NET+OS Kernel Guide

90000788_B
3/18/2010

Contents

Contents

Contents

Using This Guide

Review this section for basic information about this guide, as well as for general support contact information.

About this guide

This guide provides comprehensive information about NET+OS kernel based on ThreadX from Express Logic, Inc. NET+OS, a network software suite optimized for the NET+ARM chip, is part of the NET+Works integrated product family.

Who should read this guide

This guide is intended for software developers of embedded real-time applications. You should be familiar with standard real-time operating system functions and the C programming language.

To complete the tasks described in this guide you must have:

- Familiarity with API programming concepts and techniques, especially for network applications and systems
- Sufficient system (user) privileges to perform the tasks described
- Access to a computer system that meets NET_OS hardware and software requirements

What's in this guide

This table describes the typographic conventions used in this guide:

To read about the	See
NET+OS kernel and its relationship to real-time embedded development	Chapter 1, "Introduction to the NET+OS Kernel"
Functional components and operation of the NET+OS kernel	Chapter 2, "Functional Components of the NET+OS Kernel"
API functions of the NET+OS kernel	Chapter 3, "NET+OS Kernel Services"
Design goals of the NET+OS kernel	Chapter 4, "NET+OS Kernel Design Goals"
NET+OS kernel constants and their values, and definitions of data types	Chapter 5, "Programming Reference Information"

Conventions used in this guide

This table describes the typographic conventions used in this guide:

This convention	Is used for
italic type	Emphasis, new terms, variables, and document titles. In command and code examples, italics indicate a placeholder (such as *filename*) where you would specify some value of your own.
monospaced type	File names, pathnames, commands, and code examples.

Related documentation

- *NET+OS Getting Started Guide* explains how to install NET+OS with Green Hills or with GNU tools, and how to build your first application.
- *NET+OS User's Guide*describes how to use NET+OS to develop programs for your application and hardware.
- *NET+OS BSP Porting Guide* describes how to port the board support package (BSP) to a new hardware application, with either Green Hills software or GNU tools.
- *NET+OS Application Software Reference Guide* describes the NET+OS software application programming interfaces (APIs).
- *NET+OS BSP Software Reference Guide* describes the board support package APIs.
- Review the documentation CD-ROM that came with your development kit for information on third-party products and other components.

- Refer to the NET+Works hardware documentation for information appropriate to the chip you are using.

Digi Information

For more information about your Digi products, or for customer service and technical support, contact Digi International.

To contact Digi International by	Use
Mail	Digi International, Inc. 11001 Bren Road East Minnetonka, MN 55343 U.S.A
World Wide Web	www.digiembedded.com
Eservice Support	www.digi.com/support/eservice/login.jsp
Telephone (U.S.)	(952) 912-3444 or (877) 912-3444
Telephone (other locations)	+1 (952) 912-3444 or +1 (877) 912-3444

Introduction to the NET+OS Kernel Guide

C H A P T E R   1

This chapter provides an overview of the NET+OS kernel and its relationship to real-time embedded development.

NET+OS kernel unique features

The NET+OS kernel is a high-performance real-time kernel designed specifically for embedded applications. Unlike other real-time kernels, the NET+OS kernel is designed to be versatile - easily scaling among small micro-controller-based applications through those that use powerful RISC and DSP processors.

What makes the NET+OS kernel so scalable? The reason is based on its underlying architecture. Because NET+OS kernel 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 the NET+OS kernel is completely determined by the application. For most applications, the instruction image of the NET+OS kernel is between 2 Kbytes and 15 Kbytes.

The picokernel architecture

What about performance? Instead of layering kernel functions on top of each other like traditional *microkernel* architectures, NET+OS kernel 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

The NET+OS kernel is primarily written 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 the NET+OS kernel to a new processor family in a very short time - usually within weeks.

Embedded applications

Embedded applications execute on microprocessors that are inside products such as cellular phones, communication equipment, automobile engines, laser printers, medical devices, and so on. Another distinction of embedded applications is that their software and hardware have a dedicated purpose.

Real-time Software

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

Multitasking

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

Tasks vs. threads

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

In contemporary operating system discussion, there are two terms that more or less replace the use of task, namely *process* and *thread*. A *process* is a completely independent program that has its own address space, while a thread is a semiindependent 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, the NET+OS kernel implements a thread model, which is both extremely efficient and practical for most real-time embedded applications. To avoid confusion, this guide does not use the term *task*. Instead, the more descriptive and contemporary name *thread* is used.

NET+OS kernel benefits

The NET+OS kernel provides many benefits to embedded applications, depending on how embedded application threads are allocated processing time.

Improved responsiveness

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

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

The NET+OS kernel provides fast and deterministic response times to important external events. The NET+OS kernel 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 NET+OS 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 the NET+OS kernel.

Increased throughput

A possible work-around to the control loop response time problem is to add more polling. This improves the responsiveness, but still doesn.t guarantee a constant worst-case response time and does nothing to enhance future modification of the application. Also, the processor is now performing even more unnecessary processing because of the extra polling. All of this unnecessary processing reduces the overall throughput of the system.

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

Processor isolation

The NET+OS kernel provides a robust processor-independent interface between the application and the underlying processor. This interface 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.

Using the NET+OS kernel 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, the NET+OS kernel 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

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

Improving time-to-market

All of the benefits of the NET+OS kernel accelerate the software development process. The NET+OS kernel 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, the NET+OS kernel is easily ported to new processor environments. This coupled with the fact the NET+OS kernel insulates applications from details of the underlying processors, makes NET+OS kernel applications highly portable. As a result, the application.s migration path is guaranteed and the original development investment is protected.

Functional Components of the NET+OS Kernel

C H A P T E R 2

This chapter describes the function of the NET+OS kernel.

Execution overview

There are four types of program execution within a NET+OS kernel application:

- Initialization
- Thread execution
- Interrupt service routines (ISRs)
- Application timers

The following figure shows each different type of program execution. More detailed information about each of these types is found in subsequent sections of this chapter.

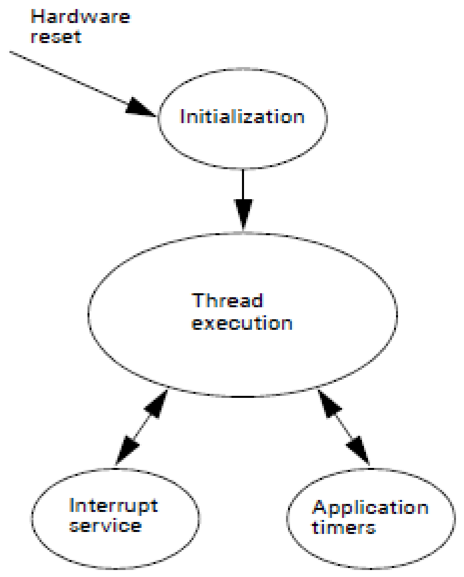

Figure 1: Types of program execution

Initialization

Initialization is the first type of program execution in a NET+OS kernel application. Initialization includes all program execution between processor reset and the entry point of the *thread scheduling loop*.

Thread execution

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

This process of continually executing and scheduling threads is the most common type of program execution in NET+OS kernel 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.

Upon 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 (ISR).

In most cases, interrupts occur during thread execution (or in the thread scheduling loop). However, interrupts may also occur inside of an executing ISR or an application timer.

Application timers

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

Memory usage

The NET+OS kernel resides along with the application program. As a result, the static memory (or fixed memory) usage of the NET+OS kernel is determined by the development tool - for example, 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
- System stack

The following figure shows an example of these memory areas.

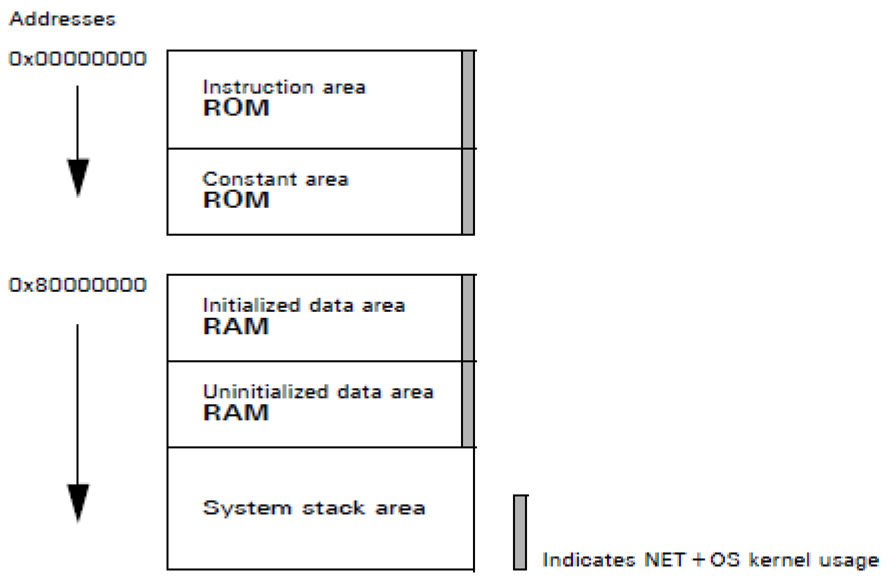

Figure 2: Static memory usage example

It is important to realize that the static memory layout in Figure 2 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 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. The system stack is used by the compiler during initialization and then by the NET+OS kernel during initialization and subsequently in ISR processing.

Dynamic memory usage

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

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

Initialization

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

Note: The NET+OS kernel attempts to use (whenever possible) the complete development tool's initialization process. This makes it easier to upgrade to new versions of the development tools in the future.

System reset

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

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

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

The 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 the NET+OS kernel. However, applications can perform preliminary processing (usually for hardware initialization) prior to entering the NET+OS kernel.

The call to tx_kernel_enter does not return, so do not place any processing after it.

The tx_kernel_enter function

The entry function coordinates initialization of various internal NET+OS kernel 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 runtime memory allocations of thread stacks, queues, and memory pools.

> **Note:** 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.

The following figure shows the entire initialization process, from system reset through application-specific initialization.

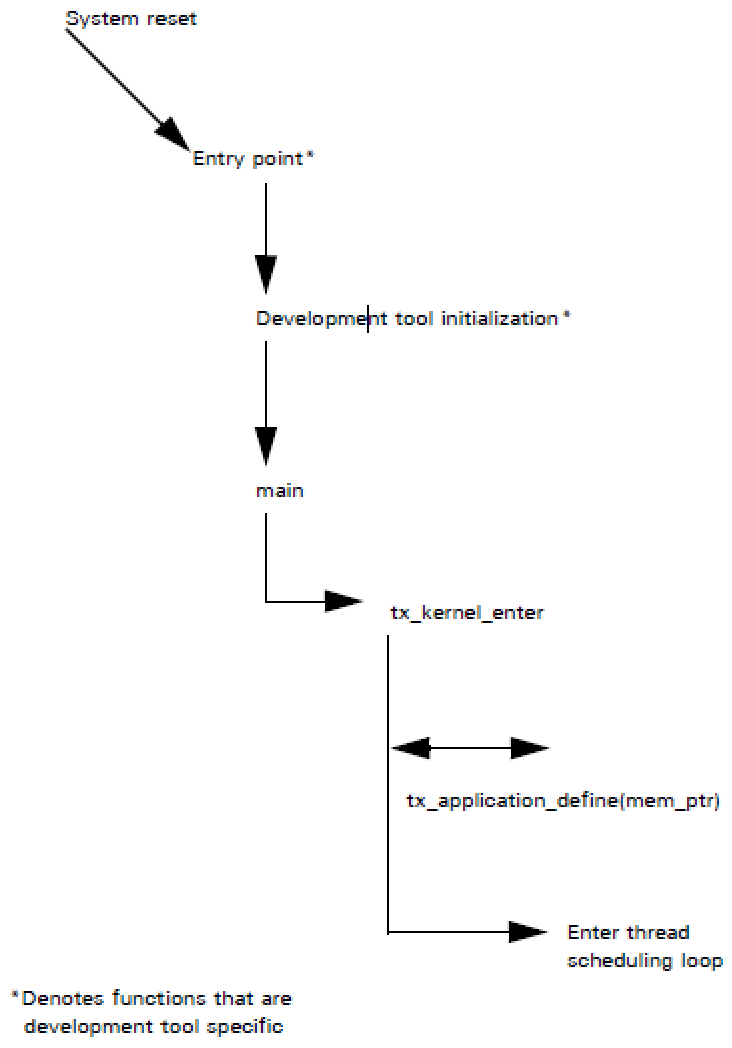
System reset
Entry point*
Development tool initialization*
main
tx_kernel_enter
tx_application_define(mem_ptr)
Enter thread scheduling loop
*Denotes functions that are development tool specific

Figure 3: Initialization process

Thread execution

Scheduling and executing application threads is the most important activity of the NET+OS kernel. A *thread* is typically defined as semi-independent program segment with a dedicated purpose. The combined processing of all threads makes an application.

Threads are created dynamically by tx_thread_create calls during initialization or during thread execution. Threads are created in either a *ready* or *suspended* state.

Thread execution states

Understanding the different processing states of threads is a key ingredient to understanding the entire multi-threaded environment. In the NET+OS kernel there are five distinct thread states:

- Ready
- Suspended
- Executing
- Terminated
- Completed

The following figure shows the thread state transition diagram for the NET+OS kernel.

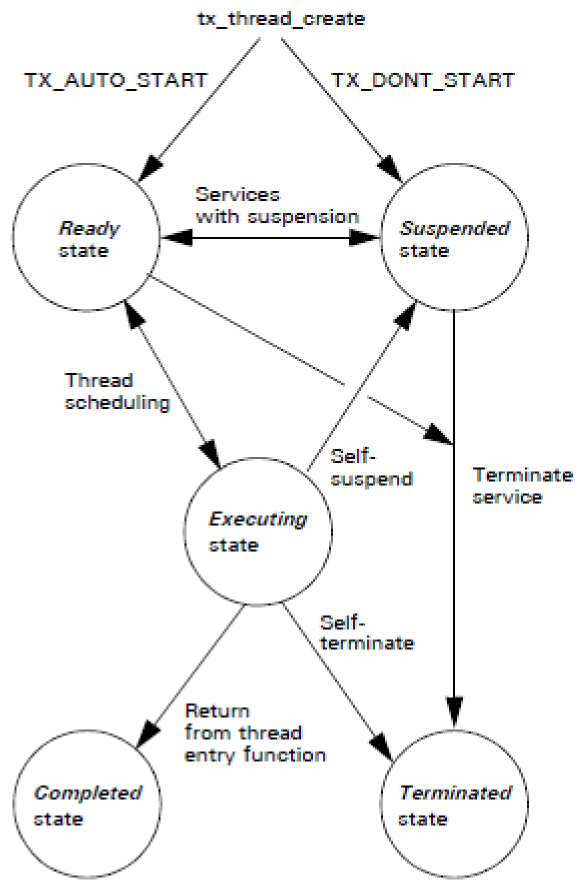

Figure 4: Thread state transition

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

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

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

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

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

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

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

Thread priorities

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

How does the NET+OS kernel determine a thread.s importance? When a thread is created, it is assigned a numerical value representing its importance or *priority*. Valid numerical priorities range between 0 and 31, where a value of 0 indicates the highest thread priority and a value of 31 represents the lowest thread priority. Threads can have the same priority as others in the application. In addition, thread priorities can be changed during run-time.

Thread scheduling

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

Round-robin scheduling

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

Time-slicing

Time-slicing provides another form of round-robin scheduling. In the NET+OS kernel, 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.

A time-slice specifies the maximum number of timer ticks (timer interrupts) that a thread can execute without giving up the processor. When a time-slice expires, all other threads of the same or higher priority levels 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 NET+OS kernel service call that causes preemption, or is itself time-sliced.

Preemption

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

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

Preemption-threshold

In order to ease some of the inherent problems of preemption, the NET+OS kernel provides a unique and advanced feature called *preemption-threshold*. which allows a thread to specify a priority ceiling for disabling preemption. Threads that have higher priorities than the ceiling are still allowed to preempt; those with priorities less than the ceiling are not allowed to preempt.

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

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

Priority inheritance

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

Thread creation

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

Thread control block

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

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

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

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

tx_run_count

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

tx_state

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

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

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

Currently executing thread

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

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

In debug sessions, users can examine the internal NET+OS kernel 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 - that is, the NET+OS kernel 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. The following figure shows a typical thread's stack.

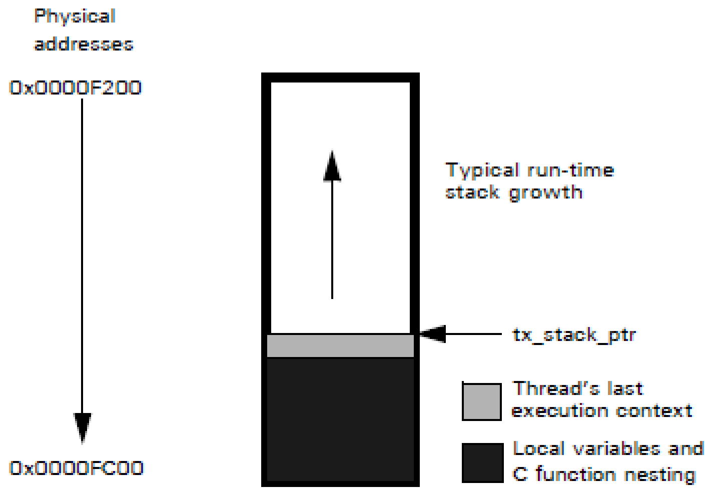
Physical addresses
0x0000F200
0x0000FC00
Typical run-time stack growth
tx_stack_ptr
Thread's last execution context
Local variables and C function nesting

Figure 5: Typical thread stack

Location of a thread stack

The location of a thread stack located 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 a very important feature because it allows applications to improve performance of important threads by placing their stack in highspeed RAM.

Size of a thread stack

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 the NET+OS kernel.
A stack of this size supports saving a thread's context and minimum amount of function calls and local variable allocation.

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

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

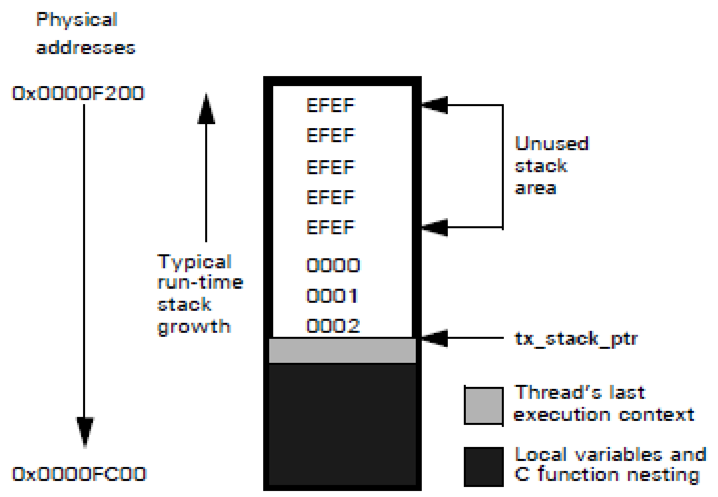

Figure 6: Stack memory area — another example

Memory pitfalls

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

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

Reentrancy

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

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 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 multi-threading. It is sometimes very tempting to assign priorities based on a perceived notion of thread importance rather than determining what is exactly required during runtime. 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, the NET+OS kernel provides a priority-based, preemptive scheduling algorithm. Lower priority threads do not execute until there are no higher-priority threads ready for execution. If a higher-priority thread is always ready, the lower-priority threads never execute. This condition is called *thread starvation*.

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

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

There are principally three distinct methods of preventing un-deterministic priority inversion in the NET+OS kernel. 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 NET+OS kernel mutex objects to protect system resources may utilize the optional mutex *priority inheritance* to eliminate un-deterministic priority inversion.

Priority overhead

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

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

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

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

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

Debugging pitfalls

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

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

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

Message queues

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

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

Each message queue is a public resource. The NET+OS kernel places no constraints on how message queues are used.

Creating message queues

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

Message size

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

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

Message queue capacity

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

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

Queue memory area

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

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

Thread suspension

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

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

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

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

Queue control block

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

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

Message destination pitfall

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

Note: This is especially lethal when a too-small message destination is on the stack.

Counting semaphores

The NET+OS kernel provides 32-bit counting semaphores that range in value between 0 and 4,294,967,295. There are two operations on 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. The NET+OS kernel 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*.

Note: 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 un-availability of the resource.

Event notification

It is also possible to use counting semaphores as event notification, in a producer consumer fashion. The consumer attempts to get the counting semaphore while the producer increases the semaphore whenever something is available. Such semaphores usually have an initial value of 0 and won.t increase until the producer has something ready for the consumer.

Creating counting semaphores

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

Thread suspension

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

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

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

Semaphore control block

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

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

Deadly embrace

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

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

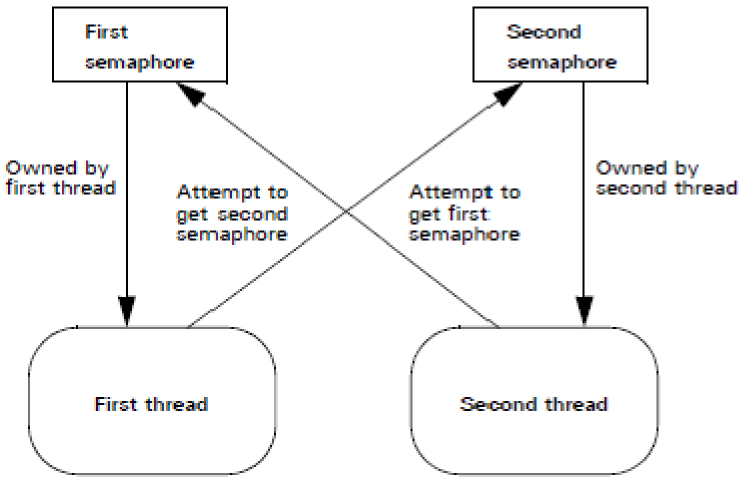

Figure 7: Example of suspended threads

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

Note: 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 the section on .Thread priority pitfalls.

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

Mutexes

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

Each mutex is a public resource. The NET+OS kernel places no constraints on how mutexes are used.

NET+OS kernel mutexes are used exclusively 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 NET+OS kernel mutex will have an ownership count of 0. Once the mutex is obtained by a thread, the ownership count is incremented once for every get operation performed on the mutex and decremented for every put operation.

Creating mutexes

NET+OS kernel mutexes are created either during initialization or during runtime by application threads. The initial condition of a mutex is always .available. Mutex creation is also where the determination is made as to whether or not the mutex implements *priority inheritance.*

Thread suspension

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

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

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

Mutex control block

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

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

Deadly embrace

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

Priority inversion

As mentioned previously, a major pitfall associated with mutual exclusion is priority inversion. This topic is discussed more fully in the section on thread priority pitfalls.

The basic problem results from a situation where a lower-priority thread has a semaphore that a higher-priority thread needs. This in itself is normal. However, threads with priorities in between them may cause the priority inversion to last a non-deterministic amount of time. Unlike semaphores discussed previously, the NET+OS kernel mutex object has optional *priority inheritance*.

The basic idea behind priority inheritance is that a lower priority thread has its priority raised temporarily to the priority of a high priority thread that wants the same mutex owned by the lower priority thread. When the lower priority thread releases the mutex, its original priority is then restored and the higher priority thread is given ownership of the mutex. This feature eliminates un-deterministic priority inversion by bounding the amount of inversion to the time the lower priority thread holds the mutex. Of course, the techniques discussed earlier in this chapter to handle un-deterministic priority inversion are also valid with mutexes as well.

Event flags

Event flags provide a powerful tool for thread synchronization. Each event flag is represented by a single bit. Event flags are arranged in groups of 32. Threads can operate on all 32 event flags in a group at the same time. Events are set by tx_event_flags_set and are retrieved by tx_event_flags_get.

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

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

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

Each event flag group is a public resource. The NET+OS kernel places no constraints on how event flag groups are used.

Creating event flag groups

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

Thread suspension

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

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

Event flag group control block

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

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

Memory block pools

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

Since memory block pools consist of fixed-size blocks, there are never any fragmentation problems. Of course, fragmentation causes behavior that is inherently un-deterministic. In addition, the time required to allocate and free a fixed-size memory block amounts to 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. The NET+OS kernel places no constraints on how pools are used.

Creating memory block pools

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

Memory block size

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

The NET+OS kernel adds a small amount of overhead - the size of a C pointer - to each memory block in the pool. In addition, the NET+OS kernel might have to pad the block size in order to keep the beginning of each memory block on proper alignment.

Pool capacity

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

Pool’s memory area

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

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

Thread suspension

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

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

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

Memory block pool control block

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

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

Overwriting memory blocks

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

Memory byte pools

NET+OS kernel 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 - that is, 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* 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. The NET+OS kernel 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 are no limits on the number of memory byte pools in an application.

Pool capacity

The number of allocatable bytes in a memory byte pool is slightly less than what was specified during creation. This is because management of the free memory area introduces some overhead.

Each free memory block in the pool requires the equivalent of two C pointers of overhead. In addition, the pool is created with two blocks, a large free block and a small permanently allocated block at the end of the memory area. This allocated block is used to improve performance of the allocation algorithm. It eliminates the need to continuously check for the end of the pool area during merging.

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

Pool's memory area

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

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

Thread suspension

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

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

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

Memory byte pool control block

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

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

Un-deterministic behavior

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

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

Overwriting memory blocks

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

Application timers

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

NET+OS 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. The NET+OS kernel places no constraints on how application timers are used.

Timer intervals

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

It is worth mentioning that the underlying hardware must have the ability to generate periodic interrupts in order for application timers to function. In some cases, the processor has a built-in periodic interrupt capability. If the processor doesn.t have this ability, the user.s board must have a peripheral device that can generate periodic interrupts.

Note: The NET+OS kernel can still function even without a periodic interrupt source. However, all timer-related processing is then disabled. This includes time-slicing, suspension time-outs, and timer services.

Timer accuracy

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

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

Timer execution

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

Creating application timers

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

Application timer control block

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

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

Excessive timers

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

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

Relative time

In addition to the application timers mentioned previously, the NET+OS kernel 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 the NET+OS kernel.

Interrupts

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

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

Interrupt control

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

NET+OS kernel managed interrupts

The NET+OS kernel provides applications with complete interrupt management. This management includes saving and restoring the context of the interrupted execution. In addition, the NET+OS kernel allows certain services to be called from within interrupt service routines (ISRs). The following is the list of NET+OS kernel services allowed from application ISRs:

tx_block_allocate
tx_block_pool_info_get
tx_block_pool_prioritize
tx_block_release
tx_byte_pool_info_get
tx_byte_pool_prioritize
tx_event_flags_info_get
tx_event_flags_get

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

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

ISR template

To manage application interrupts, several NET+OS kernel utilities must be called in the beginning and end of application ISRs. The exact format for interrupt handling varies between ports. Review the readme.txt file on the distribution disk for specific instructions on managing ISRs.

The following small code segment is typical of most NET+OS kernel-managed ISRs. In most cases, this processing is in assembly language.

```
_application_ISR_entry:
; Save context and prepare for
; NET+OS kernel use by calling the ISR
; entry function.

CALL __tx_thread_context_save

; The ISR can now call NET+OS kernel
; 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 the NET+OS kernel. This is accomplished by simply calling the entry and exit functions described in the above template.

Interrupt latency

The NET+OS kernel 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.

NET+OS Kernel Services

C H A P T E R 3

This chapter lists the API functions of the NET+OS kernel.

Summary of NET+OS kernel API functions

The following sections list the NET+OS kernel API functions according to the types of service they provide.

Memory block pool services

- TX_BLOCK_ALLOCATE
- TX_BLOCK_POOL_CREATE
- TX_BLOCK_POOL_DELETE
- TX_BLOCK_POOL_INFO_GET
- TX_BLOCK_POOL_PRIORITIZE
- TX_BLOCK_RELEASE

Memory byte pool services

- TX_BYTE_ALLOCATE
- TX_BYTE_POOL_CREATE
- TX_BYTE_POOL_DELETE
- TX_BYTE_POOL_INFO_GET
- TX_BYTE_POOL_PRIORITIZE
- TX_BYTE_RELEASE

Event flag services

- TX_EVENT_FLAGS_CREATE
- TX_EVENT_FLAGS_DELETE
- TX_EVENT_FLAGS_GET
- TX_EVENT_FLAGS_INFO_GET

Interrupt control

- TX_INTERRUPT_CONTROL

Message queue services

- TX_QUEUE_CREATE
- TX_QUEUE_DELETE
- TX_QUEUE_FLUSH
- TX_QUEUE_FRONT_SEND
- TX_QUEUE_INFO_GET

- TX_QUEUE_PRIORITIZE
- TX_QUEUE_RECEIVE
- TX_QUEUE_SEND

Semaphore services

- TX_SEMAPHORE_CREATE
- TX_SEMAPHORE_DELETE
- TX_SEMAPHORE_GET
- TX_SEMAPHORE_INFO_GET
- TX_SEMAPHORE_PRIORITIES
- TX_SEMAPHORE_PUT

Mutex services

- TX_MUTEX_CREATE
- TX_MUTEX_DELETE
- TX_MUTEX_GET
- TX_MUTEX_INFO_GET
- TX_MUTEX_PRIORITIZE
- TX_MUTEX_PUT

Thread control services

- TX_THREAD_CREATE
- TX_THREAD_DELETE
- TX_THREAD_IDENTIFY
- TX_THREAD_INFO_GET
- TX_THREAD_PREEMPTION_CHANGE
- TX_THREAD_PRIORITY_CHANGE
- TX_THREAD_RELINQUISH
- TX_THREAD_RESUME
- TX_THREAD_SLEEP
- TX_THREAD_SUSPEND
- TX_THREAD_TERMINATE
- TX_THREAD_TIME_SLICE_CHANGE
- TX_THREAD_WAIT_ABORT

Time services

- TX_TIME_GET

- TX_TIME_SET

Timer services

- TX_TIMER_ACTIVATE
- TX_TIMER_CHANGE
- TX_TIMER_CREATE
- TX_TIMER_DEACTIVATE
- TX_TIMER_DELETE
- TX_TIMER_INFO_GET

Return values and disabled error checking

The following return values are not affected by TX_DISABLE_ERROR_CHECKING, which is used to disable API error checking; other values can be completely disabled.

- TX_ACTIVATE_ERROR
- TX_DELETED
- TX_DELETE_ERROR
- TX_NO_EVENTS
- TX_NO_INSTANCE
- TX_NO_MEMORY
- TX_NOT_OWNED
- TX_QUEUE_EMPTY
- TX_QUEUE_FULL
- TX_RESUME_ERROR
- TX_SUCCESS
- TX_SUSPEND_ERROR
- TX_SUSPEND_LIFTED
- TX_WAIT_ABORTED
- TX_WAIT_ABORT_ERROR

NET+OS kernel API functions

The following pages describe the NET+OS kernel API functions. They are listed in alphabetic order with similar services grouped together. For example, all memory block services are at the beginning of this chapter.

tx_block_allocate

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

Format

UINT tx_block_allocate(TX_BLOCK_POOL **pool_ptr*,

VOID ***block_ptr*, ULONG *wait_option*)

Arguments

Arguments	**Description**
pool_ptr	Pointer to a 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 argument points to.
wait_option	Defines how the service behaves if there are no memory blocks available. The options are as follows: ■ TX_NO_WAIT (0x00000000) 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 nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until a memory block is available. ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for a memory block.

Return values

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

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

```
TX_BLOCK_POOL my_pool;
unsigned char *memory_ptr;
UINT status;
/* Allocate a memory block from my_pool. Assume that the pool has already been created
   with a call to tx_block_pool_create. */
status = tx_block_allocate(&my_pool, (VOID *) &memory_ptr, TX_NO_WAIT);
/* If status equals TX_SUCCESS, memory_ptr contains the address of the allocated block of
    memory. */
```

See also

tx_block_pool_create, tx_block_pool_delete,

tx_block_pool_info_get, tx_block_pool_prioritize, tx_block_release

tx_block_pool_create

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 formula.

Format

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

Arguments

Arguments	Description
pool_ptr	Pointer to the memory block pool
name_ptr	Pointer to the memory block pool
block_size	Number of bytes in each memory block pool
pool_start	Pointer to the starting address of the memory block pool
pool_size	Total number of bytes available for the memory block pool

Return Values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer
TX_PTR_ERROR (0x03)	Invalid starting address of the pool
TX_SIZE_ERROR (0x04)	Size of the pool is invalid
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

No

Example

```
TX_BLOCK_POOL my_pool;
UINT status;
/* Create a memory pool whose total size is 1000 bytes starting at address 0x100000. Each block in
this pool is defined to be 50 bytes long. */
status = tx_block_pool_create(&my_pool, "my_pool_name",
          50, (VOID *) 0x100000, 1000);
/* If status equals TX_SUCCESS, my_pool contains 18 memory blocks of 50 bytes each. The reason
there are not 20 blocks in the pool is because of the one overhead pointer associated with each
block.*/
```

See also

tx_block_allocate, tx_block_pool_delete, tx_block_pool_info_get,

tx_block_pool_prioritize, tx_block_release

tx_block_pool_delete

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.

Format

UINT tx_block_pool_delete(TX_BLOCK_POOL **pool_ptr*)

Arguments

Arguments	**Description**
pool_ptr	Pointer to the memory block pool

Return Values

Return Value	**Description**
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads

Preemption possible

Yes

Example

```
TX_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 is TX_SUCCESS, the memory block pool is deleted. */
```

See also
tx_block_allocate, tx_block_pool_create, tx_block_pool_info_get,
tx_block_pool_prioritize, tx_block_release

tx_block_pool_info_get

Retrieves information about the specified block memory pool.

Format

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*)

Arguments

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

Return Values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

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

/* Retrieve information about a the previously created block pool .my_pool.. */

status =     tx_block_pool_info_get(&my_pool, &name,
             &available,&total_packets,
             &first_suspended, &suspended_count,
             &next_pool);

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

See also

tx_block_pool_allocate, tx_block_pool_create, tx_block_pool_delete, tx_block_pool_prioritize, tx_block_release

tx_block_pool_prioritize

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.

Format

UINT tx_block_pool_prioritize(TX_BLOCK_POOL **pool_ptr*)

Arguments

Arguments	**Description**
pool_ptr	Pointer to the memory block pool

Return Values

Return Value	**Description**
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_BLOCK_POOL my_pool;
UINT status;

/* Ensure that the highest priority thread will receive the next free block in this pool. */

status =    tx_block_pool_prioritize(&my_pool);

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

See also

tx_block_allocate, tx_block_pool_create, tx_block_pool_delete,
tx_block_pool_info_get, tx_block_release

tx_block_release

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

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

Format

UINT tx_block_release(VOID **blovk_ptr*)

Arguments

Arguments	Description
block_ptr	Pointer to the memory block

Return Values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_PTR_ERROR (0x02)	Invalid pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

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

/* Release a memory block back to my_pool. Assume that the pool has been created and the memory block has been allocated.*/

```
status = tx_block_release((VOID *) memory_ptr);
```

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

See also

tx_block_allocate, tx_block_pool_create, tx_block_pool_delete,
tx_block_pool_info_get, tx_block_pool_prioritize

tx_byte_allocate

Allocates the specified number of bytes from the specified byte-memory pool. The performance of this service depends on block size and the amount of fragmentation in the pool. Therefore, do not use this service during time-critical threads of execution.

Format

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

Arguments

Arguments	Description
pool_ptr	Pointer to a memory block pool
memory_ptr	Pointer to a destination memory pointer. On successful allocation, the address of the allocated memory area is placed where this argument points to.
memory_size	Number of bytes requested
wait_option	Defines how the service behaves if there are no memory blocks available. The options are as follows: ■ TX_NO_WAIT (0x00000000) 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 nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until a memory block is available. ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for a memory block.

Return values

Return Value	Description
`TX_SUCCESS (0x00)`	Success
`TX_DELETED (0x01)`	Memory block pool was deleted while thread was suspended
`TX_NO_MEMORY (0x10)`	Unable to allocate a block of memory
`TX_WAIT_ABORTED (0x1A)`	Suspension was aborted by another thread, timer, or ISR
`TX_POOL_ERROR (0x02)`	Invalid memory block pool pointer
`TX_PTR_ERROR (0x03)`	Invalid pointer (NULL) for any destination pointer
`TX_CALLER_ERROR (0x13)`	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

Yes

Example

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

```
/* Allocate a 112 byte memory area from my_pool. Assume the pool has already been created with a
call to tx_byte_pool_create.*/
```

```
    status = tx_byte_allocate(&my_pool, (VOID *) &memory_ptr,
            112, TX_NO_WAIT);
```

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

See also

tx_byte_pool_create, tx_byte_pool_delete, tx_byte_pool_info_get,

tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_create

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

Format

UINT tx_byte_pool_create(TX_BYTE_POOL **pool_ptr*,

CHAR *name_ptr, VOID *pool_start, ULONG pool_size)

Arguments

Arguments	**Description**
pool_ptr	Pointer to a memory pool
name_ptr	Pointer to the memory pool name
pool_start	Pointer to the starting address of the memory pool
pool_size	Total number of bytes available for the memory pool

Return values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory block pool pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service
TX_SIZE_ERROR (0x05)	Size of pool is invalid

Allowed from

Initialization and threads

Preemption possible

No

Example

```
        TX_BYTE_POOL my_pool;
        UINT status;
/* Create a memory pool whose total size is 2000 bytes starting at address 0x500000. */

status = tx_byte_pool_create(&my_pool, .my_pool_name.,
        (VOID *) 0x500000, 2000);

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

See also

tx_byte_allocate, tx_byte_pool_delete, tx_byte_pool_info_get,

tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_delete

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

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

Format

UINT tx_byte_pool_delete(TX_BYTE_POOL **pool_ptr*)

Arguments

Arguments	Description
pool_ptr	Pointer to the memory block pool

Return Values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads

Preemption possible

Yes

Example

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

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

See also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_info_get,

tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_info_get

Retrieves information about the specified memory byte pool.

Format

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*)

Arguments

Arguments	Description
pool_ptr	Pointer to a byte pool
name	Pointer to destination for the pointer to the byte pool 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 the 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 pointer to the thread that is first on the suspension list of this byte pool
next_pool	Pointer to destination for the pointer of the next created byte pool

Return values

Return Value	Description
`TX_SUCCESS (0x00)`	Success
`TX_POOL_ERROR (0x02)`	Invalid memory pool pointer
`TX_PTR_ERROR (0x03)`	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

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

/* Retrieve information about the block pool */

status = tx_byte_pool_info_get(&my_pool, &name,
        &available, &fragments,
        &first_suspended, &suspended_count,
        &next_pool);
```

/* If status is TX_SUCCESS, information requested is valid */

See also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete,

tx_byte_pool_prioritize, tx_byte_release

tx_byte_pool_prioritize

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.

Format

UINT tx_byte_pool_prioritize(TX_BYTE_POOL **pool_ptr*)

Arguments

Arguments	Description
pool_ptr	Pointer to a memory pool

Return Values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
        TX_BYTE_POOL my_pool;
        UINT status;
```

/* Ensure that the highest priority thread will receive the next free memory from this pool. */

status = **tx_byte_pool_prioritize**(&my_pool);

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

See also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete,

tx_byte_pool_info_get, tx_byte_release

tx_byte_release

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 runs out 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 memory area after it is released.

Format

UINT tx_byte_release(VOID **memory_ptr*)

Arguments

Arguments	Description
memory_ptr	Pointer to a destination memory pointer. On successful allocation, the address of the allocated memory area is placed where this argument points to.

Return Values

Return Value	Description
TX_SUCCESS (0x00)	Success
TX_POOL_ERROR (0x02)	Invalid memory pool pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

Yes

Example

```
unsigned char *memory_ptr;
UINT status;
/* Release a memory back to my_pool. Assume that the memory area was previously allocated from
my_pool. */
status = tx_byte_release((VOID *) memory_ptr);
/* If status equals TX_SUCCESS, the memory pointed to by memory_ptr has been returned to the
pool. */
```

See also

tx_byte_allocate, tx_byte_pool_create, tx_byte_pool_delete, tx_byte_pool_info_get, tx_byte_pool_prioritize

tx_event_flags_create

Creates a group of 32 event flags. All 32 event flags in the group are initialized to zero.

Format

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

Arguments

Arguments	Description
group_ptr	Pointer to an event flags group.
name_ptr	Pointer to the name of the event flags group.

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_GROUP_ERROR (0x06)	Invalid event group pointer - either the pointer is NULL or the event group already exists
TX_CALLER_ERROR (0x03)	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

No

Example

```
        TX_EVENT_FLAGS_GROUP my_event_group;
        UINT status;

        /* Create an event flag group. */

status = tx_event_flags_create(&my_event_group,
        "my_event_group_name");

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

See also

tx_event_flags_delete, tx_event_flags_get,

tx_event_flags_info_get, tx_event_flags_set

tx_event_flags_delete

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

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

Format

UINT tx_event_flags_delete(TX_EVENT_FLAGS_GROUP **group_ptr*,

Arguments

Arguments	Description
group_ptr	Pointer to an event flags group

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_GROUP_ERROR (0x06)	Invalid event group pointer
TX_CALLER_ERROR (0x03)	Invalid caller of this service

Allowed from

Threads

Preemption possible

Yes

Example

```
        TX_EVENT_FLAGS_GROUP my_event_flag_group;
        UINT status;

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

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

See also

tx_event_flags_create, tx_event_flags_get,

tx_event_flags_info_get, tx_event_flags_set

tx_event_flags_get

Retrieves event flags from the specified event flag group. Each event flag group contains 32 event flags. Each flag is represented by a single bit.

Format

UINT tx_event_flags_get(TX_EVENT_FLAGS_GROUP **group_ptr*,

ULONG *requested_flags*, UINT *get_option*,

ULONG **actual_flags_ptr*, ULONG *wait_option*)

Arguments

Arguments	Description
group_ptr	Pointer to the event flags group
requested_flags	32-bit unsigned variable that represents the requested event flags.
get_option	Number of bytes requested
wait_option	Specifies whether any or all 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 options are as follows: ■ TX_NO_WAIT (0x00000000) 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 nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until the event flags are available. ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for the event flags.

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_DELETED (0x01)	Event flag group was deleted while thread was suspended
TX_NOEVENTS (0x07)	Unable to get the specified events
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_GROUP_ERROR (0x06)	Invalid event group pointer - either the pointer is NULL or the event group already exists
TX_PTR_ERROR (0x03)	Invalid pointer for actual event flags
TX_WAIT_ERROR (0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread
TX_OPTION_ERROR (0x08)	Invalid get option specified

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

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

See also

tx_event_flags_create, tx_event_flags_delete,

tx_event_flags_info_get, tx_event_flags_set

tx_event_flags_info_get

Retrieves information about the specified event flags group.

Format

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*)

Arguments

Arguments	Description
group_ptr	Pointer to a an event flags group
name	Pointer to destination for the pointer to the event flag group.s name
current_flags	Pointer to destination for the current set flags in the event flag group
first_suspended	Pointer to the destination for the pointer to the thread that is first on the suspension list of this event flag group
suspended_count	Pointer to destination for the pointer to the thread that is first on the suspension list of this event flag group
next_group	Pointer to destination for the pointer of the next created event flag group

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_GROUP_ERROR (0x06)	Invalid event group pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

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

```
/* Retrieve information about a the previously created event flag group .my_event_group.. */

          status =   tx_event_flags_info_get(&my_event_group, &name,
                    &current_flags,
                    &first_suspended, &suspended_count,
                    &next_group);

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

See also

tx_event_flags_create, tx_event_flags_delete, tx_event_flags_get,

tx_event_flags_set

tx_event_flags_set

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

Format

UINT tx_event_flags_set TX_EVENT_FLAGS_GROUP**group_ptr*,

ULONG*flags_to_set*,UNIT *set_option)*

Arguments

Arguments	**Description**
group_ptr	Pointer to a an event flags group
flat_to_set	Event flags to set or clear, based on the set option selected
set_option	Specifies whether the event flags specified are ANDed or ORed into the current event flags of the group: ■ TX_AND (0x02) The event flags are ANDed into the current event flags in the group. This option is typically used to clear event flags in a group. ■ TX_OR (0x00) The event flags are ORed with the current event in the group.

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_GROUP_ERROR (0x06)	Invalid event group pointer
TX_OPTION_ERROR (0x03)	Invalid get option specified

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

```
TX_EVENT_FLAGS_GROUP my_event_flags_group; UINT status;

          /* Set event flags 0, 4, and 8. */

status = tx_event_flags_set(&my_event_flags_group,
          0x111, TX_OR);

/* If status equals TX_SUCCESS, the event flags have been set and any suspended thread whose

request was satisfied has been resumed. */
```

See also

tx_event_flags_create, tx_event_flags_delete, tx_event_flags_get,

tx_event_flags_info_get

tx_interrupt_control

Enables or disables specified interrupts.

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, because doing so could cause unpredictable results.

Format

UINT tx_interrupt_control (UINT *new_posture*)

Arguments

Arguments	Description
new_posture	Specifies whether interrupts are disabled or enabled. Valid values include TX_INT_DISABLE and TX_INT_ENABLE. The actual values for these parameters are port specific. In addition, some system may support addition interrupt disable postures.

Return Values

Return Values	Description
previous_posture	The previous interrupt posture to the caller. This allows for restoring the previous posture after interrupts are disabled.

Allowed from

Threads, timers, and ISRs

Preemption possible

No

Example

```
UINT my_old_posture;
/* Lockout interrupts */
my_old_posture =          tx_interrupt_control(TX_INT_DISABLE);
/* Perform critical operations that need interrupts locked-out */
/* Restore previous interrupt lockout posture. */
        tx_interrupt_control(my_old_posture);
```

tx_mutex_create

Creates a mutex for inter-thread mutual exclusion for resource protection.

Format

UINT tx_mutex_create (TX_MUTEX **mutex_ptr*,

CHAR **name_ptr*, UINT *priority_inherit*)

Arguments

Arguments	Description
mutex_ptr	Pointer to the mutex control block
name_ptr	Pointer to the mutex name
priority_inherit	Specifies whether or not this mutex supports priority inheritance. One of the following: ■ TX_INHERIT - priority inheritance is supported ■ TX_NO_INHERIT - priority inheritance is not supported by this mutex

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_MUTEX_ERROR (0x1C)	Invalid mutex pointer - either the pointer is NULL or the mutex already exists
TX_CALLER_ERROR (0x13)	Invalid caller of this service
TX_INHERIT_ERROR (0x1F)	Invalid priority inherit parameter

Allowed from

Threads, and ISRs

Preemption possible

No

Example

```
        TX_MUTEX my_mutex;
        UINT status;

        /* Create a mutex to provide protection over a common resource. */

        status =     tx_mutex_create(&my_mutex,.my_mutex_name., TX_NO_INHERIT);

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

See also

tx_mutex_delete, tx_mutex_get, tx_mutex_info_get,

tx_mutex_prioritize, tx_mutex_put

tx_mutex_delete

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.

Format

UINT tx_mutex_delete(TX_MUTEX **mutex_ptr*)

Arguments

Arguments	Description
mutex_ptr	Pointer to the mutex

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_MUTEX_ERROR (0x1C)	Invalid mutex pointer - either the pointer is NULL or the mutex already exists
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads

Preemption possible

Yes

Example

```
        TX_MUTEX my_mutex;
        UINT status;
        /* Delete a mutex. Assume that the mutex has already been created. */
        status =     tx_mutex_delete(&my_mutex);
/* If status equals TX_SUCCESS, the mutex is deleted. */
```

See also

tx_mutex_create, tx_mutex_get, tx_mutex_info_get,

tx_mutex_prioritize, tx_mutex_put

tx_mutex_get

Tries 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.

Format

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

Arguments

Arguments	**Description**
mutex_ptr	Pointer to the mutex
wait_option	Defines how the service behaves if the mutex is already owned by another thread. The options are as follows: ■ TX_NO_WAIT (0x00000000) Results in an immediate return from this service regardless of whether it was successful or not. This is the only valid option if the service is called from initialization. ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until the mutex is available ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for the mutex.

Usage notes

If the mutex is owned by another thread having higher priority, and if 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.

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_DELETED (0x01)	Mutex was deleted while thread was suspended
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_NOT_AVAILABLE (0x1D)	Unable to get ownership of the mutex
TX_MUTEX_ERROR (0x1C)	Invalid mutex pointer - either the pointer is NULL or the mutex already exists
TX_WAIT_ERROR (0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread

Allowed from

Initialization and threads

Preemption possible

Yes

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Obtain exclusive ownership of the mutex "my_mutex..
If the .my_mutex. is not available, suspend until it becomes available. */

status =    tx_mutex_get(&my_mutex, TX_WAIT_FOREVER);
```

See also

tx_mutex_create, tx_mutex_delete, tx_mutex_info_get,

tx_mutex_prioritize, tx_mutex_put

tx_mutex_info_get

Retrieves information from the specified mutex.

Format

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*)

Arguments

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

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_MUTEX_ERROR (0x1C)	Invalid mutex pointer - either the pointer is NULL or the mutex already exists
TX_PTR_ERROR (0x13)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_MUTEX my_mutex;
CHAR *name;
ULONG count;
TX_THREAD *owner;
TX_THREAD *first_suspended;
```

```
        ULONG suspended_count;
        TX_MUTEX *next_mutex;
        UINT status;

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

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

See also
tx_mutex_create, tx_mutex_delete, tx_mutex_get,

tx_mutex_prioritize, tx_mutex_put

tx_mutex_prioritize

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.

Format

UINT tx_mutex_prioritize(TX_MUTEX **mutex_ptr*)

Arguments

Arguments	**Description**
mutex_ptr	Pointer to the mutex

Return Values

Return Values	**Description**
TX_SUCCESS (0x00)	Success
TX_MUTEX_ERROR (0x1C)	Invalid mutex pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Ensure that the highest priority thread will receive ownership of the mutex when it becomes avail-
able. */

status = tx_mutex_prioritize(&my_mutex);

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

See also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_info_get,

tx_mutex_put

tx_mutex_put

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.

Format

UINT tx_mutex_put(TX_MUTEX *mutex_ptr*)

Arguments

Arguments	Description
mutex_ptr	Pointer to the mutex

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_MUTEX_ERROR (0x1C)	Invalid mutex pointer
TX_NOT_OWNED (0x1E)	Mutex is not allowed by the caller

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_MUTEX my_mutex;
UINT status;

/* Release ownership of .my_mutex.. */

status = tx_mutex_put(&my_mutex);

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

See also

tx_mutex_create, tx_mutex_delete, tx_mutex_get, tx_mutex_info_get,

tx_mutex_prioritize

tx_queue_create

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.

Format

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

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue
name_ptr	Pointer to the message queue name
message_size	Size of the messages in the queue. Sizes range from 1 to 16 words (where a word is 32 bits). Message size options are: ■ TX_1_ULONG (0x01) ■ TX_2_ULONG (0x02) ■ TX_4_ULONG (0x04) ■ TX_8_ULONG (0x08) ■ TX_16_ULONG (0x10)
queue_start	Pointer to the starting address of the message queue
queue_size	Total number of bytes available for the message queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_PTR_ERROR (0x03)	Invalid pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service
TX_SIZE_ERROR (0x05)	Size of pool is invalid

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 mes-
sage in this queue is defined to be 4 32-bit words long. */

status = tx_queue_create(&my_queue, .my_queue_name.,
                 TX_4_ULONG, (VOID *) 0x300000, 2000);

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

See also

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

tx_queue_delete

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.

Format

UINT tx_queue_delete(TX_QUEUE *queue_ptr)

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads

Preemption possible

Yes

Example

```
        TX_QUEUE my_queue;
        UINT status;

/* Delete entire message queue. Assume that the queue has already been created with a call to
tx_queue_create. */

        status = tx_queue_delete(&my_queue);

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

See also

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

tx_queue_flush

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.

Format

UINT tx_queue_flush(TX_QUEUE *queue_ptr*)

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

Yes

Example

```
TX_QUEUE my_queue;
UINT status;

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

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

See also

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

tx_queue_front_send

Sends a message to the front location of the specified message queue. The message sent is copied to the front of the queue from the memory area specified by the source pointer.

Format

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

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue
source_ptr	Pointer to the message
wait_option	Defines how the service behaves if the message queue is full. The options are as follows: ■ TX_NO_WAIT (0x00000000) Results in an immediate return from this service regardless of whether it was successful or not. This is the only valid option if the service is called from a nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until there is room in the queue ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_DELETED (0X01)	Invalid caller of this service
TX_QUEUE_FULL(0x0B)	Unable to send message because the queue was full
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_QUEUE_ERROR (0x09)	Invalid message queue pointer
TX_PTR_ERROR (0x03)	Invalid source pointer for message
TX_WAIT_ERROR(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

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

```
/* Send a message to the front of .my_queue.. Return immediately, regardless of success. This wait
option is used for calls from initialization, timers, and ISRs. */

status = tx_queue_front_send(&my_queue, my_message,
         TX_NO_WAIT);

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

See also

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

tx_queue_info_get

Retrieves information about the specified message queue.

Format

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

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue
name	Pointer to destination for the pointer to the queue name
enqueued	Pointer to destination for the number of messages currently in the queue
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this queue
suspended_count	Pointer to destination for the number of threads currently suspended on this queue
next_queue	Pointer to destination for the pointer of the next created queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

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

/* Retrieve information about a the previously created message queue "my_queue." */

status = tx_queue_info_get(&my_queue, &name,
            &enqueued,
            &first_suspended, &suspended_count,
            &next_queue);

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

See also

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

tx_queue_prioritize

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.

Format

UINT tx_queue_prioritize(TX_QUEUE **queue_ptr*)

UINT tx_queue_flush(TX_QUEUE **queue_ptr*)

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
        TX_QUEUE my_queue;
        UINT status;

/* Ensure that the highest priority thread will receive the next message placed on this queue. */

status = tx_queue_prioritize(&my_queue);

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

See also

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

tx_queue_receive

Retrieves a message from the specified message queue. The message retrieved is copied from the queue into the memory area specified by the destination pointer.

The specified destination memory area must be large enough to hold the message.

Otherwise, if the destination is not large enough, memory corruption occurs in the following memory area.

Format

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

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue
destination_ptr	Pointer to the location of where to copy the message
wait_option	Defines how the service behaves if the queue is empty. The options are as follows: ■ TX_NO_WAIT (0x00000000) Results in an immediate return from this service regardless of whether it was successful or not. This is the only valid option if the service is called from a nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until a message is available ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for a message

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_DELETED (0X01)	Message queue was deleted while thread was suspended
TX_QUEUE_EMPTY (0x0A)	Unable to retrieve a message because the queue was empty
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_QUEUE_ERROR (0x09)	Invalid message queue pointer
TX_PTR_ERROR (0x03)	Invalid source pointer for message
TX_WAIT_ERROR(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

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

/* Retrieve a message from .my_queue.. If the queue is empty, suspend until a message is present.
Note that this suspension is only possible from application threads. */

status = tx_queue_receive(&my_queue, my_message,
         TX_WAIT_FOREVER);

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

See also

tx_queue_create, tx_queue_delete, tx_queue_flush,

tx_queue_front_send, tx_queue_info_get, tx_queue_prioritize,

tx_queue_send

tx_queue_send

Sends a message to the specified message queue. The message sent is copied to the queue from the memory area specified by the source pointer.

Format

UINT tx_queue_send(TX_QUEUE **queue_ptr*,

VOID **source_ptr*, ULONG *wait_option*)

Arguments

Arguments	Description
queue_ptr	Pointer to the message queue
source_ptr	Pointer to the message
wait_option	Defines how the service behaves if the queue is full. The options are as follows: ■ TX_NO_WAIT (0x00000000) Results in an immediate return from this service regardless of whether it was successful or not. This is the only valid option if the service is called from a nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until there is room in the queue ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_QUEUE_ERROR (0x01)	Invalid message queue pointer
TX_DELETED (0X01)	Message queue was deleted while thread was suspended
TX_QUEUE_FULL (0x0B)	Unable to send a message because the queue was full
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_QUEUE_ERROR (0x09)	Invalid message queue pointer
TX_PTR_ERROR (0x103)	Invalid source pointer for message
TX_WAIT_ERROR (0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

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

            /* Send a message to .my_queue.. Return immediately, regardless of success. This
            wait option is used for calls from initialization, timers, and ISRs. */

status =    tx_queue_send(&my_queue, my_message, TX_NO_WAIT);

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

See also

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

tx_semaphore_create

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

Format

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

Arguments

Arguments	**Description**
semaphore_ptr	Pointer to the semaphore
name_ptr	Pointer to the semaphore name
initial_count	The initial count for this semaphore. Values range from 0x00000000 through 0xFFFFFFFF.

Return Values

Return Values	**Description**
TX_SUCCESS (0x00)	Success
TX_SEMAPHORE_ERROR (0x0C)	Invalid semaphore pointer - either the pointer is NULL or the semaphore already exists
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

No

Example

```
        TX_SEMAPHORE my_semaphore;
        UINT status;

/* Create a counting semaphore whose initial value is 1. This is typically the technique used to make a
binary semaphore. Binary semaphores are used to provide protection over a common resource. */

        status =    tx_semaphore_create(&my_semaphore,
                "my_semaphore_name", 1);

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

See also
tx_semaphore_delete, tx_semaphore_get, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_delete

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.

Format

UINT tx_semaphore_delete(TX_SEMAPHORE **semaphore_ptr*)

UINT tx_queue_flush(TX_QUEUE **queue_ptr*)

Arguments

Arguments	Description
semaphore_ptr	Pointer to the semaphore

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_SEMAPHORE_ERROR (0x0C)	Invalid semaphore counting 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 is TX_SUCCESS, the counting semaphore is deleted */
```

See also

tx_semaphore_create, tx_semaphore_get, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_get

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

Format

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

Arguments

Arguments	Description
semaphore_ptr	Pointer to the semaphore
wait_option	Defines how the service behaves if there are no instances of the semaphore available (that is, the semaphore count is zero). The options are as follows: ■ TX_NO_WAIT (0x00000000) Results in an immediate return from this service regardless of whether it was successful or not. This is the only valid option if the service is called from a nonthread (initialization, timer, or ISR). ■ TX_WAIT_FOREVER (0xFFFFFFFF) Causes the calling thread to suspend indefinitely until a semaphore instance is available ■ *timeout_value* (0x00000001-0xFFFFFFFE) Specifies the maximum number of timer-ticks to stay suspended while waiting for a sermaphore instance

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_SEMAPHORE_ERROR (0x0C)	Invalid semaphore pointer - either the pointer is NULL or the semaphore already exists
TX_WAIT_ERROR (0x04))	A wait option other than TX_NO_WAIT was specified on a call from a non-thread
TX_DELETED (0x01)	Counting semaphore was deleted while thread was suspended
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_NO_INSTANCE (0xOD)	Unable to retrieve an instance of the counting semaphore (count is zero)

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

```
        TX_SEMAPHORE my_semaphore;
        UINT status;

        /* Get a semaphore instance from the semaphore "my_semaphore." If the sema
                phore 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 sema
        phore. */
```

See also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_info_get

Retrieves information about the specified semaphore.

Format

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)*

Arguments

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

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_SEMAPHORE_ERROR (0x0C)	Invalid semaphore counting pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_SEMAPHORE my_semaphore;
CHAR *name;
ULONG current_value;
```

```
        TX_THREAD *first_suspended;
        ULONG suspended_count;
        TX_SEMAPHORE *next_semaphore;
        UINT status;

/* Retrieve information about a the previously created semaphore .my_semaphore.. */

        status =    tx_semaphore_info_get(&my_semaphore, &name,
                    &current_value,
                    &first_suspended, &suspended_count,
                    &next_semaphore);

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

See also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_get,

tx_semaphore_prioritize, tx_semaphore_put

tx_semaphore_prioritize

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.

Format

UINT tx_semaphore_prioritize(TX_SEMAPHORE **semaphore_ptr*)

Arguments

Arguments	Description
semaphore_ptr	Pointer to the semaphore

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_SEMAPHORE_ERROR (0x0C)	Invalid semaphore counting 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 sema
phore. */

status =    tx_semaphore_prioritize(&my_semaphore);

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

See also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_get,

tx_semaphore_info_get, tx_semaphore_put

tx_semaphore_put

Puts an instance into the specified counting semaphore, which increments the counting semaphore by one.

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

Format

UINT tx_semaphore_put(TX_SEMAPHORE **semaphore_ptr*)

Arguments

Arguments	**Description**
semaphore_ptr	Pointer to the counting semaphore

Return Values

Return Values	**Description**
TX_SUCCESS (0x00)	Success
TX_SEMAPHORE_ERROR (0x0C)	Invalid semaphore counting pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

```
TX_SEMAPHORE my_semaphore;
UINT status;

/* Increment the counting semaphore .my_semaphore.. */

status = tx_semaphore_put(&my_semaphore);
```

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

See also

tx_semaphore_create, tx_semaphore_delete, tx_semaphore_info_get, tx_semaphore_prioritize, tx_semaphore_get

tx_thread_create

Creates an application thread that starts execution at the specified task entry function. The stack, priority, preemption, and time-slice are among the attributes specified.

In addition, the initial execution state of the thread is also specified.

Format

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*)

Arguments

Arguments	Description
thread_ptr	Pointer to the application thread control
name_ptr	Pointer to the name of the thread
entry_function	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	Pointer to the 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. Threads must have at least TX_MINIMUM_STACK bytes to execute.
priority	Numerical priority of thread. Values range from 0 through 31, where a value of 0 represents the highest priority.
preempt_ threshold	Highest priority level (0-31) of disabled preemption. Only priorities higher than this level are allowed to preempt this thread. This value must be less than or equal to the specified priority. Typically, it is the same as the priority.
time_slice	Number of timer-ticks this thread is allowed to execute without checking to see if there are any other threads of the same priority ready to execute. Ready threads with priorities equal to or less than the preemption threshold are also given a chance to execute when a time-slice occurs. Legal timeslices selections range from 1 through 0xFFFFFFFF. A value of TX_NO_TIME_SLICE (a value of 0) disables time-slicing of this thread.
auto_start	Specifies whether the thread starts immediately or stays in a pure suspended state: ■ TX_AUTO_START (0x01) ■ TX_DONT_START (0x00)

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0xOE)	Invalid thread control pointer - either the pointer is NULL or the thread already exists
TX_THREAD_ERROR (0xOE)	Invalid threat priority
TX_THREAD_ERROR (0xOE)	Invalid starting address of the entry point, or the stack area is invalid (typically, NULL)
TX_THREAD_ERROR (0xOE)	Invalid preemption threshold specified
TX_THREAD_ERROR (0xOE)	Invalid auto-start selection
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Return Values	Description
TX_SIZE_ERROR (0x05)	Size of stack area is invalid

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 preemp
        tion threshold is set up to allow preemption at priorities above 15. Time-slicing is
        disabled. This thread is automatically put into a ready condition. */
        status =    tx_thread_create(&my_thread, "my_thread_name",
                    my_thread_entry, 0x1234,
                    (VOID *) 0x400000, 1000,
                    15, 15, TX_NO_TIME_SLICE,
                    TX_AUTO_START);

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

/* Thread.s entry function. When .my_thread. actually begins execution, control is transferred to this
function. */
VOID my_thread_entry (ULONG initial_input)
{
        /* When we get here, the value of initial_input is 0x1234. See how this was specified during
creation. */

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

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

See also

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

tx_thread_delete

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.

Format

UINT tx_thread_delete(TX_THREAD **thread_ptr*)

Arguments

Arguments	Description
thread_ptr	Pointer to the application thread

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0xOE)	Invalid application thread pointer
TX_DELETE_ERROR (0x0F)	Specified thread is not in a terminated or completed state
TX_PTR_ERROR (0x03)	Invalid starting address of the entry point, or the stack area is invalid (typically, NULL)
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads and timers

Preemption possible

No

Example

```
TX_THREAD my_thread;
UINT status;

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

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

See also

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

tx_thread_identify

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.

Format

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

Arguments

None

Return Values

Return Values	**Description**
thread_pointer	Pointer to the currently executing thread. If no thread is executing, the return value is TX_NULL.

Allowed from

Threads and ISRs

Preemption possible

No

Example

```
TX_THREAD *my_thread_ptr;

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

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

See also

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

tx_thread_info_get

Retrieves information about the specified thread.

Format

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

Arguments

Arguments	Description
thread_ptr	Pointer to the thread control block
name	Pointer to destination for the pointer to the thread name
state	Pointer to destination for the thread.s current execution state: ■ TX_READY (0x00) ■ TX_COMPLETED (0x01) ■ TX_TERMINATED (0x02) ■ TX_SUSPENDED (0x03) ■ TX_SLEEP (0x04) ■ TX_QUEUE_SUSP (0x05) ■ TX_SEMAPHORE_SUSP (0x06) ■ TX_EVENT_FLAG (0x07) ■ TX_BLOCK_MEMORY (0x08) ■ TX_BYTE_MEMORY (0x09) ■ TX_MUTEX_SUSP (0x0D) ■ TX_IO_DRIVER (0x0A)
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 the next created thread pointer
suspended_thread	Pointer to destination for pointer to the next thread in the suspension list

Return values

Return values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0xOE)	Invalid thread control pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

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

/* Retrieve information about a the previously created thread "my_thread." */

status = tx_thread_info_get(&my_thread, &name,
        &state, &priority, &preemption_threshold,
        &time_slice, &next_thread,&suspended_thread);

/* If status is TX_SUCCESS, information requested is valid */
```

See also

tx_thread_create, tx_thread_delete, tx_thread_identify,

tx_thread_preemption_change, tx_thread_priority_change,

tx_thread_relinquish, tx_thread_resume, tx_thread_sleep,

tx_thread_suspend, tx_thread_terminate,

tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_preemption_change

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.

Format

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

Arguments

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

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0xOE)	Invalid application thread pointer
TX_PTR_ERROR (0x03)	Invalid pointer to previous preemption threshold storage location
TX_THRESH_ERROR (0x18)	Invalid preemption threshold specified
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads and timers

Preemption possible

Yes

Example

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

/* Disable all preemption of the specified thread. The current preemption threshold
is returned in "my_old_threshold". Assume that "my_thread" has already been cre
ated. */

status =     tx_thread_preemption_change(&my_thread,
                    0, &my_old_threshold);

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

See also

tx_thread_create, tx_thread_delete, tx_thread_identify,

tx_thread_info_get, tx_thread_priority_change,

tx_thread_relinquish, tx_thread_resume, tx_thread_sleep,

tx_thread_suspend, tx_thread_terminate,

tx_thread_time_slice_change, tx_thread_wait_abort

tx_thread_priority_change

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

The preemption threshold of the specified thread is automatically set to the new priority. If a new threshold is desired, you must use tx_thread_preemption_change after this call.

Format

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

Arguments

Arguments	Description
thread_ptr	Pointer to the application thread
new_priority	New thread priority level (0-31)
old_priority	Pointer to a location to return the previous thread priority

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0xOE)	Invalid application thread pointer
TX_PTR_ERROR (0x03)	Invalid pointer to previous preemption threshold storage location
TX_THRESH_ERROR (0x18)	Invalid preemption threshold specified
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads and timers

Preemption possible

Yes

Example

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

/* Change the thread represented by "my_thread" to priority 0. */
```

```
status =    tx_thread_priority_change(&my_thread,
                    0, &my_old_priority);

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

See also

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

tx_thread_relinquish

Relinquishes processor control to other ready-to-run threads at the same or higher priority.

Format

VOID tx_thread_relinquish(VOID)

Arguments

VOID

Return values

VOID

Allowed from

Only the executing thread

Preemption possible

Yes

Example

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

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

VOID my_first_thread(ULONG thread_input)
{

        /* Endless loop of relinquish. */
        while(1)
        {
            /* Increment the run counter. */
            run_counter_1++;
            /* Relinquish control to other thread. */
            tx_thread_relinquish();
        }
    }
    VOID my_second_thread(ULONG thread_input)
    {
        /* Endless loop of relinquish. */
        while(1)
        {
            /* Increment the run counter. */
            run_counter_2++;
            /* Relinquish control to other thread. */
            tx_thread_relinquish();
        }
    }
```

See also

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

tx_thread_resume

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.

Format

UINT tx_thread_resume(TX_THREAD **thread_ptr*)

Arguments

Argument	Description
thread_ptr	Pointer to a suspended application thread

Return value

Return value	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0xOE)	Invalid application thread pointer
TX_SUSPEND_LIFTED (0x19)	Previously set delayed suspension was lifted
TX_RESUME_ERROR (0x12)	Specified thread is not suspended or was previously suspended by a service other than TX_THREAD_SUSPEND

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

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

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

See also

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

tx_thread_sleep

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

Format

UINT tx_thread_sleep(ULONG *timer_ticks*)

Arguments

Arguments	Description
timer_ticks	The number of ticks to suspend the calling application thread. Values range from 1 through 0xFFFFFFFF. If you specify 0 (zero), the service returns immediately.

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_WAIT_ABORTED (0x1A)	Suspension was aborted by another thread, timer, or ISR
TX_CALLER_ERROR (0x13)	Invalid caller of this service (called from a non-thread)

Allowed from

Currently executing thread

Preemption possible

Yes

Example

```
UINT status;

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

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

See also

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

tx_thread_suspend

Suspends the specified application thread. A thread can suspend itself by calling this service.

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

Once suspended, the thread must be resumed by tx_thread_resume for the thread to execute again.

Format

UINT tx_thread_suspend(TX_THREAD **thread_ptr*)

Arguments

Arguments	Description
thread_ptr	Pointer to the application thread

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_SUSPEND_ERROR (0x00)	Specified thread is in a terminated or completed state
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads and timers

Preemption possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

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

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

See also

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

tx_thread_terminate

Terminates the specified application thread regardless if the thread is suspended or not. A thread may call this service to terminate itself.

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

Format

UINT tx_thread_terminate(TX_THREAD **thread_ptr*)

Arguments

Arguments	**Description**
thread_ptr	Pointer to the application thread

Return Values

Return Values	**Description**
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0x0E)	Invalid application thread pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads and timers

Preemption possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

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

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

See also

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

tx_thread_time_slice_change

Changes 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 the other threads of the same or higher priorities have a chance to execute.

Format

UINT tx_thread_time_slice_change(TX_THREAD **thread_ptr*,

ULONG *new_time_slice*, ULONG **old_time_slice*)

Arguments

Arguments	Description
thread_ptr	Pointer to the application thread
new_time_slice	New time-slice value. 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

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0x0E)	Invalid application thread pointer
TX_PTR_ERROR (0x03)	Invalid pointer to previous time-slice storage location
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads and timers

Preemption possible

No

Example

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

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

status =     tx_thread_time_slice_change(&my_thread, 20, &my_old_time_slice);

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

See also

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

tx_thread_wait_abort

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

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

Format

UIN tx_thread_wait_abort(TX_THREAD **thread_ptr*)

Arguments

Arguments	**Description**
thread_ptr	Pointer to the application thread

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_THREAD_ERROR (0x0E)	Invalid application thread pointer
TX_WAIT_ABORT_ERROR (0x0E)	Specified thread is not in a waiting state
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

Yes

Example

```
TX_THREAD my_thread;
UINT status;

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

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

See also

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

tx_time_get

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 tx_time_set.

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

Format

```
ULONG tx_time_get(VOID)
```

Arguments

None

Return Values

Return Values	Description
system clock ticks	Value of the internal, free-running system clock

Allowed from

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 has copy of the internal system clock */
```

See also

tx_time_set

tx_time_set

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.

Format

VOID tx_time_set(ULONG *new_time*)

Arguments

Arguments	Description
new_time	New time to put in the system clock. Values range from 0 through 0xFFFFFFFF.

Return Values

None

Allowed from

Threads, timers, and ISRs

Preemption possible

No

Example

```
/* Set the internal system time to 0x1234 */

tx_time_set(0x1234);

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

See also

tx_time_get

tx_timer_activate

Activates the specified application timer. The expiration routines of timers that expire at the same time are executed in the order they were activated. If the timer is already activated, this service has no effect.

Format

UINT tx_timer_activate(TX_TIMER **timer_ptr*)

Arguments

Arguments	**Description**
timer_ptr	Pointer to the application timer

Return Values

Return Values	**Description**
TX_SUCCESS (0x00)	Success
TX_TIMER_ERROR (0x15)	Invalid application timer pointer
TX_ACTIVATE_ERROR (0x17)	Timer was already active

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_TIMER my_timer;
UINT status;

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

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

See also

tx_timer_change, tx_timer_create, tx_timer_deactivate, tx_timer_delete, tx_timer_info_get

tx_timer_change

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

After calling this server, you must call tx_timer_activate to start the timer again.

Format

UINT tx_timer_change(TX_TIMER **timer_ptr*,

ULONG *initial_ticks*, ULONG *reschedule_ticks*)

Arguments

Arguments	Description
timer_ptr	Pointer to a timer control block
initial_ticks	The initial number of ticks for timer expiration. Values range from 1 through 0xFFFFFFFF.
reschedule_ticks	Specifies the number of ticks for all timer expirations after the first. Specifying 0 (zero) makes the timer a *one-shot* timer. Otherwise, for periodic timers, values range from 1 through 0xFFFFFFFF.

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_TIMER_ERROR (0x15)	Invalid application timer pointer
TX_TICK_ERROR (0x16)	Invalid value (such as zero) specified for initial ticks
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads, timers, and ISRs

Preemption possible

No

Example

```
TX_TIMER my_timer;
UINT status;

* Change a now deactivated timer to expire every 50 timer ticks, including the initial
expiration. */
```

status = **tx_timer_change**(&my_timer,50, 50);

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

/* Activate the specified timer to get it started again. */

status = tx_timer_activate(&my_timer);
```

See also

tx_timer_activate, tx_timer_create, tx_timer_deactivate,
tx_timer_delete, tx_timer_info_get

tx_timer_create

Creates an application timer with the specified expiration function and period.

Format

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*)

Arguments

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

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_TIMER_ERROR (0x15)	Invalid application timer pointer
TX_TICK_ERROR (0x16)	Invalid value (such as zero) specified for initial ticks
TX_ACTIVATE_ERROR (0x17)	Invalid activation selected
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Initialization and threads

Preemption possible

No

Example

```
TX_TIMER my_timer;
UINT status;

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

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

See also

tx_timer_activate, tx_timer_change, tx_timer_deactivate,
tx_timer_delete, tx_timer_info_get

tx_timer_deactivate

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

Format

UINT tx_timer_deactivate(TX_TIMER **timer_ptr*)

Arguments

Arguments	Description
timer_ptr	Pointer to the application timer

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_TIMER_ERROR (0x15)	Invalid application timer pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

```
TX_TIMER my_timer;
UINT status;

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

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

See also

tx_timer_activate, tx_timer_change, tx_timer_create,
tx_timer_delete, tx_timer_info_get

tx_timer_delete

Deletes the specified application timer.

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

Format

UINT tx_timer_delete(TX_TIMER **timer_ptr*)

UINT tx_timer_deactivate(TX_TIMER **timer_ptr*)

Arguments

Arguments	Description
timer_ptr	Pointer to the application timer

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_TIMER_ERROR (0x15)	Invalid application timer pointer
TX_CALLER_ERROR (0x13)	Invalid caller of this service

Allowed from

Threads

Preemption possible

No

Example

```
        TX_TIMER my_timer;
        UINT status;

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

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

See also

tx_timer_activate, tx_timer_change, tx_timer_create, tx_timer_deactivate, tx_timer_info_get

tx_timer_info_get

Retrieves information about the specified application timer.

Format

UINT tx_timer_info_get(TX_TIMER **timer_ptr*, CHAR ***name*,
UINT **active*, ULONG **remaining_ticks*,

ULONG **reschedule_ticks*, TX_TIMER ***next_timer*)

Arguments

Arguments	Description
timer_ptr	Pointer to the application timer
name	Pointer to destination for the pointer to the timer.s name.
active	Pointer to destination for the timer active indication. If this value is TX_TRUE, the timer is active.
remaining_ticks	Pointer to destination for the number of timer ticks left before the timer expires
reschedule_ticks	Pointer to destination for the number of timer ticks that will be used to automatically reschedule this timer. If the value is zero, then the timer is a one-shot and won't be rescheduled.
next_timer	Destination for the pointer of the next created application timer

Return Values

Return Values	Description
TX_SUCCESS (0x00)	Success
TX_TIMER_ERROR (0x15)	Invalid application timer pointer
TX_PTR_ERROR (0x03)	Invalid pointer (NULL) for any destination pointer

Allowed from

Initialization, threads, timers, and ISRs

Preemption possible

No

Example

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

/* Retrieve information about a the previously created application timer .my_timer..
*/

status =    tx_timer_info_get(&my_timer, &name,
            &active,&remaining_ticks,
            &reschedule_ticks,
            &next_timer);

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

See also

tx_timer_activate, tx_timer_change, tx_timer_create,
tx_timer_deactivate, tx_timer_delete, tx_timer_info_get

NET+OS Kernel Design Goals

C H A P T E R 4

This chapter describes how the design of the NET+OS kernel contributes to its performance.

Principal design goals

The NET+OS kernel has three principal design goals: simplicity, scalability, and high performance. In many situations these goals are complementary - that is, simpler, smaller software usually gives better performance.

Simplicity

Simplicity is the most important design goal of the NET+OS kernel. Simplicity makes the NET+OS kernel very easy to use, test, and verify. In addition, it makes it easy for developers to understand exactly what is happening inside.

Scalability

The NET+OS kernel is also designed to be very scalable. Its instruction area size ranges from 2Kb to 15Kb, depending on the services actually used by the application. This enables the NET+OS kernel to support a wide range of microprocessor architectures, ranging from small micro-controllers through high performance RISC and DSP processors.

How is the NET+OS kernel so scalable? First, the NET+OS kernel is designed with a software component methodology, which allows automatic removal of whole components that are not used. Second, it places each function in a separate file to minimize each function.s interaction with the rest of the system. Because the NET+OS kernel is implemented as a C library, only the functions that are used become part of the final embedded image.

High performance

The NET+OS kernel is designed for high performance. This is achieved in a variety of ways, including algorithm optimizations, register variables, in-line assembly language, low-overhead timer interrupt handling, and optimized context switching. In addition, applications have the ability (with the conditional compilation flag TX_DISABLE_ERROR_CHECKING) to disable the basic error checking facilities of the NET+OS kernel API. This feature is very useful in the tuning phase of application development. By disabling basic error checking, a 30 percent performance boost can be achieved on most NET+OS kernel implementations. And, of course, the resulting code image is also smaller!

NET+OS kernel ANSI C library

As mentioned before, the NET+OS kernel is implemented as a C library, which must be linked with the application software. The NET+OS kernel library consists of several object files that are derived from various C source files and processor specific
assembly language files. There are also several C include files used in the C file compilation process. All of the C source and include files conform completely to the ANSI standard.

Include files

NET+OS kernel applications need access to two include files: tx_api.h and tx_port.h.

The NET+OS kernel source package also contains system include files which represent the internal component specification files.

tx_api.h file

The tx_api.h file contains all the constants, function prototypes, and object data structures. This file is the same for all processor support packages.

The mapping of the NET+OS kernel API services to the underlying error checking or core processing functions is done in tx_api.h.

tx_port.h file

The tx_port.h file, which is included by tx_api.h, contains processor- and development tool-specific information, including data type assignments and interrupt management macros used throughout the NET+OS kernel source code.
The tx_port.h file also contains the NET+OS kernel port-specific ASCII version string, _tx_version_id.

Programming Reference Information

C H A P T E R 5

This chapter describes how the design of the NET+OS kernel contributes to its performance.

NET+OS kernel constants

Listed alphabetically

TX_1_ULONG	1
TX_2_ULONG	2
TX_4_ULONG	4
TX_8_ULONG	8
TX_16_ULONG	16
TX_ACTIVATE_ERROR	0x0017
TX_AND	2
TX_AUTO_ACTIVATE	1
TX_AND_CLEAR	3
TX_AUTO_START	1
TX_BLOCK_MEMORY	8
TX_BYTE_MEMORY	9
TX_CALLER_ERROR	0x0013
TX_COMPLETED	1
TX_DELETE_ERROR	0x0011
TX_DELETED	0x0001
TX_DONT_START	0
TX_EVENT_FLAG	7
TX_FALSE	0
TX_FILE	11
TX_FOREVER	1
TX_GROUP_ERROR	0x0006
TX_INHERIT	1
TX_INHERIT_ERROR	0x001F
TX_IO_DRIVER	10

TX_MAX_PRIORITIES	32
TX_MUTEX_ERROR	0x001F
TX_MUTEX_SUSP	13
TX_NO_ACTIVATE	0
TX_NO_EVENTS	0x0007
TX_NO_INHERIT	0
TX_NO_INSTANCE	0x000D
TX_NO_MEMORY	0x0010
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NOT_AVAILABLE	0x001D
TX_NOT_OWNED	0x001E
TX_NULL	0
TX_OPTION_ERROR	0x0008
TX_OR	0
TX_OR_CLEAR	1
TX_POOL_ERROR	0x0002
TX_PRIORITY_ERROR	0x000F
TX_PTR_ERROR	0x0003
TX_QUEUE_EMPTY	0x000A
TX_QUEUE_ERROR	0x0009
TX_QUEUE_FULL	0x000B
TX_QUEUE_SUSP	5
TX_READY	0
TX_RESUME_ERROR	0x0012
TX_SEMAPHORE_ERROR	0x000C
TX_SEMAPHORE_SUSP	6
TX_SIZE_ERROR	0x0005
TX_SLEEP	4
TX_START_ERROR	0x0010
TX_SUCCESS	0x0000
TX_SUSPEND_ERROR	0x0014
TX_SUSPEND_LIFTED	0x0019
TX_SUSPENDED	3

TX_TCP_IP	12
TX_TERMINATED	2
TX_THREAD_ERROR	0x000E
TX_THRESH_ERROR	0x0018
TX_TICK_ERROR	0x0016
TX_TIMER_ERROR	0x0015
TX_TRUE	1
TX_WAIT_ABORT_ERROR	0x001B
TX_WAIT_ABORTED	0x001A
TX_WAIT_ERROR	0x0004
TX_WAIT_FOREVER	FFFFFF

Listed by value

TX_DONT_START	0
TX_FALSE	0
TX_NO_ACTIVATE	0
TX_NO_INHERIT	0
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NULL	0
TX_OR	0
TX_READY	0
TX_SUCCESS	0x0000
TX_1_ULONG	1
TX_AUTO_ACTIVATE	1
TX_AUTO_START	1
TX_COMPLETED	1
TX_FOREVER	1
TX_DELETED	0x0001
TX_INHERIT	1
TX_OR_CLEAR	1
TX_TRUE	1
TX_2_ULONG	2
TX_AND	2
TX_POOL_ERROR	0x0002
TX_TERMINATED	2
TX_AND_CLEAR	3
TX_PTR_ERROR	0x0003
TX_SUSPENDED	3
TX_4_ULONG	4
TX_SLEEP	4
TX_WAIT_ERROR	0x0004
TX_QUEUE_SUSP	5
TX_SIZE_ERROR	0x0005
TX_GROUP_ERROR	0x0006
TX_SEMAPHORE_SUSP	6
TX_EVENT_FLAG	7

TX_NO_EVENTS	0x0007
TX_8_ULONG	8
TX_BLOCK_MEMORY	8
TX_OPTION_ERROR	0x0008
TX_BYTE_MEMORY	9
TX_QUEUE_ERROR	0x0009
TX_IO_DRIVER	10
TX_QUEUE_EMPTY	0x000A
TX_FILE	11
TX_QUEUE_FULL	0x000B
TX_SEMAPHORE_ERROR	0x000C
TX_TCP_IP	12
TX_MUTEX_SUSP	13
TX_NO_INSTANCE	0x000D
TX_THREAD_ERROR	0x000E
TX_PRIORITY_ERROR	0x000F
TX_16_ULONG	6
TX_START_ERROR	0x0010
TX_NO_MEMORY	0x0010
TX_DELETE_ERROR	0x0011
TX_RESUME_ERROR	0x0012
TX_CALLER_ERROR	0x0013
TX_SUSPEND_ERROR	0x0014
TX_TIMER_ERROR	0x0015
TX_TICK_ERROR	0x0016
TX_ACTIVATE_ERROR	0x0017
TX_THRESH_ERROR	0x0018
TX_SUSPEND_LIFTED	0X0019
TX_WAIT_ABORTED	0x001A
TX_WAIT_ABORT_ERROR	0x001B
TX_MUTEX_ERROR	0x001C
TX_NOT_AVAILABLE	0x001D
TX_NOT_OWNED	0x001E
TX_INHERIT_ERROR	0x001F
TX_MAX_PRIORITIES	32
TX_WAIT_FOREVER	FFFFFFFF

NET+OS kernel data types

TX_INTERNAL_TIMER_STRUCT

```
typedef  struct TX_INTERNAL_TIMER_STRUCT
{
         ULONG                      tx_remaining_ticks;
         ULONG                      tx_re_initialize_ticks;
         VOID                       (*tx_timeout_function)(ULONG);
```

```
        ULONG                   tx_timeout_param;
        struct                  TX_INTERNAL_TIMER_STRUCT
                *tx_active_next,
                                *tx_active_previous;
        struct TX_INTERNAL_TIMER_STRUCT **tx_list_head;
} TX_INTERNAL_TIMER;
```

TX_TIMER_STRUCT

```
typedef struct TX_TIMER_STRUCT
{
        ULONG                   tx_timer_id;
        CHAR_PTR                tx_timer_name;
        TX_INTERNAL_TIMER       tx_timer_internal;
        struct TX_TIMER_STRUCT  *tx_timer_created_next,
                                *tx_timer_created_previous;
} TX_TIMER;
```

TX_QUEUE_STRUCT

```
typedef struct TX_QUEUE_STRUCT
{
        ULONG           tx_queue_id;
        CHAR_PTR        tx_queue_name;
        UINT            tx_queue_message_size;
        ULONG           tx_queue_capacity;
        ULONG           tx_queue_enqueued;
        ULONG           tx_queue_available_storage;
        ULONG_PTR       tx_queue_start;
        ULONG_PTR       tx_queue_end;
        ULONG_PTR       tx_queue_read;
        ULONG_PTR       tx_queue_write;
        struct TX_THREAD_STRUCT         *tx_queue_suspension_list;
        ULONG                           tx_queue_suspended_count;
        struct TX_QUEUE_STRUCT
                *tx_queue_created_next,
                *tx_queue_created_previous;
} TX_QUEUE;
```

TX_THREAD_STRUCT

```
typedef struct TX_THREAD_STRUCT
{
        ULONG           tx_thread_id;
        ULONG           tx_run_count;
        VOID_PTR        tx_stack_ptr;
        VOID_PTR        tx_stack_start;
        VOID_PTR        tx_stack_end;
        ULONG           tx_stack_size;
        ULONG           tx_time_slice;
        ULONG           tx_new_time_slice;
        struct TX_THREAD_STRUCT *tx_ready_next,
                *tx_ready_previous;
        TX_THREAD_PORT_EXTENSION /* See tx_port.h for details */
        CHAR_PTR                tx_thread_name;
        UINT            tx_priority;
        UINT            tx_state;
        UINT            tx_delayed_suspend;
        UINT            tx_suspending;
        UINT            tx_preempt_threshold;
        ULONG           tx_priority_bit;
        VOID            (*tx_thread_entry)(ULONG);
```

```
    ULONG           tx_entry_parameter;
    TX_INTERNAL_TIMER tx_thread_timer;
    VOID            (*tx_suspend_cleanup)
            (struct TX_THREAD_STRUCT *);
    VOID_PTR        tx_suspend_control_block;
    struct TX_THREAD_STRUCT *tx_suspended_next,
            *tx_suspended_previous;
    ULONG           tx_suspend_info;
    VOID_PTR        tx_additional_suspend_info;
    UINT            tx_suspend_option;
    UINT            tx_suspend_status;
    struct TX_THREAD_STRUCT *tx_created_next,
                            *tx_created_previous;
    VOID_PTR tx_filex_ptr;
    } TX_THREAD;
```

TX_SEMAPHORE_STRUCT

```
typedef struct TX_SEMAPHORE_STRUCT
{
    ULONG           tx_semaphore_id;
    CHAR_PTR        tx_semaphore_name;
    ULONG           tx_semaphore_count;
    struct TX_THREAD_STRUCT *tx_semaphore_suspension_list;
    ULONG           tx_semaphore_suspended_count;
    struct TX_SEMAPHORE_STRUCT *tx_semaphore_created_next,
                    *tx_semaphore_created_previous;
} TX_SEMAPHORE;
```

TX_EVENT_FLAGS_GROUP_STRUCT

```
typedef struct TX_EVENT_FLAGS_GROUP_STRUCT
{
    ULONG           tx_event_flags_id;
    CHAR_PTR        tx_event_flags_name;
    ULONG           tx_event_flags_current;
    UINT            tx_event_flags_reset_search;
    struct TX_THREAD_STRUCT         *tx_event_flags_suspension_list;
    ULONG                           tx_event_flags_suspended_count;
    struct TX_EVENT_FLAGS_GROUP_STRUCT
                    *tx_event_flags_created_next,
                    *tx_event_flags_created_previous;
} TX_EVENT_FLAGS_GROUP;
```

TX_BLOCK_POOL_STRUCT

```
typedef struct TX_BLOCK_POOL_STRUCT
{
        ULONG           tx_block_pool_id;
        CHAR_PTR        tx_block_pool_name;
        ULONG           tx_block_pool_available;
        ULONG           tx_block_pool_total;
        CHAR_PTR        tx_block_pool_available_list;
        CHAR_PTR        tx_block_pool_start;
        ULONG           tx_block_pool_size;
        ULONG           tx_block_pool_block_size;
        struct TX_THREAD_STRUCT*tx_block_pool_suspension_list;
        ULONG                   tx_block_pool_suspended_count;
        struct TX_BLOCK_POOL_STRUCT
                        *tx_block_pool_created_next,
                        *tx_block_pool_created_previous;
} TX_BLOCK_POOL;
```

TX_BYTE_POOL_STRUCT

```
typedef struct TX_BYTE_POOL_STRUCT
{
        ULONG           tx_byte_pool_id;
        CHAR_PTR        tx_byte_pool_name;
        ULONG           tx_byte_pool_available;
        ULONG           tx_byte_pool_fragments;
        CHAR_PTR        tx_byte_pool_list;
        CHAR_PTR        tx_byte_pool_search;
        CHAR_PTR        tx_byte_pool_start;
        ULONG           tx_byte_pool_size;
        struct TX_THREAD_STRUCT*tx_byte_pool_owner;
        struct TX_THREAD_STRUCT*tx_byte_pool_suspension_list;
        ULONG                   tx_byte_pool_suspended_count;
        struct TX_BYTE_POOL_STRUCT *tx_byte_pool_created_next,
                        *tx_byte_pool_created_previous;
} TX_BYTE_POOL;
```

TX_MUTEX_STRUCT

```
typedef struct TX_MUTEX_STRUCT
{
        ULONG           tx_mutex_id;
        CHAR_PTR        tx_mutex_name;
        ULONG           tx_mutex_ownership_count;
        TX_THREAD       *tx_mutex_owner;
        UINT            tx_mutex_inherit;
        UINT            tx_mutex_original_priority;
        UINT            tx_mutex_original_threshold;
        struct TX_THREAD_STRUCT
                *tx_mutex_suspension_list;
        ULONG tx_mutex_suspended_count;
        struct TX_MUTEX_STRUCT
                *tx_mutex_created_next,
        *tx_mutex_created_previous;
} TX_MUTEX;
```