©1997-2006 by Express Logic, Inc.

All rights reserved. This document and the associated ThreadX software are the sole property of Express Logic, Inc. Each contains proprietary information of Express Logic, Inc. Reproduction or duplication by any means of any portion of this document without the prior written consent of Express Logic, Inc. is expressly forbidden.

Express Logic, Inc. reserves the right to make changes to the specifications described herein at any time and without notice in order to improve design or reliability of ThreadX. The information in this document has been carefully checked for accuracy; however, Express Logic, Inc. makes no warranty pertaining to the correctness of this document.

Trademarks

ThreadX is a registered trademark of Express Logic, Inc., and picokernel, preemption-threshold, and event-chaining are trademarks of Express Logic, Inc.

All other product and company names are trademarks or registered trademarks of their respective holders.

Warranty Limitations

Express Logic, Inc. makes no warranty of any kind that the ThreadX products will meet the USER’s requirements, or will operate in the manner specified by the USER, or that the operation of the ThreadX products will operate uninterrupted or error free, or that any defects that may exist in the ThreadX products will be corrected after the warranty period. Express Logic, Inc. makes no warranties of any kind, either expressed or implied, including but not limited to the implied warranties of merchantability and fitness for a particular purpose, with respect to the ThreadX products. No oral or written information or advice given by Express Logic, Inc., its dealers, distributors, agents, or employees shall create any other warranty or in any way increase the scope of this warranty, and licensee may not rely on any such information or advice.

Part Number: 000-1001
Revision 5.0
Contents

About This Guide 13
- Organization 13
- Guide Conventions 14
- ThreadX Data Types 15
- Customer Support Center 16
  Latest Product Information 16
  What We Need From You 16
  Where to Send Comments About This Guide 17

1 Introduction to ThreadX 19
- ThreadX Unique Features 20
  picokernel™ Architecture 20
  ANSI C Source Code 20
  Advanced Technology 20
  Not A Black Box 21
  The RTOS Standard 22
- Embedded Applications 22
  Real-time Software 22
  Multitasking 22
  Tasks vs. Threads 23
- ThreadX Benefits 23
  Improved Responsiveness 24
  Software Maintenance 24
  Increased Throughput 24
  Processor Isolation 25
  Dividing the Application 25
  Ease of Use 26
  Improve
2 Installation and Use of ThreadX  27
- Host Considerations 28
- Target Considerations 28
- Product Distribution 29
- ThreadX Installation 30
- Using ThreadX 31
- Small Example System 32
- Troubleshooting 34
- Configuration Options 34
- ThreadX Version ID 40

3 Functional Components of ThreadX  41
- Execution Overview 44
  - Initialization 44
  - Thread Execution 44
  - Interrupt Service Routines (ISR) 44
  - Initialization 45
  - Application Timers 46
- Memory Usage 46
  - Static Memory Usage 46
  - Dynamic Memory Usage 48
- Initialization 48
  - System Reset Vector 48
  - Development Tool Initialization 49
  - main Function 49
  - tx_kernel_enter 49
  - Application Definition Function 50
  - Interrupts 50
- Thread Execution 50
  - Thread Execution States 52
  - Thread Entry/Exit Notification 54
Contents

Thread Priorities 54
Thread Scheduling 55
Round-robin Scheduling 55
Time-Slicing 55
Preemption 56
Preemption-Threshold™ 56
Priority Inheritance 57
Thread Creation 57
Thread Control Block TX_THREAD 57
Currently Executing Thread 59
Thread Stack Area 59
Memory Pitfalls 62
Optional Run-time Stack Checking 62
Reentrancy 62
Thread Priority Pitfalls 63
Priority Overhead 64
Run-time Thread Performance Information 65
Debugging Pitfalls 66

Message Queues 67
Creating Message Queues 68
Message Size 68
Message Queue Capacity 68
Queue Memory Area 69
Thread Suspension 69
Queue Send Notification 70
Queue Event-chaining™ 70
Run-time Queue Performance Information 71
Queue Control Block TX_QUEUE 72
Message Destination Pitfall 72

Counting Semaphores 72
Mutual Exclusion 73
Event Notification 73
Creating Counting Semaphores 74
Thread Suspension 74
Semaphore Put Notification 74
Semaphore Event-chaining™ 75
Run-time Semaphore Performance Information 75
Semaphore Control Block TX_SEMAPHORE 76
Deadly Embrace 76
Priority Inversion 78
Mutexes 78
Mutex Mutual Exclusion 79
Creating Mutexes 79
Thread Suspension 79
Run-time Mutex Performance Information 80
Mutex Control Block TX_MUTEX 81
Deadly Embrace 81
Priority Inversion 81

Event Flags 82
Creating Event Flags Groups 83
Thread Suspension 83
Event Flags Set Notification 83
Event Flags Event-chaining™ 84
Run-time Event Flags Performance Information 84
Event Flags Group Control Block
TX_EVENT_FLAGS_GROUP 85

Memory Block Pools 85
Creating Memory Block Pools 86
Memory Block Size 86
Pool Capacity 86
Pool's Memory Area 87
Thread Suspension 87
Run-time Block Pool Performance Information 87
Memory Block Pool Control Block TX_BLOCK_POOL 88
Overwriting Memory Blocks 89

Memory Byte Pools 89
Creating Memory Byte Pools 89
Pool Capacity 90
Pool's Memory Area 90
Thread Suspension 90
Run-time Byte Pool Performance Information 91
Memory Byte Pool Control Block TX_BYTE_POOL 92
Un-deterministic Behavior 92
Over-writing Memory Blocks 93

Application Timers 93
Timer Intervals 93
Timer Accuracy 94
Timer Execution 94
Creating Application Timers 94
Run-time Application Timer Performance Information 95
Contents

Application Timer Control Block TX_TIMER 95
Excessive Timers 96

- Relative Time 96
- Interrupts 96
  - Interrupt Control 97
  - ThreadX Managed Interrupts 97
  - ISR Template 99
  - High-frequency Interrupts 100
  - Interrupt Latency 100

4 Description of ThreadX Services 101

5 Device Drivers for ThreadX 295

- Device Driver Introduction 296
- Driver Functions 296
  - Driver Initialization 297
  - Driver Control 297
  - Driver Access 297
  - Driver Input 297
  - Driver Output 298
  - Driver Interrupts 298
  - Driver Status 298
  - Driver Termination 298
- Simple Driver Example 298
  - Simple Driver Initialization 299
  - Simple Driver Input 300
  - Simple Driver Output 301
  - Simple Driver Shortcomings 302
- Advanced Driver Issues 303
  - I/O Buffering 303
  - Circular Byte Buffers 303
  - Circular Buffer Input 303
  - Circular Output Buffer 305
  - Buffer I/O Management 306
  - TX_IO_BUFFER 306
  - Buffered I/O Advantage 307
  - Buffered Driver Responsibilities 307
6 Demonstration System for ThreadX 311

- Overview 312
- Application Define 312
  Initial Execution 313
- Thread 0 314
- Thread 1 314
- Thread 2 314
- Threads 3 and 4 315
- Thread 5 315
- Threads 6 and 7 316
- Observing the Demonstration 316
- Distribution file: demo_threadx.c 317

A ThreadX API Services 323

- Entry Function 324
- Block Memory Services 324
- Byte Memory Services 324
- Event Flags Services 325
- Interrupt Control 325
- Mutex Services 325
- Queue Services 326
- Semaphore Services 326
- Thread Control Services 327
- Time Services 328
- Timer Services 328

B ThreadX Constants 329

- Alphabetic Listings 330
- Listing by Value 332
Contents

C ThreadX Data Types 335
  TX_BLOCK_POOL 336
  TX_BYTE_POOL 336
  TX_EVENT_FLAGS_GROUP 337
  TX_MUTEX 337
  TX_QUEUE 338
  TX_SEMAPHORE 339
  TX_THREAD 339
  TX_TIMER 341
  TX_TIMER_INTERNAL 341

D ASCII Character Codes 343
  ASCII Character Codes in HEX 344

Index 345
Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Template for Application Development</td>
<td>33</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Types of Program Execution</td>
<td>46</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Memory Area Example</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4</td>
<td>Initialization Process</td>
<td>51</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Thread State Transition</td>
<td>52</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Typical Thread Stack</td>
<td>60</td>
</tr>
<tr>
<td>Figure 7</td>
<td>Stack Preset to 0xEFEF</td>
<td>61</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Example of Suspended Threads</td>
<td>77</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Simple Driver Initialization</td>
<td>300</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Simple Driver Input</td>
<td>301</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Simple Driver Output</td>
<td>302</td>
</tr>
<tr>
<td>Figure 12</td>
<td>Logic for Circular Input Buffer</td>
<td>304</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Logic for Circular Output Buffer</td>
<td>305</td>
</tr>
<tr>
<td>Figure 14</td>
<td>I/O Buffer</td>
<td>306</td>
</tr>
<tr>
<td>Figure 15</td>
<td>Input-Output Lists</td>
<td>308</td>
</tr>
</tbody>
</table>
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.
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.

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.

- **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

Latest Product Information
Visit the Express Logic web site and select the “Support” menu option to find the latest online support information, including information about the latest ThreadX product releases.

What We Need From You
Please supply us with the following information in an email message so we can more efficiently resolve your support request:

1. A detailed description of the problem, including frequency of occurrence and whether it can be reliably reproduced.
2. A detailed description of any changes to the application and/or ThreadX that preceded the problem.
3. The contents of the _tx_version_id string found in the tx_port.h file of your distribution. This string will provide us valuable information regarding your run-time environment.
4. The contents in RAM of the _tx_build_options ULONG variable. This variable will give us information on how your ThreadX library was built.
Where to Send Comments About This Guide

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 support@expresslogic.com

Enter “ThreadX User Guide” in the subject line.
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 20
  - picokernel™ Architecture 20
  - ANSI C Source Code 20
  - Advanced Technology 20
  - Not A Black Box 21
  - The RTOS Standard 22

- Embedded Applications 22
  - Real-time Software 22
  - Multitasking 22
  - Tasks vs. Threads 23

- ThreadX Benefits 23
  - Improved Responsiveness 24
  - Software Maintenance 24
  - Increased Throughput 24
  - Processor Isolation 25
  - Dividing the Application 25
  - Ease of Use 26
  - Improve Time-to-market 26
  - Protecting the Software Investment 26
ThreadX Unique Features

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

ThreadX is scalable 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**

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!

**Advanced Technology**

The following are highlights of the ThreadX advanced technology:

- Simple picokernel architecture
- Automatic scaling (small footprint)
- Deterministic processing
- Fast real-time performance
ThreadX Unique Features

• Preemptive and cooperative scheduling
• Flexible thread priority support (32-1024)
• Dynamic system object creation
• Unlimited number of system objects
• Optimized interrupt handling
• Preemption-threshold™
• Priority inheritance
• Event-chaining™
• Fast software timers
• Run-time memory management
• Run-time performance monitoring
• Run-time stack analysis
• Built-in system trace
• Vast processor support
• Vast development tool support
• Completely endian neutral

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. With 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.
The RTOS Standard

Because of its versatility, high-performance picokernel architecture, advanced technology, and demonstrated portability, ThreadX is deployed in more than 300,000,000 devices today. This effectively makes ThreadX the RTOS standard for deeply embedded applications.

Embedded Applications

Embedded applications execute on microprocessors buried within products such as wireless communication devices, 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 called the real-time software. 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 external events.

Multitasking

As mentioned, embedded applications have a dedicated purpose. To fulfill this purpose, the software must perform a variety of 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 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 may refer to an internal program segment.

In contemporary operating system discussion, there are two terms that more or less replace the use of task: 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 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 it 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 multithreaded environments like ThreadX increase overhead and have a negative impact on total system throughput. But in some cases, multithreading actually reduces overhead by eliminating all of the redundant polling that occurs in control loop environments. The overhead associated with multithreaded 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 and/or development tool environments. This, coupled with the fact that 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.
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 28
- Target Considerations 28
- Product Distribution 29
- ThreadX Installation 30
- Using ThreadX 31
- Small Example System 32
- Troubleshooting 34
- Configuration Options 34
- ThreadX Version ID 40
Host Considerations

Embedded software is usually developed on Windows or Linux (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 from within the development tool debugger. After download, the debugger is responsible for providing target execution control (go, halt, breakpoint, etc.) as well as access to memory and processor registers.

Most development tool debuggers communicate with the target hardware via on-chip debug (OCD) connections such as JTAG (IEEE 1149.1) and Background Debug Mode (BDM). Debuggers also communicate with target hardware through In-Circuit Emulation (ICE) connections. Both OCD and ICE connections provide robust solutions with minimal intrusion on the target resident software.

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_threadx.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_initialize_low_level` 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_threadx.txt` file for any additional host system considerations and/or options.*

**Product Distribution**

ThreadX is shipped on a single CD-ROM. 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 content 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_threadx.txt**  
  Text file containing specific information about the ThreadX port, including information about the target processor and the development tools.
ThreadX Installation

Installation of ThreadX is straightforward. The following instructions apply to virtually any installation. However, examine the readme_threadx.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.
Using ThreadX

Application software needs access to the ThreadX library file (usually \texttt{tx.a} or \texttt{tx.lib}) and the C include files \texttt{tx_api.h} and \texttt{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 \texttt{tx_api.h} during compilation and link with the ThreadX run-time library \texttt{tx.a} (or \texttt{tx.lib}).

There are four steps required to build a ThreadX application:

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

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

   \textit{The ThreadX entry function \texttt{tx_kernel_enter} does not return. So be sure not to place any processing or function calls after it.}

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

4. **Step 4:** Compile application source and link with the ThreadX run-time library \texttt{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 33 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.
Although this is a simple example, it provides a good template for real application development. Once again, please see the `readme_threadx.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 simulated environment.

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

If the demonstration system does not execute properly, the following are some troubleshooting tips:

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. Rebuild the ThreadX library with TX_ENABLE_STACK_CHECKING defined. This will enable the built-in ThreadX stack checking.
4. Temporarily bypass any recent changes to see if the problem disappears or changes. Such information should prove useful to Express Logic support engineers.

Follow the procedures outlined in “What We Need From You” on page 16 to send the information gathered from the troubleshooting steps.

Configuration Options

There are several configuration options when building the ThreadX library and the application using ThreadX. The options below can be defined in the application source, on the command line, or within the tx_user.h include file.
Options defined in `tx_user.h` are applied only if the application and ThreadX library are built with `TX_INCLUDE_USER_DEFINE_FILE` defined.

Review the `readme_threadx.txt` file for additional options for your specific version of ThreadX. The following describes each configuration option in detail:
Define  | Meaning
--- | ---
**TX_DISABLE_ERROR_CHECKING**  | Bypasses basic service call error checking. When defined in the application source, all basic parameter error checking is disabled. This may improve performance by as much as 30% and may also reduce the image size. Of course, this option should only be used after the application is thoroughly debugged. By default, this option is not defined.

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

**TX_MAX_PRIORITIES**  | Defines the priority levels for ThreadX. Legal values range from 32 through 1024 (inclusive) and must be evenly divisible by 32. Increasing the number of priority levels supported increases the RAM usage by 128 bytes for every group of 32 priorities. However, there is only a negligible effect on performance. By default, this value is set to 32 priority levels.
### Configuration Options

<table>
<thead>
<tr>
<th>Define</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_MINIMUM_STACK</td>
<td>Defines the minimum stack size (in bytes). It is used for error checking when threads are created. The default value is port-specific and is found in <code>tx_port.h</code>.</td>
</tr>
<tr>
<td>TX_TIMER_THREAD_STACK_SIZE</td>
<td>Defines the stack size (in bytes) of the internal ThreadX system timer thread. This thread processes all thread sleep requests as well as all service call timeouts. In addition, all application timer callback routines are invoked from this context. The default value is port-specific and is found in <code>tx_port.h</code>.</td>
</tr>
<tr>
<td>TX_TIMER_THREAD_PRIORITY</td>
<td>Defines the priority of the internal ThreadX system timer thread. The default value is priority 0—the highest priority in ThreadX. The default value is defined in <code>tx_port.h</code>.</td>
</tr>
<tr>
<td>TX_TIMER_PROCESS_IN_ISR</td>
<td>When defined, eliminates the internal system timer thread for ThreadX. This results in improved performance on timer events and smaller RAM requirements because the timer stack and control block are no longer needed. However, using this option moves all the timer expiration processing to the timer ISR level. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_REACTIVATE_INLINE</td>
<td>When defined, performs reactivation of ThreadX timers in-line instead of using a function call. This improves performance but slightly increases code size. By default, this option is not defined.</td>
</tr>
<tr>
<td>Define</td>
<td>Meaning</td>
</tr>
<tr>
<td>----------------------------------------</td>
<td>-----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TX_DISABLE_STACK_FILLING</td>
<td>When defined, disables placing the 0xEF value in each byte of each thread's stack when created. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_ENABLE_STACK_CHECKING</td>
<td>When defined, enables ThreadX run-time stack checking, which includes analysis of how much stack has been used and examination of data pattern “fences” before and after the stack area. If a stack error is detected, the registered application stack error handler is called. This option does result in slightly increased overhead and code size. Review the tx_thread_stack_error_notify API for more information. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_DISABLE_PREEMPTION_THRESHOLD</td>
<td>When defined, disables the preemption-threshold feature and slightly reduces code size and improves performance. Of course, the preemption-threshold capabilities are no longer available. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_DISABLE_REDUNDANT_CLEARING</td>
<td>When defined, removes the logic for initializing ThreadX global C data structures to zero. This should only be used if the compiler’s initialization code sets all un-initialized C global data to zero. Using this option slightly reduces code size and improves performance during initialization. By default, this option is not defined.</td>
</tr>
<tr>
<td>Define</td>
<td>Meaning</td>
</tr>
<tr>
<td>---------------------------------------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>TX_DISABLE_NOTIFY_CALLBACKS</td>
<td>When defined, disables the notify callbacks for various ThreadX objects. Using this option slightly reduces code size and improves performance. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO</td>
<td>When defined, enables the gathering of performance information on block pools. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO</td>
<td>When defined, enables the gathering of performance information on byte pools. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO</td>
<td>When defined, enables the gathering of performance information on event flags groups. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_MUTEX_ENABLE_PERFORMANCE_INFO</td>
<td>When defined, enables the gathering of performance information on mutexes. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_QUEUE_ENABLE_PERFORMANCE_INFO</td>
<td>When defined, enables the gathering of performance information on queues. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO</td>
<td>When defined, enables the gathering of performance information on semaphores. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_THREAD_ENABLE_PERFORMANCE_INFO</td>
<td>Defined, enables the gathering of performance information on threads. By default, this option is not defined.</td>
</tr>
<tr>
<td>TX_TIMER_ENABLE_PERFORMANCE_INFO</td>
<td>Defined, enables the gathering of performance information on timers. By default, this option is not defined.</td>
</tr>
</tbody>
</table>
ThreadX Version ID

The ThreadX version ID can be found in the readme_threadx.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 44
  - Initialization 44
  - Thread Execution 44
  - Interrupt Service Routines (ISR) 44
- Memory Usage 46
  - Static Memory Usage 46
  - Dynamic Memory Usage 48
- Initialization 48
  - System Reset Vector 48
  - Development Tool Initialization 49
  - main Function 49
  - tx_kernel_enter 49
  - Application Definition Function 50
  - Interrupts 50
- Thread Execution 50
  - Thread Execution States 52
  - Thread Entry/Exit Notification 54
  - Thread Priorities 54
  - Thread Scheduling 55
  - Round-robin Scheduling 55
  - Time-Slicing 55
  - Preemption 56
  - Preemption-Threshold™ 56
  - Priority Inheritance 57
  - Thread Creation 57
Thread Control Block TX_THREAD 57
Currently Executing Thread 59
Thread Stack Area 59
Memory Pitfalls 62
Optional Run-time Stack Checking 62
Reentrancy 62
Thread Priority Pitfalls 63
Priority Overhead 64
Run-time Thread Performance Information 65
Debugging Pitfalls 67

- Message Queues 67
  Creating Message Queues 68
  Message Size 68
  Message Queue Capacity 68
  Queue Memory Area 69
  Thread Suspension 69
  Queue Send Notification 70
  Queue Event-chaining™ 70
  Run-time Queue Performance Information 71
  Queue Control Block TX_QUEUE 72
  Message Destination Pitfall 72

- Counting Semaphores 72
  Mutual Exclusion 73
  Event Notification 73
  Creating Counting Semaphores 74
  Thread Suspension 74
  Semaphore Put Notification 74
  Semaphore Event-chaining™ 75
  Run-time Semaphore Performance Information 75
  Semaphore Control Block TX_SEMAPHORE 76
  Deadly Embrace 76
  Priority Inversion 78

- Mutexes 78
  Mutex Mutual Exclusion 79
  Creating Mutexes 79
  Thread Suspension 79
  Run-time Mutex Performance Information 80
  Mutex Control Block TX_MUTEX 81
  Deadly Embrace 81
  Priority Inversion 81

- Event Flags 82
Creating Event Flags Groups 83
Thread Suspension 83
Event Flags Set Notification 83
Event Flags Event-chaining™ 84
Run-time Event Flags Performance Information 84
Event Flags Group Control Block TX_EVENT_FLAGS_GROUP 85

- Memory Block Pools 85
  Creating Memory Block Pools 86
  Memory Block Size 86
  Pool Capacity 86
  Pool's Memory Area 87
  Thread Suspension 87
  Run-time Block Pool Performance Information 87
  Memory Block Pool Control Block TX_BLOCK_POOL 88
  Overwriting Memory Blocks 89

- Memory Byte Pools 89
  Creating Memory Byte Pools 89
  Pool Capacity 90
  Pool's Memory Area 90
  Thread Suspension 90
  Run-time Byte Pool Performance Information 91
  Memory Byte Pool Control Block TX_BYTE_POOL 92
  Un-deterministic Behavior 92
  Overwriting Memory Blocks 93

- Application Timers 93
  Timer Intervals 93
  Timer Accuracy 94
  Timer Execution 94
  Creating Application Timers 94
  Run-time Application Timer Performance Information 95
  Application Timer Control Block TX_TIMER 95
  Excessive Timers 96

- Relative Time 96

- Interrupts 97
  Interrupt Control 97
  ThreadX Managed Interrupts 97
  ISR Template 99
  High-frequency Interrupts 100
  Interrupt Latency 100
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 45 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. After the thread is finished (or another higher-priority thread becomes ready), execution transfers back to the thread scheduling loop 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. On detection of an interrupt, the processor saves key information about the current 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.

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.

FIGURE 2. Types of Program Execution
Application Timers

Application Timers are similar to ISRs, except the 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 47 shows an example of these memory areas.

It is important to understand 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.

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
In the compiler’s initialization process, this portion of the constant area is used to set up 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 set up immediately following the initialized and uninitialized data areas.

FIGURE 3. Memory Area Example

*Express Logic, Inc.*
The system stack is used by the compiler during initialization, 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 may greatly enhance its performance.

**Initialization**

Understanding the initialization process is important. The initial hardware environment is set up 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 Vector**

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 set up. Remember their initial values are retrieved from the constant area. Exact initialization processing is development tool specific.

**main Function**

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 do not 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 51 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. A thread is typically defined as a semi-independent program segment with a dedicated purpose. The combined processing of all threads makes an application.

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

Initialization Process

- System Reset Vector
  - entry point*
    - development tool initialization*
      - main()
        - tx_kernel_enter()
          - tx_application_define(mem_ptr)

* denotes functions that are development-tool specific

FIGURE 4. Initialization Process
Understanding the different processing states of threads is a key ingredient to understanding the entire multithreaded environment. In ThreadX there are five distinct thread states: `ready`, `suspended`, `executing`, `terminated`, and `completed`. Figure 5 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 in ready state. When this happens, ThreadX executes the thread, which then 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.

At any given moment, only one thread is in an *executing* state. This is because a thread in the *executing* state has control of the underlying processor.

Threads 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. After the cause for suspension is removed, the thread is placed back in a *ready* state.

A thread in a *completed* state is a thread that has completed its processing and returned from its entry function. 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 the thread 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 Entry/Exit Notification

Some applications may find it advantageous to be notified when a specific thread is entered for the first time, when it completes, or is terminated. ThreadX provides this ability through the tx_thread_entry_exit_notify service. This service registers an application notification function for a specific thread, which is called by ThreadX whenever the thread starts running, completes, or is terminated. After being invoked, the application notification function can perform the application-specific processing. This typically involves informing another application thread of the event via a ThreadX synchronization primitive.

Thread Priorities

As mentioned before, a thread is 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.

ThreadX determines a thread's importance when the thread is created by assigning a numerical value representing its priority. The maximum number of ThreadX priorities is configurable from 32 through 1024 in increments of 32. The actual maximum number of priorities is determined by the TX_MAX_PRIORITIES constant during compilation of the ThreadX library. Having a larger number of priorities does not significantly increase processing overhead. However, for each group of 32 priority levels an additional 128 bytes of RAM is required to manage them. For example, 32 priority levels require 128 bytes of RAM, 64 priority levels require 256 bytes of RAM, and 96 priority levels requires 384 bytes of RAM.

By default, ThreadX has 32 priority levels, ranging from priority 0 through priority 31. Numerically
smaller values imply higher priority. Hence, priority 0 represents the highest priority, while priority 
(TX_MAX_PRIORITIES-1) represents the lowest priority.

Multiple threads can have the same priority relying on cooperative scheduling or timeslicing. In addition, thread priorities can be changed during run-time.

Thread Scheduling
ThreadX schedules threads based on 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
ThreadX supports round-robin scheduling of multiple threads having the same priority. This is accomplished through cooperative calls to tx_thread_relinquish. This service gives all other ready threads of the same priority a chance to execute before the tx_thread_relinquish caller executes again.

Time-Slicing
Time-slicing is another form of round-robin scheduling. A time-slice specifies the maximum number of timer ticks (timer interrupts) that a thread can execute without giving up the processor. 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. 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. Because time-slicing is only useful in cases in which 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™**

To ease some of the inherent problems of preemption, ThreadX provides a unique and advanced feature called *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-threshold can be changed during run-time.

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 is no limit 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, avoid using local storage for control blocks because after the function returns, 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 during debug, in which looking at certain members is useful. The following are some of the more useful control block members:

**tx_thread_run_count**

contains a counter of the number of many times the thread has been scheduled. An increasing counter indicates the thread is being scheduled and executed.

**tx_thread_state**

contains the state of the associated thread. The following lists the possible thread states:

<table>
<thead>
<tr>
<th>State</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_READY</td>
<td>0x00</td>
</tr>
<tr>
<td>TX_COMPLETED</td>
<td>0x01</td>
</tr>
<tr>
<td>TX_TERMINATED</td>
<td>0x02</td>
</tr>
<tr>
<td>TX_SUSPENDED</td>
<td>0x03</td>
</tr>
<tr>
<td>TX_SLEEP</td>
<td>0x04</td>
</tr>
<tr>
<td>TX_QUEUE_SUSP</td>
<td>0x05</td>
</tr>
<tr>
<td>TX_SEMAPHORE_SUSP</td>
<td>0x06</td>
</tr>
<tr>
<td>TX_EVENT_FLAG</td>
<td>0x07</td>
</tr>
<tr>
<td>TX_BLOCK_MEMORY</td>
<td>0x08</td>
</tr>
<tr>
<td>TX_BYTE_MEMORY</td>
<td>0x09</td>
</tr>
<tr>
<td>TX_MUTEX_SUSP</td>
<td>0x0D</td>
</tr>
</tbody>
</table>
Of course there are many other interesting fields in the thread control block, including the stack pointer, time-slice value, priorities, etc. Users are welcome to review control block members, but modifications are strictly prohibited!

There is no equate for the “executing” state mentioned earlier in this section. It is not necessary because 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 which thread 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 on page 60 shows a typical thread’s stack.

Where a thread stack is located in memory is 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 an important feature because it allows applications to improve performance of important threads by placing their stack in high-speed RAM.

How big a stack should be 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 saving a thread’s context and minimum amount of function calls and local variable allocation.

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

After the application is debugged, it is possible to tune the thread stack 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 stack was actually used by finding the area of the stack where the data pattern is still intact. Figure 7 shows a stack preset to 0xEFEF after thorough thread execution.

**Stack Memory Area**

(another example)

![Stack Memory Area Diagram](image)

**FIGURE 7. Stack Preset to 0xEFEF**

By default, ThreadX initializes every byte of each thread stack with a value of 0xEF.
Memory Pitfalls

The stack requirements for threads can be 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 be avoided.

In most cases, an overflowed stack causes thread execution to corrupt memory adjacent (usually before) its stack area. The results are 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 all thread stacks are large enough.

Optional Run-time Stack Checking

ThreadX provides the ability to check each thread’s stack for corruption during run-time. By default, ThreadX fills every byte of thread stacks with a 0xEF data pattern during creation. If the application builds the ThreadX library with `TX_ENABLE_STACK_CHECKING` defined, ThreadX will examine each thread’s stack for corruption as it is suspended or resumed. If stack corruption is detected, ThreadX will call the application’s stack error handling routine as specified by the call to `tx_thread_stack_error_notify`. Otherwise, if no stack error handler was specified, ThreadX will call the internal `_tx_thread_stack_error_handler` routine.

Reentrancy

One of the real beauties of multithreading 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**.

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 set.
Thread Execution

up. 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 multithreading. 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 thread 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 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 multithreading 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, it goes back and waits for another message.

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 before the execution of each thread. 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 an 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.

**Run-time Thread Performance Information**

ThreadX provides optional run-time thread performance information. If the ThreadX library and application is built with `TX_THREAD_ENABLE_PERFORMANCE_INFO` defined, ThreadX accumulates the following information:

Total number for the overall system:

- thread resumptions
- thread suspensions
- service call preemptions
- interrupt preemptions
Functional Components of ThreadX

- priority inversions
- time-slices
- relinquishes
- thread timeouts
- suspension aborts
- idle system returns
- non-idle system returns

Total number for each thread:
- resumptions
- suspensions
- service call preemptions
- interrupt preemptions
- priority inversions
- time-slices
- thread relinquishes
- thread timeouts
- suspension aborts

This information is available at run-time through the services `tx_thread_performance_info_get` and `tx_thread_performance_system_info_get`. Thread performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of service call preemptions might suggest the thread’s priority and/or preemption-threshold is too low. Furthermore, a relatively low number of idle system returns might suggest that lower priority threads are not suspending enough.

Debugging Pitfalls

Debugging multithreaded 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 using a conditional
breakpoint to see if the calling thread is the one to
debug.

Much of this is being handled in multithreading
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
multithreading. Whenever unexplained behavior is
observed, it is usually a good first guess to increase
stack sizes for all threads—especially the stack size
of the last thread to execute!

It is also a good idea to build the ThreadX library with
\texttt{TX\_ENABLE\_STACK\_CHECKING} defined. This will
help isolate stack corruption problems as early in the
processing as possible!

### 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 \textit{mailbox}.

Messages are copied to a queue by \texttt{tx\_queue\_send}
and are copied from a queue by \texttt{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 is no limit 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 through 16 32-bit words inclusive. 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 an important queue in high-speed RAM 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 Send Notification

Some applications may find it advantageous to be notified whenever a message is placed on a queue. ThreadX provides this ability through the `tx_queue_send_notify` service. This service registers the supplied application notification function with the specified queue. ThreadX will subsequently invoke this application notification function whenever a message is sent to the queue. The exact processing within the application notification function is determined by the application; however, it typically consists of resuming the appropriate thread for processing the new message.

Queue Event-chaining™

The notification capabilities in ThreadX can be used to chain various synchronization events together. This is typically useful when a single thread must process multiple synchronization events.

For example, suppose a single thread is responsible for processing messages from five different queues and must also suspend when no messages are available. This is easily accomplished by registering an application notification function for each queue and introducing an additional counting semaphore. Specifically, the application notification function performs a `tx_semaphore_put` whenever it is called (the semaphore count represents the total number of messages in all five queues). The processing thread suspends on this semaphore via the `tx_semaphore_get` service. When the semaphore is available (in this case, when a message is available!), the processing thread is resumed. It then interrogates each queue for a message, processes the found message, and performs another `tx_semaphore_get` to wait for the next message. Accomplishing this without event-chaining is quite difficult and likely would require more threads and/or additional application code.
In general, event-chaining results in fewer threads, less overhead, and smaller RAM requirements. It also provides a highly flexible mechanism to handle synchronization requirements of more complex systems.

Run-time Queue Performance Information

ThreadX provides optional run-time queue performance information. If the ThreadX library and application is built with `TX_QUEUE_ENABLE_PERFORMANCE_INFO` defined, ThreadX accumulates the following information:

Total number for the overall system:
- messages sent
- messages received
- queue empty suspensions
- queue full suspensions
- queue full error returns (suspension not specified)
- queue timeouts

Total number for each queue:
- messages sent
- messages received
- queue empty suspensions
- queue full suspensions
- queue full error returns (suspension not specified)
- queue timeouts

This information is available at run-time through the services `tx_queue_performance_info_get` and `tx_queue_performance_system_info_get`. Queue performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of “queue full suspensions”
suggests an increase in the queue size might be beneficial.

**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 important to ensure 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 will not increase until the producer has something ready for the consumer. Semaphores used for event notification may also benefit from use of the tx_semaphore_ceiling_put service call. This service ensures that the semaphore count never exceeds the value supplied in the call.
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 is no limit 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.

After 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 Put Notification

Some applications may find it advantageous to be notified whenever a semaphore is put. ThreadX provides this ability through the `tx_semaphore_put_notify` service. This service registers the supplied application notification function with the specified semaphore. ThreadX will subsequently invoke this application notification function whenever the semaphore is put. The exact processing within the application notification function is determined by the application; however, it typically consists of resuming the appropriate thread for processing the new semaphore put event.
The notification capabilities in ThreadX can be used to chain various synchronization events together. This is typically useful when a single thread must process multiple synchronization events.

For example, instead of having separate threads suspend for a queue message, event flags, and a semaphore, the application can register a notification routine for each object. When invoked, the application notification routine can then resume a single thread, which can interrogate each object to find and process the new event.

In general, event-chaining results in fewer threads, less overhead, and smaller RAM requirements. It also provides a highly flexible mechanism to handle synchronization requirements of more complex systems.

ThreadX provides optional run-time semaphore performance information. If the ThreadX library and application is built with TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO defined, ThreadX accumulates the following information.

Total number for the overall system:
- semaphore puts
- semaphore gets
- semaphore get suspensions
- semaphore get timeouts

Total number for each semaphore:
- semaphore puts
- semaphore gets
- semaphore get suspensions
- semaphore get timeouts
This information is available at run-time through the services `tx_semaphore_performance_info_get` and `tx_semaphore_performance_system_info_get`. Semaphore performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of “semaphore get timeouts” might suggest that other threads are holding resources too long.

**Semaphore Control Block TX_SEMAPHORE**

The characteristics of each counting semaphore are found in its control block. It contains 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 in which two or more threads are suspended indefinitely while attempting to get semaphores already owned by each other.

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 77 illustrates this example.
For real-time systems, deadly embraces can be prevented by placing certain restrictions on how threads obtain semaphores. Threads can only have one semaphore at a time. Alternatively, threads can own multiple semaphores if they 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 63.

The basic problem results from a situation in which 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-threshold, 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: `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. 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. After the mutex is obtained by a thread, the ownership count is incremented once for every successful get operation performed on the mutex and decremented for every successful 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.” A mutex may also be created with priority inheritance selected.

Thread Suspension

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

After 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. 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.
ThreadX provides optional run-time mutex performance information. If the ThreadX library and application is built with `TX_MUTEX_ENABLE_PERFORMANCE_INFO` defined, ThreadX accumulates the following information.

Total number for the overall system:
- mutex puts
- mutex gets
- mutex get suspensions
- mutex get timeouts
- mutex priority inversions
- mutex priority inheritances

Total number for each mutex:
- mutex puts
- mutex gets
- mutex get suspensions
- mutex get timeouts
- mutex priority inversions
- mutex priority inheritances

This information is available at run-time through the services `tx_mutex_performance_info_get` and `tx_mutex_performance_system_info_get`. Mutex performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of “mutex get timeouts” might suggest that other threads are holding resources too long.
Mutex Control Block TX_MUTEX

The characteristics of each mutex are found in its control block. It contains 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 on page 76 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 63.

The basic problem results from a situation in which 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 undeterministic 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 undeterministic 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 flags retrieval is specified in the `tx_event_flags_get` call.

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

**Creating Event Flags Groups**

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

**Thread Suspension**

Application threads can suspend while attempting to get any logical combination of event flags from a group. After 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.

*i*  

*All suspended threads on an event flags group are reviewed when its event flags are set. This, of course, introduces additional overhead. Therefore, it is good practice to limit the number of threads using the same event flags group to a reasonable number.*

**Event Flags Set Notification**

Some applications may find it advantageous to be notified whenever an event flag is set. ThreadX provides this ability through the `tx_event_flags_set_notify` service. This service registers the supplied application notification function with the specified event flags group. ThreadX will subsequently invoke this application notification function whenever an event flag in the group is set. The exact processing within the application notification function is determined by the application, but it typically consists of resuming the appropriate thread for processing the new event flag.
Event Flags Event-chaining™

The notification capabilities in ThreadX can be used to “chain” various synchronization events together. This is typically useful when a single thread must process multiple synchronization events.

For example, instead of having separate threads suspend for a queue message, event flags, and a semaphore, the application can register a notification routine for each object. When invoked, the application notification routine can then resume a single thread, which can interrogate each object to find and process the new event.

In general, event-chaining results in fewer threads, less overhead, and smaller RAM requirements. It also provides a highly flexible mechanism to handle synchronization requirements of more complex systems.

Run-time Event Flags Performance Information

ThreadX provides optional run-time event flags performance information. If the ThreadX library and application is built with `TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO` defined, ThreadX accumulates the following information.

Total number for the overall system:
- event flags sets
- event flags gets
- event flags get suspensions
- event flags get timeouts

Total number for each event flags group:
- event flags sets
- event flags gets
- event flags get suspensions
- event flags get timeouts
This information is available at run-time through the services `tx_event_flags_performance_info_get` and `tx_event_flags_performance_system_info_get`. Event Flags performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of timeouts on the `tx_event_flags_get` service might suggest that the event flags suspension timeout is too short.

The characteristics of each event flags group are found in its control block. It contains information such as the current event flags 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.

Because 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 block 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 is no limit 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 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 provides. 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 the thread is 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.

### Run-time Block Pool Performance Information

ThreadX provides optional run-time block pool performance information. If the ThreadX library and application is built with `TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO`
defined, ThreadX accumulates the following information.

Total number for the overall system:
- blocks allocated
- blocks released
- allocation suspensions
- allocation timeouts

Total number for each block pool:
- blocks allocated
- blocks released
- allocation suspensions
- allocation timeouts

This information is available at run-time through the services `tx_block_pool_performance_info_get` and `tx_block_pool_performance_system_info_get`. Block pool performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of "allocation suspensions" might suggest that the block pool is too small.

**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 available and the memory pool block 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 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 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 cannot be called from ISRs.

Creating Memory Byte Pools

Memory byte pools are created either during initialization or during run-time by application threads. There is no limit 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 ensure 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 provides. 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 the threads are 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.

**Run-time Byte Pool Performance Information**

ThreadX provides optional run-time byte pool performance information. If the ThreadX library and application is built with `TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO` defined, ThreadX accumulates the following information.

Total number for the overall system:
- allocations
- releases
- fragments searched
- fragments merged
- fragments created
- allocation suspensions
- allocation timeouts

Total number for each byte pool:
- allocations
- releases
- fragments searched
- fragments merged
- fragments created
- allocation suspensions
- allocation timeouts
This information is available at run-time through the services `tx_byte_pool_performance_info_get` and `tx_byte_pool_performance_system_info_get`. Byte pool performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application. For example, a relatively high number of “allocation suspensions” might suggest that the byte pool is too small.

### 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 exists, 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 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 important to ensure 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 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_initialize_low_level assembly file.

It is worth mentioning that the underlying hardware must have the ability to generate periodic interrupts 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. Because 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 the order they were activated.

Creating Application Timers

Application timers are created either during initialization or during run-time by application threads. There is no limit on the number of application timers in an application.
ThreadX provides optional run-time application timer performance information. If the ThreadX library and application are built with TX_TIMER_ENABLE_PERFORMANCE_INFO defined, ThreadX accumulates the following information.

Total number for the overall system:
- activations
- deactivations
- reactivations (periodic timers)
- expirations
- expiration adjustments

Total number for each application timer:
- activations
- deactivations
- reactivations (periodic timers)
- expirations
- expiration adjustments

This information is available at run-time through the services tx_timer_performance_info_get and tx_timer_performance_system_info_get. Application Timer performance information is useful in determining if the application is behaving properly. It is also useful in optimizing the application.

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 typically 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, 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. The application knows such an event is present through hardware interrupts.

An interrupt is an asynchronous change in processor execution. Typically, when an interrupt occurs, the
Interrupts

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 an interrupt that 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:

```c
#include <tx_api.h>

void thread_xISR(void* arg)
{
    int32_t intNum;

    // Get the interrupt number
    intNum = tx_interrupt_get_num();

    // Call the ThreadX service
    tx_block_allocate;
    tx_block_pool_info_get;
    tx_block_pool_prioritize;
    tx_block_pool_performance_info_get;
    tx_block_pool_performance_system_info_get;
    tx_block_release;
    tx_byte_pool_info_get;
    tx_byte_pool_performance_info_get;
    tx_byte_pool_performance_system_info_get;
    tx_byte_pool_prioritize;
}
```
Suspension is not allowed from ISRs. Therefore, the \textit{wait_option} parameter for all ThreadX service calls made from an ISR must be set to \texttt{TX\_NO\_WAIT}.
ISR Template

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. Review the `readme_threadx.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.
Functional Components of ThreadX

(application_ISR_vector_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 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.
Description of ThreadX Services

This chapter contains a description of all ThreadX services in alphabetic order. Their names are designed so all similar services are grouped together. In the “Return Values” section in the following descriptions, values in **BOLD** are not affected by the `TX_DISABLE_ERROR_CHECKNG` define used to disable API error checking; while values shown in non-bold are completely disabled. In addition, a “Yes” listed under the “**Preemption Possible**” heading indicates that calling the service may resume a higher-priority thread, thus preempting the calling thread.

- `tx_block_allocate`
  *Allocate fixed-size block of memory 108*

- `tx_block_pool_create`
  *Create pool of fixed-size memory blocks 112*

- `tx_block_pool_delete`
  *Delete memory block pool 114*

- `tx_block_pool_info_get`
  *Retrieve information about block pool 116*

- `tx_block_pool_performance_info_get`
  *Get block pool performance information 118*

- `tx_block_pool_performance_system_info_get`
  *Get block pool system performance information 120*

- `tx_block_pool_prioritize`
  *Prioritize block pool suspension list 122*

- `tx_block_release`
  *Release fixed-size block of memory 124*

- `tx_byte_allocate`
  *Allocate bytes of memory 126*
Description of ThreadX Services

- `tx_byte_pool_create`
  - Create memory pool of bytes 130

- `tx_byte_pool_delete`
  - Delete memory byte pool 132

- `tx_byte_pool_info_get`
  - Retrieve information about byte pool 134

- `tx_byte_pool_performance_info_get`
  - Get byte pool performance information 136

- `tx_byte_pool_performance_system_info_get`
  - Get byte pool system performance information 138

- `tx_byte_pool_prioritize`
  - Prioritize byte pool suspension list 140

- `tx_byte_release`
  - Release bytes back to memory pool 142

- `tx_event_flags_create`
  - Create event flags group 144

- `tx_event_flags_delete`
  - Delete event flags group 146

- `tx_event_flags_get`
  - Get event flags from event flags group 148

- `tx_event_flags_info_get`
  - Retrieve information about event flags group 152

- `tx_event_flags_performance_info_get`
  - Get event flags group performance information 154

- `tx_event_flags_performance_system_info_get`
  - Retrieve performance system information 156

- `tx_event_flags_set`
  - Set event flags in an event flags group 158

- `tx_event_flags_set_notify`
  - Notify application when event flags are set 160

- `tx_interrupt_control`
  - Enable and disable interrupts 162
tx_mutex_create
  Create mutual exclusion mutex 164

tax_mutex_delete
  Delete mutual exclusion mutex 166

tax_mutex_get
  Obtain ownership of mutex 168

tax_mutex_info_get
  Retrieve information about mutex 170

tax_mutex_performance_info_get
  Get mutex performance information 172

tax_mutex_performance_system_info_get
  Get mutex system performance information 174

tax_mutex_prioritize
  Prioritize mutex suspension list 176

tax_mutex_put
  Release ownership of mutex 178

tax_queue_create
  Create message queue 180

tax_queue_delete
  Delete message queue 182

tax_queue_flush
  Empty messages in message queue 184

tax_queue_front_send
  Send message to the front of queue 186

tax_queue_info_get
  Retrieve information about queue 190

tax_queue_performance_info_get
  Get queue performance information 192

tax_queue_performance_system_info_get
  Get queue system performance information 194

tax_queue_prioritize
  Prioritize queue suspension list 196

Express Logic, Inc.
**Description of ThreadX Services**

- `tx_queue_receive`  
  *Get message from message queue 198*

- `tx_queue_send`  
  *Send message to message queue 202*

- `tx_queue_send_notify`  
  *Notify application when message is sent to queue 206*

- `tx_semaphore_ceiling_put`  
  *Place an instance in counting semaphore with ceiling 208*

- `tx_semaphore_create`  
  *Create counting semaphore 210*

- `tx_semaphore_delete`  
  *Delete counting semaphore 212*

- `tx_semaphore_get`  
  *Get instance from counting semaphore 214*

- `tx_semaphore_info_get`  
  *Retrieve information about semaphore 218*

- `tx_semaphore_performance_info_get`  
  *Get semaphore performance information 220*

- `tx_semaphore_performance_system_info_get`  
  *Get semaphore system performance information 222*

- `tx_semaphore_prioritize`  
  *Prioritize semaphore suspension list 224*

- `tx_semaphore_put`  
  *Place an instance in counting semaphore 226*

- `tx_semaphore_put_notify`  
  *Notify application when semaphore is put 228*

- `tx_thread_create`  
  *Create application thread 230*

- `tx_thread_delete`  
  *Delete application thread 234*

- `tx_thread_entry_exit_notify`  
  *Notify application upon thread entry and exit 236*
tx_thread_identify
  Retrieves pointer to currently executing thread 238

tx_thread_info_get
  Retrieve information about thread 240

tx_thread_performance_info_get
  Get thread performance information 244

tx_thread_performance_system_info_get
  Get thread system performance information 248

tx_thread_preemption_change
  Change preemption-threshold of application thread 252

tx_thread_priority_change
  Change priority of application thread 254

tx_thread_relinquish
  Relinquish control to other application threads 256

tx_thread_reset
  Reset thread 258

tx_thread_resume
  Resume suspended application thread 260

tx_thread_sleep
  Suspend current thread for specified time 262

tx_thread_stack_error_notify
  Register thread stack error notification callback 264

tx_thread_suspend
  Suspend application thread 266

tx_thread_terminate
  Terminates application thread 268

tx_thread_time_slice_change
  Changes time-slice of application thread 270

tx_thread_wait_abort
  Abort suspension of specified thread 272

tx_time_get
  Retrieves the current time 274
Description of ThreadX Services

- **tx_time_set**
  - *Sets the current time* 276

- **tx_timer_activate**
  - *Activate application timer* 278

- **tx_timer_change**
  - *Change application timer* 280

- **tx_timer_create**
  - *Create application timer* 282

- **tx_timer_deactivate**
  - *Deactivate application timer* 284

- **tx_timer_delete**
  - *Delete application timer* 286

- **tx_timer_info_get**
  - *Retrieve information about an application timer* 288

- **tx_timer_performance_info_get**
  - *Get timer performance information* 290

- **tx_timer_performance_system_info_get**
  - *Get timer system performance information* 292
Tx_block_allocate

Allocate 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.

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 0xFFFFFFFF)

Selecting TX_NO_WAIT results in an immediate return from this service regardless if it was successful or not. *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 (1-0xFFFFFFFF) specifies the maximum number of timer-ticks to
Memory Blocks

stay suspended while waiting for a memory block.

Return Values

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```c
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_performance_info_get
- tx_block_pool_performance_system_info_get
- tx_block_pool_prioritize
- tx_block_release
tx_block_pool_create

Create pool of fixed-size memory blocks

Prototype

```c
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

```c
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_performance_info_get,
tx_block_pool_performance_system_info_get, tx_block_pool_prioritize,
tx_block_release
tx_block_pool_delete

Delete memory block 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

```c
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_performance_info_get`
- `tx_block_pool_performance_system_info_get`
- `tx_block_pool_prioritize`
- `tx_block_release`
Description of ThreadX Services

**tx_block_pool_info_get**

Retrieve information about block pool

**Prototype**

```c
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.

Supplying a TX_NULL for any parameter indicates the parameter is not required.
Return Values

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
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 the previously created
   block pool "my_pool." */
status = tx_block_pool_info_get(&my_pool, &name,
                             &available,&total_blocks,
                             &first_suspended, &suspended_count,
                             &next_pool);

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

See Also

tx_block_allocate, tx_block_pool_create, tx_block_pool_delete,
tx_block_pool_info_get, tx_block_pool_performance_info_get,
tx_block_pool_performance_system_info_get, tx_block_pool_prioritize,
tx_block_release
tx_block_pool_performance_info_get
Get block pool performance information

Prototype
UINT tx_block_pool_performance_info_get(TX_BLOCK_POOL *pool_ptr,
ULONG *allocates, ULONG *releases,
ULONG *suspensions, ULONG *timeouts)

Description
This service retrieves performance information about the specified
memory block pool.

The ThreadX library and application must be built with
TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO defined for this
service to return performance information.

Input Parameters
pool_ptr Pointer to previously created memory block pool.
allocates Pointer to destination for the number of allocate
requests performed on this pool.
releases Pointer to destination for the number of release
requests performed on this pool.
suspensions Pointer to destination for the number of thread
allocation suspensions on this pool.
timeouts Pointer to destination for the number of allocate
suspension timeouts on this pool.

Supplying a TX_NULL for any parameter indicates that the parameter is
not required.
Return Values

<table>
<thead>
<tr>
<th>TX_SUCCESS</th>
<th>(0x00)</th>
<th>Successful block pool performance get.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_PTR_ERROR</td>
<td>(0x03)</td>
<td>Invalid block pool pointer.</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>(0xFF)</td>
<td>The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
TX_BLOCK_POOL my_pool;
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created block pool. */
status = tx_block_pool_performance_info_get(&my_pool, &allocates,
                                           &releases,
                                           &suspensions,
                                           &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

`tx_block_allocate, tx_block_pool_create, tx_block_pool_delete, tx_block_pool_info_get, tx_block_pool_performance_info_get, tx_block_pool_performance_system_info_get, tx_block_release`
tx_block_pool_performance_system_info_get

Get block pool system performance information

Prototype

UINT tx_block_pool_performance_system_info_get(ULONG *allocates,
                        ULONG *releases, ULONG *suspensions, ULONG *timeouts);

Description

This service retrieves performance information about all memory block pools in the application.

The ThreadX library and application must be built with TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.

Input Parameters

allocates Pointer to destination for the total number of allocate requests performed on all block pools.
releases Pointer to destination for the total number of release requests performed on all block pools.
suspensions Pointer to destination for the total number of thread allocation suspensions on all block pools.
timeouts Pointer to destination for the total number of allocate suspension timeouts on all block pools.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.

Return Values

TX_SUCCESS (0x00) Successful block pool system performance get.
TX_FEATURE_NOT_ENABLED (0xFF) The system was not compiled with performance information enabled.
Allowed From

Initialization, threads, timers, and ISRs

Example

```c
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all the block pools in
   the system. */
status = tx_block_pool_performance_system_info_get(&allocates,
                                                  &releases,
                                                  &suspensions,
                                                  &timeouts);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */
```

See Also

tx_block_allocate, tx_block_pool_create, tx_block_pool_delete,
tx_block_pool_info_get, tx_block_pool_performance_info_get,
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

```c
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_pool_performance_info_get,
 tx_block_pool_performance_system_info_get, tx_block_release
tx_block_release

Description of ThreadX Services

---

**tx_block_release**

Release fixed-size block of memory

**Prototype**

```c
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 blocks 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

```c
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_performance_info_get`
- `tx_block_pool_performance_system_info_get`
- `tx_block_pool_prioritize`
tx_byte_allocate

Allocate bytes of memory

Prototype

```c
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
memory byte 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 0xFFFFFFFFE)

  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 (1-0xFFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the memory.

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS (0x00)</td>
<td>Successful memory allocation.</td>
</tr>
<tr>
<td>TX_DELETED (0x01)</td>
<td>Memory pool was deleted while thread was suspended.</td>
</tr>
<tr>
<td>TX_NO_MEMORY (0x10)</td>
<td>Service was unable to allocate the memory within the specified time to wait.</td>
</tr>
<tr>
<td>TX_WAIT_ABORTED (0x1A)</td>
<td>Suspension was aborted by another thread, timer, or ISR.</td>
</tr>
<tr>
<td>TX_POOL_ERROR (0x02)</td>
<td>Invalid memory pool pointer.</td>
</tr>
<tr>
<td>TX_PTR_ERROR (0x03)</td>
<td>Invalid pointer to destination pointer.</td>
</tr>
<tr>
<td>TX_SIZE_ERROR (0X05)</td>
<td>Requested size is zero or larger than the pool.</td>
</tr>
<tr>
<td>TX_WAIT_ERROR (0x04)</td>
<td>A wait option other than TX_NO_WAIT was specified on a call from a non-thread.</td>
</tr>
<tr>
<td>TX_CALLER_ERROR (0x13)</td>
<td>Invalid caller of this service.</td>
</tr>
</tbody>
</table>

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_performance_info_get,
rx_byte_pool_performance_system_info_get, tx_byte_pool_prioritize,
tx_byte_release
tx_byte_pool_create

Create 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 byte 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

```c
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_performance_info_get,
tx_byte_pool_performance_system_info_get, tx_byte_pool_prioritize,
tx_byte_release
tx_byte_pool_delete

Delete memory byte pool

Prototype

UINT tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr)

Description

This service deletes the specified memory byte 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

```c
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_performance_info_get`,
- `tx_byte_pool_performance_system_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.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Memory Bytes

Return Values

TX_SUCCESS (0x00) Successful pool information retrieve.
TX_POOL_ERROR (0x02) Invalid memory pool 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 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_performance_info_get,
tx_byte_pool_performance_system_info_get, tx_byte_pool_prioritize,
tx_byte_release
tx_byte_pool_performance_info_get

Get byte pool performance information

Prototype

UINT tx_byte_pool_performance_info_get(TX_BYTE_POOL *pool_ptr,
ULONG *allocates, ULONG *releases,
ULONG *fragments_searched, ULONG *merges, ULONG *splits,
ULONG *suspensions, ULONG *timeouts);

Description

This service retrieves performance information about the specified memory byte pool.

The ThreadX library and application must be built with TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.

Input Parameters

- pool_ptr: Pointer to previously created memory byte pool.
- allocates: Pointer to destination for the number of allocate requests performed on this pool.
- releases: Pointer to destination for the number of release requests performed on this pool.
- fragments_searched: Pointer to destination for the number of internal memory fragments searched during allocation requests on this pool.
- merges: Pointer to destination for the number of internal memory blocks merged during allocation requests on this pool.
- splits: Pointer to destination for the number of internal memory blocks split (fragments) created during allocation requests on this pool.
- suspensions: Pointer to destination for the number of thread allocation suspensions on this pool.
- timeouts: Pointer to destination for the number of allocate suspension timeouts on this pool.
Supplying a TX_NULL for any parameter indicates the parameter is not required.

**Return Values**

- **TX_SUCCESS** (0x00) Successful byte pool performance get.
- **TX_PTR_ERROR** (0x03) Invalid byte pool pointer.
- **TX_FEATURE_NOT_ENABLED** (0xFF) The system was not compiled with performance information enabled.

**Allowed From**

Initialization, threads, timers, and ISRs

**Example**

```c
TX_BYTE_POOL my_pool;
ULONG fragments_searched;
ULONG merges;
ULONG splits;
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created byte pool. */
status = tx_byte_pool_performance_info_get(&my_pool,
                                          &fragments_searched,
                                          &merges, &splits,
                                          &allocates, &releases,
                                          &suspensions,&timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

**See Also**

- tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete,
- tx_byte_pool_info_get, tx_byte_pool_performance_system_info_get,
- tx_byte_pool_prioritize, tx_byte_release
tx_byte_pool_performance_system_info_get

Get byte pool system performance information

Prototype

UINT tx_byte_pool_performance_system_info_get(ULONG *allocates,
     ULONG *releases, ULONG *fragments_searched, ULONG *merges,
     ULONG *splits, ULONG *suspensions, ULONG *timeouts);

Description

This service retrieves performance information about all memory byte pools in the system.

The ThreadX library and application must be built with TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.

Input Parameters

allocates  Pointer to destination for the number of allocate requests performed on this pool.

releases   Pointer to destination for the number of release requests performed on this pool.

fragments_searched  Pointer to destination for the total number of internal memory fragments searched during allocation requests on all byte pools.

merges     Pointer to destination for the total number of internal memory blocks merged during allocation requests on all byte pools.

splits     Pointer to destination for the total number of internal memory blocks split (fragments) created during allocation requests on all byte pools.

suspensions  Pointer to destination for the total number of thread allocation suspensions on all byte pools.

timeouts   Pointer to destination for the total number of allocate suspension timeouts on all byte pools.
Supplying a TX_NULL for any parameter indicates the parameter is not required.

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>(0x00) Successful byte pool performance get.</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>(0xFF) The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
ULONG fragments_searched;
ULONG merges;
ULONG splits;
ULONG allocates;
ULONG releases;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all byte pools in the system. */
status = tx_byte_pool_performance_system_info_get(&fragments_searched,
                                             &merges, &splits, &allocates, &releases,
                                             &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

- `tx_byte_allocate`, `tx_byte_pool_create`, `tx_byte_pool_delete`,
- `tx_byte_pool_info_get`, `tx_byte_pool_performance_info_get`,
- `tx_byte_pool_prioritize`, `tx_byte_release`
tx_byte_pool_prioritize

Prioritize 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

```c
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_pool_performance_info_get,
tx_byte_pool_performance_system_info_get, tx_byte_release
tx_byte_release

---

Release bytes back to memory pool

**Prototype**

```c
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

```c
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_performance_info_get`
- `tx_byte_pool_performance_system_info_get`
- `tx_byte_pool_prioritize`
tx_event_flags_create

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

/* If status equals TX_SUCCESS, my_event_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_performance_info_get,
          tx_event_flags_performance_system_info_get, tx_event_flags_set,
          tx_event_flags_set_notify
tx_event_flags_delete

Delete event flags group

Prototype

UINT tx_event_flags_delete(TX_EVENT_FLAGS_GROUP *group_ptr)

Description

This service deletes the specified event flags 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 flags group.

Input Parameters

- **group_ptr**: Pointer to a previously created event flags group.

Return Values

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

Allowed From

Threads

Preemption Possible

Yes
Event Flags

Example

```c
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT status;

/* Delete event flags 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_performance_info_get`
- `tx_event_flags_performance_system_info_get`
- `tx_event_flags_set`
- `tx_event_flags_set_notify`
**tx_event_flags_get**

Get event flags from event flags group

**Prototype**

```c
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 flags group. Each event flags 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 flags group.

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

- **get_option**  
  Specifies whether all or any of the requested event flags are 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:

<table>
<thead>
<tr>
<th>Option</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_NO_WAIT</td>
<td>0x00000000</td>
</tr>
<tr>
<td>TX_WAIT_FOREVER</td>
<td>0xFFFFFFFF</td>
</tr>
<tr>
<td>timeout value</td>
<td>(0x00000001 through 0xFFFFFFF)</td>
</tr>
</tbody>
</table>

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 (1-0xFFFFFFF) specifies the maximum number of timer-ticks to stay suspended while waiting for the event flags.

**Return Values**

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>TX_SUCCESS Successful event flags get.</td>
</tr>
<tr>
<td>0x01</td>
<td>TX_DELETED Event flags group was deleted while thread was suspended.</td>
</tr>
<tr>
<td>0x07</td>
<td>TX_NO_EVENTS Service was unable to get the specified events within the specified time to wait.</td>
</tr>
<tr>
<td>0x1A</td>
<td>TX_WAIT_ABORTED Suspension was aborted by another thread, timer, or ISR.</td>
</tr>
<tr>
<td>0x06</td>
<td>TX_GROUP_ERROR Invalid event flags group pointer.</td>
</tr>
<tr>
<td>0x03</td>
<td>TX_PTR_ERROR Invalid pointer for actual event flags.</td>
</tr>
<tr>
<td>0x04</td>
<td>TX_WAIT_ERROR A wait option other than TX_NO_WAIT was specified on a call from a non-thread.</td>
</tr>
<tr>
<td>0x08</td>
<td>TX_OPTION_ERROR Invalid get-option was specified.</td>
</tr>
</tbody>
</table>
Allowed From
Initialization, threads, timers, and ISRs

Preemption Possible
Yes

Example

```c
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_performance_info_get,
tx_event_flags_performance_system_info_get, tx_event_flags_set,
tx_event_flags_set_notify
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 flags group's name.
- current_flags: Pointer to destination for the current set flags in the event flags group.
- first_suspended: Pointer to destination for the pointer to the thread that is first on the suspension list of this event flags group.
- suspended_count: Pointer to destination for the number of threads currently suspended on this event flags group.
- next_group: Pointer to destination for the pointer of the next created event flags group.

Supposing a TX_NULL for any parameter indicates that the parameter is not required.
Return Values

**TX_SUCCESS** (0x00) Successful event group information retrieval.

**TX_GROUP_ERROR** (0x06) Invalid event group pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```c
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 the previously created event flags 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_performance_info_get,
- tx_event_flags_performance_system_info_get, tx_event_flags_set,
- tx_event_flags_set_notify
**tx_event_flags_performance_info_get**

Get event flags group performance information

**Prototype**

```c
UINT tx_event_flags_performance_info_get(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG *sets, ULONG *gets, ULONG *suspensions, ULONG *timeouts);
```

**Description**

This service retrieves performance information about the specified event flags group.

ThreadX library and application must be built with `TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO` defined for this service to return performance information.

**Input Parameters**

- `group_ptr` Pointer to previously created event flags group.
- `sets` Pointer to destination for the number of event flags set requests performed on this group.
- `gets` Pointer to destination for the number of event flags get requests performed on this group.
- `suspensions` Pointer to destination for the number of thread event flags get suspensions on this group.
- `timeouts` Pointer to destination for the number of event flags get suspension timeouts on this group.

Supplying a `TX_NULL` for any parameter indicates that the parameter is not required.
Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>0x00</td>
<td>Successful event flags group performance get.</td>
</tr>
<tr>
<td>TX_PTR_ERROR</td>
<td>0x03</td>
<td>Invalid event flags group pointer.</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>0xFF</td>
<td>The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
TX_EVENT_FLAGS_GROUP my_event_flag_group;
ULONG sets;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created event flag group. */
status = tx_event_flags_performance_info_get(&my_event_flag_group, &sets, &gets, &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

- tx_event_flags_create
- tx_event_flags_delete
- tx_event_flags_get
- tx_event_flags_info_get
- tx_event_flags_performance_system_info_get
- tx_event_flags_set
- tx_event_flags_set_notify
tx_event_flags_performance_system_info_get

Retrieve performance system information

Prototype

UINT tx_event_flags_performance_system_info_get(ULONG *sets,
ULONG *gets, ULONG *suspensions, ULONG *timeouts);

Description

This service retrieves performance information about all event flags
groups in the system.

ThreadX library and application must be built with
TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO defined for this
service to return performance information.

Input Parameters

sets
Pointer to destination for the total number of
event flags set requests performed on all groups.

gets
Pointer to destination for the total number of
event flags get requests performed on all groups.

suspensions
Pointer to destination for the total number of
thread event flags get suspensions on all groups.

timeouts
Pointer to destination for the total number of
event flags get suspension timeouts on all
groups.

Supplying a TX_NULL for any parameter indicates that the parameter is
not required.

Return Values

TX_SUCCESS (0x00) Successful event flags
system performance get.

TX_FEATURE_NOT_ENABLED (0xFF) The system was not
compiled with performance
information enabled.
Event Flags

Allowed From

Initialization, threads, timers, and ISRs

Example

ULONG sets;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all previously created event flag groups. */
status = tx_event_flags_performance_system_info_get(&sets, &gets,
& suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */

See Also

tx_event_flags_create, tx_event_flags_delete, tx_event_flags_get,
tx_event_flags_info_get, tx_event_flags_performance_info_get,
tx_event_flags_set, tx_event_flags_set_notify
tx_event_flags_set

Set event flags in an event flags group

Prototype

\[
\text{UINT } \text{tx_event_flags_set}(\text{TX_EVENT_FLAGS_GROUP *group_ptr,\
ULONG flags_to_set,UINT set_option})
\]

Description

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

Input Parameters

- **group_ptr**: Pointer to the previously created event flags 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 ANDed 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 ORed with the current event in the group.

Return Values

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

**Allowed From**
Initialization, threads, timers, and ISRs

**Preemption Possible**
Yes

**Example**
```c
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_event_flags_performance_info_get`,
- `tx_event_flags_performance_system_info_get`, `tx_event_flags_set_notify`
tx_event_flags_set_notify

Notify application when event flags are set

Prototype

UINT  tx_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr,
                               VOID (*events_set_notify)(TX_EVENT_FLAGS_GROUP *));

Description

This service registers a notification callback function that is called whenever one or more event flags are set in the specified event flags group. The processing of the notification callback is defined by the application.

Input Parameters

group_ptr  Pointer to previously created event flags group.

events_set_notify  Pointer to application’s event flags set notification function. If this value is TX_NULL, notification is disabled.

Return Values

TX_SUCCESS  (0x00)  Successful registration of event flags set notification.

TX_GROUP_ERROR  (0x06)  Invalid event flags group pointer.

TX_FEATURE_NOT_ENABLED(0xFF)  The system was compiled with notification capabilities disabled.
Event Flags

Allowed From
Initialization, threads, timers, and ISRs

Example

```c
TX_EVENT_FLAGS_GROUP my_group;

/* Register the "my_event_flags_set_notify" function for monitoring event flags set in the event flags group "my_group." */
status = tx_event_flags_set_notify(&my_group,
                                  my_event_flags_set_notify);

/* If status is TX_SUCCESS the event flags set notification function was successfully registered. */

void my_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr)
/* One or more event flags was set in this group! */
```

See Also

`tx_event_flags_create, tx_event_flags_delete, tx_event_flags_get, tx_event_flags_info_get, tx_event_flags_performance_info_get, tx_event_flags_performance_system_info_get, tx_event_flags_set`
tx_interrupt_control

Enable and disable 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_threadx.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.
Interrupt Control

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 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_performance_info_get,
tx_mutex_performance_system_info_get, tx_mutex_prioritize,
tx_mutex_put
```

 Mutex
tx_mutex_delete

Delete 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_performance_info_get,
  tx_mutex_performance_system_info_get, tx_mutex_prioritize,
  tx_mutex_put
tx_mutex_get

Obtain ownership of 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.

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 (1-0xFFFFFFFFE) 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 within the specified time to wait.

**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 and threads and timers

Preemption Possible

Yes

Example

```c
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_performance_info_get, tx_mutex_performance_system_info_get, tx_mutex_prioritize, tx_mutex_put
tx_mutex_info_get

Retrieve information about 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.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.

Return Values

TX_SUCCESS (0x00) Successful mutex information retrieval.
TX_MUTEX_ERROR (0x1C) Invalid mutex pointer.
Mutex

Allowed From
Initialization, threads, timers, and ISRs

Preemption Possible
No

Example

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

```c
/* Retrieve information about 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
tax_mutex_create, tx_mutex_delete, tx_mutex_get,
tax_mutex_performance_info_get,
tax_mutex_performance_system_info_get, tx_mutex_prioritize,
tax_mutex_put
**tx_mutex_performance_info_get**

Get mutex performance information

**Prototype**

```
UINT tx_mutex_performance_info_get(TX_MUTEX *mutex_ptr, ULONG *puts, 
                                  ULONG *gets, ULONG *suspensions, ULONG *timeouts, 
                                  ULONG *inversions, ULONG *inheritances);
```

**Description**

This service retrieves performance information about the specified mutex.

The ThreadX library and application must be built with `TX_MUTEX_ENABLE_PERFORMANCE_INFO` defined for this service to return performance information.

**Input Parameters**

- **mutex_ptr**
  
  Pointer to previously created mutex.

- **puts**
  
  Pointer to destination for the number of put requests performed on this mutex.

- **gets**
  
  Pointer to destination for the number of get requests performed on this mutex.

- **suspensions**
  
  Pointer to destination for the number of thread mutex get suspensions on this mutex.

- **timeouts**
  
  Pointer to destination for the number of mutex get suspension timeouts on this mutex.

- **inversions**
  
  Pointer to destination for the number of thread priority inversions on this mutex.

- **inheritances**
  
  Pointer to destination for the number of thread priority inheritance operations on this mutex.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>0x00</td>
<td>Successful mutex performance get.</td>
</tr>
<tr>
<td>TX_PTR_ERROR</td>
<td>0x03</td>
<td>Invalid mutex pointer.</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>0xFF</td>
<td>The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
TX_MUTEX my_mutex;
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;
ULONG inversions;
ULONG inheritances;

/* Retrieve performance information on the previously created mutex. */
status = tx_mutex_performance_info_get(&my_mutex_ptr, &puts, &gets, &suspensions, &timeouts, &inversions, &inheritances);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_info_get,
tx_mutex_performance_system_info_get, tx_mutex_prioritize,
tx_mutex_put
tx_mutex_performance_system_info_get

Get mutex system performance information

Prototype

UINT  tx_mutex_performance_system_info_get(ULONG *puts, ULONG *gets,
ULONG *suspensions, ULONG *timeouts,
ULONG *inversions, ULONG *inheritances);

Description

This service retrieves performance information about all the mutexes in the system.

The ThreadX library and application must be built with TX_MUTEX_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.

Input Parameters

puts  Pointer to destination for the total number of put requests performed on all mutexes.

gets  Pointer to destination for the total number of get requests performed on all mutexes.

suspensions  Pointer to destination for the total number of thread mutex get suspensions on all mutexes.

timeouts  Pointer to destination for the total number of mutex get suspension timeouts on all mutexes.

inversions  Pointer to destination for the total number of thread priority inversions on all mutexes.

inheritances  Pointer to destination for the total number of thread priority inheritance operations on all mutexes.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Mutex

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS (0x00)</td>
<td>Successful mutex system performance get.</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED (0xFF)</td>
<td>The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;
ULONG inversions;
ULONG inheritances;

/* Retrieve performance information on all previously created mutexes. */
status = tx_mutex_performance_system_info_get(&puts, &gets, &suspensions, &timeouts, &inversions, &inheritances);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_info_get, tx_mutex_performance_info_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

```c
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_performance_info_get`,
- `tx_mutex_performance_system_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

```c
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_performance_info_get`,
- `tx_mutex_performance_system_info_get`, `tx_mutex_prioritize`
tx_queue_create

Create 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 numerical values from 1 through 16, inclusive.
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", 4, (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_performance_info_get,
tx_queue_performance_system_info_get, tx_queue_prioritize,
tx_queue_receive, tx_queue_send, tx_queue_send_notify
tx_queue_delete

Delete 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

```c
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_performance_info_get`, `tx_queue_performance_system_info_get`, `tx_queue_prioritize`, `tx_queue_receive`, `tx_queue_send`, `tx_queue_send_notify`
tx_queue_flush

Empty messages in 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

<table>
<thead>
<tr>
<th>Status</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>0x00</td>
<td>Successful message queue flush.</td>
</tr>
<tr>
<td>TX_QUEUE_ERROR</td>
<td>0x09</td>
<td>Invalid message queue pointer.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes
Example

```c
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_performance_info_get`
- `tx_queue_performance_system_info_get`
- `tx_queue_prioritize`
- `tx_queue_receive`
- `tx_queue_send`
- `tx_queue_send_notify`
tx_queue_front_send

Send 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 0xFFFFFFFF)

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 (1-0xFFFFFFFF) 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 for the duration of the specified time to wait.
- **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

```c
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_performance_info_get`,
- `tx_queue_performance_system_info_get`, `tx_queue_pprioritize`,
- `tx_queue_receive`, `tx_queue_send`, `tx_queue_send_notify`
tx_queue_info_get

Retrieve information about 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.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.

Return Values

- **TX_SUCCESS** (0x00): Successful queue information get.
- **TX_QUEUE_ERROR** (0x09): Invalid message queue pointer.
Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

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

/* Retrieve information about the previously created message queue "my_queue." */
status = tx_queue_info_get(&my_queue, &name,
&enqueued, &available_storage,
&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_performance_info_get,
tx_queue_performance_system_info_get, tx_queue_prioritize,
tx_queue_receive, tx_queue_send, tx_queue_send_notify
tx_queue_performance_info_get

Get queue performance information

Prototype

UINT tx_queue_performance_info_get(TX_QUEUE *queue_ptr,
ULONG *messages_sent, ULONG *messages_received,
ULONG *empty_suspensions, ULONG *full_suspensions,
ULONG *full_errors, ULONG *timeouts);

Description

This service retrieves performance information about the specified queue.

\[\text{The ThreadX library and application must be built with TX_QUEUE_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.}\]

Input Parameters

- **queue_ptr**
  Pointer to previously created queue.
- **messages_sent**
  Pointer to destination for the number of send requests performed on this queue.
- **messages_received**
  Pointer to destination for the number of receive requests performed on this queue.
- **empty_suspensions**
  Pointer to destination for the number of queue empty suspensions on this queue.
- **full_suspensions**
  Pointer to destination for the number of queue full suspensions on this queue.
- **full_errors**
  Pointer to destination for the number of queue full errors on this queue.
- **timeouts**
  Pointer to destination for the number of thread suspension timeouts on this queue.

\[\text{Supplying a TX_NULL for any parameter indicates that the parameter is not required.}\]
Return Values

TX_SUCCESS (0x00) Successful queue performance get.

TX_PTR_ERROR (0x03) Invalid queue pointer.

TX_FEATURE_NOT_ENABLED (0xFF) The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
TX_QUEUE my_queue;
ULONG messages_sent;
ULONG messages_received;
ULONG empty_suspensions;
ULONG full_suspensions;
ULONG full_errors;
ULONG timeouts;

/* Retrieve performance information on the previously created queue. */
status = tx_queue_performance_info_get(&my_queue, &messages_sent,
                           &messages_received, &empty_suspensions,
                           &full_suspensions, &full_errors, &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush,
tx_queue_front_send, tx_queue_info_get,
tx_queue_performance_system_info_get, tx_queue_prioritize,
tx_queue_receive, tx_queue_send, tx_queue_send_notify
tx_queue_performance_system_info_get

Get queue system performance information

Prototype

```c
UINT tx_queue_performance_system_info_get(ULONG *messages_sent,
                                          ULONG *messages_received,
                                          ULONG *empty_suspensions,
                                          ULONG *full_suspensions, ULONG *full_errors,
                                          ULONG *timeouts);
```

Description

This service retrieves performance information about all the queues in the system.

The ThreadX library and application must be built with
TX_QUEUE_ENABLE_PERFORMANCE_INFO defined for this service
to return performance information.

Input Parameters

- **messages_sent**: Pointer to destination for the total number of send requests performed on all queues.
- **messages_received**: Pointer to destination for the total number of receive requests performed on all queues.
- **empty_suspensions**: Pointer to destination for the total number of queue empty suspensions on all queues.
- **full_suspensions**: Pointer to destination for the total number of queue full suspensions on all queues.
- **full_errors**: Pointer to destination for the total number of queue full errors on all queues.
- **timeouts**: Pointer to destination for the total number of thread suspension timeouts on all queues.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Return Values

**TX_SUCCESS** (0x00) Successful queue system performance get.

**TX_FEATURE_NOT_ENABLED** (0xFF) The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
ULONG messages_sent;
ULONG messages_received;
ULONG empty_suspensions;
ULONG full_suspensions;
ULONG full_errors;
ULONG timeouts;

/* Retrieve performance information on all previously created queues. */
status = tx_queue_performance_system_info_get(&messages_sent,
                                             &messages_received,
                                             &empty_suspensions,
                                             &full_suspensions,
                                             &full_errors,
                                             &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

tx_queue_create, tx_queue_delete, tx_queue_flush,
tx_queue_front_send, tx_queue_info_get,
tx_queue_performance_info_get, tx_queue_prioritize, tx_queue_receive,
tx_queue_send, tx_queue_send_notify
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

```c
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_performance_info_get,
tx_queue_performance_system_info_get, tx_queue_receive,
tx_queue_send, tx_queue_send_notify
tx_queue_receive

Get message from message queue

Prototype

```c
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 0xFFFFFFFFE)

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

Return Values

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

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_performance_info_get,
tx_queue_performance_system_info_get, tx_queue_prioritize,
tx_queue_send, tx_queue_send_notify
tx_queue_send

Send message to message queue

Prototype

```c
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 0xFFFFFFFFE)

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 (1-0xFFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.
Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>0x00</td>
<td>Successful sending of message.</td>
</tr>
<tr>
<td>TX_DELETED</td>
<td>0x01</td>
<td>Message queue was deleted while thread was suspended.</td>
</tr>
<tr>
<td>TX_QUEUE_FULL</td>
<td>0x0B</td>
<td>Service was unable to send message because the queue was full for the duration of the specified time to wait.</td>
</tr>
<tr>
<td>TX_WAIT_ABORTED</td>
<td>0x1A</td>
<td>Suspension was aborted by another thread, timer, or ISR.</td>
</tr>
<tr>
<td>TX_QUEUE_ERROR</td>
<td>0x09</td>
<td>Invalid message queue pointer.</td>
</tr>
<tr>
<td>TX_PTR_ERROR</td>
<td>0x03</td>
<td>Invalid source pointer for message.</td>
</tr>
<tr>
<td>TX_WAIT_ERROR</td>
<td>0x04</td>
<td>A wait option other than TX_NO_WAIT was specified on a call from a non-thread.</td>
</tr>
</tbody>
</table>

Allowed From

- Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```c
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_performance_info_get`,
- `tx_queue_performance_system_info_get`, `tx_queue_prioritize`,
- `tx_queue_receive`, `tx_queue_send_notify`
tx_queue_send_notify

Notify application when message is sent to queue

Prototype

```c
UINT tx_queue_send_notify(TX_QUEUE *queue_ptr,
                           VOID (*queue_send_notify)(TX_QUEUE *));
```

Description

This service registers a notification callback function that is called whenever a message is sent to the specified queue. The processing of the notification callback is defined by the application.

Input Parameters

- `queue_ptr`  Pointer to previously created queue.
- `queue_send_notify`  Pointer to application’s queue send notification function. If this value is TX_NULL, notification is disabled.

Return Values

- **TX_SUCCESS**  (0x00) Successful registration of queue send notification.
- **TX_QUEUE_ERROR**  (0x09) Invalid queue pointer.
- **TX_FEATURE_NOT_ENABLED**  (0xFF) The system was compiled with notification capabilities disabled.

Allowed From

Initialization, threads, timers, and ISRs
Example

```c
TX_QUEUE my_queue;

/* Register the "my_queue_send_notify" function for monitoring messages sent to the queue "my_queue." */
status = tx_queue_send_notify(&my_queue, my_queue_send_notify);

/* If status is TX_SUCCESS the queue send notification function was successfully registered. */
void my_queue_send_notify(TX_QUEUE *queue_ptr)
{
    /* A message was just sent to this queue! */
}
```

See Also

tax_queue_create, tx_queue_delete, tx_queue_flush,
tax_queue_front_send, tx_queue_info_get,
tax_queue_performance_info_get,
tax_queue_performance_system_info_get, tx_queue_prioritize,
tax_queue_receive, tx_queue_send
**tx_semaphore_ceiling_put**  
Place an instance in counting semaphore with ceiling

**Prototype**

```c
UINT tx_semaphore_ceiling_put(TX_SEMAPHORE *semaphore_ptr,
                              ULONG ceiling);
```

**Description**

This service puts an instance into the specified counting semaphore, which in reality increments the counting semaphore by one. If the counting semaphore’s current value is greater than or equal to the specified ceiling, the instance will not be put and a TX_CEILING_EXCEEDED error will be returned.

**Input Parameters**

- `semaphore_ptr`: Pointer to previously created semaphore.
- `ceiling`: Maximum limit allowed for the semaphore (valid values range from 1 through 0xFFFFFFFF).

**Return Values**

- **TX_SUCCESS** (0x00): Successful semaphore ceiling put.
- **TX_CEILING_EXCEEDED** (0x21): Put request exceeds ceiling.
- **TX_INVALID_CEILING** (0x22): An invalid value of zero was supplied for ceiling.
- **TX_SEMAPHORE_ERROR** (0x03): Invalid semaphore pointer.

**Allowed From**

Initialization, threads, timers, and ISRs
Counting Semaphores

Example

```c
TX_SEMAPHORE my_semaphore;

/* Increment the counting semaphore "my_semaphore" but make sure that it never exceeds 7 as specified in the call. */
status = tx_semaphore_ceiling_put(&my_semaphore, 7);

/* If status is TX_SUCCESS the semaphore count has been incremented. */
```

See Also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_get,
tx_semaphore_info_get, tx_semaphore_performance_info_get,
tx_semaphore_performance_system_info_get, tx_semaphore_prioritize,
tx_semaphore_put, tx_semaphore_put_notify
tx_semaphore_create

Create 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

```c
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_ceiling_put, tx_semaphore_delete, tx_semaphore_get, 
tx_semaphore_info_get, tx_semaphore_performance_info_get, 
tx_semaphore_performance_system_info_get, tx_semaphore_prioritize, 
tx_semaphore_put, tx_semaphore_put_notify
tx_semaphore_delete

Delete 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_ceiling_put, tx_semaphore_create, tx_semaphore_get,
tx_semaphore_info_get, tx_semaphore_performance_info_get,
tx_semaphore_performance_system_info_get
```

Express Logic, Inc.
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

<table>
<thead>
<tr>
<th>semaphore_ptr</th>
<th>Pointer to a previously created counting semaphore.</th>
</tr>
</thead>
<tbody>
<tr>
<td>wait_option</td>
<td>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:</td>
</tr>
</tbody>
</table>

| TX_NO_WAIT     | 0x00000000 |
| TX_WAIT_FOREVER| 0xFFFFFFFF |
| timeout value  | 0x00000001 through 0xFFFFFFFF |

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 (1-0xFFFFFFFF) specifies the maximum number of timer-ticks to stay suspended while waiting for a semaphore instance.
Return Values

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```c
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. */
```
Description of ThreadX Services

See Also

  tx_semaphore_ceiling_put, tx_semaphore_create, tx_semaphore_delete,
  tx_semaphore_info_get, tx_semaphore_performance_info_get,
  tx_semaphore_prioritize, tx_semaphore_put, tx_semaphore_put_notify
**tx_semaphore_info_get**

Retrieve information about semaphore

**Prototype**

```c
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.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Return Values

TX_SUCCESS (0x00) Successful semaphore information retrieval.

TX_SEMAPHORE_ERROR (0x0C) Invalid semaphore 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 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_ceiling_put, tx_semaphore_create, tx_semaphore_delete,
tx_semaphore_get, tx_semaphore_performance_info_get,
tx_semaphore_performance_system_info_get, tx_semaphore_prioritize,
tx_semaphore_put, tx_semaphore_put_notify
tx_semaphore_performance_info_get

Get semaphore performance information

Prototype

UINT tx_semaphore_performance_info_get(TX_SEMAPHORE *semaphore_ptr,
ULONG *puts, ULONG *gets,
ULONG *suspensions, ULONG *timeouts);

Description

This service retrieves performance information about the specified semaphore.

Note: The ThreadX library and application must be built with
TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.

Input Parameters

- semaphore_ptr: Pointer to previously created semaphore.
- puts: Pointer to destination for the number of put requests performed on this semaphore.
- gets: Pointer to destination for the number of get requests performed on this semaphore.
- suspensions: Pointer to destination for the number of thread suspensions on this semaphore.
- timeouts: Pointer to destination for the number of thread suspension timeouts on this semaphore.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Return Values

- **TX_SUCCESS** (0x00) Successful semaphore performance get.
- **TX_PTR_ERROR** (0x03) Invalid semaphore pointer.
- **TX_FEATURE_NOT_ENABLED** (0xFF) The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
TX_SEMAPHORE my_semaphore;
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on the previously created semaphore. */
status = tx_semaphore_performance_info_get(&my_semaphore, &puts, &gets, &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

- tx_semaphore_ceiling_put, tx_semaphore_create, tx_semaphore_delete,
- tx_semaphore_get, tx_semaphore_info_get,
- tx_semaphore_performance_system_info_get, tx_semaphore_prioritize,
- tx_semaphore_put, tx_semaphore_put_notify
tx_semaphore_performance_system_info_get

Get semaphore system performance information

Prototype

UINT tx_semaphore_performance_system_info_get(ULONG *puts, ULONG *gets, ULONG *suspensions, ULONG *timeouts);

Description

This service retrieves performance information about all the semaphores in the system.

Input Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>puts</td>
<td>Pointer to destination for the total number of put requests performed on all semaphores.</td>
</tr>
<tr>
<td>gets</td>
<td>Pointer to destination for the total number of get requests performed on all semaphores.</td>
</tr>
<tr>
<td>suspensions</td>
<td>Pointer to destination for the total number of thread suspensions on all semaphores.</td>
</tr>
<tr>
<td>timeouts</td>
<td>Pointer to destination for the total number of thread suspension timeouts on all semaphores.</td>
</tr>
</tbody>
</table>

Supplying a TX_NULL for any parameter indicates that the parameter is not required.

The ThreadX library and application must be built with TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO defined for this service to return performance information.
Return Values

- **TX_SUCCESS** (0x00) Successful semaphore system performance get.
- **TX_FEATURE_NOT_ENABLED** (0xFF) The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
ULONG puts;
ULONG gets;
ULONG suspensions;
ULONG timeouts;

/* Retrieve performance information on all previously created semaphores. */
status = tx_semaphore_performance_system_info_get(&puts, &gets,
                                                  &suspensions, &timeouts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

- `tx_semaphore_ceiling_put`
- `tx_semaphore_create`
- `tx_semaphore_delete`
- `tx_semaphore_get`
- `tx_semaphore_info_get`
- `tx_semaphore_performance_info_get`
- `tx_semaphore_prioritize`
- `tx_semaphore_put`
- `tx_semaphore_put_notify`
**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 semaphore 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

```c
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_ceiling_put`
- `tx_semaphore_create`
- `tx_semaphore_delete`
- `tx_semaphore_info_get`
- `tx_semaphore_performance_info_get`
- `tx_semaphore_performance_system_info_get`
- `tx_semaphore_prioritize`
- `tx_semaphore_get`
- `tx_semaphore_put_notify`
tx_semaphore_put_notify

Notify application when semaphore is put

Prototype

```c
UINT tx_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr,
                           VOID (*semaphore_put_notify)(TX_SEMAPHORE *));
```

Description

This service registers a notification callback function that is called whenever the specified semaphore is put. The processing of the notification callback is defined by the application.

Input Parameters

- `semaphore_ptr` Pointer to previously created semaphore.
- `semaphore_put_notify` Pointer to application’s semaphore put notification function. If this value is TX_NULL, notification is disabled.

Return Values

- `TX_SUCCESS` (0x00) Successful registration of semaphore put notification.
- `TX_SEMAPHORE_ERROR` (0x0C) Invalid semaphore pointer.
- `TX_FEATURE_NOT_ENABLED` (0xFF) The system was compiled with notification capabilities disabled.

Allowed From

Initialization, threads, timers, and ISRs
Example

```c
TX_SEMAPHORE my_semaphore;

/* Register the "my_semaphore_put_notify" function for monitoring
the put operations on the semaphore "my_semaphore." */
status = tx_semaphore_put_notify(&my_semaphore,
                               my_semaphore_put_notify);

/* If status is TX_SUCCESS the semaphore put notification function
was successfully registered. */

void my_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr)
{
    /* The semaphore was just put! */
}
```

See Also

tx_semaphore_ceiling_put, tx_semaphore_create, tx_semaphore_delete,
tx_semaphore_get, tx_semaphore_info_get,
tx_semaphore_performance_info_get,
tx_semaphore_performance_system_info_get, tx_semaphore_prioritize,
tx_semaphore_put
tx_thread_create

Create 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 (TX_MAX_PRIORITIES-1), where a value of 0 represents the highest priority.
- **preempt_threshold**: Highest priority level (0 through (TX_MAX_PRIORITIES-1)) of disabled
Thread Control

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-slice values range from 1 to 0xFFFFFFFF (inclusive). 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 through (TX_MAX_PRIORITIES-1)). |
| 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 of threads
   with priorities ranging from 0 through 14. 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 this function returns, the corresponding
       thread is placed into a "completed" state. */
}

See Also

tx_thread_delete, tx_thread_entry_exit_notify, tx_thread_identify,
tx_thread_info_get, tx_thread_performance_info_get,
tx_thread_performance_system_info_get,
tx_thread_preemption_change, tx_thread_priority_change,
tx_thread_relinquish, tx_thread_reset, tx_thread_resume,
tx_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend,
tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort
tx_thread_delete

Delete 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

```c
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_entry_exit_notify, tx_thread_identify,
tax_thread_info_get, tx_thread_performance_info_get,
tax_thread_performance_system_info_get,
tax_thread_preemption_change, tx_thread_priority_change,
tax_thread_relinquish, tx_thread_reset, tx_thread_resume,
tax_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend,
tax_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort
tx_thread_entry_exit_notify

Notify application upon thread entry and exit

Prototype

```
UINT tx_thread_entry_exit_notify(TX_THREAD *thread_ptr,
                                VOID (*entry_exit_notify)(TX_THREAD *, UINT))
```

Description

This service registers a notification callback function that is called whenever the specified thread is entered or exits. The processing of the notification callback is defined by the application.

Input Parameters

- `thread_ptr`: Pointer to previously created thread.
- `entry_exit_notify`: Pointer to application’s thread entry/exit notification function. The second parameter to the entry/exit notification function designates if an entry or exit is present. The value `TX_THREAD_ENTRY (0x00)` indicates the thread was entered, while the value `TX_THREAD_EXIT (0x01)` indicates the thread was exited. If this value is `TX_NULL`, notification is disabled.

Return Values

- `TX_SUCCESS (0x00)`: Successful registration of the thread entry/exit notification function.
- `TX_THREAD_ERROR (0x0E)`: Invalid thread pointer.
- `TX_FEATURE_NOT_ENABLED (0xFF)`: The system was compiled with notification capabilities disabled.

Allowed From

Initialization, threads, timers, and ISRs
Example

TX_THREAD     my_thread;

/* Register the "my_entry_exit_notify" function for monitoring 
   the entry/exit of the thread "my_thread." */
status = tx_thread_entry_exit_notify(&my_thread,
                                     my_entry_exit_notify);

/* If status is TX_SUCCESS the entry/exit notification function was 
   successfully registered. */
void my_entry_exit_notify(TX_THREAD *thread_ptr, UINT condition)
{
    /* Determine if the thread was entered or exited. */
    if (condition == TX_THREAD_ENTRY)
        /* Thread entry! */
    else if (condition == TX_THREAD_EXIT)
        /* Thread exit! */
}

See Also

tx_thread_create, tx_thread_delete, tx_thread_entry_exit_notify,
.tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get,
.tx_thread_performance_system_info_get,
.tx_thread_preemption_change, tx_thread_priority_change,
.tx_thread_relinquish, tx_thread_reset, tx_thread_resume,
.tx_thread_sleep, tx_thread_stack_error_notify, 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

```c
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_entry_exit_notify,
tx_thread_info_get, tx_thread_performance_info_get,
tx_thread_performance_system_info_get,
tx_thread_preemption_change, tx_thread_priority_change,
tx_thread_relinquish, tx_thread_reset, tx_thread_resume,
tx_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend,
tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort
tx_thread_info_get

Retrieve information about 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)
- `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.

Supplying a TX_NULL for any parameter indicates that the parameter is not required.

Return Values

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS (0x00)</td>
<td>Successful thread information retrieval.</td>
</tr>
<tr>
<td>TX_THREAD_ERROR (0x0E)</td>
<td>Invalid thread control pointer.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```c
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 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. */
```

Express Logic, Inc.
Description of ThreadX Services

See Also

tx_thread_create, tx_thread_delete, tx_thread_entry_exit_notify,
tx_thread_identify, tx_thread_performance_info_get,
rx_thread_performance_system_info_get,
rt_thread_preemption_change, tx_thread_priority_change,
rx_thread_relinquish, tx_thread_reset, tx_thread_resume,
rt_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend,
rt_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort
tx_thread_performance_info_get

Get thread performance information

Prototype

```
UINT tx_thread_performance_info_get(TX_THREAD *thread_ptr,
    ULONG *resumptions, ULONG *suspensions,
    ULONG *solicited_preemptions, ULONG *interrupt_preemptions,
    ULONG *priority_inversions, ULONG *time_slices,
    ULONG *relinquishes, ULONG *timeouts, ULONG *wait_aborts,
    TX_THREAD **last_preempted_by);
```

Description

This service retrieves performance information about the specified thread.

The ThreadX library and application must be built with
`TX_THREAD_ENABLE_PERFORMANCE_INFO` defined in order for
this service to return performance information.

Input Parameters

- `thread_ptr` Pointer to previously created thread.
- `resumptions` Pointer to destination for the number of
  resumptions of this thread.
- `suspensions` Pointer to destination for the number of
  suspensions of this thread.
- `solicited_preemptions` Pointer to destination for the number of
  preemptions as a result of a ThreadX API
  service call made by this thread.
- `interrupt_preemptions` Pointer to destination for the number of
  preemptions of this thread as a result of
  interrupt processing.
- `priority_inversions` Pointer to destination for the number of
  priority
  inversions of this thread.
- `time_slices` Pointer to destination for the number of time-
  slices of this thread.
- `relinquishes` Pointer to destination for the number of thread
  relinquishes performed by this thread.
Thread Control

### timeouts
Pointer to destination for the number of suspension timeouts on this thread.

### wait_aborts
Pointer to destination for the number of wait aborts performed on this thread.

### last_preempted_by
Pointer to destination for the thread pointer that last preempted this thread.

> Supplying a TX_NULL for any parameter indicates that the parameter is not required.

#### Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>TX_SUCCESS</strong></td>
<td>0x00</td>
<td>Successful thread performance get.</td>
</tr>
<tr>
<td><strong>TX_PTR_ERROR</strong></td>
<td>0x03</td>
<td>Invalid thread pointer.</td>
</tr>
<tr>
<td><strong>TX_FEATURE_NOT_ENABLED</strong></td>
<td>0xFF</td>
<td>The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

#### Allowed From

- Initialization, threads, timers, and ISRs
Example

TX_THREAD     my_thread;
ULONG         resumptions;
ULONG         suspensions;
ULONG         solicited_preemptions;
ULONG         interrupt_preemptions;
ULONG         priority_inversions;
ULONG         time_slices;
ULONG         relinquishes;
ULONG         timeouts;
ULONG         wait_aborts;
TX_THREAD     *last_preempted_by;

/* Retrieve performance information on the previously created thread. */
status = tx_thread_performance_info_get(&my_thread, &resumptions,
                                        &suspensions,
                                        &solicited_preemptions, &interrupt_preemptions,
                                        &priority_inversions, &time_slices,
                                        &relinquishes, &timeouts,
                                        &wait_aborts, &last_preempted_by);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */

See Also

  tx_thread_create, tx_thread_delete, tx_thread_entry_exit_notify,
  tx_thread_identify, tx_thread_info_get,
  tx_thread_performance_system_info_get,
  tx_thread_preemption_change, tx_thread_priority_change,
  tx_thread_relinquish, tx_thread_reset, tx_thread_resume,
  tx_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend,
  tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort
**Description of ThreadX Services**

**tx_thread_performance_system_info_get**

Get thread system performance information

**Prototype**

```c
UINT tx_thread_performance_system_info_get(ULONG *resumptions,
                                          ULONG *suspensions,
                                          ULONG *solicited_preemptions,
                                          ULONG *interrupt_preemptions,
                                          ULONG *priority_inversions,
                                          ULONG *time_slices,
                                          ULONG *relinquishes,
                                          ULONG *wait_aborts,
                                          ULONG *non_idle_returns,
                                          ULONG *idle_returns);
```

**Description**

This service retrieves performance information about all the threads in the system.

The ThreadX library and application must be built with `TX_THREAD_ENABLE_PERFORMANCE_INFO` defined in order for this service to return performance information.

**Input Parameters**

- **resumptions**: Pointer to destination for the total number of thread resumptions.
- **suspensions**: Pointer to destination for the total number of thread suspensions.
- **solicited_preemptions**: Pointer to destination for the total number of thread preemptions as a result of a thread calling a ThreadX API service.
- **interrupt_preemptions**: Pointer to destination for the total number of thread preemptions as a result of interrupt processing.
- **priority_inversions**: Pointer to destination for the total number of thread priority inversions.
- **time_slices**: Pointer to destination for the total number of thread time-slices.
- **relinquishes**: Pointer to destination for the total number of thread relinquishes.
- **wait_aborts**: Pointer to destination for the total number of non-idle returns.
- **non_idle_returns**: Pointer to destination for the total number of idle returns.
- **idle_returns**: Pointer to destination for the total number of idle returns.
Thread Control

- **timeouts**: Pointer to destination for the total number of thread suspension timeouts.
- **wait_aborts**: Pointer to destination for the total number of thread wait aborts.
- **non_idle_returns**: Pointer to destination for the number of times a thread returns to the system when another thread is ready to execute.
- **idle_returns**: Pointer to destination for the number of times a thread returns to the system when no other thread is ready to execute (idle system).

> Supplying a TX_NULL for any parameter indicates that the parameter is not required.

### Return Values

- **TX_SUCCESS** (0x00): Successful thread system performance get.
- **TX_FEATURE_NOT_ENABLED** (0xFF): The system was not compiled with performance information enabled.

### Allowed From

Initialization, threads, timers, and ISRs
Example

ULONG resumptions;
ULONG suspensions;
ULONG solicited_preemptions;
ULONG interrupt_preemptions;
ULONG priority_inversions;
ULONG time_slices;
ULONG relinquishes;
ULONG timeouts;
ULONG wait_aborts;
ULONG non_idle_returns;
ULONG idle_returns;

/* Retrieve performance information on all previously created
   thread. */
status = tx_thread_performance_system_info_get(&resumptions,
   &suspensions,
   &solicited_preemptions, &interrupt_preemptions,
   &priority_inversions, &time_slices, &relinquishes,
   &timeouts, &wait_aborts, &non_idle_returns,
   &idle_returns);

/* If status is TX_SUCCESS the performance information was
   successfully retrieved. */

See Also

tx_thread_create, tx_thread_delete, tx_thread_entry_exit_notify, 
tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get, 
tx_thread_preemption_change, tx_thread_priority_change,
tx_thread_relinquish, tx_thread_reset, tx_thread_resume, 
tx_thread_sleep, tx_thread_stack_error_notify, 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**

```c
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.

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 through (TX_MAX_PRIORITIES-1)).
- **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 through (TX_MAX_PRIORITIES-1)) 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.
Thread Control

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_entry_exit_notify,
mx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get,
tx_thread_performance_system_info_get, tx_thread_priority_change,
tx_thread_relinquish, tx_thread_reset, tx_thread_resume,
tx_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend,
tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort
**tx_thread_priority_change**

Change priority of application thread

**Prototype**

```c
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 (TX_MAX_PRIORITIES-1), 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 through (TX_MAX_PRIORITIES-1)).
- **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 through (TX_MAX_PRIORITIES-1)).
- **TX_PTR_ERROR** (0x03) Invalid pointer to previous priority storage location.
- **TX_CALLER_ERROR** (0x13) Invalid caller of this service.
### Thread Control

**Allowed From**

Threads and timers

**Preemption Possible**

Yes

**Example**

```c
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_entry_exit_notify, tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get, tx_thread_performance_system_info_get, tx_thread_preemption_change, tx_thread_relinquish, tx_thread_reset, tx_thread_resume, tx_thread_sleep, tx_thread_stack_error_notify, 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**

```c
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**

None

**Return Values**

None

**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_entry_exit_notify,
tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get,
rx_thread_performance_system_info_get,
rx_thread_preemption_change, tx_thread_priority_change,
rx_thread_reset, tx_thread_resume, tx_thread_sleep,
rx_thread_stack_error_notify, tx_thread_suspend, tx_thread_terminate,
rx_thread_time_slice_change, tx_thread_wait_abort
**tx_thread_reset**

*Reset thread*

**Prototype**

```c
UINT tx_thread_reset(TX_THREAD *thread_ptr);
```

**Description**

This service resets the specified thread to execute at the entry point defined at thread creation. The thread must be in either a TX_COMPLETED or TX_TERMINATED state for it to be reset.

*The thread must be resumed for it to execute again.*

**Input Parameters**

- `thread_ptr`: Pointer to a previously created thread.

**Return Values**

- **TX_SUCCESS** (0x00): Successful thread reset.
- **TX_NOT_DONE** (0x20): Specified thread is not in a TX_COMPLETED or TX_TERMINATED state.
- **TX_THREAD_ERROR** (0x0E): Invalid thread pointer.
- **TX_CALLER_ERROR** (0x13): Invalid caller of this service.

**Allowed From**

Threads
Example

```c
TX_THREAD my_thread;

/* Reset the previously created thread "my_thread." */
status = tx_thread_reset(&my_thread);

/* If status is TX_SUCCESS the thread is reset. */
```

See Also

`tx_thread_create, tx_thread_delete, tx_thread_entry_exit_notify, tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get, tx_thread_performance_system_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_resume, tx_thread_sleep, tx_thread_stack_error_notify, tx_thread_suspend, tx_thread_terminate, tx_thread_time_slice_change, tx_thread_wait_abort`
**tx_thread_resume**

Resume suspended application thread

**Prototype**

```c
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

```c
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_entry_exit_notify`
- `tx_thread_identify`, `tx_thread_info_get`, `tx_thread_performance_info_get`
- `tx_thread_preemption_change`, `tx_thread_priority_change`
- `tx_thread_relinquish`, `tx_thread_reset`, `tx_thread_sleep`
- `tx_thread_stack_error_notify`, `tx_thread_suspend`, `tx_thread_terminate`
- `tx_thread_time_slice_change`, `tx_thread_wait_abort`
tx_thread_sleep

Suspend 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 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

```c
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_entry_exit_notify`,
`tx_thread_identify`, `tx_thread_info_get`, `tx_thread_performance_info_get`,
`tx_thread_performance_system_info_get`,
`tx_thread_preemption_change`, `tx_thread_priority_change`,
`tx_thread_relinquish`, `tx_thread_reset`, `tx_thread_resume`,
`tx_thread_stack_error_notify`, `tx_thread_suspend`, `tx_thread_terminate`,
`tx_thread_time_slice_change`, `tx_thread_wait_abort`
tx_thread_stack_error_notify

Register thread stack error notification callback

Prototype

UINT tx_thread_stack_error_notify(VOID (*error_handler)(TX_THREAD *));

Description

This service registers a notification callback function for handling thread stack errors. When ThreadX detects a thread stack error during execution, it will call this notification function to process the error. Processing of the error is completely defined by the application. Anything from suspending the violating thread to resetting the entire system may be done.

\[\text{The ThreadX library must be built with }\]
\[\text{TX_ENABLE_STACK_CHECKING defined in order for this service to return performance information.}\]

Input Parameters

error_handler Pointer to application’s stack error handling function. If this value is TX_NULL, the notification is disabled.

Return Values

| TX_SUCCESS (0x00) | Successful thread reset. |
| TX_FEATURE_NOT_ENABLED (0xFF) | The system was not compiled with performance information enabled. |

Allowed From

Initialization, threads, timers, and ISRs
Example

```c
void my_stack_error_handler(TX_THREAD *thread_ptr);

/* Register the "my_stack_error_handler" function with ThreadX 
so that thread stack errors can be handled by the application. */
status = tx_thread_stack_error_notify(my_stack_error_handler);

/* If status is TX_SUCCESS the stack error handler is registered. */
```

See Also

- `tx_thread_create`
- `tx_thread_delete`
- `tx_thread_entry_exit_notify`
- `tx_thread_identify`
- `tx_thread_info_get`
- `tx_thread_performance_info_get`
- `tx_thread_preformance_system_info_get`
- `tx_thread_preemption_change`
- `tx_thread_priority_change`
- `tx_thread_relinquish`
- `tx_thread_reset`
- `tx_thread_resume`
- `tx_thread_sleep`
- `tx_thread_suspend`
- `tx_thread_terminate`
- `tx_thread_time_slice_change`
- `tx_thread_wait_abort`
Description of ThreadX Services

**tx_thread_suspend**

Suspend application thread

**Prototype**

```c
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.*

After being suspended, the thread must be resumed by `tx_thread_resume` 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**

Initialization, threads, timers, and ISRs

**Preemption Possible**

Yes
Thread Control

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_entry_exit_notify,
tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get,
rx_thread_performance_system_info_get,
tx_thread_preemption_change, tx_thread_priority_change,
tx_thread_relinquish, tx_thread_release, tx_thread_resume,
tx_thread_sleep, tx_thread_stack_error_notify, tx_thread_terminate,
tx_thread_time_slice_change, tx_thread_wait_abort
description of threadx services

\textbf{tx\_thread\_terminate}

\begin{center}
Terminates application thread
\end{center}

\textbf{Prototype}

\begin{verbatim}
UINT tx_thread_terminate(TX_THREAD *thread_ptr)
\end{verbatim}

\textbf{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.

\begin{center}
\textit{After being terminated, the thread must be reset for it to execute again.}
\end{center}

\textbf{Input Parameters}

\begin{itemize}
\item \textbf{thread\_ptr} Pointer to application thread.
\end{itemize}

\textbf{Return Values}

\begin{itemize}
\item \textbf{TX\_SUCCESS} (0x00) Successful thread terminate.
\item \textbf{TX\_THREAD\_ERROR} (0x0E) Invalid application thread pointer.
\item \textbf{TX\_CALLER\_ERROR} (0x13) Invalid caller of this service.
\end{itemize}

\textbf{Allowed From}

Threads and timers

\textbf{Preemption Possible}

Yes
Example

```c
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 reset. */
```

See Also

tx_thread_create, tx_thread_delete, tx_thread_entry_exit_notify,
tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get,
tx_thread_performance_system_info_get,
tx_thread_preemption_change, tx_thread_priority_change,
tx_thread_relinquish, tx_thread_reset, tx_thread_resume,
tx_thread_sleep, tx_thread_stack_error_notify, 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.

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 change.
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**

```c
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_entry_exit_notify, tx_thread_identify, tx_thread_info_get, tx_thread_performance_info_get, tx_thread_performance_system_info_get, tx_thread_preemption_change, tx_thread_priority_change, tx_thread_relinquish, tx_thread_reset, tx_thread_resume, tx_thread_sleep, tx_thread_stack_error_notify, 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.

This service does not release explicit 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

```c
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_entry_exit_notify`
- `tx_thread_identify`
- `tx_thread_info_get`
- `tx_thread_performance_info_get`
- `tx_thread_performance_system_info_get`
- `tx_thread_preemption_change`
- `tx_thread_priority_change`
- `tx_thread_relinquish`
- `tx_thread_reset`
- `tx_thread_resume`
- `tx_thread_sleep`
- `tx_thread_stack_error_notify`
- `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.

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

```c
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. */
\texttt{tx\_time\_set(0x1234);}  
/* Current time now contains 0x1234 until the next timer interrupt. */

See Also

\texttt{tx\_time\_get}
tx_timer_activate

Activate 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

<table>
<thead>
<tr>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS (0x00)</td>
<td>Successful application timer activation.</td>
</tr>
<tr>
<td>TX_TIMER_ERROR (0x15)</td>
<td>Invalid application timer pointer.</td>
</tr>
<tr>
<td>TX_ACTIVATE_ERROR (0x17)</td>
<td>Timer was already active.</td>
</tr>
</tbody>
</table>

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_performance_info_get,
tx_timer_performance_system_info_get

Express Logic, Inc.
tx_timer_change

Change 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

```c
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_performance_info_get, tx_timer_performance_system_info_get`
Description of ThreadX Services

(tx_timer_create)

Create 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

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>0x00</td>
<td>Successful application timer creation.</td>
</tr>
<tr>
<td>TX_TIMER_ERROR</td>
<td>0x15</td>
<td>Invalid application timer pointer. Either the pointer is NULL or the timer is already created.</td>
</tr>
<tr>
<td>TX_TICK_ERROR</td>
<td>0x16</td>
<td>Invalid value (a zero) supplied for initial ticks.</td>
</tr>
<tr>
<td>TX_ACTIVATE_ERROR</td>
<td>0x17</td>
<td>Invalid activation selected.</td>
</tr>
<tr>
<td>TX_CALLER_ERROR</td>
<td>0x13</td>
<td>Invalid caller of this service.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization and threads

Preemption Possible
No

Example

```c
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_performance_info_get, tx_timer_performance_system_info_get`
tx_timer_deactivate

Deactivate 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_performance_info_get,
tx_timer_performance_system_info_get
tx_timer_delete

Delete 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

```c
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_performance_info_get,
  tx_timer_performance_system_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

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>timer_ptr</td>
<td>Pointer to a previously created application timer.</td>
</tr>
<tr>
<td>name</td>
<td>Pointer to destination for the pointer to the timer's name.</td>
</tr>
<tr>
<td>active</td>
<td>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.</td>
</tr>
<tr>
<td>remaining_ticks</td>
<td>Pointer to destination for the number of timer ticks left before the timer expires.</td>
</tr>
<tr>
<td>reschedule_ticks</td>
<td>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.</td>
</tr>
<tr>
<td>next_timer</td>
<td>Pointer to destination for the pointer of the next created application timer.</td>
</tr>
</tbody>
</table>

Note: Supplying a TX_NULL for any parameter indicates that the parameter is not required.
Return Values

TX_SUCCESS (0x00) Successful timer information retrieval.
TX_TIMER_ERROR (0x15) Invalid application timer 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 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, tx_timer_performance_info_get,
tx_timer_performance_system_info_get
tx_timer_performance_info_get

Get timer performance information

Prototype

UINT tx_timer_performance_info_get(TX_TIMER *timer_ptr,
ULONG *activates, ULONG *reactivates,
ULONG *deactivates, ULONG *expiration,
ULONG *expiration_adjusts);

Description

This service retrieves performance information about the specified
application timer.

The ThreadX library and application must be built with
TX_TIMER_ENABLE_PERFORMANCE_INFO defined for this service to
return performance information.

Input Parameters

timer_ptr Pointer to previously created timer.
activates Pointer to destination for the number of activation
requests performed on this timer.
reactivates Pointer to destination for the number of
automatic reactivations performed on this
periodic timer.
deactivates Pointer to destination for the number of
deactivation requests performed on this timer.
expiration Pointer to destination for the number of
expirations of this timer.
expiration_adjusts Pointer to destination for the number of internal
expiration adjustments performed on this timer.
These adjustments are done in the timer
interrupt processing for timers that are larger
than the default timer list size (by default timers
with expirations greater than 32 ticks).
Supplied a TX_NULL for any parameter indicates the parameter is not required.

Return Values

<table>
<thead>
<tr>
<th>Return Value</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SUCCESS</td>
<td>0x00</td>
<td>Successful timer performance get.</td>
</tr>
<tr>
<td>TX_PTR_ERROR</td>
<td>0x03</td>
<td>Invalid timer pointer.</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>0xFF</td>
<td>The system was not compiled with performance information enabled.</td>
</tr>
</tbody>
</table>

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
TX_TIMER my_timer;
ULONG activates;
ULONG reactivates;
ULONG deactivates;
ULONG expirations;
ULONG expiration_adjusts;

/* Retrieve performance information on the previously created timer. */
status = tx_timer_performance_info_get(&my_timer, &activates, &reactivates, &deactivates, &expirations, &expiration_adjusts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

tx_timer_activate, tx_timer_change, tx_timer_create,
tx_timer_deactivate, tx_timer_delete, tx_timer_info_get,
tx_timer_performance_system_info_get
tx_timer_performance_system_info_get

Get timer system performance information

Prototype

UINT tx_timer_performance_system_info_get(ULONG *activates,
                   ULONG *reactivates, ULONG *deactivates,
                   ULONG *expirations, ULONG *expiration_adjusts);

Description

This service retrieves performance information about all the application
	timers in the system.

\[\text{The ThreadX library and application must be built with}
\text{TX_TIMER_ENABLE_PERFORMANCE_INFO defined for this service to}
\text{return performance information.}\]

Input Parameters

activates Pointer to destination for the total number of
activation requests performed on all timers.

reactivates Pointer to destination for the total number of
automatic reactivation performed on all periodic
timers.

deactivates Pointer to destination for the total number of
deactivation requests performed on all timers.

expirations Pointer to destination for the total number of
expirations on all timers.

expiration_adjusts Pointer to destination for the total number of
internal expiration adjustments performed on all
timers. These adjustments are done in the timer
interrupt processing for timers that are larger
than the default timer list size (by default timers
with expirations greater than 32 ticks).

\[\text{Supplying a TX_NULL for any parameter indicates that the parameter}
\text{is not required.}\]
Return Values

**TX_SUCCESS** (0x00) Successful timer system performance get.

**TX_FEATURE_NOT_ENABLED** (0xFF) The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```c
ULONG activates;
ULONG reactivates;
ULONG deactivates;
ULONG expirations;
ULONG expiration_adjusts;

/* Retrieve performance information on all previously created timers. */
status = tx_timer_performance_system_info_get(&activates, &reactivates, &deactivates, &expirations, &expiration_adjusts);

/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

See Also

tx_timer_activate, tx_timer_change, tx_timer_create,
tx_timer_deactivate, tx_timer_delete, tx_timer_info_get,
tx_timer_performance_info_get
Description of ThreadX Services
This chapter contains a description of device drivers for ThreadX. The information presented in this chapter is designed to help developers write application specific drivers. The following lists the device driver topics covered in this chapter:

- Device Driver Introduction 296
- Driver Functions 296
  - Driver Initialization 297
  - Driver Control 297
  - Driver Access 297
  - Driver Input 297
  - Driver Output 298
  - Driver Interrupts 298
  - Driver Status 298
  - Driver Termination 298
- Simple Driver Example 298
  - Simple Driver Initialization 299
  - Simple Driver Input 300
  - Simple Driver Output 301
  - Simple Driver Shortcomings 302
- Advanced Driver Issues 303
  - I/O Buffering 303
  - Circular Byte Buffers 303
  - Circular Buffer Input 303
  - Circular Output Buffer 305
  - Buffer I/O Management 306
  - TX_IO_BUFFER 306
  - Buffered I/O Advantage 307
  - Buffered Driver Responsibilities 307
  - Interrupt Management 309
  - Thread Suspension 309
Device 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 *Device Drivers*.

Device 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 device drivers and write their own specific drivers.

Driver Functions

ThreadX device 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 device 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 device 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 device 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 principal 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 principal 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
- Device 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 being terminated, the driver must not be called again until it is re-initialized.

Simple Driver Example
An example is the best way to describe a device driver. In this example, the driver assumes a simple serial hardware device with a configuration register,
Simple Driver Example

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 300 for the initialization source code of the simple driver.
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.

```c
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.

```c
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 device driver example illustrates the basic idea of a ThreadX device driver. However, because the simple device 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 device drivers.
Advanced Driver Issues

As mentioned previously, device 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 canceling 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.

```c
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.

```c
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_ptr > &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 device 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 64 KBytes 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 device 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 308 shows simple input and
Device Drivers for ThreadX

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.
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 312
- Application Define 312
  - Initial Execution 313
- Thread 0 314
- Thread 1 314
- Thread 2 314
- Threads 3 and 4 315
- Thread 5 315
- Threads 6 and 7 316
- Observing the Demonstration 316
- Distribution file: demo_threadx.c 317
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_threadx.c and is designed to illustrate how ThreadX is used in an embedded multithread environment. The demonstration consists of initialization, eight threads, one byte pool, one block pool, one queue, one semaphore, one mutex, and one event flags group.

Except for the thread’s stack size, the demonstration application is identical on all ThreadX supported processors.

The complete listing of demo_threadx.c, including the line numbers referenced throughout the remainder of this chapter, is displayed on page 318 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
The demonstration system does not create any other additional ThreadX objects. However, an actual application may create system objects during runtime 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 $thread\_2$.

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 87 and 93). 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 \texttt{thread\_0} (through \texttt{event\_flags\_0}), and repeating the sequence. Notice that \texttt{thread\_5} suspends whenever the event flag in \texttt{event\_flags\_0} is not available (line 298).

### Threads 6 and 7

The function \texttt{thread\_6_and\_7\_entry} marks the entry point of both \texttt{thread\_6} and \texttt{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 \texttt{mutex\_0} twice, sleeping for 2 timer ticks, releasing \texttt{mutex\_0} twice, and repeating the sequence. Notice that each thread suspends whenever \texttt{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:

- \texttt{thread\_0\_counter}
- \texttt{thread\_1\_counter}
- \texttt{thread\_2\_counter}
- \texttt{thread\_3\_counter}
- \texttt{thread\_4\_counter}
- \texttt{thread\_5\_counter}
- \texttt{thread\_6\_counter}
- \texttt{thread\_7\_counter}
Each of these counters should continue to increase as the demonstration executes, with \textit{thread\_1\_counter} and \textit{thread\_2\_counter} increasing at the fastest rate.

\textbf{Distribution file: \texttt{demo\_threadx.c}}

This section displays the complete listing of \texttt{demo\_threadx.c}, including the line numbers referenced throughout this chapter.
Demonstration System for ThreadX

/* 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 flag 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_2_counter;
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. */

/* Define what the initial system looks like. */

void tx_application_define(void *first_unused_memory)
{
/* Create a byte memory pool from which to allocate the thread stacks. */

tx_byte_pool_create("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. */
072 /* Allocate the stack for thread 0. */
073 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
074
075 /* Create the main thread. */
076 tx_thread_create(thread_0, "thread 0", thread_0_entry, 0,
077 pointer, DEMO_STACK_SIZE,
078 1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);
079
080 /* Allocate the stack for thread 1. */
081 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
082
083 /* 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. */
084 tx_thread_create(thread_1, "thread 1", thread_1_entry, 1,
085 pointer, DEMO_STACK_SIZE,
086 16, 16, 4, TX_AUTO_START);
087
088 /* Allocate the stack for thread 2. */
089 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
090 tx_thread_create(thread_2, "thread 2", thread_2_entry, 2,
091 pointer, DEMO_STACK_SIZE,
092 16, 16, 4, TX_AUTO_START);
093
094 /* Allocate the stack for thread 3. */
095 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
096
097 /* 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. */
098 tx_thread_create(thread_3, "thread 3", thread_3_and_4_entry, 3,
099 pointer, DEMO_STACK_SIZE,
100 8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);
101
102 /* Allocate the stack for thread 4. */
103 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
104 tx_thread_create(thread_4, "thread 4", thread_3_and_4_entry, 4,
105 pointer, DEMO_STACK_SIZE,
106 8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);
107
108 /* Allocate the stack for thread 5. */
109 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
110
111 /* Create thread 5. This thread simply pends on an event flag, which will be set by thread 0. */
112 tx_thread_create(thread_5, "thread 5", thread_5_entry, 5,
113 pointer, DEMO_STACK_SIZE,
114 4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);
115
116 /* Allocate the stack for thread 6. */
117 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
118
119 /* Create threads 6 and 7. These threads compete for a ThreadX mutex. */
120 tx_thread_create(thread_6, "thread 6", thread_6_and_7_entry, 6,
121 pointer, DEMO_STACK_SIZE,
122 8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);
123
124 /* Allocate the stack for thread 7. */
125 tx_byte_allocate(byte_pool_0, &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);
126
127 /* Allocate the message queue. */
128 tx_byte_allocate(byte_pool_0, &pointer, DEMO_QUEUE_SIZE*sizeof(ULONG), TX_NO_WAIT);
129
130 /* Create the message queue shared by threads 1 and 2. */
131 tx_queue_create(queue_0, "queue 0", TX_1_ULONG, pointer, DEMO_QUEUE_SIZE*sizeof(ULONG));
132
133 /* 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. */

/* Create the mutex used by thread 6 and 7 without priority inheritance. */

/* Allocate the memory for a small block pool. */

/* Create the mutex used by thread 6 and 7 without priority inheritance. */

/* Create a block memory pool to allocate a message buffer from. */

/* Define the test threads. */

/* This thread simply sits in while-forever-sleep loop. */

/* Increment the thread counter. */

/* Sleep for 10 ticks. */

/* Set event flag 0 to wake up thread 5. */

/* Check status. */

/* This thread simply sends messages to a queue shared by thread 2. */

/* Increment the thread counter. */

/* Send message to queue 0. */

/* Check completion status. */

/* Increment the message 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 3 ticks to hold the semaphore. */
        tx_thread_sleep(3);

        /* 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;

    /* Increment the thread counter. */
    if (thread_input == 5)
        thread_5_counter++;

    /* Get the semaphore with suspension. */
    status = tx_semaphore_get(&semaphore_0, TX_WAIT_FOREVER);

    /* Check status. */
    if (status != TX_SUCCESS)
        break;

    /* Sleep for 3 ticks to hold the semaphore. */
    tx_thread_sleep(3);

    /* Release the semaphore. */
    status = tx_semaphore_put(&semaphore_0);

    /* Check status. */
    if (status != TX_SUCCESS)
        break;
}
/* 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;
    }
}
ThreadX API Services

- Entry Function 324
- Block Memory Services 324
- Byte Memory Services 324
- Event Flags Services 325
- Interrupt Control 325
- Mutex Services 325
- Queue Services 326
- Semaphore Services 326
- Thread Control Services 327
- Time Services 328
- Timer Services 328
## ThreadX API Services

### Entry Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>VOID tx_kernel_enter(VOID);</td>
<td></td>
</tr>
</tbody>
</table>

### Block Memory Services

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT tx_block_allocate(TX_BLOCK_POOL *pool_ptr, VOID **block_ptr, ULONG wait_option);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_block_pool_create(TX_BLOCK_POOL *pool_ptr, CHAR *name_ptr, ULONG block_size, VOID *pool_start, ULONG pool_size);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_block_pool_delete(TX_BLOCK_POOL *pool_ptr);</td>
<td></td>
</tr>
<tr>
<td>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);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_block_pool_performance_info_get(TX_BLOCK_POOL *pool_ptr, ULONG *allocates, ULONG *releases, ULONG *suspensions, ULONG *timeouts);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_block_pool_performance_system_info_get(ULONG *allocates, ULONG *releases, ULONG *suspensions, ULONG *timeouts);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_block_pool_prioritize(TX_BLOCK_POOL *pool_ptr);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_block_release(VOID *block_ptr);</td>
<td></td>
</tr>
</tbody>
</table>

### Byte Memory Services

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>UINT tx_byte_allocate(TX_BYTE_POOL *pool_ptr, VOID **memory_ptr, ULONG memory_size, ULONG wait_option);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_byte_pool_create(TX_BYTE_POOL *pool_ptr, CHAR *name_ptr, VOID *pool_start, ULONG pool_size);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_byte_pool_delete(TX_BYTE_POOL *pool_ptr);</td>
<td></td>
</tr>
<tr>
<td>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);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_byte_pool_performance_info_get(TX_BYTE_POOL *pool_ptr, ULONG *allocates, ULONG *releases, ULONG *fragments_searched, ULONG *merges, ULONG *splits, ULONG *suspensions, ULONG *timeouts);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_byte_pool_performance_system_info_get(ULONG *allocates, ULONG *releases, ULONG *fragments_searched, ULONG *merges, ULONG *splits, ULONG *suspensions, ULONG *timeouts);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_byte_pool_prioritize(TX_BYTE_POOL *pool_ptr);</td>
<td></td>
</tr>
<tr>
<td>UINT tx_byte_release(VOID *memory_ptr);</td>
<td></td>
</tr>
</tbody>
</table>
**ThreadX API Services** 325

**Event Flags 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_performance_info_get(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG *sets, ULONG *gets, ULONG *suspensions, ULONG *timeouts);`
- `UINT tx_event_flags_performance_system_info_get(ULONG *sets, ULONG *gets, ULONG *suspensions, ULONG *timeouts);`
- `UINT tx_event_flags_set(TX_EVENT_FLAGS_GROUP *group_ptr, ULONG flags_to_set, UINT set_option);`
- `UINT tx_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr, VOID (*events_set_notify)(TX_EVENT_FLAGS_GROUP *));`

**Interrupt Control**

- `UINT tx_interrupt_control(UINT new_posture);`

**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_performance_info_get(TX_MUTEX *mutex_ptr, ULONG *puts, ULONG *gets, ULONG *suspensions, ULONG *timeouts, ULONG *inversions, ULONG *inheritances);`
- `UINT tx_mutex_performance_system_info_get(ULONG *puts, ULONG *gets, ULONG *suspensions, ULONG *timeouts, ULONG *inversions, ULONG *inheritances);`
- `UINT tx_mutex_prioritize(TX_MUTEX *mutex_ptr);`
- `UINT tx_mutex_put(TX_MUTEX *mutex_ptr);`

Express Logic, Inc.
### ThreadX API Services

**Queue Services**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tx_queue_create(TX_QUEUE *queue_ptr, CHAR *name_ptr, UINT message_size, VOID *queue_start, ULONG queue_size);</code></td>
<td>Create a queue</td>
</tr>
<tr>
<td><code>tx_queue_delete(TX_QUEUE *queue_ptr);</code></td>
<td>Delete a queue</td>
</tr>
<tr>
<td><code>tx_queue_flush(TX_QUEUE *queue_ptr);</code></td>
<td>Flush a queue</td>
</tr>
<tr>
<td><code>tx_queue_front_send(TX_QUEUE *queue_ptr, VOID *source_ptr, ULONG wait_option);</code></td>
<td>Send a message to the front of a queue</td>
</tr>
<tr>
<td><code>tx_queue_info_get(TX_QUEUE *queue_ptr, CHAR **name, ULONG *enqueued, TX_THREAD **first_suspended, ULONG *suspended_count, TX_QUEUE **next_queue);</code></td>
<td>Get queue information</td>
</tr>
<tr>
<td><code>tx_queue_performance_info_get(TX_QUEUE *queue_ptr, ULONG *messages_sent, ULONG *messages_received, ULONG *empty_suspensions, ULONG *full_suspensions, ULONG *full_errors, ULONG *timeouts);</code></td>
<td>Get performance information</td>
</tr>
<tr>
<td><code>tx_queue_performance_system_info_get(ULONG *messages_sent, ULONG *messages_received, ULONG *empty_suspensions, ULONG *full_suspensions, ULONG *full_errors, ULONG *timeouts);</code></td>
<td>Get system performance information</td>
</tr>
<tr>
<td><code>tx_queue_prioritize(TX_QUEUE *queue_ptr);</code></td>
<td>Prioritize a queue</td>
</tr>
<tr>
<td><code>tx_queue_receive(TX_QUEUE *queue_ptr, VOID *destination_ptr, ULONG wait_option);</code></td>
<td>Receive a message from a queue</td>
</tr>
<tr>
<td><code>tx_queue_send(TX_QUEUE *queue_ptr, VOID *source_ptr, ULONG wait_option);</code></td>
<td>Send a message to a queue</td>
</tr>
<tr>
<td><code>tx_queue_send_notify(TX_QUEUE *queue_ptr, VOID (*queue_send_notify)(TX_QUEUE *));</code></td>
<td>Send a notification when a message is sent</td>
</tr>
</tbody>
</table>

**Semaphore Services**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tx_semaphore_ceiling_put(TX_SEMAPHORE *semaphore_ptr, ULONG ceiling);</code></td>
<td>Put a value into a semaphore</td>
</tr>
<tr>
<td><code>tx_semaphore_create(TX_SEMAPHORE *semaphore_ptr, CHAR *name_ptr, ULONG initial_count);</code></td>
<td>Create a semaphore</td>
</tr>
<tr>
<td><code>tx_semaphore_delete(TX_SEMAPHORE *semaphore_ptr);</code></td>
<td>Delete a semaphore</td>
</tr>
<tr>
<td><code>tx_semaphore_get(TX_SEMAPHORE *semaphore_ptr, ULONG wait_option);</code></td>
<td>Get a value from a semaphore</td>
</tr>
<tr>
<td><code>tx_semaphore_info_get(TX_SEMAPHORE *semaphore_ptr, CHAR **name, ULONG *current_value, TX_THREAD **first_suspended, ULONG *suspended_count, TX_SEMAPHORE **next_semaphore);</code></td>
<td>Get semaphore information</td>
</tr>
<tr>
<td><code>tx_semaphore_performance_info_get(TX_SEMAPHORE *semaphore_ptr, ULONG *puts, ULONG *gets, ULONG *suspensions, ULONG *timeouts);</code></td>
<td>Get performance information</td>
</tr>
<tr>
<td><code>tx_semaphore_performance_system_info_get(ULONG *puts, ULONG *gets, ULONG *suspensions, ULONG *timeouts);</code></td>
<td>Get system performance information</td>
</tr>
<tr>
<td><code>tx_semaphore_prioritize(TX_SEMAPHORE *semaphore_ptr);</code></td>
<td>Prioritize a semaphore</td>
</tr>
</tbody>
</table>
ThreadX API Services

ThreadX Control Services

UINT  tx_semaphore_put(TX_SEMAPHORE *semaphore_ptr);
UINT  tx_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr,
                             VOID (*semaphore_put_notify)(TX_SEMAPHORE *));

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_entry_exit_notify(TX_THREAD *thread_ptr,
                                 VOID (*thread_entry_exit_notify)(TX_THREAD *, UINT));
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_performance_info_get(TX_THREAD *thread_ptr,
                                     ULONG *resumptions, ULONG *suspensions,
                                     ULONG *interrupt_preemptions,
                                     ULONG *request_inversions,ULONG *time_slices, ULONG *relinquished,
                                     ULONG *wait_aborts);
UINT  tx_thread_performance_system_info_get(ULONG *resumptions,
                                    ULONG *suspensions, ULONG *interrupt_preemptions,
                                    ULONG *request_inversions,ULONG *time_slices, ULONG *relinquished,
                                    ULONG *wait_aborts, TX_THREAD **last_preempted_by);
UINT  tx_thread_priority_change(TX_THREAD *thread_ptr,
                               UINT new_priority,  UINT *old_priority);
VOID   tx_thread_relinquish(VOID);
UINT   tx_thread_reset(TX_THREAD *thread_ptr);
UINT   tx_thread_resume(TX_THREAD *thread_ptr);
UINT   tx_thread_sleep(UULONG timer_ticks);
UINT   tx_thread_stack_error_notify(VOID (*stack_error_handler)(TX_THREAD *));
UINT   tx_thread_suspend(TX_THREAD *thread_ptr);
UINT   tx_thread_terminate(TX_THREAD *thread_ptr);
ThreadX API Services

**Time Services**

- `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);`

**Timer Services**

- `UINT tx_time_get(VOID);`
- `VOID tx_time_set(ULONG new_time);`

- `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);`
- `UINT tx_timer_performance_info_get(TX_TIMER *timer_ptr, ULONG *activates, ULONG *reactivates, ULONG *deactivates, ULONG *expirations, ULONG *expiration_adjusts);`
- `UINT tx_timer_performance_system_info_get(ULONG *activates, ULONG *reactivates, ULONG *deactivates, ULONG *expirations, ULONG *expiration_adjusts);`
ThreadX Constants

- Alphabetic Listings 330
- Listing by Value 332
Alphabetic Listings

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_1_ULONG</td>
<td>1</td>
</tr>
<tr>
<td>TX_2_ULONG</td>
<td>2</td>
</tr>
<tr>
<td>TX_4_ULONG</td>
<td>4</td>
</tr>
<tr>
<td>TX_8_ULONG</td>
<td>8</td>
</tr>
<tr>
<td>TX_16_ULONG</td>
<td>16</td>
</tr>
<tr>
<td>TX_ACTIVATE_ERROR</td>
<td>0x17</td>
</tr>
<tr>
<td>TX_AND</td>
<td>2</td>
</tr>
<tr>
<td>TX_AND_CLEAR</td>
<td>3</td>
</tr>
<tr>
<td>TX_AUTO_ACTIVATE</td>
<td>1</td>
</tr>
<tr>
<td>TX_AUTO_START</td>
<td>1</td>
</tr>
<tr>
<td>TX_BLOCK_MEMORY</td>
<td>8</td>
</tr>
<tr>
<td>TX_BYTE_MEMORY</td>
<td>9</td>
</tr>
<tr>
<td>TX_CALLER_ERROR</td>
<td>0x13</td>
</tr>
<tr>
<td>TX_CEILING_EXCEEDED</td>
<td>0x21</td>
</tr>
<tr>
<td>TX_COMPLETED</td>
<td>1</td>
</tr>
<tr>
<td>TX_DELETE_ERROR</td>
<td>0x11</td>
</tr>
<tr>
<td>TX_DELETED</td>
<td>0x01</td>
</tr>
<tr>
<td>TX_DONT_START</td>
<td>0</td>
</tr>
<tr>
<td>TX_EVENT_FLAG</td>
<td>7</td>
</tr>
<tr>
<td>TX_FALSE</td>
<td>0</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>0xFF</td>
</tr>
<tr>
<td>TX_FILE</td>
<td>11</td>
</tr>
<tr>
<td>TX_GROUP_ERROR</td>
<td>0x06</td>
</tr>
<tr>
<td>TX_INHERIT</td>
<td>1</td>
</tr>
<tr>
<td>TX_INHERIT_ERROR</td>
<td>0x1F</td>
</tr>
<tr>
<td>TX_INVALID_CEILING</td>
<td>0x22</td>
</tr>
<tr>
<td>TX_IO_DRIVER</td>
<td>10</td>
</tr>
<tr>
<td>TX_LOOP_FOREVER</td>
<td>1</td>
</tr>
<tr>
<td>TX_MUTEX_ERROR</td>
<td>0x1C</td>
</tr>
<tr>
<td>TX_MUTEX_SUSP</td>
<td>13</td>
</tr>
<tr>
<td>TX_NO_ACTIVATE</td>
<td>0</td>
</tr>
<tr>
<td>ThreadX Constants</td>
<td></td>
</tr>
<tr>
<td>-------------------</td>
<td></td>
</tr>
<tr>
<td>TX_NO_EVENTS</td>
<td>0x07</td>
</tr>
<tr>
<td>TX_NO_INHERIT</td>
<td>0</td>
</tr>
<tr>
<td>TX_NO_INSTANCE</td>
<td>0x0D</td>
</tr>
<tr>
<td>TX_NO_MEMORY</td>
<td>0x10</td>
</tr>
<tr>
<td>TX_NO_TIME_SLICE</td>
<td>0</td>
</tr>
<tr>
<td>TX_NO_WAIT</td>
<td>0</td>
</tr>
<tr>
<td>TX_NOT_AVAILABLE</td>
<td>0x1D</td>
</tr>
<tr>
<td>TX_NOT_DONE</td>
<td>0x20</td>
</tr>
<tr>
<td>TX_NOT_OWNED</td>
<td>0x1E</td>
</tr>
<tr>
<td>TX_NULL</td>
<td>0</td>
</tr>
<tr>
<td>TX_OPTION_ERROR</td>
<td>0x08</td>
</tr>
<tr>
<td>TX_OR</td>
<td>0</td>
</tr>
<tr>
<td>TX_OR_CLEAR</td>
<td>1</td>
</tr>
<tr>
<td>TX_POOL_ERROR</td>
<td>0x02</td>
</tr>
<tr>
<td>TX_PRIORITY_ERROR</td>
<td>0x0F</td>
</tr>
<tr>
<td>TX_PTR_ERROR</td>
<td>0x03</td>
</tr>
<tr>
<td>TX_QUEUE_EMPTY</td>
<td>0x0A</td>
</tr>
<tr>
<td>TX_QUEUE_ERROR</td>
<td>0x09</td>
</tr>
<tr>
<td>TX_QUEUE_FULL</td>
<td>0x0B</td>
</tr>
<tr>
<td>TX_QUEUE_SUSP</td>
<td>5</td>
</tr>
<tr>
<td>TX_READY</td>
<td>0</td>
</tr>
<tr>
<td>TX_RESUME_ERROR</td>
<td>0x12</td>
</tr>
<tr>
<td>TX_SEMAPHORE_ERROR</td>
<td>0x0C</td>
</tr>
<tr>
<td>TX_SEMAPHORE_SUSP</td>
<td>6</td>
</tr>
<tr>
<td>TX_SIZE_ERROR</td>
<td>0x05</td>
</tr>
<tr>
<td>TX_SLEEP</td>
<td>4</td>
</tr>
<tr>
<td>TX_STACK_FILL</td>
<td>0xEF</td>
</tr>
</tbody>
</table>
TX_SUSPENDED 3
TX_TCP_IP 12
TX_TERMINATED 2
TX_THREAD_ENTRY 0
TX_THREAD_ERROR 0x0E
TX_THREAD_EXIT 1
TX_THRESH_ERROR 0x18
TX_TICK_ERROR 0x16
TX_TIMER_ERROR 0x15
TX_TRUE 1
TX_WAIT_ABORT_ERROR 0x1B
TX_WAIT_ABORTED 0x1A
TX_WAIT_ERROR 0x04
TX_WAIT_FOREVER 0xFFFFFFFFUL

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 0x00
TX_THREAD_ENTRY 0
TX_1 ULONG 1
TX_AUTO_ACTIVATE 1
TX_AUTO_START 1
TX_COMPLETED 1
TX_INHERIT 1
### ThreadX Constants

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_LOOP_FOREVER</td>
<td>1</td>
</tr>
<tr>
<td>TX_DELETED</td>
<td>0x01</td>
</tr>
<tr>
<td>TX_OR_CLEAR</td>
<td>1</td>
</tr>
<tr>
<td>TX_THREAD_EXIT</td>
<td>1</td>
</tr>
<tr>
<td>TX_TRUE</td>
<td>1</td>
</tr>
<tr>
<td>TX_2_ULONG</td>
<td>2</td>
</tr>
<tr>
<td>TX_AND</td>
<td>2</td>
</tr>
<tr>
<td>TX_POOL_ERROR</td>
<td>0x02</td>
</tr>
<tr>
<td>TX_TERMINATED</td>
<td>2</td>
</tr>
<tr>
<td>TX_AND_CLEAR</td>
<td>3</td>
</tr>
<tr>
<td>TX_PTR_ERROR</td>
<td>0x03</td>
</tr>
<tr>
<td>TX_SUSPENDED</td>
<td>3</td>
</tr>
<tr>
<td>TX_4_ULONG</td>
<td>4</td>
</tr>
<tr>
<td>TX_SLEEP</td>
<td>4</td>
</tr>
<tr>
<td>TX_WAIT_ERROR</td>
<td>0x04</td>
</tr>
<tr>
<td>TX_QUEUE_SUSP</td>
<td>5</td>
</tr>
<tr>
<td>TX_SIZE_ERROR</td>
<td>0x05</td>
</tr>
<tr>
<td>TX_GROUP_ERROR</td>
<td>0x06</td>
</tr>
<tr>
<td>TX_SEMAPHORE_SUSP</td>
<td>6</td>
</tr>
<tr>
<td>TX_EVENT_FLAG</td>
<td>7</td>
</tr>
<tr>
<td>TX_NO_EVENTS</td>
<td>0x07</td>
</tr>
<tr>
<td>TX_8_ULONG</td>
<td>8</td>
</tr>
<tr>
<td>TX_BLOCK_MEMORY</td>
<td>8</td>
</tr>
<tr>
<td>TX_OPTION_ERROR</td>
<td>0x08</td>
</tr>
<tr>
<td>TX_BYTE_MEMORY</td>
<td>9</td>
</tr>
<tr>
<td>TX_QUEUE_ERROR</td>
<td>0x09</td>
</tr>
<tr>
<td>TX_IO_DRIVER</td>
<td>10</td>
</tr>
<tr>
<td>TX_QUEUE_EMPTY</td>
<td>0x0A</td>
</tr>
<tr>
<td>TX_FILE</td>
<td>11</td>
</tr>
<tr>
<td>TX_QUEUE_FULL</td>
<td>0x0B</td>
</tr>
<tr>
<td>TX_TCP_IP</td>
<td>12</td>
</tr>
</tbody>
</table>
ThreadX User Guide

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>TX_SEMAPHORE_ERROR</td>
<td>0x0C</td>
</tr>
<tr>
<td>TX_MUTEX_SUSP</td>
<td>13</td>
</tr>
<tr>
<td>TX_NO_INSTANCE</td>
<td>0x0D</td>
</tr>
<tr>
<td>TX_THREAD_ERROR</td>
<td>0x0E</td>
</tr>
<tr>
<td>TX_PRIORITY_ERROR</td>
<td>0x0F</td>
</tr>
<tr>
<td>TX_16_ULONG</td>
<td>16</td>
</tr>
<tr>
<td>TX_NO_MEMORY</td>
<td>0x10</td>
</tr>
<tr>
<td>TX_START_ERROR</td>
<td>0x10</td>
</tr>
<tr>
<td>TX_DELETE_ERROR</td>
<td>0x11</td>
</tr>
<tr>
<td>TX_RESUME_ERROR</td>
<td>0x12</td>
</tr>
<tr>
<td>TX_CALLER_ERROR</td>
<td>0x13</td>
</tr>
<tr>
<td>TX_SUSPEND_ERROR</td>
<td>0x14</td>
</tr>
<tr>
<td>TX_TIMER_ERROR</td>
<td>0x15</td>
</tr>
<tr>
<td>TX_TICK_ERROR</td>
<td>0x16</td>
</tr>
<tr>
<td>TX_ACTIVATE_ERROR</td>
<td>0x17</td>
</tr>
<tr>
<td>TX_THRESH_ERROR</td>
<td>0x18</td>
</tr>
<tr>
<td>TX_SUSPEND_LIFTED</td>
<td>0x19</td>
</tr>
<tr>
<td>TX_WAIT_ABORTED</td>
<td>0x1A</td>
</tr>
<tr>
<td>TX_WAIT_ABORT_ERROR</td>
<td>0x1B</td>
</tr>
<tr>
<td>TX_MUTEX_ERROR</td>
<td>0x1C</td>
</tr>
<tr>
<td>TX_NOT_AVAILABLE</td>
<td>0x1D</td>
</tr>
<tr>
<td>TX_NOT_OWNED</td>
<td>0x1E</td>
</tr>
<tr>
<td>TX_INHERIT_ERROR</td>
<td>0x1F</td>
</tr>
<tr>
<td>TX_NOT_DONE</td>
<td>0x20</td>
</tr>
<tr>
<td>TX_CEILING_EXCEEDED</td>
<td>0x21</td>
</tr>
<tr>
<td>TX_INVALID_CEILING</td>
<td>0x22</td>
</tr>
<tr>
<td>TX_FEATURE_NOT_ENABLED</td>
<td>0xFF</td>
</tr>
<tr>
<td>TX_STACK_FILL</td>
<td>0xEFEFEFEFEFUL</td>
</tr>
<tr>
<td>TX_WAIT_FOREVER</td>
<td>0xFFFFFFFFUL</td>
</tr>
</tbody>
</table>
ThreadX Data Types

- TX_BLOCK_POOL 336
- TX_BYTE_POOL 336
- TX_EVENT_FLAGS_GROUP 337
- TX_MUTEX 337
- TX_QUEUE 338
- TX_SEMAPHORE 339
- TX_THREAD 339
- TX_TIMER 341
- TX_TIMER_INTERNAL 341
typedef struct TX_BLOCK_POOL_STRUCT
{
    ULONG tx_block_pool_id;
    CHAR *tx_block_pool_name;
    ULONG tx_block_pool_available;
    ULONG tx_block_pool_total;
    UCHAR *tx_block_pool_available_list;
    UCHAR *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;
#endif
    TX_BLOCK_POOL_EXTENSION /* Port defined */
} TX_BLOCK_POOL;

typedef struct TX_BYTE_POOL_STRUCT
{
    ULONG tx_byte_pool_id;
    CHAR *tx_byte_pool_name;
    ULONG tx_byte_pool_available;
    ULONG tx_byte_pool_fragments;
    UCHAR *tx_byte_pool_list;
    UCHAR *tx_byte_pool_search;
    UCHAR *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;
#endif
    TX_BYTE_POOL_EXTENSION /* Port defined */
} TX_BYTE_POOL;
typedef struct TX_EVENT_FLAGS_GROUP_STRUCT
{
    ULONG tx_event_flags_group_id;
    CHAR *tx_event_flags_group_name;
    ULONG tx_event_flags_group_current;
    UINT tx_event_flags_group_reset_search;
    struct TX_THREAD_STRUCT *tx_event_flags_group_suspension_list;
    ULONG tx_event_flags_group_suspended_count;
    struct TX_EVENT_FLAGS_GROUP_STRUCT *tx_event_flags_group_created_next,
    *tx_event_flags_group_created_previous;
    ULONG tx_event_flags_group_delayed_clear;
    #ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO
    ULONG tx_event_flags_group_performance_set_count;
    ULONG tx_event_flags_group_performance_get_count;
    ULONG tx_event_flags_group_performance_suspension_count;
    ULONG tx_event_flags_group_performance_timeout_count;
    #endif
    #ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_event_flags_group_set_notify)(struct TX_EVENT_FLAGS_GROUP_STRUCT *);
    #endif
    #endif
    TX_EVENT_FLAGS_GROUP_EXTENSION /* Port defined */
} TX_EVENT_FLAGS_GROUP;

typedef struct TX_MUTEX_STRUCT
{
    ULONG tx_mutex_id;
    CHAR *tx_mutex_name;
    ULONG tx_mutex_ownership_count;
    TX_MUTEX_EXTENSION /* Port defined */
} TX_MUTEX;
ThreadX Data Types

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;
struct TX_MUTEX_STRUCT
  *tx_mutex_created_next,
  *tx_mutex_created_previous;
ULONG tx_mutex_suspended_count;
struct TX_MUTEX_STRUCT
  *tx_mutex_owned_next,
  *tx_mutex_owned_previous;

#ifdef TX_MUTEX_ENABLE_PERFORMANCE_INFO
  ULONG tx_mutex_performance_put_count;
  ULONG tx_mutex_performance_get_count;
  ULONG tx_mutex_performance_suspension_count;
  ULONG tx_mutex_performance_timeout_count;
  ULONG tx_mutex_performance_priority_inversion_count;
  ULONG tx_mutex_performance_priority_inheritance_count;
#endif

TX_MUTEX_EXTENSION /* Port defined */

TX_MUTEX;

typedef struct TX_QUEUE_STRUCT
{
  ULONG tx_queue_id;
  CHAR *tx_queue_name;
  UINT tx_queue_message_size;
  ULONG tx_queue_capacity;
  ULONG tx_queue_enqueued;
  ULONG tx_queue_available_storage;
  ULONG *tx_queue_start;
  ULONG *tx_queue_end;
  ULONG *tx_queue_read;
  ULONG *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;
#endif TX_QUEUE_ENABLE_PERFORMANCE_INFO
  ULONG tx_queue_performance_messages_sent_count;


---

THREADX User Guide
typedef struct TX_SEMAPHORE_STRUCT
{
    ULONG tx_semaphore_id;
    CHAR *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_THREAD_STRUCT
{
    ULONG tx_thread_id;
    ULONG tx_thread_run_count;
    VOID *tx_thread_stack_ptr;
    VOID *tx_thread_stack_start;
} TX_THREAD;
VOID *tx_thread_stack_end;
ULONG tx_thread_stack_size;
ULONG tx_thread_time_slice;
ULONG tx_thread_new_time_slice;
struct TX_THREAD_STRUCT
  *tx_thread_ready_next,
  *tx_thread_ready_previous;
TX_THREAD_EXTENSION_0 /* Port defined */

CHAR *tx_thread_name;
UINT tx_thread_priority;
UINT tx_thread_state;
UINT tx_thread_delayed_suspend;
UINT tx_thread_suspending;
UINT tx_thread_preempt_threshold;
VOID *tx_thread_stack_highest_ptr;
VOID (*tx_thread_entry)(ULONG);
ULONG tx_thread_entry_parameter;
TX_TIMER_INTERNAL tx_thread_timer;
VOID (*tx_thread_suspend_cleanup)(struct TX_THREAD_STRUCT *);
VOID *tx_thread_suspend_control_block;
struct TX_THREAD_STRUCT
  *tx_thread_suspended_next,
  *tx_thread_suspended_previous;
ULONG tx_thread_suspend_info;
VOID *tx_thread_additional_suspend_info;
UINT tx_thread_suspend_option;
UINT tx_thread_suspend_status;
TX_THREAD_EXTENSION_1 /* Port defined */

struct TX_THREAD_STRUCT
  *tx_thread_created_next,
  *tx_thread_created_previous;

TX_THREAD_EXTENSION_2 /* Port defined */

VOID *tx_thread_filex_ptr;
UINT tx_thread_original_priority;
UINT tx_thread_original_preempt_threshold;
ULONG tx_thread_owned_mutex_count;
struct TX_MUTEX_STRUCT*tx_thread_owned_mutex_list;

#ifdef TX_THREAD_ENABLE_PERFORMANCE_INFO
ULONG tx_thread_performance_resume_count;
ULONG tx_thread_performance_suspend_count;
ULONG tx_thread_performance_solicited_preemption_count;
ULONG tx_thread_performance_interrupt_preemption_count;
#endif
typedef struct TX_THREAD_STRUCT
{
    ULONG tx_thread_performance_priority_inversion_count;
    struct TX_THREAD_STRUCT
        *tx_thread_performance_last_preempting_thread;
    ULONG tx_thread_performance_time_slice_count;
    ULONG tx_thread_performance_relinquish_count;
    ULONG tx_thread_performance_timeout_count;
    ULONG tx_thread_performance_wait_abort_count;
} TX_THREAD;

#define TX_THREAD_EXTENSION_3 /* Port defined */

TX_TIMER
typedef struct TX_TIMER_STRUCT
{
    ULONG tx_timer_id;
    CHAR *tx_timer_name;
    TX_TIMER_INTERNAL tx_timer_internal;
    struct TX_TIMER_STRUCT
        *tx_timer_created_next,
        *tx_timer_created_previous;
} TX_TIMER;

#define TX_TIMER_ENABLE_PERFORMANCE_INFO
    ULONG tx_timer_performance_activate_count;
    ULONG tx_timer_performance_reactivate_count;
    ULONG tx_timer_performance_deactivate_count;
    ULONG tx_timer_performance_expiration_count;
    ULONG tx_timer_performance_expiration_adjust_count;
#endif

TX_TIMER_INTERNAL
typedef struct TX_TIMER_INTERNAL_STRUCT
{
    ULONG tx_timer_internal_remaining_ticks;
    ULONG tx_timer_internal_re_initialize_ticks;
    VOID (*tx_timer_internal_timeout_function)(ULONG);
    ULONG tx_timer_internal_timeout_param;
    struct TX_TIMER_INTERNAL_STRUCT
        *tx_timer_internal_active_next,
        *tx_timer_internal_active_previous;
} TX_TIMER_INTERNAL;
struct TX_TIMER_INTERNAL_STRUCT
    *tx_timer_internal_list_head;
} TX_TIMER_INTERNAL;
ASCII Character Codes

ASCII Character Codes in HEX 344
## ASCII Character Codes in HEX

<table>
<thead>
<tr>
<th>least significant nibble</th>
<th>0_</th>
<th>1_</th>
<th>2_</th>
<th>3_</th>
<th>4_</th>
<th>5_</th>
<th>6_</th>
<th>7_</th>
</tr>
</thead>
<tbody>
<tr>
<td>_0</td>
<td>NUL</td>
<td>DLE</td>
<td>SP</td>
<td>0</td>
<td>@</td>
<td>P</td>
<td>,</td>
<td>p</td>
</tr>
<tr>
<td>_1</td>
<td>SOH</td>
<td>DC1</td>
<td>!</td>
<td>1</td>
<td>A</td>
<td>Q</td>
<td>a</td>
<td>q</td>
</tr>
<tr>
<td>_2</td>
<td>STX</td>
<td>DC2</td>
<td>*</td>
<td>2</td>
<td>B</td>
<td>R</td>
<td>b</td>
<td>r</td>
</tr>
<tr>
<td>_3</td>
<td>ETX</td>
<td>DC3</td>
<td>#</td>
<td>3</td>
<td>C</td>
<td>S</td>
<td>c</td>
<td>s</td>
</tr>
<tr>
<td>_4</td>
<td>EOT</td>
<td>DC4</td>
<td>$</td>
<td>4</td>
<td>D</td>
<td>T</td>
<td>d</td>
<td>t</td>
</tr>
<tr>
<td>_5</td>
<td>ENQ</td>
<td>NAK</td>
<td>%</td>
<td>5</td>
<td>E</td>
<td>U</td>
<td>e</td>
<td>u</td>
</tr>
<tr>
<td>_6</td>
<td>ACK</td>
<td>SYN</td>
<td>&amp;</td>
<td>6</td>
<td>F</td>
<td>V</td>
<td>f</td>
<td>v</td>
</tr>
<tr>
<td>_7</td>
<td>BEL</td>
<td>ETB</td>
<td>'</td>
<td>7</td>
<td>G</td>
<td>W</td>
<td>g</td>
<td>w</td>
</tr>
<tr>
<td>_8</td>
<td>BS</td>
<td>CAN</td>
<td>(</td>
<td>8</td>
<td>H</td>
<td>X</td>
<td>h</td>
<td>x</td>
</tr>
<tr>
<td>_9</td>
<td>HT</td>
<td>EM</td>
<td>)</td>
<td>9</td>
<td>I</td>
<td>Y</td>
<td>i</td>
<td>y</td>
</tr>
<tr>
<td>_A</td>
<td>LF</td>
<td>SUB</td>
<td>*</td>
<td>:</td>
<td>J</td>
<td>Z</td>
<td>j</td>
<td>z</td>
</tr>
<tr>
<td>_B</td>
<td>VT</td>
<td>ESC</td>
<td>+</td>
<td>;</td>
<td>K</td>
<td>[</td>
<td>K</td>
<td>}</td>
</tr>
<tr>
<td>_C</td>
<td>FF</td>
<td>FS</td>
<td>,</td>
<td>&lt;</td>
<td>L</td>
<td>\</td>
<td>l</td>
<td></td>
</tr>
<tr>
<td>_D</td>
<td>CR</td>
<td>GS</td>
<td>-</td>
<td>=</td>
<td>M</td>
<td>]</td>
<td>m</td>
<td>}</td>
</tr>
<tr>
<td>_E</td>
<td>SO</td>
<td>RS</td>
<td>.</td>
<td>&gt;</td>
<td>N</td>
<td>^</td>
<td>n</td>
<td>~</td>
</tr>
<tr>
<td>_F</td>
<td>SI</td>
<td>US</td>
<td>/</td>
<td>?</td>
<td>O</td>
<td>_</td>
<td>o</td>
<td>DEL</td>
</tr>
</tbody>
</table>
Symbols

__tx_thread_context_restore 100
__tx_thread_context_save 100
__application_ISR_entry 100
__tx_thread_context_restore 299
__tx_thread_context_save 299
__tx_thread_stack_error_handler 62
__tx_version_id 40

A

abort suspension of specified thread 272
accelerated development
  benefit of ThreadX 26
activate an application timer 278
activations
  number of 95
  total number of 95
adding assign/release facilities in the
device driver 297
advanced driver issue 303
alleviating some buffer allocation
processing 309
allocate bytes of memory 126
allocate fixed-size block of memory 108
allocation algorithm 90
allocation of processing 22
allocation suspensions
  high number of 88, 92
  number of 88, 91
  total number of 88, 91
allocation timeouts
  number of 88, 91
  total number of 88, 91
allocations
  number of 91
  total number of 91
ANSI C 20
application define 312
application definition function 50
application downloaded to target 28
application entry point 48
application linked 28
application located on host 28
application notification registration 70
application output request 305
application resources 73, 79
application run-time behavior 63
application specific modifications 21
application timer control block 95
application timers 29, 46, 93, 94
application-specific modifications 21
application-specific processing 54
architecture
  non-layering picokernel 20
ASCII character codes in HEX 344
assembly language 20
asynchronous events 96
available 79

B

Background Debug Mode (BDM) 28
basic service call error checking
  disable 36
basic thread suspension 53
BDM (Background Debug Mode) 28
binary semaphores 73, 78
black-box problem elimination 21
block memory services 324
block size 86
Block TX_MUTEX 81
Block TX_THREAD 57
blocks allocated
  number of 88
  total number of 88
blocks released
  number of 88
  total number of 88
buffer I/O management 306
buffered device drivers 307
buffered driver responsibilities 307
buffered I/O advantage 307
buffered output 298
buffering messages 69
byte memory area 303
byte memory services 324

C
C library 20
C main function 31
C pointers 86, 90
C source code 20
change an application timer 280
change priority of an application thread 254
changes time-slice of application thread 270
changing the baud rate of a serial device 297
circular buffer input 303
circular buffers 303, 305
circular byte buffers 303
circular output buffer 305
clock tick 32
compiled application 28
compiler tool 46
completed state 52, 53
configuration options 34
constant 46
constant area 46
context of last execution 59
context switch overhead 64
context switches 24, 64, 65
context switching and polling 25
counting semaphore
  delete 212
  get instance from 214
  get performance information 220
  get system performance information 222
  notify application when put 228
  place instance in 226
  place instance with ceiling 208
  prioritize suspension list 224
  retrieve information about 218
counting semaphores 72, 73, 76, 79, 299
create a memory pool of bytes 130
create a message queue 180
create an application thread 230
create an application timer 282
create an event flags group 144
create mutual exclusion mutex 164
create pool of fixed-size memory blocks 112
creating application timers 94
Index

creating counting semaphores 74
creating event flags groups 83
creating memory block pools 86
creating memory byte pools 89
creating message queues 68
creating mutexes 79
critical sections 73, 79
current device status 298
current thread point 315
currently executing thread 59
Customer Support Center 16

deterministic 85
deterministic real-time behavior 92
deterministic response times 24
development tool
  compiler 46
  linker 46
  locator 46
development tool initialization 48, 49
development tools 46
device drivers 303
device error counts 298
device interrupt frequency 309
device interrupts 303, 309
disabling basic service call error
  checking 36
disabling notify callbacks for ThreadX
  objects 39
disabling the preemption-threshold feature
  defining 38
disabling 0xEF value in byte of thread stack
  defining 38
distribution file 317
demo.threadx.c 317
dividing the application 25
division of application into threads 26
DMA 303
driver access 296, 297
driver control 296, 297
driver example 298, 299
driver functions 296
driver initialization 296, 297
driver input 296, 297
driver input interface 309
driver interrupts 296, 298
driver output 296, 298
driver status 296, 298
driver termination 296, 298
dynamic memory 46, 48
dynamic memory usage 48

E

ease of use
  ThreadX 26
elimination of internal system timer thread for ThreadX
  defining 37
embedded applications 22
  allocation of processor between tasks 22
definition 22
definition of 22
  multitasking 22
embraces avoided 77
empty messages in a message queue 184
enable and disable interrupts 162
enabling performance gathering information on mutexes 39
enabling performance information gathering on block pools 39
enabling performance information gathering on byte pools 39
enabling performance information gathering on event flags groups 39
enabling performance information gathering on event flags
  get suspensions 84
  number of 84
  total number of 84
  get timeouts 84
  number of 84
  total number of 84
  gets 84
  number of 84
  total number of 84
  group
  create 144
  group control block 85
  groups 31
  set notification 83
  sets
  number of 84
  total number of 84
  notification 73, 78
  _flags_0 316
  chaining 70
  advantages of 71, 75, 84
  example of suspended threads 77
  example system 32
  excessive timers 96
  exchanging a free buffer for an input buffer 309
  executing state 52, 53
  execution
    initialization 45
    interrupt service routines (ISR) 44
  execution context 60

get event flags from group 148
get performance information 154
notify application when set 160
retrieve information about group 152
retrieve performance system information 156
set flags in group 158
event flags get suspensions
  number of 84
  total number of 84
event flags get timeouts
  number of 84
  total number of 84
event flags gets
  number of 84
  total number of 84
  group
  create
<table>
<thead>
<tr>
<th>Index</th>
<th>349</th>
</tr>
</thead>
<tbody>
<tr>
<td>execution overview 44</td>
<td>function call nesting 60</td>
</tr>
<tr>
<td>expiration adjustments</td>
<td>function calls 59</td>
</tr>
<tr>
<td>number of 95</td>
<td></td>
</tr>
<tr>
<td>total number of 95</td>
<td></td>
</tr>
<tr>
<td>expirations</td>
<td></td>
</tr>
<tr>
<td>number of 95</td>
<td></td>
</tr>
<tr>
<td>total number of 95</td>
<td></td>
</tr>
<tr>
<td>external events 64</td>
<td></td>
</tr>
<tr>
<td>F</td>
<td></td>
</tr>
<tr>
<td>fast memory 48</td>
<td></td>
</tr>
<tr>
<td>faster time to market</td>
<td></td>
</tr>
<tr>
<td>benefit of ThreadX 26</td>
<td></td>
</tr>
<tr>
<td>FIFO order 69, 74, 79, 87, 91</td>
<td></td>
</tr>
<tr>
<td>first_unused_memory 33</td>
<td></td>
</tr>
<tr>
<td>first-available RAM 50</td>
<td></td>
</tr>
<tr>
<td>first-fit memory allocation 89</td>
<td></td>
</tr>
<tr>
<td>first-in-first-out (FIFO) 55</td>
<td></td>
</tr>
<tr>
<td>fixed-size block of memory allocation of 108</td>
<td></td>
</tr>
<tr>
<td>fixed-size blocks 86</td>
<td></td>
</tr>
<tr>
<td>fixed-size memory 85</td>
<td></td>
</tr>
<tr>
<td>fixed-size memory blocks create pool of 112</td>
<td></td>
</tr>
<tr>
<td>fixed-sized messages 68</td>
<td></td>
</tr>
<tr>
<td>fragmentation 85</td>
<td></td>
</tr>
<tr>
<td>definition of 89</td>
<td></td>
</tr>
<tr>
<td>fragmented pool 90</td>
<td></td>
</tr>
<tr>
<td>fragments created</td>
<td></td>
</tr>
<tr>
<td>number of 91</td>
<td></td>
</tr>
<tr>
<td>total number of 91</td>
<td></td>
</tr>
<tr>
<td>fragments merged</td>
<td></td>
</tr>
<tr>
<td>number of 91</td>
<td></td>
</tr>
<tr>
<td>total number of 91</td>
<td></td>
</tr>
<tr>
<td>fragments searched</td>
<td></td>
</tr>
<tr>
<td>number of 91</td>
<td></td>
</tr>
<tr>
<td>total number of 91</td>
<td></td>
</tr>
<tr>
<td>G</td>
<td></td>
</tr>
<tr>
<td>gathering of performance information on semaphores 39</td>
<td></td>
</tr>
<tr>
<td>get a message from message queue 198</td>
<td></td>
</tr>
<tr>
<td>get block pool performance information 118</td>
<td></td>
</tr>
<tr>
<td>get block pool system performance information 120</td>
<td></td>
</tr>
<tr>
<td>get byte pool performance information 136</td>
<td></td>
</tr>
<tr>
<td>get byte pool system performance information 138</td>
<td></td>
</tr>
<tr>
<td>get event flags from event flags group 148</td>
<td></td>
</tr>
<tr>
<td>get event flags group performance information 154</td>
<td></td>
</tr>
<tr>
<td>get instance from counting semaphore 214</td>
<td></td>
</tr>
<tr>
<td>get mutex performance information 172</td>
<td></td>
</tr>
<tr>
<td>get mutex system performance information 174</td>
<td></td>
</tr>
<tr>
<td>get queue performance information 192</td>
<td></td>
</tr>
<tr>
<td>get queue system performance information 194</td>
<td></td>
</tr>
<tr>
<td>get semaphore performance information 220</td>
<td></td>
</tr>
<tr>
<td>get semaphore system performance information 222</td>
<td></td>
</tr>
<tr>
<td>get thread performance information 244</td>
<td></td>
</tr>
<tr>
<td>get thread system performance information 248</td>
<td></td>
</tr>
<tr>
<td>get timer performance information 290</td>
<td></td>
</tr>
<tr>
<td>get timer system performance information 292</td>
<td></td>
</tr>
<tr>
<td>getting started 27</td>
<td></td>
</tr>
<tr>
<td>global data structures 28</td>
<td></td>
</tr>
</tbody>
</table>

Express Logic, Inc.
global variables 47
globals 63

H
hardware devices 296
hardware interrupt 46
hardware interrupts 298
heterogeneous 54
hidden system thread 96
high throughput I/O 307
highest priority thread 314
high-frequency interrupts 100
host computers 28
host considerations 28

I
I/O buffer 306
I/O buffering 303
I/O drivers 296
ICE (In-Circuit Emulation) 28
idle system returns
  low number of 66
  number of 66
IEEE 1149.1 28
improved responsiveness
  ThreadX benefit 24
In-Circuit Emulation (ICE) 28
increased throughput 24
in-house kernels 21
initial execution 313
initialization 44, 45, 48
initialization process 48
initialized data 46, 47
input and output notifications 299
input buffer 303, 304
input buffer list 307
input byte requests 304
input bytes 298
input characters 303
input semaphore 300
input-output lists 308
installation
  troubleshooting 34
  enable and disable 162
  interrupt frequency 309
  interrupt latency 100
  interrupt management 309
  interrupt preemptions
    number of 65, 66
  interrupt service routines 44, 45
  interrupt vector handlers 299
interrupting 56
interrupts 44, 50, 96
invalid pointer 63
ISR
  handling the transmit complete interrupt 301
ISR template 99
ISRs 44
  memory cannot be called from 89

J
JTAG 28

L
large local data 62
linker tool 46
linking multiple packets 306
<table>
<thead>
<tr>
<th>Index</th>
<th>351</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux 28</td>
<td></td>
</tr>
<tr>
<td>Linux development platform 30</td>
<td></td>
</tr>
<tr>
<td>local storage 58</td>
<td></td>
</tr>
<tr>
<td>local variable allocation 60</td>
<td></td>
</tr>
<tr>
<td>local variables 59</td>
<td></td>
</tr>
<tr>
<td>locator tool 46</td>
<td></td>
</tr>
<tr>
<td>locking out interrupts 305</td>
<td></td>
</tr>
<tr>
<td>logic for circular input buffer 304</td>
<td></td>
</tr>
<tr>
<td>logic for circular output buffer 305</td>
<td></td>
</tr>
<tr>
<td>logical AND/OR operation 82</td>
<td></td>
</tr>
<tr>
<td>lower-priority threads</td>
<td></td>
</tr>
<tr>
<td>not suspending 66</td>
<td></td>
</tr>
<tr>
<td>low-level initialization 49</td>
<td></td>
</tr>
<tr>
<td>M</td>
<td></td>
</tr>
<tr>
<td>main 33, 49, 51</td>
<td></td>
</tr>
<tr>
<td>main function 49</td>
<td></td>
</tr>
<tr>
<td>malloc calls 89</td>
<td></td>
</tr>
<tr>
<td>memory 53</td>
<td></td>
</tr>
<tr>
<td>memory areas 46</td>
<td></td>
</tr>
<tr>
<td>memory block in cache 86</td>
<td></td>
</tr>
<tr>
<td>memory block pool 85</td>
<td></td>
</tr>
<tr>
<td>delete 114</td>
<td></td>
</tr>
<tr>
<td>get performance information about 118</td>
<td></td>
</tr>
<tr>
<td>get system performance for 120</td>
<td></td>
</tr>
<tr>
<td>prioritize suspension list 122</td>
<td></td>
</tr>
<tr>
<td>release fixed size block 124</td>
<td></td>
</tr>
<tr>
<td>retrieve information about 116</td>
<td></td>
</tr>
<tr>
<td>memory block pool control block 88</td>
<td></td>
</tr>
<tr>
<td>memory block pools 85</td>
<td></td>
</tr>
<tr>
<td>memory block size 86</td>
<td></td>
</tr>
<tr>
<td>memory byte pool 89, 92</td>
<td></td>
</tr>
<tr>
<td>allocate 126</td>
<td></td>
</tr>
<tr>
<td>create 130</td>
<td></td>
</tr>
<tr>
<td>get performance information 136</td>
<td></td>
</tr>
<tr>
<td>get system performance information 138</td>
<td></td>
</tr>
<tr>
<td>prioritize suspension list 140</td>
<td></td>
</tr>
<tr>
<td>release bytes to pool 142</td>
<td></td>
</tr>
<tr>
<td>memory byte pool control block 92</td>
<td></td>
</tr>
<tr>
<td>memory pitfalls 62</td>
<td></td>
</tr>
<tr>
<td>memory pools 31, 48, 50</td>
<td></td>
</tr>
<tr>
<td>memory usage 46</td>
<td></td>
</tr>
<tr>
<td>merging of adjacent memory blocks 89</td>
<td></td>
</tr>
<tr>
<td>message destination pitfall 72</td>
<td></td>
</tr>
<tr>
<td>message queue 67</td>
<td></td>
</tr>
<tr>
<td>create 180</td>
<td></td>
</tr>
<tr>
<td>delete 182</td>
<td></td>
</tr>
<tr>
<td>empty messages from 184</td>
<td></td>
</tr>
<tr>
<td>get message from queue 198</td>
<td></td>
</tr>
<tr>
<td>get queue performance information 192</td>
<td></td>
</tr>
<tr>
<td>get system performance information 194</td>
<td></td>
</tr>
<tr>
<td>notify application when message is sent to queue 206</td>
<td></td>
</tr>
<tr>
<td>prioritize suspension list 196</td>
<td></td>
</tr>
<tr>
<td>retrieve information about 190</td>
<td></td>
</tr>
<tr>
<td>send message to front of queue 186</td>
<td></td>
</tr>
<tr>
<td>send message to queue 202</td>
<td></td>
</tr>
<tr>
<td>message queue capacity 68</td>
<td></td>
</tr>
<tr>
<td>message size 68</td>
<td></td>
</tr>
<tr>
<td>messages received</td>
<td></td>
</tr>
<tr>
<td>total number of 71</td>
<td></td>
</tr>
<tr>
<td>messages sent</td>
<td></td>
</tr>
<tr>
<td>total number of 71</td>
<td></td>
</tr>
<tr>
<td>microkernel vs. picokernel architecture 20</td>
<td></td>
</tr>
<tr>
<td>minimum stack size 60</td>
<td></td>
</tr>
<tr>
<td>defining 37</td>
<td></td>
</tr>
<tr>
<td>misuse of thread priorities 63</td>
<td></td>
</tr>
<tr>
<td>multiple buffers 306</td>
<td></td>
</tr>
<tr>
<td>multiple synchronization events 70</td>
<td></td>
</tr>
<tr>
<td>multitasking 22</td>
<td></td>
</tr>
<tr>
<td>multithreaded 52</td>
<td></td>
</tr>
<tr>
<td>multithreaded environment 25</td>
<td></td>
</tr>
<tr>
<td>multithreading 63, 64, 67</td>
<td></td>
</tr>
<tr>
<td>mutex</td>
<td></td>
</tr>
</tbody>
</table>
create 164
delete 166
get information about 170
get ownership of 168
get performance information 172
get system performance information 174
prioritize suspension list 176
release ownership of 178
mutex get suspensions
  number of 80
  total number of 80
mutex get timeouts
  high number of 80
  number of 80
  total number of 80
mutex gets
  number of 80
  total number of 80
mutex mutual exclusion 79
mutex priority inheritances
  number of 80
  total number of 80
mutex priority inversions
  number of 80
  total number of 80
mutex puts
  number of 80
  total number of 80
mutex services 325
mutex_0 316
mutuexes 31, 50, 53, 57, 64, 78, 79
mutual exclusion 73, 76, 78, 81
my_thread_entry 33

N
non-idle system returns
  number of 66
non-reentrant 63
notify application upon thread entry and exit 236
notify application when event flags are set 160
notify application when message is sent to queue 206
notify application when semaphore is put 228
number of threads 57

O
observing the demonstration 316
obtain ownership of mutex 168
OCD 28
OCD (on chip debug) 28
on-chip debug 28
one-shot timer 93
optimized driver ISRs 303
optimizing applications 71
order of thread execution 313
output buffer 305
output buffer list 307
output bytes 298
output semaphore 299
overhead 90
  associated with multithreaded kernels 25
  reduction due to multithreading 25
overhead impact of multithreaded environments 25
overview 312
ThreadX 20
overwriting memory blocks 89, 93
own 78
ownership count 79
Index

P

packet I/O 303
performance of embedded microprocessors 306
periodic interrupt 29
periodic timers 93
periodics 46
physical memory 48
picokernel 20
picokernel architecture 20
pitfall 78, 81
place an instance in counting semaphore 226
place an instance in counting semaphore with ceiling 208
polling
definition of 298
polling as workaround to control loop response time 24
pool capacity 86, 90
pool memory area 87, 90
portability of ThreadX 20, 26
preemption 55, 56
preemption-threshold 56, 57, 64, 65, 78
changing during run-time 57
too low 66
preemptive scheduling 24
premium package 29
priorities
thread control block field 59
prioritize block pool suspension list 122
prioritize byte pool suspension list 140
prioritize mutex suspension list 176
prioritize queue suspension list 196
prioritize semaphore suspension list 224
priority 54
priority ceiling 56
priority inheritance 57, 64, 81
priority inversion 56, 63, 78, 81
priority inversions
number of 66
priority levels for ThreadX
defining 36
priority of internal ThreadX timer thread
defining 37
priority overhead 64
priority zero 96
priority-based scheduling 24
process
definition of 23
process oriented operating system 23
processing bandwidth 63, 100
processing time allocation prior to real-time kernels 24
processor allocation 25
processor allocation logic 25
processor isolation 25
processor reset 44
processor-independent interface provided by ThreadX 25
producer-consumer 73
product distribution 29
program execution
types of 44
protecting the software investment
ThreadX guarantees migration path 26
public resource 68, 73, 78, 93
memory blocks 86
memory byte pool 89

Q

queue control 72
queue empty suspensions
total number of 71
queue event-chaining 70
queue full error returns
total number of 71
queue full suspensions 71
total number of 71
queue memory area 69
queue messages 53
queue performance information 71
queue send notification 70
queue services 326
queue timeouts
total number of 71
queue_0 314
queues 31, 48, 50

R

RAM
first available 50
initialized data area 47
placing stack in 60
queue memory area in 69
requirements 28
reactivation of ThreadX timers in-line
   defining 37
reactivations (periodic timers)
   number of 95
total number of 95
read and write pointers 304
read pointer 304
readme_threadx.txt 28, 29, 30, 33, 34, 35, 40, 99
ready state 52, 53
ready thread 44
real-time 85
definition of 22
real-time software
definition of 22
real-time systems 44, 56
device drivers embedded in 296
re-creating thread 53
recursive algorithms 62
redundant polling 25
reentrancy of threads 62
reentrant 62
reentrant function 62
register thread stack error notification
callback 264
relative time 96
release a fixed-size block of memory 124
release bytes back to memory pool 142
release ownership of mutex 178
releases
   number of 91
total number of 91
Relinquish control to other application
threads 256
removing logic for initializing ThreadX
global C data structures 38
reset 48, 50
reset thread 258
responsive processing 57
re-starting thread 53
resume suspended application thread 260
retrieve information about an application
timer 288
retrieve information about block pool 116
retrieve information about event flags
group 152
retrieve information about mutex 170
retrieve information about queue 190
retrieve information about semaphore 218
retrieve information about thread 240
retrieve performance system information
about event flags group 156
retrieves current time
time retrieve 274
retrieves pointer to currently executing thread 238
ROM
instruction area location 46
location of instruction area 47
ROM requirements for target 28
round-robin scheduling 55
RTOS standard 22
run-time
preemption-threshold changing during 57
run-time application timer performance 95
run-time behavior 25, 63
run-time block pool performance 87
run-time byte pool performance 91
run-time configuration 92
run-time control. 297
run-time eEvent flags performance 84
run-time image 20
run-time mutex performance 80
run-time queue performance 71
run-time semaphore performance 75
run-time stack checking 38, 62
run-time statistics 298
run-time status 298
run-time thread performance 65

S
scalability 20
scaling among micro-controller-based applications 20
scheduling 50
scheduling loop 59
scheduling threads 44
seeking a new sector on a disk 297
semaphore control block 76, 81
semaphore event-chaining 75
semaphore get suspensions
number of 75
total number of 75
semaphore get timeouts
high number of 76
number of 75
total number of 75
semaphore gets
number of 75
total number of 75
semaphore put notification 74
semaphore puts
number of 75
total number of 75
semaphore services 326
semaphore_0 315
semaphores 31, 50, 53, 78
semi-independent program segment 50
send a message to message queue 202
send message to the front of queue 186
service call preemptions 66
number of 65, 66
service call time-outs 29
set event flags in an event flag group 158
sets the current time 276
setting both the read and write buffer pointers to beginning address of buffer 303
setting the output semaphore 301
simple 302
simple driver initialization 299, 300
simple driver input 300
simple driver output 301, 302
simple driver shortcomings 302
simplifying development with threads 26
T

tailoring kernel with assembly language 20

T

target

definition of 22, 23
ThreadX does not use term 23
tasks vs. threads 23
terminated state 52, 53
terminates an application thread 268
Thread 315

thread

abort suspension of 272
calculate priority of 254
changes time slice of 270
critical sections 56
definition of 23
generate performance information 244
generate system performance 248
highest priority 314
notify application when entering and exiting 236
register stack error notification 264
relinquish control to other threads 256
reset 258
resume suspended 260
retrieves pointer to executing thread 238
stack area 60
stack for saving context of execution 59
stack of 58, 59

size and location of I/O buffers 306
size of ThreadX 20
slow memory 48
software maintenance 24
stack 44
stack areas
  preset with data pattern prior to creating threads 61
stack corruption 62
stack error handler 38
stack error handling routine 62
stack memory area 61
stack pointer 59
stack preset 61
stack size 67, 312
stack size of internal ThreadX timer thread defining 37
stack space 58
stacks 48, 50
starvation 56
  of threads 63
starving threads 63
static memory 46
static memory usage 46
static variables 47
statics 63
suspend an application thread 266
suspended current thread for specified time 262
suspended state 52, 53
suspension 98
suspension aborts
  number of 66
system reset 48, 51
system stack 28, 46, 47
system throughput
  impact on 25
suspend 266
suspend for specified time 262
term that replaces task 23
terminate 268
thread 0 314
thread 1 314
thread 2 314
thread 3 315
thread 4 315
thread 5 315
thread 6 316
thread 7 316
thread control 57
thread control block fields 59
thread control services 327
thread counters 316
thread creation 57
Thread Entry/Exit Notification 54
thread execution 44, 50, 315
thread execution states 52
thread identity 315
thread model 23
thread preemption 53
thread priorities 54, 63
thread priority pitfalls 63
thread relinquishes
  number of 66
thread resumptions
  number of 65
thread scheduling 55
thread scheduling loops 44, 49
thread stack area 59
thread stack sizes 61
thread starvation 63
thread state transition 52
thread states 52
thread suspension 69, 74, 83, 87, 90, 309
thread suspensions
  number of 65
thread timeouts
  number of 66
thread_0 314, 316
thread_0_entry 314
thread_1 314
thread_1_counter 317
thread_1_entry 314
thread_2 314
thread_2_counter 317
thread_2_entry 314
thread_3 315
thread_4 315
thread_5 314, 315, 316
thread_5_entry 315
thread_6 316
thread_7 316
threads 31, 50, 54, 57
  number of 57
  simplifying development with 26
ThreadX
  block memory pool 306
  constants 329
  data types 15
demo application 34
deployed in 300 million devices 22
distribution contents 29
ease of use 26
initialization 312
installation 30
installation of 30
instruction image of 20
managed interrupts 97
overview 20
portability 20
portability of 26
premium 29
primary purpose of 22
processor-independent interface 25
RTOS standard for deeply embedded applications 22
services 101
size of 20
Standard 29
supported processors 312
synchronization primitive 54
using 31
version ID 40
ThreadX benefit 23
accelerated development 26
faster time to market 26
improve time-to-market 26
improved responsiveness 24
throughput reduction 25
tick counter 96
time
set 276
suspension for 53
time services 328
time slicing 55
service call function 29
time-outs 46, 69
service call 29
timer
activate 278
change 280
create 282
deactivate 284
delete 286
get performance information 290
get system performance information 292
retrieve information about 288
timer accuracy 94
timer execution 94
timer intervals 93
timer related functions 29
timer services 94, 328
timer setup 93
timer ticks 55, 93, 94, 96
timers 50
time-slice 55, 59
number of 66
time-slices
number of 66
time-slicing 94
transmiting and receiving data with
buffers 306
transmitting or receiving individual packets of data 306
troubleshooting 34
installation 34
tips 34
where to send information 34
tx.a 30, 31
tx.lib 30, 31
TX_AND_CLEAR 82
tx_api.h 30, 31, 33, 57, 72, 76, 81, 85, 88, 92, 95
tx_application_define 31, 33, 49, 50, 51, 297, 299, 312, 313
TX_AUTO_START 313
tx_block_allocate 97, 108
TX_BLOCK_MEMORY (0x08) 58
TX_BLOCK_POOL 88, 336
tx_block_pool_create 112, 122
tx_block_pool_delete 114
TX_BLOCK_POOL_ENABLE_
PERFORMANCE_INFO 39, 87
tx_block_pool_info_get 97, 116
tx_block_pool_performance_info_get 88, 97, 118
tx_block_pool_performance_system_info_get 88, 97, 120
tx_block_pool_prioritize 87, 97, 122
Index

359

tax_block_release 97, 124
tax_byte_allocate 126, 134
TX_BYTE_MEMORY (0x09) 58
TX_BYTE_POOL 92, 336, 337
tax_byte_pool_create 130, 140
ntax_byte_pool_delete 132
TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO 91
ntax_byte_pool_info_get 97
ntax_byte_pool_performance_info_get 92, 97, 136
ntax_byte_pool_performance_system_info_get 92, 97, 138
tax_byte_pool_prioritize 91, 97, 140
tax_byte_release 142
TX_COMPLETED (0x01) 58
TX_DISABLE_ERROR_CHECKING 36
TX_DISABLE_ERROR_CHECKING 101
TX_DISABLE_NOTIFY_CALLBACKS 39
TX_DISABLE_PREEMPTION_THRESHOLD 38
TX_DISABLE_REDUNDANT_CLEARING 38
TX_DISABLE_STACK_FILLING 38
TX_ENABLE_STACK_CHECKING 38, 62
TX_EVENT_FLAG (0x07) 58
tax_event_flags_create 144, 152
tax_event_flags_delete 146
TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO 39, 84
tax_event_flags_get 82, 98, 148
TX_EVENT_FLAGS_GROUP 85, 337
tax_event_flags_info_get 98, 152
tax_event_flags_performance 154
tax_event_flags_performance_info_get 85, 98
tax_event_flags_performance_system_info_get 85, 98, 156
tax_event_flags_set 82, 98, 158
ntax_event_flags_set_notify 83, 98, 160
rax_ill assembly file 93
TX_INCLUDE_USER_DEFINE_FILE 35
tax_initialize_low_level 29
tax_interrupt_control 97, 98, 162
TX_IO_BUFFER 306
tax_kernel_entry 31, 33, 49, 51
TX_MAX_PRIORITIES 36
TX_MINIMUM_STACK 37, 60
TX_MUTEX 337, 338
tax_mutex_create 164
tax_mutex_delete 166
TX_MUTEX_ENABLE_PERFORMANCE_INFO 39, 80
tax_mutex_get 78, 168
tax_mutex_info_get 170
tax_mutex_performance_info_get 80, 98, 172
tax_mutex_performance_system_info_get 80, 98, 174
tax_mutex_prioritize 79, 176
tax_mutex_put 78, 178
TX_MUTEX_SUSP (0x0D) 58
tax_next_buffer 307
tax_next_packet 306
TX OR CONSUME 82
tax_port.h 15, 30, 31, 37
TX_QUEUE 72, 338, 339
tax_queue_create 180
tax_queue_delete 182
TX_QUEUE_ENABLE_PERFORMANCE_INFO 39, 71
tax_queue_flush 184
tx_queue_front_send 98, 186
tx_queue_info_get 98, 190
tx_queue_performance_info_get 71, 98, 192
TX_QUEUE_SUSP (0x05) 58
TX_READY (0x00) 58
TX_SEMAPHORE 76, 339
TX_SEMAPHORE_SUSP (0x06) 58
TX_SLEEP (0x04) 58
TX_SUSPENDED (0x03) 58
TX_TERMINATED (0x02) 58
TX_THREAD 48, 339, 341

TX_SEMAPHORE_ENABLE_PERFORMANCE_INFO 39, 75
tx_semaphore_get 70, 72, 98, 214
tx_semaphore_info_get 98, 218
tx_semaphore_performance_info_get 76, 98, 220
TX_TIMER 95, 341
tx_timer_activate 98, 278, 288
tx_timer_change 98, 280
TX_THREAD 48, 339, 341

TX_THREAD_ENABLE_PERFORMANCE_INFO 39, 65
tx_thread_create 33, 50, 230, 240
tx_thread_current_ptr 59, 67
tx_thread_delete 234, 272
TX_THREAD_SUSPENDED (0x03) 58
TX_THREAD_TERMINATED (0x02) 58
TX_THREAD 48, 339, 341
Index

TX_TIMER_ENABLE_PERFORMANCE_INFO 39, 95
Tx_timer_info_get 98, 288
TX_TIMER_INTERNAL 341, 342
Tx_timer_performance_info_get 95, 98, 290
Tx_timer_performance_system_info_get 9, 5, 98, 292
TX_TIMER_PROCESS_IN_ISR 37
TX_TIMER_THREAD_PRIORITY 37
TX_TIMER_THREAD_STACK_SIZE 37
tx_user.h 34, 35
types of program execution 44
typical thread stack 60

U
UART 303
un-deterministic behavior 85, 92
un-deterministic priority inversion 57, 64, 82
uninitialized data 46, 47
Unix 28
Unix development platform 30
unnecessary processing due to extra poling 25
unpredictable behavior 50
user-supplied main function 49
using a semaphore to control driver access 297
using ThreadX 31

V
version ID 40

W
watchdog services 46