with buffered drivers with the same I/O buffers. On transmit, application software provides the driver with one or more buffers to transmit. When the application software requests input, the driver returns the input data in I/O buffers.

In some applications, it may be useful to build a driver input interface that requires the application to exchange a free buffer for an input buffer from the *driver. This might alleviate some buffer allocation processing inside of the driver.*

Interrupt Management

In some applications, the device interrupt frequency may prohibit writing the ISR in C or to interact with ThreadX on each interrupt. For example, if it takes 25us to save and restore the interrupted context, it would not be advisable to perform a full context save if the interrupt frequency was 50us. In such cases, a small assembly language ISR is used to handle most of the device interrupts. This low-overhead ISR would only interact with ThreadX when necessary.

A similar discussion can be found in the interrupt management discussion at the end of Chapter 3.

Thread Suspension

In the simple driver example presented earlier in this chapter, the caller of the input service suspends if a character is not available. In some applications, this might not be acceptable.

For example, if the thread responsible for processing input from a driver also has other duties, suspending on just the driver input is probably not going to work. Instead, the driver needs to be customized to request processing similar to the way other processing requests are made to the thread.

In most cases, the input buffer is placed on a linked list and an “input event” message is sent to the thread’s input queue.

CHAPTER 6

Demonstration System for ThreadX

This chapter contains a description of the demonstration system that is delivered with all ThreadX processor support packages. The following lists specific demonstration areas that are covered in this chapter:

Overview

Each ThreadX product distribution contains a demonstration system that runs on all supported microprocessors.

This example system is defined in the distribution file ***demo.c*** and is designed to illustrate how ThreadX is used in an embedded multi-thread environment. The demonstration consists of initialization, eight threads, one byte pool, one block pool, one queue, one semaphore, one mutex, and one event flag group.

It is worthwhile to mention that—except for the thread's stack size—the demonstration application is identical on all ThreadX supported processors.

The complete listing of ***demo.c***, including the line numbers referenced throughout the remainder of this chapter, is displayed on page 244 and following.

Application Define

The ***tx_application_define*** function executes after the basic ThreadX initialization is complete. It is responsible for setting up all of the initial system resources, including threads, queues, semaphores, mutexes, event flags, and memory pools.

The demonstration system's ***tx_application_define*** (*line numbers 60-164*) creates the demonstration objects in the following order:

```
byte_pool_0
thread_0
thread_1
thread_2
thread_3
thread_4
thread_5
thread_6
```

```
thread_7
queue_0
semaphore_0
event_flags_0
mutex_0
block_pool_0
```

The demonstration system does not create any other additional ThreadX objects. However, an actual application may create system objects during run-time inside of executing threads.

Initial Execution

All threads are created with the **TX_AUTO_START** option. This makes them initially ready for execution. After ***tx_application_define*** completes, control is transferred to the thread scheduler and from there to each individual thread.

The order in which the threads execute is determined by their priority and the order that they were created. In the demonstration system, ***thread_0*** executes first because it has the highest priority (*it was created with a priority of 1*). After ***thread_0*** suspends, ***thread_5*** is executed, followed by the execution of ***thread_3***, ***thread_4***, ***thread_6***, ***thread_7***, ***thread_1***, and finally ***thread_2***.

*Notice that even though **thread_3** and **thread_4** have the same priority (both created with a priority of 8), **thread_3** executes first. This is because **thread_3** was created and became ready before **thread_4**. Threads of equal priority execute in a FIFO fashion.*

Thread 0

The function ***thread_0_entry*** marks the entry point of the thread *(lines 167-190*). ***Thread_0*** is the first thread in the demonstration system to execute. Its processing is simple: it increments its counter, sleeps for 10 timer ticks, sets an event flag to wake up ***thread_5***, then repeats the sequence.

Thread_0 is the highest priority thread in the system. When its requested sleep expires, it will preempt any other executing thread in the demonstration.

Thread 1

The function ***thread_1_entry*** marks the entry point of the thread *(lines 193-216*). ***Thread_1*** is the second-to-last thread in the demonstration system to execute. Its processing consists of incrementing its counter, sending a message to ***thread_2*** (*through* ***queue_0***), and repeating the sequence. Notice that ***thread_1*** suspends whenever ***queue_0*** becomes full (*line 207*).

Thread 2

The function ***thread_2_entry*** marks the entry point of the thread *(lines 219-243*). ***Thread_2*** is the last thread in the demonstration system to execute. Its processing consists of incrementing its counter, getting a message from ***thread_1*** (through ***queue_0***), and repeating the sequence. Notice that ***thread_2*** suspends whenever ***queue_0*** becomes empty (*line 233*).

Although ***thread_1*** and ***thread_2*** share the lowest priority in the demonstration system (*priority 16*), they are also the only threads that are ready for execution most of the time. They are also the only threads created with time-slicing (*lines 74 and 82*). Each thread is allowed to execute for a maximum of 4 timer ticks before the other thread is executed.

Threads 3 and 4

The function ***thread_3_and_4_entry*** marks the entry point of both ***thread_3*** and ***thread_4*** *(lines 246-280*). Both threads have a priority of 8, which makes them the third and fourth threads in the demonstration system to execute. The processing for each thread is the same: incrementing its counter, getting ***semaphore_0***, sleeping for 2 timer ticks, releasing ***semaphore_0***, and repeating the sequence. Notice that each thread suspends whenever ***semaphore_0*** is unavailable (*line 264*).

Also both threads use the same function for their main processing. This presents no problems because they both have their own unique stack, and C is naturally reentrant. Each thread determines which one it is by examination of the thread input parameter (*line 258*), which is setup when they are created (*lines 102 and 109*).

It is also reasonable to obtain the current thread point during thread execution and compare it with the control block's address to determine thread identity.

Thread 5

The function ***thread_5_entry*** marks the entry point of the thread *(lines 283-305*). ***Thread_5*** is the second thread in the demonstration system to execute. Its processing consists of incrementing its counter, getting an event flag from ***thread_0*** (through ***event_flags_0***), and repeating the sequence. Notice that ***thread_5*** suspends whenever the event flag in ***event_flags_0*** is not available (*line 298*).

Threads 6 and 7

The function ***thread_6_and_7_entry*** marks the entry point of both ***thread_6*** and ***thread_7*** *(lines 307-358*). Both threads have a priority of 8, which makes them the fifth and sixth threads in the demonstration system to execute. The processing for each thread is the same: incrementing its counter, getting ***mutex_0*** twice, sleeping for 2 timer ticks, releasing ***mutex_0*** twice, and repeating the sequence. Notice that each thread suspends whenever ***mutex_0*** is unavailable (*line 325*).

Also both threads use the same function for their main processing. This presents no problems because they both have their own unique stack, and C is naturally reentrant. Each thread determines which one it is by examination of the thread input parameter (*line 319*), which is setup when they are created (*lines 126 and 133*).

Observing the Demonstration

Each of the demonstration threads increments its own unique counter. The following counters may be examined to check on the demo's operation:

```
thread_0_counter
thread_1_counter
thread_2_counter
thread_3_counter
thread_4_counter
thread_5_counter
thread_6_counter
```

```
    thread_7_counter
```

Each of these counters should continue to increase as the demonstration executes, with ***thread_1_counter*** and ***thread_2_counter*** increasing at the fastest rate.

Distribution file: demo.c

This section displays the complete listing of ***demo.c***, including the line numbers referenced throughout this chapter.

```
/* This is a small demo of the high-performance ThreadX kernel.  It includes examples of eight
threads of different priorities, using a message queue, semaphore, mutex, event flags group,
byte pool, and block pool.  */

#include"tx_api.h"

#define   DEMO_STACK_SIZE         1024
#define   DEMO_BYTE_POOL_SIZE     9120
#define   DEMO_BLOCK_POOL_SIZE    100
#define   DEMO_QUEUE_SIZE         100

/* Define the ThreadX object control blocks...  */

TX_THREAD               thread_0;
TX_THREAD               thread_1;
TX_THREAD               thread_2;
TX_THREAD               thread_3;
TX_THREAD               thread_4;
TX_THREAD               thread_5;
TX_THREAD               thread_6;
TX_THREAD               thread_7;
TX_QUEUE                queue_0;
TX_SEMAPHORE            semaphore_0;
TX_MUTEX                mutex_0;
TX_EVENT_FLAGS_GROUP    event_flags_0;
TX_BYTE_POOL            byte_pool_0;
TX_BLOCK_POOL           block_pool_0;

/* Define the counters used in the demo application...  */

ULONG               thread_0_counter;
ULONG               thread_1_counter;
ULONG               thread_1_messages_sent;
ULONG               thread_2_counter;
ULONG               thread_2_messages_received;
ULONG               thread_3_counter;
ULONG               thread_4_counter;
ULONG               thread_5_counter;
ULONG               thread_6_counter;
ULONG               thread_7_counter;

/* Define thread prototypes.  */

void    thread_0_entry(ULONG thread_input);
void    thread_1_entry(ULONG thread_input);
void    thread_2_entry(ULONG thread_input);
void    thread_3_and_4_entry(ULONG thread_input);
void    thread_5_entry(ULONG thread_input);
void    thread_6_and_7_entry(ULONG thread_input);


/* Define main entry point.  */

int main()
{

     /* Enter the ThreadX kernel.  */
     tx_kernel_enter();
}

/* Define what the initial system looks like.  */
void    tx_application_define(void *first_unused_memory)
{

CHAR    *pointer;

     /* Create a byte memory pool from which to allocate the thread stacks.  */
     tx_byte_pool_create(&byte_pool_0, "byte pool 0", first_unused_memory,
                              DEMO_BYTE_POOL_SIZE);

     /* Put system definition stuff in here, e.g. thread creates and other assorted
        create information.  */
```

```

    /* Allocate the stack for thread 0.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create the main thread.  */
    tx_thread_create(&thread_0, "thread 0", thread_0_entry, 0,
                    pointer, DEMO_STACK_SIZE,
                    1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 1.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create threads 1 and 2. These threads pass information through a ThreadX
         message queue.  It is also interesting to note that these threads have a time
         slice.  */
    tx_thread_create(&thread_1, "thread 1", thread_1_entry, 1,
                    pointer, DEMO_STACK_SIZE,
                    16, 16, 4, TX_AUTO_START);

    /* Allocate the stack for thread 2.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
    tx_thread_create(&thread_2, "thread 2", thread_2_entry, 2,
                    pointer, DEMO_STACK_SIZE,
                    16, 16, 4, TX_AUTO_START);

    /* Allocate the stack for thread 3.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create threads 3 and 4.  These threads compete for a ThreadX counting semaphore.
         An interesting thing here is that both threads share the same instruction area.  */
    tx_thread_create(&thread_3, "thread 3", thread_3_and_4_entry, 3,
                    pointer, DEMO_STACK_SIZE,
                    8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 4.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    tx_thread_create(&thread_4, "thread 4", thread_3_and_4_entry, 4,
                    pointer, DEMO_STACK_SIZE,
                    8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 5.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create thread 5.  This thread simply pends on an event flag which will be set
         by thread_0.  */
    tx_thread_create(&thread_5, "thread 5", thread_5_entry, 5,
                    pointer, DEMO_STACK_SIZE,
                    4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 6.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    /* Create threads 6 and 7.  These threads compete for a ThreadX mutex.  */
    tx_thread_create(&thread_6, "thread 6", thread_6_and_7_entry, 6,
                    pointer, DEMO_STACK_SIZE,
                    8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the stack for thread 7.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

    tx_thread_create(&thread_7, "thread 7", thread_6_and_7_entry, 7,
                    pointer, DEMO_STACK_SIZE,
                    8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

    /* Allocate the message queue.  */
    tx_byte_allocate(&byte_pool_0, &pointer, DEMO_QUEUE_SIZE*sizeof(ULONG), TX_NO_WAIT);

    /* Create the message queue shared by threads 1 and 2.  */
    tx_queue_create(&queue_0, "queue 0", TX_1_ULONG, pointer, DEMO_QUEUE_SIZE*sizeof(ULONG));

    /* Create the semaphore used by threads 3 and 4.  */
```

```
     tx_semaphore_create(&semaphore_0, "semaphore 0", 1);

     /* Create the event flags group used by threads 1 and 5.  */
     tx_event_flags_create(&event_flags_0, "event flags 0");

     /* Create the mutex used by thread 6 and 7 without priority inheritance.  */
     tx_mutex_create(&mutex_0, "mutex 0", TX_NO_INHERIT);

     /* Allocate the memory for a small block pool.  */
     tx_byte_allocate(&byte_pool_0, &pointer, DEMO_BLOCK_POOL_SIZE, TX_NO_WAIT);

     /* Create a block memory pool to allocate a message buffer from.  */
     tx_block_pool_create(&block_pool_0, "block pool 0", sizeof(ULONG), pointer,
                         DEMO_BLOCK_POOL_SIZE);

     /* Allocate a block and release the block memory.  */
     tx_block_allocate(&block_pool_0, &pointer, TX_NO_WAIT);

     /* Release the block back to the pool.  */
     tx_block_release(pointer);
}

/* Define the test threads.  */
void    thread_0_entry(ULONG thread_input)
{

UINT    status;


     /* This thread simply sits in while-forever-sleep loop.  */
     while(1)
     {

          /* Increment the thread counter.  */
          thread_0_counter++;

          /* Sleep for 10 ticks.  */
          tx_thread_sleep(10);

          /* Set event flag 0 to wakeup thread 5.  */
          status =  tx_event_flags_set(&event_flags_0, 0x1, TX_OR);

          /* Check status.  */
          if (status != TX_SUCCESS)
               break;
     }
}


void    thread_1_entry(ULONG thread_input)
{

UINT    status;


     /* This thread simply sends messages to a queue shared by thread 2.  */
     while(1)
     {

          /* Increment the thread counter.  */
          thread_1_counter++;

          /* Send message to queue 0.  */
          status =  tx_queue_send(&queue_0, &thread_1_messages_sent, TX_WAIT_FOREVER);

          /* Check completion status.  */
          if (status != TX_SUCCESS)
               break;

          /* Increment the message sent.  */
          thread_1_messages_sent++;
     }
```

```
}


void    thread_2_entry(ULONG thread_input)
{

ULONG   received_message;
UINT    status;

     /* This thread retrieves messages placed on the queue by thread 1.  */
     while(1)
     {

          /* Increment the thread counter.  */
          thread_2_counter++;

          /* Retrieve a message from the queue.  */
          status = tx_queue_receive(&queue_0, &received_message, TX_WAIT_FOREVER);

          /* Check completion status and make sure the message is what we
               expected.  */
          if ((status != TX_SUCCESS) || (received_message != thread_2_messages_received))
               break;

          /* Otherwise, all is okay.  Increment the received message count.  */
          thread_2_messages_received++;
     }
}


void    thread_3_and_4_entry(ULONG thread_input)
{

UINT    status;


     /* This function is executed from thread 3 and thread 4.  As the loop
          below shows, these function compete for ownership of semaphore_0.  */
     while(1)
     {

          /* Increment the thread counter.  */
          if (thread_input == 3)
               thread_3_counter++;
          else
               thread_4_counter++;

          /* Get the semaphore with suspension.  */
          status =  tx_semaphore_get(&semaphore_0, TX_WAIT_FOREVER);

          /* Check status.  */
          if (status != TX_SUCCESS)
               break;

          /* Sleep for 2 ticks to hold the semaphore.  */
          tx_thread_sleep(2);

          /* Release the semaphore.  */
          status =  tx_semaphore_put(&semaphore_0);

          /* Check status.  */
          if (status != TX_SUCCESS)
               break;
     }
}


void    thread_5_entry(ULONG thread_input)
{

UINT    status;
ULONG   actual_flags;
```

```


    /* This thread simply waits for an event in a forever loop.  */
    while(1)
    {

         /* Increment the thread counter.  */
         thread_5_counter++;

         /* Wait for event flag 0.  */
         status =  tx_event_flags_get(&event_flags_0, 0x1, TX_OR_CLEAR,
                                     &actual_flags, TX_WAIT_FOREVER);

         /* Check status.  */
         if ((status != TX_SUCCESS) || (actual_flags != 0x1))
              break;
    }
}

void    thread_6_and_7_entry(ULONG thread_input)
{

UINT    status;


    /* This function is executed from thread 6 and thread 7.  As the loop
         below shows, these function compete for ownership of mutex_0.  */
    while(1)
    {

         /* Increment the thread counter.  */
         if (thread_input == 6)
              thread_6_counter++;
         else
              thread_7_counter++;

         /* Get the mutex with suspension.  */
         status =  tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);

         /* Check status.  */
         if (status != TX_SUCCESS)
              break;

         /* Get the mutex again with suspension.  This shows
              that an owning thread may retrieve the mutex it
              owns multiple times.  */
         status =  tx_mutex_get(&mutex_0, TX_WAIT_FOREVER);

         /* Check status.  */
         if (status != TX_SUCCESS)
              break;

         /* Sleep for 2 ticks to hold the mutex.  */
         tx_thread_sleep(2);

         /* Release the mutex.  */
         status =  tx_mutex_put(&mutex_0);

         /* Check status.  */
         if (status != TX_SUCCESS)
              break;

         /* Release the mutex again.  This will actually
              release ownership since it was obtained twice.  */
         status =  tx_mutex_put(&mutex_0);

         /* Check status.  */
         if (status != TX_SUCCESS)
              break;
    }
}
```

CHAPTER 7

Internal Composition of ThreadX

Source code products without supporting documentation have limited usefulness. Furthermore, complicated coding standards or software design make source code products equally hard to use. This chapter contains a clear and concise description of the internal composition of ThreadX.

ThreadX Design Goals

ThreadX has three principal design goals: simplicity, scalability in size, and high performance. In many situations these goals are complementary; i.e. simpler, smaller software usually gives better performance.

Simplicity

Simplicity is the most important design goal of ThreadX. It makes ThreadX easy to use, test, and verify. In addition, it makes it easy for developers to understand exactly what is happening inside. This takes the mystery out of multi-threading, which contrasts sharply with the “black-box” approach so prevalent in the industry.

Scalability

ThreadX is also designed to be scalable. Its instruction area size ranges from 2KBytes through 15Kbytes, depending on the services actually used by the application. This enables ThreadX to support a wide range of microprocessor architectures, ranging from small micro-controllers through high-performance RISC and DSP processors.

How is ThreadX so scalable? First, ThreadX is designed with a software component methodology, which allows automatic removal of whole components that are not used. Second, it places each function in a separate file to minimize each function’s interaction with the rest of the system. Because ThreadX is implemented as a C library, only the functions that are used become part of the final embedded image.

High Performance

ThreadX is designed for high performance. This is achieved in a variety of ways, including algorithm optimizations, register variables, in-line assembly language, low-overhead timer interrupt handling, and optimized context switching. In addition, applications have the ability (with the conditional compilation flag **TX_DISABLE_ERROR_CHECKING**) to disable the basic error checking facilities of the ThreadX API. This feature is very useful in the tuning phase of application development. By disabling basic error checking, a 30 percent performance boost can be achieved on most ThreadX implementations. And, of course, the resulting code image is also smaller!

ThreadX ANSI C Library

As mentioned before, ThreadX is implemented as a C library, which must be linked with the application software. The ThreadX library consists of 146 object files that are derived from 138 C source files and eight (8) processor specific assembly language files. There are also ten C include files that are used in the C file compilation process. All the C source and include files conform completely to the ANSI standard.

System Include Files

ThreadX applications need access to two include files: ***tx_api.h*** and ***tx_port.h***. The *tx_api.h* file contains all the constants, function prototypes, and object data structures. This file is generic; i.e., it is the same for all processor support packages.

The *tx_port.h* file is included by *tx_api.h*. It contains processor and/or development tool specific information, including data type assignments and interrupt management macros that are used throughout the ThreadX C source code. The *tx_port.h* file also contains the ThreadX port-specific ASCII version string, ***_tx_version_id***.

*The mapping of the ThreadX API services to the underlying error checking or core processing functions is done in **tx_api.h**.*

The ThreadX source package also contains eight (8) system include files. These files represent the internal component specification files, which are discussed later in this chapter.

System Entry

From the application's point of view, the entry point of ThreadX is the function ***tx_kernel_enter***. However, this function is contained in the initialization file so its real name is ***_tx_initialize_kernel_enter***. Typically, this function is called from the application main routine with interrupts still disabled from the hardware reset and compiler start-up processing.

The entry function is responsible for calling the processor-specific, low-level initialization and the high-level C initialization. After all the initialization is complete, this function transfers control to the ThreadX scheduling loop.

Application Definition

ThreadX applications are required to provide their own ***tx_application_define*** function. This function is responsible for setting up the initial threads and other system objects. This function is called from the high-level C initialization mentioned previously.

Avoid enabling interrupts inside of the ***tx_application_define*** *function. If interrupts are enabled, unpredictable results may occur.*

Software Components

Express Logic utilizes a software component methodology in its products. A software component is somewhat similar to an object or class in C++. Each component provides a set of action functions that operate on the internal data of the component. In general, components are not allowed access to the global data of other components. The one exception to this rule is the thread component. For performance reasons, information like the currently running thread is accessed directly by other ThreadX components.

What makes up a ThreadX component? Each ThreadX component is comprised of a specification include file, an initialization function, and one or more action functions. As mentioned previously, each ThreadX function is defined in its own file.

If it were not for the design goal of scalable code size, component files would likely contain more than one function. In general, Express Logic recommends a "more than one function per-file" approach to application development.

ThreadX Components

There are nine functional ThreadX components. Each component has the same basic construction, and its processing and data structures are easily distinguished from those of other components. The following lists ThreadX software components:

Initialize
Thread
Timer
Queue
Semaphore
Mutex
Event Flags
Block Memory
Byte Memory

Component Specification File

Each ThreadX software component has a specification file. The specification file is a standard C include file that contains all component constants, data types, external and internal component function prototypes, and even the component's global data definitions.

The specification file is included in all component files and in files of other components that need to access the individual component's functions.

Component Initialization

Each component has an initialization function, which is responsible for initializing all of the component's internal global C data. In addition, all component global data instantiation takes place inside of the component's initialization file. This is accomplished with conditional compilation in the component's specification file as well as a special define in its initialization file.

If none of the component's services are used by the application, only the component's small initialization function is included in the application's run-time image.

Component Body Functions

A variable number of the component body or "action" functions complete the composition of a ThreadX software component.

As a general rule, component body functions are the only functions allowed to access the global data of the component. All interaction with other components must use access functions defined in the other component's specification file.

Coding Conventions

All ThreadX software conforms to a strict set of coding conventions. This makes it easier to understand and maintain. In addition, it provides a reasonable template for application software conventions.

ThreadX File Names

All ThreadX C file names take the form

TX_c[x].C

where **c** represents the first initial letter of the component and **[x]** represents a variable number of supplemental initial letters used to identify the function contained in the file. For example, file ***tx_tc.c*** contains the function ***_tx_thread_create*** and file ***tx_ike.c*** contains the function ***_tx_initialize_kernel_enter***.

Component specification file names are slightly different, taking on the form

TX_ccc.H

where the **ccc** field represents the first three characters of the component's name. For example, the file ***tx_tim.h*** contains the timer component specification.

The file naming conventions make it easy to distinguish ThreadX files from all other application source files.

ThreadX Name Space

In a similar vein, all ThreadX functions and global data have a leading _***tx*** in their name. This keeps ThreadX global symbols separate from the application symbols and in one contiguous area of load map created by the linker.

Most development tools will insert an additional underscore in front of all global symbols.

For ANSI compliance and greater compiler compatibility, all symbolic names in ThreadX are limited to 31 characters.

ThreadX Constants

All ThreadX constants have the form

TX_NAME or **TX_C_NAME**

and are comprised of capital letters, numerics, and underscores. System constants (defined in ***tx_api.h*** or ***tx_port.h***) take the form

TX_NAME

For example, the system-wide constant associated with a successful service call return is **TX_SUCCESS**.

Component constants (defined in component specification files) take on the form

TX_C_NAME

where **C** represents the capitalized entire component name. For example, **TX_INITIALIZE_IN_PROGRESS** is specific to the initialization component and is defined in the file ***tx_ini.h***.

ThreadX Struct and Typedef Names

ThreadX C *structure* and *typedef* names are similar to the component-specific constant names described previously. System wide typedefs have the form

TX_C_NAME

Just like the constant names, the **C** stands for the capitalized entire component name. For example, the queue control structure typedef is called **TX_QUEUE**.

To limit the number of ThreadX include files an application must deal with, the component specific typedefs that would normally be defined in the component specification files are contained in ***tx_api.h***.

For greater readability, primitive data types like **UINT**, **ULONG**, **VOID**, etc., do not require the leading **TX_** modifier. All primitive ThreadX data types are defined in the file ***tx_port.h***.

ThreadX Member Names

ThreadX structure member names are all lower case and take on the form

tx_c_name

where **c** is the entire component name (which is also the same as the parent structure or typedef name). For example, the thread identification field in the **TX_THREAD** structure is named ***tx_thread_id***.

ThreadX Global Data

Each ThreadX component has a small amount of global C data elements. All global data elements are lower-case and have the form ***_tx_c_name***. Like other ThreadX names, the ***c*** represents the entire component name. For example, the current thread pointer is part of the thread control component and is named ***_tx_thread_current_ptr*** and defined in the file ***tx_thr.h***.

ThreadX Local Data

Readability is the only requirement imposed on local data elements, i.e. data defined inside of ThreadX C functions. The most frequently used of these elements are typically assigned the *register* modifier if supported by the target compiler.

ThreadX Function Names

All ThreadX component function names have the form

_tx_c_name

ThreadX functions are in lower-case, where the ***c*** represents the entire component name. For example, the function that creates new application threads is named ***_tx_thread_create***.

Source Code Indentation

The standard indentation increment in ThreadX source code is four spaces. Tab characters are avoided in order to make the source code less sensitive to text editors. In addition, the source code is also designed to use indentation and white-space for greater readability.

Comments

In general, each C statement in the ThreadX source code has a meaningful comment. Each source file also contains a comment header that contains a description of the file, revision history, and the component it belongs to. Figure 16 on page 263 shows the file header for the thread create file, ***tx_tc.c***.

```
/**************************************************************************/
/**                                                                       */
/** ThreadX Component                                                     */
/**                                                                       */
/**   Thread Control (THR)                                                */
/**                                                                       */
/**************************************************************************/
/**************************************************************************/
/*                                                                        */
/*  FUNCTION                                               RELEASE        */
/*                                                                        */
/*    _tx_thread_create                                   PORTABLE C      */
/*                                                           3.0          */
/*  AUTHOR                                                                */
/*                                                                        */
/*    William E. Lamie, Express Logic, Inc.                               */
/*                                                                        */
/*  DESCRIPTION                                                           */
/*                                                                        */
/*    This function creates a thread and places it on the list of created */
/*    threads.                                                            */
/*                                                                        */
/*  INPUT                                                                 */
/*                                                                        */
/*    thread_ptr                            Thread control block pointer  */
/*    name                                  Pointer to thread name string */
/*    entry_function                        Entry function of the thread  */
/*    entry_input                           32-bit input value to thread  */
/*    stack_start                           Pointer to start of stack     */
/*    stack_size                            Stack size in bytes           */
/*    priority                              Priority of thread (0-31)     */
/*    preempt_threshold                     Preemption-threshold          */
/*    time_slice                            Thread time-slice value       */
/*    auto_start                            Automatic start selection     */
/*                                                                        */
/*  OUTPUT                                                                */
/*                                                                        */
/*    return status                         Thread create return status   */
/*                                                                        */
/*  CALLS                                                                 */
/*                                                                        */
/*    _tx_thread_stack_build                Build initial thread stack    */
/*    _tx_thread_resume                     Resume automatic start thread */
/*    _tx_thread_system_return              Return to system on preemption*/
/*                                                                        */
/*  CALLED BY                                                             */
/*                                                                        */
/*    Application Code                                                    */
/*    _tx_timer_initialize                  Create system timer thread    */
/*                                                                        */
/*  RELEASE HISTORY                                                       */
/*                                                                        */
/*    DATE              NAME                      DESCRIPTION             */
/*                                                                        */
/*  12-31-1996     William E. Lamie         Initial Version 3.0           */
/*                                                                        */
/**************************************************************************/
```

FIGURE 16. ThreadX File Header Example

Initialization Component

This component is responsible for performing all ThreadX initialization. This processing includes setting-up processor specific resources as well as calling all of the other component initialization functions. Once basic ThreadX initialization is complete, the application ***tx_application_define*** function is called to perform application specific initialization. The thread scheduling loop is entered after all initialization is complete.

TX_INI.H

This is the specification file for the ThreadX Initialization Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the initialization component is defined in this file and consists of the following data elements:

_tx_initialize_unused_memory

This VOID pointer contains the first memory address available to the application after ThreadX is initialized. The contents of this variable is passed into the application's ***tx_application_define*** function.

TX_IHL.C

This file contains ***_tx_initialize_high_level***, which is responsible for calling all other ThreadX component initialization functions and the application definition function, ***tx_application_define***.

TX_IKE.C

This file contains ***_tx_initialize_kernel_enter***, which coordinates the initialization and start-up processing of ThreadX. Note that the ***tx_kernel_enter*** function used by the application is mapped to this routine.

TX_ILL.[S, ASM]

This file contains ***_tx_initialize_low_level***, which handles all assembly language initialization processing. This file is processor and development tool specific.

Thread Component

This component is responsible for all thread management activities, including thread creation, scheduling, and interrupt management. The thread component is the most processor/compiler-specific of all ThreadX components, hence, it has the most assembly language files.

TX_THR.H

This is the specification file for the ThreadX Thread Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the thread component is defined in this file and consists of the following data elements:

_tx_thread_system_stack_ptr

This VOID pointer contains the address of the system stack pointer. The system stack is used inside of the ThreadX scheduling loop and inside of interrupt processing.

_tx_thread_current_ptr

This TX_THREAD pointer contains the address of the currently running thread's control block. If this pointer is NULL, the system is idle.

_tx_thread_execute_ptr

This TX_THREAD pointer contains the address of the next thread to execute and is used by the scheduling loop to determine which thread to execute next.

_tx_thread_created_ptr

This TX_THREAD pointer is the head pointer of the created thread list. The list is a doubly-linked, circular list of all created thread control blocks.

_tx_thread_created_count

This ULONG contains the number of currently created threads in the system.

_tx_thread_system_state

This ULONG contains the current system state. It is set during initialization and during interrupt processing to disable internal thread switching inside of the ThreadX services.

_tx_thread_preempted_map

This ULONG represents each of the 32 thread priority levels in ThreadX with a single bit. A set bit indicates that a thread of the corresponding priority level was preempted when it had preemption-threshold in force.

_tx_thread_priority_map

This ULONG represents each of the 32 thread priority levels in ThreadX with a single bit. It is used to find the next lower priority ready thread when a higher-priority thread suspends.

_tx_thread_highest_priority

This UINT contains the priority of the highest priority thread ready for execution.

_tx_thread_lowest_bit

This array of UCHARs contains a table lookup for quickly finding the lowest bit set in a byte. This is used in examination of the _tx_thread_priority_map to find the next ready priority group.

_tx_thread_priority_list

This array of TX_THREAD list-head pointers is directly indexed by thread priority. If an entry is non-NULL, there is at least one thread at that priority ready for execution. The threads in each priority list are managed in a doubly-linked, circular list of thread control blocks. The thread in the front of the list represents the next thread to execute for that priority.

_tx_thread_preempt_disable

This UINT is an internal mechanism for ThreadX services to enter into internal critical section processing. This reduces the amount of time interrupts need to be disabled inside of ThreadX services.

_tx_thread_special_string

This array of CHAR contains initials of various people and institutions that have helped make ThreadX possible.

TX_TC.C

This file contains ***_tx_thread_create***, which is responsible for creating application threads.

TX_TCR.[S,ASM]

This file contains ***_tx_thread_context_restore***, which is responsible for processing at the end of managed ISRs. This function is processor/compiler specific and is typically written in assembly language.

TX_TCS.[S,ASM]	This file contains ***_tx_thread_context_save***, which is responsible for saving the interrupted context in the beginning of ISR processing. This function is processor/compiler specific and is typically written in assembly language.
TX_TDEL.C	This file contains ***_tx_thread_delete***, which is responsible fore deleting a previously created thread.
TX_TI.C	This file contains ***_tx_thread_initialize***, which is responsible for basic thread component initialization.
TX_TIC.[S,ASM]	This file contains ***_tx_thread_interrupt_control***, which is responsible for enabling and disabling processor interrupts.
TX_TIDE.C	This file contains ***_tx_thread_identify***, which is responsible for returning the value of ***_tx_thread_current_ptr***.
TX_TIG.C	This file contains ***_tx_thread_info_get***, which is responsible for returning various information about a thread.
TX_TPC.[S,ASM]	This file contains ***_tx_thread_preempt_check***, which determines if preemption occurred while processing a lower level interrupt. This function is processor/compiler specific and is written in assembly language. In addition, this function is optional and is not needed for most ports.

TX_TPCH.C	This file contains ***_tx_thread_preemption_change***, which is responsible for changing the preemption-threshold of the specified thread.
TX_TPRCH.C	This file contains ***_tx_thread_priority_change***, which is responsible for changing the priority of the specified thread.
TX_TR.C	This file contains ***_tx_thread_resume***, which is responsible for making the specified thread ready for execution. This function is called from other ThreadX components as well as the thread resume API service.
TX_TRA.C	This file contains ***_tx_thread_resume_api***, which is responsible for processing application resume thread requests.
TX_TREL.C	This file contains ***_tx_thread_relinquish***, which is responsible for placing the current thread behind all other threads of the same priority that are ready for execution.
TX_TS.[S,ASM]	This file contains ***_tx_thread_schedule***, which is responsible for scheduling and restoring the last context of the highest-priority thread ready for execution. This function is processor/compiler specific and is written in assembly language.
TX_TSA.C	This file contains ***_tx_thread_suspend_api***, which is responsible for processing application thread suspend requests.

TX_TSB.[S,ASM]	This file contains ***_tx_thread_stack_build***, which is responsible for creating each thread's initial stack frame. The initial stack frame causes an interrupt context return to the beginning of the ***_tx_thread_shell_entry*** function. This function then calls the specified application thread entry function. The ***_tx_thread_stack_build*** function is processor/ compiler specific and is written in assembly language.
TX_TSE.C	This file contains ***_tx_thread_shell_entry***, which is responsible for calling the specified application thread entry function. If the thread entry function returns, ***_tx_thread_shell_entry*** suspends the thread in the "finished" state.
TX_TSLE.C	This file contains ***_tx_thread_sleep***, which is responsible for processing all application thread sleep requests.
TX_TSR.[S,ASM]	This file contains ***_tx_thread_system_return***, which is responsible for saving a thread's minimal context and exiting to the ThreadX scheduling loop. This function is processor/compiler specific and is written in assembly language.
TX_TSUS.C	This file contains ***_tx_thread_suspend***, which is responsible for processing all thread suspend requests from internal ThreadX components and the application software.
TX_TT.C	This file contains ***_tx_thread_terminate***, which is responsible for processing all thread terminate requests.

TX_TTO.C	This file contains ***_tx_thread_timeout***, which is responsible for processing all suspension time-out conditions.
TX_TTS.C	This file contains ***_tx_thread_time_slice***, which is responsible for processing thread time-slicing.
TX_TTSC.C	This file contains ***_tx_thread_time_slice_change***, which is responsible for requests to change a thread's time-slice.
TX_TWA.C	This file contains ***_tx_thread_wait_abort***, which is responsible for breaking the wait condition of the specified thread.
TXE_TC.C	This file contains ***_txe_thread_create***, which is responsible for checking the thread create requests for errors.
TXE_TDEL.C	This file contains ***_txe_thread_delete***, which is responsible for checking the thread delete requests for errors.
TXE_TIG.C	This file contains ***_txe_thread_info_get***, which is responsible for checking thread information get requests for errors.
TXE_TPCH.C	This file contains ***_txe_thread_preemption_change***, which is responsible for checking preemption change requests for errors.

TXE_TRA.C	This file contains ***_txe_thread_resume_api***, which is responsible for checking thread resume requests for errors.
TXE_TREL.C	This file contains ***_txe_thread_relinquish***, which is responsible for checking thread relinquish requests for errors.
TXE_TRPC.C	This file contains ***_txe_thread_priority_change***, which is responsible for checking priority change requests for errors.
TXE_TSA.C	This file contains ***_txe_thread_suspend_api***, which is responsible for checking thread suspend requests for errors.
TXE_TT.C	This file contains ***_txe_thread_terminate***, which is responsible for checking thread terminate requests for errors.
TXE_TTSC.C	This file contains ***_txe_thread_time_slice_change***, which is responsible for checking time-slice changes for errors.
TXE_TWA.C	This file contains ***_txe_thread_wait_abort***, which is responsible for checking thread wait abort requests for errors.

Timer Component

This component is responsible for all timer management activities, including thread time-slicing, thread sleeps, API service time-outs, and application timers. The timer component has one processor/compiler-specific function that is responsible for handling the physical timer interrupt.

TX_TIM.H

This is the specification file for the ThreadX Timer Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the timer component is defined in this file and consists of the following data elements:

_tx_timer_system_clock

This ULONG contains a tick counter that increments on each timer interrupt.

_tx_timer_time_slice

This ULONG contains the time-slice of the current thread. If this value is zero, no time-slice is active.

_tx_timer_expired_time_slice

This UINT is set if a time-slice expiration is detected in the timer interrupt handling. It is cleared once the time-slice has been processed in the ISR.

_tx_timer_list

This array of active timer linked-list head pointers is indexed by the timer's relative time displacement from the current time pointer. Each timer expiration list is maintained in a doubly-linked, circular fashion.

_tx_timer_list_start

This TX_INTERNAL_TIMER head pointer contains the address of the first timer list. It is used to reset the _tx_timer_current_ptr to the beginning of _tx_timer_list when a wrap condition is detected.

_tx_timer_list_end

This TX_INTERNAL_TIMER head pointer contains the address of the end of the _tx_timer_list array. It is used to signal when to reset the _tx_timer_current_ptr to the beginning of the _tx_timer_list.

_tx_timer_current_ptr

This TX_INTERNAL_TIMER head pointer points to an active timer list in the _tx_timer_list array. If a timer interrupt occurs and this entry is non-NULL, one or more timers have possibly expired. This pointer is positioned to point at the next timer list head pointer after each timer interrupt.

_tx_timer_expired

This UINT flag is set in the timer ISR when a timer has expired. It is cleared in the timer system thread after the expiration has been processed.

_tx_timer_thread

This TX_THREAD structure is the control block for the internal timer thread. This thread is setup during initialization and is used to process all timer expirations.

_tx_timer_stack_start

This VOID pointer represents the starting address of the internal timer thread's stack.

_tx_timer_stack_size

This ULONG represents the size of the internal timer thread's stack. This variable contains the value specified by TX_TIMER_THREAD_STACK_SIZE, which is defined inside of tx_port.h or on the command line.

_tx_timer_priority

This UINT represents the priority of the internal timer thread.

_tx_timer_created_ptr

This TX_TIMER pointer is the head pointer of the created application timer list. The list is a doubly-linked, circular list of all created timer control blocks.

_tx_timer_created_count

This ULONG represents the number of created application timers.

_tx_timer_thread_stack_area

This character array allocates space for the system timer's stack. The size of the array is defined by TX_TIMER_THREAD_STACK_SIZE, and the **_tx_timer_stack_start** and **_tx_timer_stack_end** pointers point to the beginning and end of this array.

TX_TA.C

This file contains ***_tx_timer_activate***, which is responsible for processing all timer activate requests (thread sleeps, time-outs, and application timers).

TX_TAA.C	This file contains ***_tx_timer_activate_api***, which is responsible for processing application timer activate requests.
TX_TD.C	This file contains ***_tx_timer_deactivate***, which is responsible for processing all timer deactivate requests (time-outs and application timers).
TX_TDA.C	This file contains ***_tx_timer_deactivate_api***, which is responsible for processing application timer deactivate requests.
TX_TIMCH.C	This file contains ***_tx_timer_change***, which is responsible for processing application timer change requests.
TX_TIMCR.C	This file contains ***_tx_timer_create***, which is responsible for processing application timer create requests.
TX_TIMD.C	This file contains ***_tx_timer_delete***, which is responsible for processing application timer delete requests.
TX_TIMEG.C	This file contains ***_tx_time_get***, which is responsible for processing requests to read the system clock, ***_tx_timer_system_clock***.
TX_TIMES.C	This file contains ***_tx_time_set***, which is responsible for processing requests to set the ***_tx_timer_system_clock*** to a specified value.

TX_TIMI.C	This file contains _***tx_timer_initialize***, which is responsible for initialization of the timer component.
TX_TIMIG.C	This file contains _***tx_timer_info_get***, which is responsible for retrieving information about a timer.
TX_TIMIN.[S,ASM]	This file contains _***tx_timer_interrupt***, which is responsible for processing actual timer interrupts. The interrupt processing is typically optimized to reduce overhead if neither a timer nor a time-slice has expired.
TX_TTE.C	This file contains _***tx_timer_thread_entry***, which is responsible for the processing of the internal timer thread.
TXE_TAA.C	This file contains _***txe_timer_activate_api***, which is responsible for checking application timer activate requests for errors
TXE_TDA.C	This file contains _***txe_timer_deactivate_api***, which is responsible for checking application timer deactivate requests for errors.
TXE_TIMD.C	This file contains _***txe_timer_delete***, which is responsible for checking application timer delete requests for errors.
TXE_TIMI.C	This file contains _***txe_timer_info_get***, which is responsible for checking application timer information get requests.

TXE_TMCH.C

This file contains _***txe_timer_change***, which is responsible for checking application timer change requests for errors.

TXE_TMCR.C

This file contains _***txe_timer_create***, which is responsible for checking application timer create requests for errors.

Queue Component

This component is responsible for all queue management activities, including queue creation, deletion, and message sending/receiving.

TX_QUE.H

This is the specification file for the ThreadX Queue Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the queue component is defined in this file and consists of the following data elements:

_tx_queue_created_ptr

This TX_QUEUE pointer is the head pointer of the created queue list. The list is a doubly-linked, circular list of all created queue control blocks.

_tx_queue_created_count

This ULONG represents the number of created application queues.

TX_QC.C

This file contains _***tx_queue_create***, which is responsible for processing queue create requests.

TX_QCLE.C	This file contains _***tx_queue_cleanup***, which is responsible for processing queue suspension time-outs, queue-suspended thread termination, and thread wait abort requests.
TX_QD.C	This file contains _***tx_queue_delete***, which is responsible for processing queue deletion requests.
TX_QF.C	This file contains _***tx_queue_flush***, which is responsible for processing queue flush requests.
TX_QFS.C	This file contains _***tx_queue_front_send***, which is responsible for processing requests to send a message to the front of a queue.
TX_QI.C	This file contains _***tx_queue_initialize***, which is responsible for initialization of the queue component.
TX_QIG.C	This file contains _***tx_queue_info_get***, which is responsible for retrieving information about a queue.
TX_QP.C	This file contains _***tx_queue_prioritize***, which is responsible for finding the highest priority thread suspended on a queue and placing it at the front of the suspension list.
TX_QR.C	This file contains _***tx_queue_receive***, which is responsible for processing queue receive requests.

TX_QS.C	This file contains _***tx_queue_send***, which is responsible for processing queue send requests.
TXE_QC.C	This file contains _***txe_queue_create***, which is responsible for checking queue create requests for errors.
TXE_QD.C	This file contains _***txe_queue_delete***, which is responsible for checking queue delete requests for errors.
TXE_QF.C	This file contains _***txe_queue_flush***, which is responsible for checking queue flush requests for errors.
TXE_QFS.C	This file contains _***txe_queue_front_send***, which is responsible for checking queue front send requests for errors.
TXE_QIG.C	This file contains _***txe_queue_info_get***, which is responsible for checking queue information retrieve requests for errors.
TXE_QP.C	This file contains _***txe_queue_prioritize***, which is responsible for checking queue prioritize requests for errors.
TXE_QR.C	This file contains _***txe_queue_receive***, which is responsible for checking queue receive requests for errors.

TXE_QS.C

This file contains ***_txe_queue_send***, which is responsible for checking queue send requests for errors.

Semaphore Component

This component is responsible for all semaphore management activities, including semaphore creation, deletion, semaphore gets, and semaphore puts.

TX_SEM.H

This is the specification file for the ThreadX Semaphore Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the semaphore component is defined in this file and consists of the following data elements:

_tx_semaphore_created_ptr

This TX_SEMAPHORE pointer is the head pointer of the created semaphore list. The list is a doubly-linked, circular list of all created semaphore control blocks.

_tx_semaphore_created_count

This ULONG represents the number of created application semaphores.

TX_SC.C

This file contains ***_tx_semaphore_create***, which is responsible for processing semaphore create requests.

TX_SCLE.C	This file contains _***tx_semaphore_cleanup***, which is responsible for processing semaphore suspension time-outs, semaphore-suspended thread termination, and thread wait abort requests.
TX_SD.C	This file contains _***tx_semaphore_delete***, which is responsible for processing semaphore deletion requests.
TX_SG.C	This file contains _***tx_semaphore_get***, which is responsible for processing semaphore get requests.
TX_SI.C	This file contains _***tx_semaphore_initialize***, which is responsible for initialization of the semaphore component.
TX_SIG.C	This file contains _***tx_semaphore_info_get***, which is responsible for semaphore information retrieval requests.
TX_SP.C	This file contains _***tx_semaphore_put***, which is responsible for semaphore put requests.
TX_SPRI.C	This file contains _***tx_semaphore_prioritize***, which is responsible for finding the highest priority thread suspended on a semaphore and placing it at the front of the suspension list.
TXE_SC.C	This file contains _***txe_semaphore_create***, which is responsible for checking semaphore create requests for errors.

TXE_SD.C	This file contains ***_txe_semaphore_delete***, which is responsible for checking semaphore delete requests for errors.
TXE_SG.C	This file contains ***_txe_semaphore_get***, which is responsible for checking semaphore get requests for errors.
TXE_SIG.C	This file contains ***_txe_semaphore_info_get***, which is responsible for checking semaphore information retrieval requests for errors.
TXE_SP.C	This file contains ***_txe_semaphore_put***, which is responsible for checking semaphore put requests for errors.
TXE_SPRI.C	This file contains ***_txe_semaphore_prioritize***, which is responsible for checking semaphore prioritize requests for errors.

Mutex Component

This component is responsible for all mutex management activities, including mutex creation, deletion, mutex gets, and mutex puts.

TX_MUT.H	This is the specification file for the ThreadX Mutex Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the mutex component is defined in this file and consists of the following data elements:

_tx_mutex_created_ptr

This TX_MUTEX pointer is the head pointer of the created mutex list. The list is a doubly-linked, circular list of all created mutex control blocks.

_tx_mutex_created_count

This ULONG represents the number of created application mutexes.

TX_MC.C

This file contains ***_tx_mutex_create***, which is responsible for processing mutex create requests.

TX_MCLE.C

This file contains ***_tx_mutex_cleanup***, which is responsible for processing mutex suspension time-outs, mutex-suspended thread termination, and thread wait abort requests.

TX_MD.C

This file contains ***_tx_mutex_delete***, which is responsible for processing mutex deletion requests.

TX_MG.C

This file contains ***_tx_mutex_get***, which is responsible for processing mutex get requests.

TX_MI.C

This file contains ***_tx_mutex_initialize***, which is responsible for initialization of the mutex component.

TX_MIG.C	This file contains ***_tx_mutex_info_get***, which is responsible for mutex information retrieval requests.
TX_MP.C	This file contains ***_tx_mutex_put***, which is responsible for mutex put requests.
TX_MPC.C	This file contains ***_tx_mutex_priority_change***, which is used by the mutex priority-inheritance logic to modify thread priorities.
TX_MPRI.C	This file contains ***_tx_mutex_prioritize***, which is responsible for finding the highest priority thread suspended on a mutex and placing it at the front of the suspension list.
TXE_MC.C	This file contains ***_txe_mutex_create***, which is responsible for checking mutex create requests for errors.
TXE_MD.C	This file contains ***_txe_mutex_delete***, which is responsible for checking mutex delete requests for errors.
TXE_MG.C	This file contains ***_txe_mutex_get***, which is responsible for checking mutex get requests for errors.
TXE_MIG.C	This file contains ***_txe_mutex_info_get***, which is responsible for checking mutex information retrieval requests for errors.

TXE_MP.C

This file contains ***_txe_mutex_put***, which is responsible for checking mutex put requests for errors.

TXE_MPRI.C

This file contains ***_txe_mutex_prioritize***, which is responsible for checking mutex prioritize requests for errors.

Event Flag Component

This component is responsible for all event flag management activities, including event flag creation, deletion, setting, and retrieval.

TX_EVE.H

This is the specification file for the ThreadX Event Flags Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the event flags component is defined in this file and consists of the following data elements:

_tx_event_flags_created_ptr

This TX_EVENT_FLAGS_GROUP pointer is the head pointer of the created event flags list. The list is a doubly-linked, circular list of all created event flags control blocks.

_tx_event_flags_created_count

This ULONG represents the number of created application event flags.

TX_EFC.C	This file contains ***_tx_event_flags_create***, which is responsible for processing event flag create requests.
TX_EFCLE.C	This file contains ***_tx_event_flags_cleanup***, which is responsible for processing event flag suspension time-outs, event-flag-suspended thread termination, and thread wait abort requests.
TX_EFD.C	This file contains ***_tx_event_flags_delete***, which is responsible for processing event flag deletion requests.
TX_EFG.C	This file contains ***_tx_event_flags_get***, which is responsible for processing event flag retrieval requests.
TX_EFI.C	This file contains ***_tx_event_flags_initialize***, which is responsible for initialization of the event flags component.
TX_EFIG.C	This file contains ***_tx_event_flags_info_get***, which is responsible for event flag information retrieval.
TX_EFS.C	This file contains ***_tx_event_flags_set***, which is responsible for processing event flag setting requests.
TXE_EFC.C	This file contains ***_txe_event_flags_create***, which is responsible for checking event flags create requests for errors.

TXE_EFD.C

This file contains ***_txe_event_flags_delete***, which is responsible for checking event flags delete requests for errors.

TXE_EFG.C

This file contains ***_txe_event_flags_get***, which is responsible for checking event flag retrieval requests for errors.

TXE_EFIG.C

This file contains ***_txe_event_flags_info_get***, which is responsible for checking event flag information retrieval requests for errors.

TXE_EFS.C

This file contains ***_txe_event_flags_set***, which is responsible for checking event flag setting requests for errors.

Block Memory Component

This component is responsible for all block memory management activities, including block pool creation, deletion, block allocates, and block releases.

TX_BLO.H

This is the specification file for the ThreadX Block Memory Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the block memory component is defined in this file and consists of the following data elements:

_tx_block_pool_created_ptr

This TX_BLOCK_POOL pointer is the head pointer of the created block memory pool list. The list is a doubly-linked, circular list of all created block pool control blocks.

_tx_block_pool_created_count

This ULONG represents the number of created application block memory pools.

TX_BA.C

This file contains ***_tx_block_allocate***, which is responsible for processing block allocation requests.

TX_BPC.C

This file contains ***_tx_block_pool_create***, which is responsible for processing block memory pool create requests.

TX_BPCLE.C

This file contains ***_tx_block_pool_cleanup***, which is responsible for processing block memory suspension time-outs, block-memory-suspended thread termination, and thread wait abort requests.

TX_BPD.C

This file contains ***_tx_block_pool_delete***, which is responsible for processing block memory pool delete requests.

TX_BPI.C

This file contains ***_tx_block_pool_initialize***, which is responsible for initialization of the block memory pool component.

TX_BPIG.C

This file contains ***_tx_block_pool_info_get***, which is responsible for block pool information retrieval.

TX_BPP.C	This file contains ***_tx_block_pool_prioritize***, which is responsible for finding the highest priority thread suspended on a block pool and moving it to the front of the suspension list.
TX_BR.C	This file contains ***_tx_block_release***, which is responsible for processing block release requests.
TXE_BA.C	This file contains ***_txe_block_allocate***, which is responsible for checking block allocate requests for errors.
TXE_BPC.C	This file contains ***_txe_block_pool_create***, which is responsible for checking block memory pool create requests for errors.
TXE_BPD.C	This file contains ***_txe_block_pool_delete***, which is responsible for checking block memory pool delete requests for errors.
TXE_BPIG.C	This file contains ***_txe_block_pool_info_get***, which is responsible for checking block pool information retrieval requests for errors.
TXE_BPP.C	This file contains ***_txe_block_pool_prioritize***, which is responsible for checking block pool prioritize requests for errors.
TXE_BR.C	This file contains ***_txe_block_release***, which is responsible for checking block memory release request for errors.

Byte Memory Component

This component is responsible for all byte memory management activities, including byte pool creation, deletion, byte allocates, and byte releases.

TX_BYT.H

This is the specification file for the ThreadX Byte Memory Component. All component constants, external interfaces, and data structures are defined in this file.

The global data for the byte memory component is defined in this file and consists of the following data elements:

_tx_byte_pool_created_ptr

This TX_BYTE_POOL pointer is the head pointer of the created byte memory pool list. The list is a doubly-linked, circular list of all created byte pool control blocks.

_tx_byte_pool_created_count

This ULONG represents the number of created application byte memory pools.

TX_BYTA.C

This file contains ***_tx_byte_allocate***, which is responsible for processing byte memory allocation requests.

TX_BYTC.C

This file contains ***_tx_byte_pool_create***, which is responsible for processing byte memory pool create requests.

TX_BYTCL.C	This file contains ***_tx_byte_pool_cleanup***, which is responsible for processing byte memory suspension time-outs, byte-memory-suspended thread termination, and thread wait abort requests.
TX_BYTD.C	This file contains ***_tx_byte_pool_delete***, which is responsible for processing byte memory pool delete requests.
TX_BYTI.C	This file contains ***_tx_byte_pool_initialize***, which is responsible for initialization of the byte memory pool component.
TX_BYTIG.C	This file contains ***_tx_byte_pool_info_get***, which is responsible for retrieving information about a byte pool.
TX_BYTPP.C	This file contains ***_tx_byte_pool_prioritize***, which is responsible for finding the highest priority thread suspended on a byte pool and moving it to the front of the suspension list.
TX_BYTR.C	This file contains ***_tx_byte_release***, which is responsible for processing byte release requests.
TX_BYTS.C	This file contains ***_tx_byte_pool_search***, which is responsible for searching through the byte memory pool for a large enough area of free bytes. Fragmented blocks are merged as the search proceeds through the memory area.

TXE_BTYA.C	This file contains _***txe_byte_allocate***, which is responsible for checking byte allocate requests for errors.
TXE_BYTC.C	This file contains _***txe_byte_pool_create***, which is responsible for checking byte memory pool create requests for errors.
TXE_BYTD.C	This file contains _***txe_byte_pool_delete***, which is responsible for checking byte memory pool delete requests for errors.
TXE_BYTG.C	This file contains _***txe_byte_pool_info_get***, which is responsible for checking byte pool information retrieval requests for errors.
TXE_BYTP.C	This file contains _***txe_byte_pool_prioritize***, which is responsible for checking byte pool prioritize requests for errors.
TXE_BYTR.C	This file contains _***txe_byte_release***, which is responsible for checking byte memory release requests for errors.

APPENDIX A

ThreadX API Services

Entry Function

```
VOID      tx_kernel_enter(VOID);
```

Byte Memory Services

```
UINT      tx_byte_allocate(TX_BYTE_POOL *pool_ptr,
             VOID **memory_ptr,
             ULONG memory_size, ULONG wait_option);

UINT      tx_byte_pool_create(TX_BYTE_POOL *pool_ptr,
             CHAR *name_ptr,
             VOID *pool_start, ULONG pool_size);

UINT      tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr);

UINT      tx_byte_pool_info_get(TX_BYTE_POOL *pool_ptr,
             CHAR **name, ULONG *available_bytes,
             ULONG *fragments, TX_THREAD **first_suspended,
             ULONG *suspended_count,
             TX_BYTE_POOL **next_pool);

UINT      tx_byte_pool_prioritize(TX_BYTE_POOL *pool_ptr);

UINT      tx_byte_release(VOID *memory_ptr);
```

Block Memory Services

```
UINT      tx_block_allocate(TX_BLOCK_POOL *pool_ptr,
             VOID **block_ptr, ULONG wait_option);

UINT      tx_block_pool_create(TX_BLOCK_POOL *pool_ptr,
             CHAR *name_ptr, ULONG block_size,
             VOID *pool_start, ULONG pool_size);

UINT      tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr);

UINT      tx_block_pool_info_get(TX_BLOCK_POOL *pool_ptr,
             CHAR **name,
             ULONG *available_blocks, ULONG *total_blocks,
             TX_THREAD **first_suspended,
             ULONG *suspended_count,
             TX_BLOCK_POOL **next_pool);

UINT      tx_block_pool_prioritize(TX_BLOCK_POOL *pool_ptr);

UINT      tx_block_release(VOID *block_ptr);
```

Event Flag Services

```
UINT      tx_event_flags_create(TX_EVENT_FLAGS_GROUP *group_ptr,
             CHAR *name_ptr);

UINT      tx_event_flags_delete(TX_EVENT_FLAGS_GROUP *group_ptr);

UINT      tx_event_flags_get(TX_EVENT_FLAGS_GROUP *group_ptr,
             ULONG requested_flags, UINT get_option,
             ULONG *actual_flags_ptr, ULONG wait_option);

UINT      tx_event_flags_info_get(TX_EVENT_FLAGS_GROUP *group_ptr,
             CHAR **name, ULONG *current_flags,
             TX_THREAD **first_suspended,
             ULONG *suspended_count,
             TX_EVENT_FLAGS_GROUP **next_group);

UINT      tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr,
             ULONG flags_to_set, UINT set_option);
```

Interrupt Control

```
UINT      tx_interrupt_control(UINT new_posture);
```

Message Queue Services

```
UINT      tx_queue_create(TX_QUEUE *queue_ptr, CHAR *name_ptr,
             UINT message_size, VOID *queue_start,
             ULONG queue_size);

UINT      tx_queue_delete(TX_QUEUE *queue_ptr);

UINT      tx_queue_flush(TX_QUEUE *queue_ptr);

UINT      tx_queue_front_send(TX_QUEUE *queue_ptr, VOID *source_ptr,
             ULONG wait_option);

UINT      tx_queue_info_get(TX_QUEUE *queue_ptr, CHAR **name,
             ULONG *enqueued, ULONG *available_storage,
             TX_THREAD **first_suspended,
             ULONG *suspended_count, TX_QUEUE **next_queue);

UINT      tx_queue_prioritize(TX_QUEUE *queue_ptr);

UINT      tx_queue_receive(TX_QUEUE *queue_ptr,
             VOID *destination_ptr, ULONG wait_option);

UINT      tx_queue_send(TX_QUEUE *queue_ptr, VOID *source_ptr,
             ULONG wait_option);
```

Semaphore Services

```
UINT      tx_semaphore_create(TX_SEMAPHORE *semaphore_ptr,
             CHAR *name_ptr, ULONG initial_count);

UINT      tx_semaphore_delete(TX_SEMAPHORE *semaphore_ptr);

UINT      tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr,
             ULONG wait_option);

UINT      tx_semaphore_info_get(TX_SEMAPHORE *semaphore_ptr, CHAR
             **name,
             ULONG *current_value,
             TX_THREAD **first_suspended,
             ULONG *suspended_count,
             X_SEMAPHORE **next_semaphore);

UINT      tx_semaphore_prioritize(TX_SEMAPHORE *semaphore_ptr);

UINT      tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr);
```

Mutex Services

```
UINT      tx_mutex_create(TX_MUTEX *mutex_ptr, CHAR *name_ptr,
             UINT inherit);

UINT      tx_mutex_delete(TX_MUTEX *mutex_ptr);

UINT      tx_mutex_get(TX_MUTEX *mutex_ptr, ULONG wait_option);

UINT      tx_mutex_info_get(TX_MUTEX *mutex_ptr, CHAR **name,
             ULONG *count, TX_THREAD **owner,
             TX_THREAD **first_suspended,
             ULONG *suspended_count,
             TX_MUTEX **next_mutex);

UINT      tx_mutex_prioritize(TX_MUTEX *mutex_ptr);

UINT      tx_mutex_put(TX_MUTEX *mutex_ptr);
```

Thread Control Services

```
UINT      tx_thread_create(TX_THREAD *thread_ptr, CHAR *name_ptr,
             VOID (*entry_function)(ULONG), ULONG entry_input,
             VOID *stack_start, ULONG stack_size,
             UINT priority, UINT preempt_threshold,
             ULONG time_slice, UINT auto_start);

UINT      tx_thread_delete(TX_THREAD *thread_ptr);
             TX_THREAD  *tx_thread_identify(VOID);

UINT      tx_thread_info_get(TX_THREAD *thread_ptr, CHAR **name,
             UINT *state, ULONG *run_count, UINT *priority,
             UINT *preemption_threshold, ULONG *time_slice,
             TX_THREAD **next_thread,
             TX_THREAD **next_suspended_thread);

UINT      tx_thread_preemption_change(TX_THREAD *thread_ptr,
             UINT new_threshold, UINT *old_threshold);

UINT      tx_thread_priority_change(TX_THREAD *thread_ptr,
             UINT new_priority,  UINT *old_priority);
             VOID  tx_thread_relinquish(VOID);

UINT  tx_thread_resume(TX_THREAD *thread_ptr);

UINT      tx_thread_sleep(ULONG timer_ticks);

UINT      tx_thread_suspend(TX_THREAD *thread_ptr);

UINT      tx_thread_terminate(TX_THREAD *thread_ptr);

UINT      tx_thread_time_slice_change(TX_THREAD *thread_ptr,
             ULONG new_time_slice, ULONG *old_time_slice);

UINT      tx_thread_wait_abort(TX_THREAD *thread_ptr);
```

Time Services

```
ULONG     tx_time_get(VOID);
             VOID tx_time_set(ULONG new_time);
```

Timer Services

```
UINT      tx_timer_activate(TX_TIMER *timer_ptr);
             UINT  tx_timer_change(TX_TIMER *timer_ptr,
             ULONG initial_ticks,
             ULONG reschedule_ticks);
             UINT  tx_timer_create(TX_TIMER *timer_ptr,
             CHAR *name_ptr,
             VOID (*expiration_function)(ULONG),
             ULONG expiration_input, ULONG initial_ticks,
             ULONG reschedule_ticks, UINT auto_activate);

UINT      tx_timer_deactivate(TX_TIMER *timer_ptr);

UINT      tx_timer_delete(TX_TIMER *timer_ptr);

UINT      tx_timer_info_get(TX_TIMER *timer_ptr, CHAR **name,
             UINT *active, ULONG *remaining_ticks,
             ULONG *reschedule_ticks,
             TX_TIMER **next_timer);
```

ThreadX Constants

Alphabetic Listings

TX_1_ULONG	1
TX_2_ULONG	2
TX_4_ULONG	4
TX_8_ULONG	8
TX_16_ULONG	16
TX_ACTIVATE_ERROR	0x0017
TX_AND	2
TX_AUTO_ACTIVATE	1
TX_AND_CLEAR	3
TX_AUTO_START	1
TX_BLOCK_MEMORY	8
TX_BYTE_MEMORY	9
TX_CALLER_ERROR	0x0013
TX_COMPLETED	1
TX_DELETE_ERROR	0x0011
TX_DELETED	0x0001
TX_DONT_START	0
TX_EVENT_FLAG	7
TX_FALSE	0
TX_FILE	11
TX_FOREVER	1
TX_GROUP_ERROR	0x0006
TX_INHERIT	1
TX_INHERIT_ERROR	0x001F
TX_IO_DRIVER	10
TX_MAX_PRIORITIES	32
TX_MUTEX_ERROR	0x001F
TX_MUTEX_SUSP	13
TX_NO_ACTIVATE	0
TX_NO_EVENTS	0x0007
TX_NO_INHERIT	0

TX_NO_INSTANCE	0x000D
TX_NO_MEMORY	0x0010
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NOT_AVAILABLE	0x001D
TX_NOT_OWNED	0x001E
TX_NULL	0
TX_OPTION_ERROR	0x0008
TX_OR	0
TX_OR_CLEAR	1
TX_POOL_ERROR	0x0002
TX_PRIORITY_ERROR	0x000F
TX_PTR_ERROR	0x0003
TX_QUEUE_EMPTY	0x000A
TX_QUEUE_ERROR	0x0009
TX_QUEUE_FULL	0x000B
TX_QUEUE_SUSP	5
TX_READY	0
TX_RESUME_ERROR	0x0012
TX_SEMAPHORE_ERROR	0x000C
TX_SEMAPHORE_SUSP	6
TX_SIZE_ERROR	0x0005
TX_SLEEP	4
TX_START_ERROR	0x0010
TX_SUCCESS	0x0000
TX_SUSPEND_ERROR	0x0014
TX_SUSPEND_LIFTED	0x0019
TX_SUSPENDED	3
TX_TCP_IP	12
TX_TERMINATED	2
TX_THREAD_ERROR	0x000E

TX_THRESH_ERROR	0x0018
TX_TICK_ERROR	0x0016
TX_TIMER_ERROR	0x0015
TX_TRUE	1
TX_WAIT_ABORT_ERROR	0x001B
TX_WAIT_ABORTED	0x001A
TX_WAIT_ERROR	0x0004
TX_WAIT_FOREVER	FFFFFFFF

Listing by Value

TX_DONT_START	0
TX_FALSE	0
TX_NO_ACTIVATE	0
TX_NO_INHERIT	0
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NULL	0
TX_OR	0
TX_READY	0
TX_SUCCESS	0x0000
TX_1_ULONG	1
TX_AUTO_ACTIVATE	1
TX_AUTO_START	1
TX_COMPLETED	1
TX_FOREVER	1
TX_DELETED	0x0001
TX_INHERIT	1
TX_OR_CLEAR	1
TX_TRUE	1
TX_2_ULONG	2
TX_AND	2

TX_POOL_ERROR	0x0002
TX_TERMINATED	2
TX_AND_CLEAR	3
TX_PTR_ERROR	0x0003
TX_SUSPENDED	3
TX_4_ULONG	4
TX_SLEEP	4
TX_WAIT_ERROR	0x0004
TX_QUEUE_SUSP	5
TX_SIZE_ERROR	0x0005
TX_GROUP_ERROR	0x0006
TX_SEMAPHORE_SUSP	6
TX_EVENT_FLAG	7
TX_NO_EVENTS	0x0007
TX_8_ULONG	8
TX_BLOCK_MEMORY	8
TX_OPTION_ERROR	0x0008
TX_BYTE_MEMORY	9
TX_QUEUE_ERROR	0x0009
TX_IO_DRIVER	10
TX_QUEUE_EMPTY	0x000A
TX_FILE	11
TX_QUEUE_FULL	0x000B
TX_SEMAPHORE_ERROR	0x000C
TX_TCP_IP	12
TX_MUTEX_SUSP	13
TX_NO_INSTANCE	0x000D
TX_THREAD_ERROR	0x000E
TX_PRIORITY_ERROR	0x000F
TX_16_ULONG	16
TX_START_ERROR	0x0010

TX_NO_MEMORY	0x0010
TX_DELETE_ERROR	0x0011
TX_RESUME_ERROR	0x0012
TX_CALLER_ERROR	0x0013
TX_SUSPEND_ERROR	0x0014
TX_TIMER_ERROR	0x0015
TX_TICK_ERROR	0x0016
TX_ACTIVATE_ERROR	0x0017
TX_THRESH_ERROR	0x0018
TX_SUSPEND_LIFTED	0X0019
TX_WAIT_ABORTED	0x001A
TX_WAIT_ABORT_ERROR	0x001B
TX_MUTEX_ERROR	0x001C
TX_NOT_AVAILABLE	0x001D
TX_NOT_OWNED	0x001E
TX_INHERIT_ERROR	0x001F
TX_MAX_PRIORITIES	32
TX_WAIT_FOREVER	FFFFFFFF

ThreadX Data Types

```
typedef  struct TX_INTERNAL_TIMER_STRUCT
{
    ULONG                       tx_remaining_ticks;
    ULONG                       tx_re_initialize_ticks;
    VOID                        (*tx_timeout_function)(ULONG);
    ULONG                        tx_timeout_param;
    struct TX_INTERNAL_TIMER_STRUCT *tx_active_next,
               *tx_active_previous;
    struct TX_INTERNAL_TIMER_STRUCT **tx_list_head;
} TX_INTERNAL_TIMER;


typedef  struct TX_TIMER_STRUCT
{
    ULONG                      tx_timer_id;
    CHAR_PTR                   tx_timer_name;
    TX_INTERNAL_TIMER          tx_timer_internal;
    struct TX_TIMER_STRUCT    *tx_timer_created_next,
                                *tx_timer_created_previous;
} TX_TIMER;


typedef struct TX_QUEUE_STRUCT
{
    ULONG       tx_queue_id;
    CHAR_PTR    tx_queue_name;
    UINT        tx_queue_message_size;
    ULONG       tx_queue_capacity;
    ULONG       tx_queue_enqueued;
    ULONG       tx_queue_available_storage;
    ULONG_PTR   tx_queue_start;
    ULONG_PTR   tx_queue_end;
    ULONG_PTR   tx_queue_read;
    ULONG_PTR   tx_queue_write;
    struct TX_THREAD_STRUCT  *tx_queue_suspension_list;
    ULONG                    tx_queue_suspended_count;
    struct TX_QUEUE_STRUCT
                *tx_queue_created_next,
                 *tx_queue_created_previous;
} TX_QUEUE;
```

```
typedef  struct TX_THREAD_STRUCT
{
    ULONG       tx_thread_id;
    ULONG       tx_run_count;
    VOID_PTR    tx_stack_ptr;
    VOID_PTR    tx_stack_start;
    VOID_PTR    tx_stack_end;
    ULONG       tx_stack_size;
    ULONG       tx_time_slice;
    ULONG       tx_new_time_slice;
    struct TX_THREAD_STRUCT  *tx_ready_next,
               *tx_ready_previous;
    TX_THREAD_PORT_EXTENSION  /* See tx_port.h for details */
    CHAR_PTR    tx_thread_name;
    UINT        tx_priority;
    UINT        tx_state;
    UINT        tx_delayed_suspend;
    UINT        tx_suspending;
    UINT        tx_preempt_threshold;
    ULONG       tx_priority_bit;
    VOID        (*tx_thread_entry)(ULONG);
    ULONG       tx_entry_parameter;
    TX_INTERNAL_TIMER  tx_thread_timer;
    VOID        (*tx_suspend_cleanup)
                  (struct TX_THREAD_STRUCT *);
    VOID_PTR    tx_suspend_control_block;
    struct TX_THREAD_STRUCT *tx_suspended_next,
               *tx_suspended_previous;
    ULONG       tx_suspend_info;
    VOID_PTR    tx_additional_suspend_info;
    UINT        tx_suspend_option;
    UINT        tx_suspend_status;
    struct TX_THREAD_STRUCT *tx_created_next,
               *tx_created_previous;
    VOID_PTR   tx_filex_ptr;

} TX_THREAD;


typedef struct TX_SEMAPHORE_STRUCT
{
    ULONGtx_semaphore_id;
    CHAR_PTR    tx_semaphore_name;
    ULONG       tx_semaphore_count;
    struct TX_THREAD_STRUCT  *tx_semaphore_suspension_list;
    ULONG                    tx_semaphore_suspended_count;
    struct TX_SEMAPHORE_STRUCT *tx_semaphore_created_next,
*tx_semaphore_created_previous;
} TX_SEMAPHORE;


typedef struct TX_EVENT_FLAGS_GROUP_STRUCT
{
    ULONG       tx_event_flags_id;
    CHAR_PTR    tx_event_flags_name;
    ULONG       tx_event_flags_current;
    UINT        tx_event_flags_reset_search;
    struct TX_THREAD_STRUCT  *tx_event_flags_suspension_list;
    ULONG                     tx_event_flags_suspended_count;
    struct TX_EVENT_FLAGS_GROUP_STRUCT
                *tx_event_flags_created_next,
               *tx_event_flags_created_previous;
} TX_EVENT_FLAGS_GROUP;


typedef struct TX_BLOCK_POOL_STRUCT
{
    ULONG       tx_block_pool_id;
    CHAR_PTR    tx_block_pool_name;
    ULONG       tx_block_pool_available;
    ULONG       tx_block_pool_total;
    CHAR_PTR    tx_block_pool_available_list;
    CHAR_PTR    tx_block_pool_start;
    ULONG       tx_block_pool_size;
    ULONG       tx_block_pool_block_size;
    struct TX_THREAD_STRUCT*tx_block_pool_suspension_list;
    ULONG                   tx_block_pool_suspended_count;
    struct TX_BLOCK_POOL_STRUCT
                *tx_block_pool_created_next,
                *tx_block_pool_created_previous;
} TX_BLOCK_POOL;
typedef struct TX_BYTE_POOL_STRUCT
{
    ULONG       tx_byte_pool_id;
    CHAR_PTR    tx_byte_pool_name;
    ULONG       tx_byte_pool_available;
    ULONG       tx_byte_pool_fragments;
    CHAR_PTR    tx_byte_pool_list;
    CHAR_PTR    tx_byte_pool_search;
    CHAR_PTR    tx_byte_pool_start;
    ULONG       tx_byte_pool_size;
    struct TX_THREAD_STRUCT*tx_byte_pool_owner;
    struct TX_THREAD_STRUCT*tx_byte_pool_suspension_list;
    ULONG                   tx_byte_pool_suspended_count;
    struct TX_BYTE_POOL_STRUCT *tx_byte_pool_created_next,
                *tx_byte_pool_created_previous;
} TX_BYTE_POOL;
```

```
typedef struct TX_MUTEX_STRUCT
{
    ULONG   tx_mutex_id;
    CHAR_PTR        tx_mutex_name;
    ULONG   tx_mutex_ownership_count;
    TX_THREAD       *tx_mutex_owner;
    UINT            tx_mutex_inherit;
    UINT            tx_mutex_original_priority;
    UINT            tx_mutex_original_threshold;
    struct TX_THREAD_STRUCT
    *tx_mutex_suspension_list;
    ULONG   tx_mutex_suspended_count;
    struct TX_MUTEX_STRUCT
    *tx_mutex_created_next,
    *tx_mutex_created_previous;

} TX_MUTEX;
```

APPENDIX D

ThreadX Source Files

ThreadX C Include Files

TX_API.H	Application Interface Include
TX_BLO.H	Block Memory Component Include
TX_BYT.H	Byte Memory Component Include
TX_EVE.H	Event Flag Component Include
TX_INI.H	Initialize Component Include
TX_MUT.H	Mutex Component Include
TX_PORT.H	Port Specific Include (processor specific
TX_QUE.H	Queue Component Include
TX_THR.H	Thread Control Component Include
TX_TIM.H	Timer Component Include
TX_SEM.H	Semaphore Component Include

ThreadX C Source Files

TX_BA.C	Block Memory Allocate
TX_BPC.C	Block Pool Create
TX_BPCLE.C	Block Pool Cleanup
TX_BPD.C	Block Pool Delete
TX_BPI.C	Block Pool Initialize
TX_BPIG.C	Block Pool Information Get
TX_BPP.C	Block Pool Prioritize
TX_BR.C	Block Memory Release
TXE_BA.C	Block Allocate Error Checking
TXE_BPC.C	Block Pool Create Error Checking
TXE_BPD.C	Block Pool Delete Error Checking
TXE_BPIG.C	Block Pool Information Get Error Checking
TXE_BPP.C	Block Pool Prioritize Error Checking
TXE_BR.C	Block Release Error Checking

TX_BYTA.C	Byte Memory Allocate
TX_BYTC.C	Byte Pool Create
TX_BYTCL.C	Byte Pool Cleanup
TX_BYTD.C	Byte Pool Delete
TX_BYTI.C	Byte Pool Initialize
TX_BYTIG.C	Byte Pool Information Get
TX_BYTPP.C	Byte Pool Prioritize
TX_BYTR.C	Byte Memory Release
TX_BYTS.C	Byte Pool Search
TXE_BYTA.C	Byte Allocate Error Checking
TXE_BYTC.C	Byte Pool Create Error Checking
TXE_BYTD.C	Byte Pool Delete Error Checking
TXE_BYTG.C	Byte Pool Information Get Error Checking
TXE_BYTP.C	Byte Pool Prioritize Error Checking
TXE_BYTR.C	Byte Pool Release Error Checking
TX_EFC.C	Event Flag Create
TX_EFCLE.C	Event Flag Cleanup
TX_EFD.C	Event Flag Delete
TX_EFG.C	Event Flag Get
TX_EFI.C	Event Flag Initialize
TX_EFIG.C	Event Flag Information Get
TX_EFS.C	Event Flag Set
TXE_EFC.C	Event Flag Create Error Checking
TXE_EFD.C	Event Flag Delete Error Checking
TXE_EFG.C	Event Flag Get Error Checking
TXE_EFIG.C	Event Flag Information Get Error Checking
TXE_EFS.C	Event Flag Set Error Checking
TX_IHL.C	Initialize High Level
TX_IKE.C	Initialize Kernel Entry Point

TX_SC.C	Semaphore Create
TX_SCLE.C	Semaphore Cleanup
TX_SD.C	Semaphore Delete
TX_SG.C	Semaphore Get
TX_SI.C	Semaphore Initialize
TX_SIG.C	Semaphore Information Get
TX_SP.C	Semaphore Put
TX_SPRI.C	Semaphore Prioritize
TXE_SC.C	Semaphore Create Error Checking
TXE_SD.C	Semaphore Delete Error Checking
TXE_SG.C	Semaphore Get Error Checking
TXE_SIG.C	Semaphore Information Get Error Checking
TXE_SP.C	Semaphore Put Error Checking
TXE_SPRI.C	Semaphore Prioritize Error Checking
TX_MC.C	Mutex Create
TX_MCLE.C	Mutex Cleanup
TX_MD.C	Mutex Delete
TX_MG.C	Mutex Get
TX_MI.C	Mutex Initialize
TX_MIG.C	Mutex Information Get
TX_MP.C	Mutex Put
TX_MPC.C	Mutex Priority Change
TX_MPRI.C	Mutex Prioritize
TXE_MC.C	Mutex Create Error Checking
TXE_MD.C	Mutex Delete Error Checking
TXE_MG.C	Mutex Get Error Checking
TXE_MIG.C	Mutex Information Get Error Checking
TXE_MP.C	Mutex Put Error Checking
TXE_MPRI.C	Mutex Prioritize Error Checking

TX_QC.C	Queue Create
TX_QCLE.C	Queue Cleanup
TX_QD.C	Queue Delete
TX_QF.C	Queue Flush
TX_QFS.C	Queue Front Send
TX_QI.C	Queue Initialize
TX_QIG.C	Queue Information Get
TX_QP.C	Queue Prioritize
TX_QR.C	Queue Receive
TX_QS.C	Queue Send
TXE_QC.C	Queue Create Error Checking
TXE_QD.C	Queue Delete Error Checking
TXE_QF.C	Queue Flush Error Checking
TXE_QFS.C	Queue Front Send Error Checking
TXE_QIG.C	Queue Information Get Error Checking
TXE_QP.C	Queue Prioritize Error Checking
TXE_QR.C	Queue Receive Error Checking
TXE_QS.C	Queue Send Error Checking
TX_TA.C	Timer Activate
TX_TAA.C	Timer Activate API
TX_TD.C	Timer Deactivate
TX_TDA.C	Timer Deactivate API
TX_TIMCH.C	Timer Change
TX_TIMCR.C	Timer Create
TX_TIMD.C	Timer Delete
TX_TIMI.C	Timer Initialize
TX_TIMIG.C	Timer Information Get
TX_TTE.C	Timer Thread Entry
TXE_TAA.C	Timer Activate API Error Checking
TXE_TMCH.C	Timer Change Error Checking

TXE_TMCR.C	Timer Create Error Checking
TXE_TDA.C	Timer Deactivate API Error Checking
TXE_TIMD.C	Timer Delete Error Checking
TXE_TIMI.C	Timer Information Get Error Checking
TX_TIMEG.C	Time Get
TX_TIMES.C	Time Set
TX_TC.C	Thread Create
TX_TDEL.C	Thread Delete
TX_TI.C	Thread Initialize
TX_TIDE.C	Thread Identify
TX_TIG.C	Thread Information Get
TX_TPCH.C	Thread Preemption Change
TX_TPRCH.C	Thread Priority Change
TX_TR.C	Thread Resume
TX_TRA.C	Thread Resume API
TX_TREL.C	Thread Relinquish
TX_TSA.C	Thread Suspend API
TX_TSE.C	Thread Shell Entry
TX_TSLE.C	Thread Sleep
TX_TSUS.C	Thread Suspend
TX_TT.C	Thread Terminate
TX_TTO.C	Thread Time-out
TX_TTS.C	Thread Time Slice
TX_TTSC.C	Thread Time-slice Change
TX_TWA.C	Thread Wait Abort
TXE_TC.C	Thread Create Error Checking
TXE_TDEL.C	Thread Delete Error Checking
TXE_TIG.C	Thread Information Get Error Checking
TXE_TPCH.C	Thread Preemption Change Error Checking

TXE_TRA.C	Thread Resume API Error Checking
TXE_TREL.C	Thread Relinquish Error Checking
TXE_TRPC.C	Thread Priority Change Error Checking
TXE_TSA.C	Thread Suspend API Error Checking
TXE_TT.C	Thread Terminate Error Checking
TXE_TTSC.C	Thread Time-slice Change Error Checking
TXE_TWA.C	Thread Wait Abort Error Checking

ThreadX Port Specific Assembly Language Files

TX_ILL.[S,ASM,SRC]	Initialize Low Level
TX_TCR.[S,ASM,SRC]	Thread Contest Restore
TX_TCS.[S,ASM,SRC]	Thread Context Save
TX_TIC.[S,ASM,SRC]	Thread Interrupt Control
TX_TIMIN.[S,ASM,SRC]	Timer Interrupt Handling
TX_TPC.[S,ASM,SRC]	Thread Preempt Check (optional)
TX_TS.[S,ASM,SRC]	Tread Scheduler
TX_TSB.[S,ASM,SRC]	Thread Stack Build
TX_TSR.[S,ASM,SRC]	Thread System Return

APPENDIX E

ASCII Character Codes

ASCII Character Codes in HEX

most significant nibble

least significant nibble

	0_	**1_**	**2_**	**3_**	**4_**	**5_**	**6_**	**7_**
_0	NUL	DLE	SP	0	@	P	'	p
_1	SOH	DC1	!	1	A	Q	a	q
_2	STX	DC2	"	2	B	R	b	r
_3	ETX	DC3	#	3	C	S	c	s
_4	EOT	DC4	$	4	D	T	d	t
_5	ENQ	NAK	%	5	E	U	e	u
_6	ACK	SYN	&	6	F	V	f	v
_7	BEL	ETB	'	7	G	W	g	w
_8	BS	CAN	(	8	H	X	h	x
_9	HT	EM	)	9	I	Y	i	y
_A	LF	SUB	*	:	J	Z	j	z
_B	VT	ESC	+	;	K	[	K	}
_C	FF	FS	,	<	L	\	l	|
_D	CR	GS	-	=	M	]	m	}
_E	SO	RS	.	>	N	^	n	~
_F	SI	US	/	?	O	_	o	DEL

Index