SIM7070_SIM7080_SIM7090 Series_ThreadX API

LPWA Module

SIMCom Wireless Solutions Limited
SIMCom Headquarters Building, Building 3, No. 289 Linhong Road, Changning District, Shanghai P.R. China
Tel: 86-21-31575100
support@simcom.com
www.simcom.com

Document Title:	SIM7070_SIM7080_SIM7090 Series_ThreadX API
Version:	1.00
Date:	2021.09.18
Status:	Released

SIMCom Wireless Solutions Limited
SIMCom Headquarters Building, Building 3, No. 289 Linhong Road, Changning District, Shanghai P.R. China
Tel: +86 21 31575100
Email: simcom@simcom.com

For more information, please visit:
https://www.simcom.com/download/list-863-en.html

For technical support, or to report documentation errors, please visit:
https://www.simcom.com/ask/ or email to: support@simcom.com

About Document

Version History

Version	Date	Owner	What is new
V1.00	2021.09.18	Wenjie.lai	First Release

Scope

This document applies to the following products

Name	Type	Size(mm)	Comments
SIM7080G	CAT-M/NB	17.6*15.7*2.3	N/A
SIM7070G/SIM7070E	CAT-M/NB/GPRS	24*24*2.4	N/A
SIM7070G-NG	NB/GPRS	24*24*2.4	N/A
SIM7090G	CAT-M/NB	14.8*12.8*2.0	N/A

Contents

About Document 3
Version History 3
Scope 3

Contents 4

1 Introduction 7
1.1 Purpose of the document 7
1.2 Related documents 7
1.3 Conventions and abbreviations 7

2 ThreadX Data Types 8
2.1 Alphabetic Listings 8
2.2 Listing by Value 10

3 ThreadX Constants 12
3.1 TX_BLOCK_POOL 12
3.2 TX_BYTE_POOL 13
3.3 TX_EVENT_FLAGS_GROUP 14
3.4 TX_MUTEX 14
3.5 TX_QUEUE 15
3.6 TX_SEMAPHORE 16
3.7 TX_THREAD 17
3.8 TX_TIMER 19
3.9 TX_TIMER_INTERNAL 20

4 ThreadX API Services 21
4.1 Block Memory Services 21
4.1.1 tx_block_allocate() 21
4.1.2 tx_block_pool_create() 23
4.1.3 tx_block_pool_delete () 24
4.1.4 tx_block_pool_info_get() 25
4.1.5 tx_block_pool_performance_info_get() 26
4.1.6 tx_block_pool_performance_system_info_get() 27
4.1.7 tx_block_pool_prioritize() 28
4.1.8 tx_block_release() 29
4.2 Byte Memory Services 29
4.2.1 tx_byte_allocate() 30
4.2.2 tx_byte_pool_create() 31
4.2.3 tx_byte_pool_delete() 32
4.2.4 tx_byte_pool_info_get() 33
4.2.5 tx_byte_pool_performance_info_get() 34
4.2.6 tx_byte_pool_performance_system_info_get() 35

4.2.7 tx_byte_pool_prioritize() 36
4.2.8 tx_byte_release() 37
4.3 Event Flags Services 38
4.3.1 tx_event_flags_create() 38
4.3.2 tx_event_flags_delete() 39
4.3.3 tx_event_flags_get() 40
4.3.4 tx_event_flags_info_get() 41
4.3.5 tx_event_flags_performance info_get() 42
4.3.6 tx_event_flags_performance_system_info_get() 44
4.3.7 tx_event_flags_set() 45
4.3.8 tx_event_flags_set_notify() 46
4.4 Interrupt Control 46
4.4.1 tx_interrupt_control() 46
4.5 Interrupt Control 47
4.5.1 tx_mutex_create() 48
4.5.2 tx_mutex_delete() 48
4.5.3 tx_mutex_get() 49
4.5.4 tx_mutex_info_get() 50
4.5.5 tx_mutex_performance_info_get() 51
4.5.6 tx_mutex_performance_system_info_get() 53
4.5.7 tx_mutex_prioritize() 54
4.5.8 tx_mutex_put() 55
4.6 Queue Services 55
4.6.1 tx_queue_create() 56
4.6.2 tx_queue_delete() 57
4.6.3 tx_queue_flush() 57
4.6.4 tx_queue_front_send() 58
4.6.5 tx_queue_info_get() 59
4.6.6 tx_queue_performance_info_get() 61
4.6.7 tx_queue_performance_system_info_get() 62
4.6.8 tx_queue_prioritize() 63
4.6.9 tx_queue_receive() 64
4.6.10 tx_queue_send() 65
4.6.11 tx_queue_send_notify() 66
4.7 Semaphore Services 67
4.7.1 tx_semaphore_ceiling_put() 67
4.7.2 tx_semaphore_create() 68
4.7.3 tx_semaphore_delete() 69
4.7.4 tx_semaphore_get() 70
4.7.5 tx_semaphore_info_get() 71
4.7.6 tx_semaphore_performance_info_get() 72
4.7.7 tx_semaphore_performance_system_info_get() 73
4.7.8 tx_semaphore_prioritize() 74
4.7.9 tx_semaphore_put() 75
4.7.10 tx_semaphore_put_notify() 76
4.8 Thread Control Services 77

4.8.1 tx_thread_create() 77
4.8.2 tx_thread_delete() 79
4.8.3 tx_thread_entry_exit_notify() 80
4.8.4 tx_thread_identify() 81
4.8.5 tx_thread_info_get() 81
4.8.6 tx_thread_performance_info_get() 83
4.8.7 tx_thread_performance_system_info_get() 85
4.8.8 tx_thread_preemption_change() 86
4.8.9 tx_thread_priority_change() 87
4.8.10 tx_thread_relinquish() 88
4.8.11 tx_thread_reset() 90
4.8.12 tx_thread_resume() 90
4.8.13 tx_thread_sleep() 91
4.8.14 tx_thread_stack_error_notify() 92
4.8.15 tx_thread_suspend() 92
4.8.16 tx_thread_terminate() 93
4.8.17 tx_thread_time_slice_change 94
4.8.18 tx_thread_wait_abort() 95
4.9 Time Services 96
4.9.1 tx_time_get() 96
4.9.2 tx_time_set() 96
4.10 Timer Services 97
4.10.1 tx_timer_activate() 97
4.10.2 tx_timer_change() 98
4.10.3 tx_timer_create() 99
4.10.4 tx_timer_deactivate() 100
4.10.5 tx_timer_delete() 100
4.10.6 tx_timer_info_get() 101
4.10.7 tx_timer_performance_info_get() 102
4.10.8 tx_timer_performance_system_info_get() 104

1 Introduction

1.1 Purpose of the document

ThreadX is a high-performance real-time kernel designed specifically for embedded applications. This document contains details the application's interface to ThreadX.

1.2 Related documents

[1] SIM7070_SIM7080_SIM7090 Series_AT Command Manual

1.3 Conventions and abbreviations

Abbreviation	Description

2 ThreadX Data Types

2.1 Alphabetic Listings

TX_1_ULONG	1
TX_2_ULONG	2
TX_4_ULONG	4
TX_8_ULONG	8
TX_16_ULONG	16
TX_ACTIVATE_ERROR	0x17
TX_AND	2
TX_AND_CLEAR	3
TX_AUTO_ACTIVATE	1
TX_AUTO_START	1
TX_BLOCK_MEMORY	8
TX_BYTE_MEMORY	9
TX_CALLER_ERROR	0x13
TX_CEILING_EXCEEDED	0x21
TX_COMPLETED	1
TX_DELETE_ERROR	0x11
TX_DELETED	0x01
TX_DONT_START	0
TX_EVENT_FLAG	7
TX_FALSE	0
TX_FEATURE_NOT_ENABLED	0xFF
TX_FILE	11
TX_GROUP_ERROR	0x06
TX_INHERIT	1
TX_INHERIT_ERROR	0x1F
TX_INVALID_CEILING	0x22
TX_IO_DRIVER	10
TX_LOOP_FOREVER	1
TX_MUTEX_ERROR	0x1C
TX_MUTEX_SUSP	13
TX_NO_ACTIVATE	0
TX_NO_EVENTS	0x07
TX_NO_INHERIT	0
TX_NO_INSTANCE	0x0D

TX_NO_MEMORY	0x10
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NOT_AVAILABLE	0x1D
TX_NOT_DONE	0x20
TX_NOT_OWNED	0x1E
TX_NULL	0
TX_OPTION_ERROR	0x08
TX_OR	0
TX_OR_CLEAR	1
TX_POOL_ERROR	0x02
TX_PRIORITY_ERROR	0x0F
TX_PTR_ERROR	0x03
TX_QUEUE_EMPTY	0x0A
TX_QUEUE_ERROR	0x09
TX_QUEUE_FULL	0x0B
TX_QUEUE_SUSP	5
TX_READY	0
TX_RESUME_ERROR	0x12
TX_SEMAPHORE_ERROR	0x0C
TX_SEMAPHORE_SUSP	6
TX_SIZE_ERROR	0x05
TX_SLEEP	4
TX_STACK_FILL	0xEFEFEFEFUL
TX_START_ERROR	0x10
TX_SUCCESS	0x00
TX_SUSPEND_ERROR	0x14
TX_SUSPEND_LIFTED	0x19
TX_SUSPENDED	3
TX_TCP_IP	12
TX_TERMINATED	2
TX_THREAD_ENTRY	0
TX_THREAD_ERROR	0x0E
TX_THREAD_EXIT	1
TX_THRESH_ERROR	0x18
TX_TICK_ERROR	0x16
TX_TIMER_ERROR	0x15
TX_TRUE	1
TX_WAIT_ABORT_ERROR	0x1B
TX_WAIT_ABORTED	0x1A
TX_WAIT_ERROR	0x04
TX_WAIT_FOREVER	0xFFFFFFFFUL

2.2 Listing by Value

TX_DONT_START	0
TX_FALSE	0
TX_NO_ACTIVATE	0
TX_NO_INHERIT	0
TX_NO_TIME_SLICE	0
TX_NO_WAIT	0
TX_NULL	0
TX_OR	0
TX_READY	0
TX_SUCCESS	0x00
TX_THREAD_ENTRY	0
TX_1_ULONG	1
TX_AUTO_ACTIVATE	1
TX_AUTO_START	1
TX_COMPLETED	1
TX_INHERIT	1
TX_LOOP_FOREVER	1
TX_DELETED	0x01
TX_OR_CLEAR	1
TX_THREAD_EXIT	1
TX_TRUE	1
TX_2_ULONG	2
TX_AND	2
TX_POOL_ERROR	0x02
TX_TERMINATED	2
TX_AND_CLEAR	3
TX_PTR_ERROR	0x03
TX_SUSPENDED	3
TX_4_ULONG	4
TX_SLEEP	4
TX_WAIT_ERROR	0x04
TX_QUEUE_SUSP	5
TX_SIZE_ERROR	0x05
TX_GROUP_ERROR	0x06
TX_SEMAPHORE_SUSP	6
TX_EVENT_FLAG	7
TX_NO_EVENTS	0x07
TX_8_ULONG	8
TX_BLOCK_MEMORY	8
TX_OPTION_ERROR	0x08
TX_BYTE_MEMORY	9
TX_QUEUE_ERROR	0x09

TX_IO_DRIVER	10
TX_QUEUE_EMPTY	0x0A
TX_FILE	11
TX_QUEUE_FULL	0x0B
TX_TCP_IP	12
TX_SEMAPHORE_ERROR	0x0C
TX_MUTEX_SUSP	13
TX_NO_INSTANCE	0x0D
TX_THREAD_ERROR	0x0E
TX_PRIORITY_ERROR	0x0F
TX_16_ULONG	16
TX_NO_MEMORY	0x10
TX_START_ERROR	0x10
TX_DELETE_ERROR	0x11
TX_RESUME_ERROR	0x12
TX_CALLER_ERROR	0x13
TX_SUSPEND_ERROR	0x14
TX_TIMER_ERROR	0x15
TX_TICK_ERROR	0x16
TX_ACTIVATE_ERROR	0x17
TX_THRESH_ERROR	0x18
TX_SUSPEND_LIFTED	0x19
TX_WAIT_ABORTED	0x1A
TX_WAIT_ABORT_ERROR	0x1B
TX_MUTEX_ERROR	0x1C
TX_NOT_AVAILABLE	0x1D
TX_NOT_OWNED	0x1E
TX_INHERIT_ERROR	0x1F
TX_NOT_DONE	0x20
TX_CEILING_EXCEEDED	0x21
TX_INVALID_CEILING	0x22
TX_FEATURE_NOT_ENABLED	0xFF
TX_STACK_FILL	0xEFEFEFEFUL
TX_WAIT_FOREVER	0xFFFFFFFFUL

3 ThreadX Constants

3.1 TX_BLOCK_POOL

```
typedef struct TX_BLOCK_POOL_STRUCT

{

    ULONG tx_block_pool_id;
    CHAR *tx_block_pool_name;
    ULONG tx_block_pool_available;
    ULONG tx_block_pool_total;
    UCHAR *tx_block_pool_available_list;
    UCHAR *tx_block_pool_start;

    ULONG tx_block_pool_size;

    ULONG tx_block_pool_block_size;
    struct TX_THREAD_STRUCT

                        *tx_block_pool_suspension_list;
    ULONG tx_block_pool_suspended_count;
    struct TX_BLOCK_POOL_STRUCT
                        *tx_block_pool_created_next,
                        *tx_block_pool_created_previous;

#ifdef TX_BLOCK_POOL_ENABLE_PERFORMANCE_INFO

    ULONG tx_block_pool_performance_allocate_count;
    ULONG tx_block_pool_performance_release_count;
    ULONG tx_block_pool_performance_suspension_count;
    ULONG tx_block_pool_performance_timeout_count;

#endif

    TX_BLOCK_POOL_EXTENSION /* Port defined   */

} TX_BLOCK_POOL;
```

3.2 TX_BYTE_POOL

```
typedef struct TX_BYTE_POOL_STRUCT

{

    ULONG tx_byte_pool_id;
    CHAR *tx_byte_pool_name;
    ULONG tx_byte_pool_available;
    ULONG tx_byte_pool_fragments;
    UCHAR *tx_byte_pool_list;
    UCHAR *tx_byte_pool_search;
    UCHAR *tx_byte_pool_start;
    ULONG tx_byte_pool_size;
    struct TX_THREAD_STRUCT
                    *tx_byte_pool_owner;
    struct TX_THREAD_STRUCT
                    *tx_byte_pool_suspension_list;
    ULONG tx_byte_pool_suspended_count
    struct TX_BYTE_POOL_STRUCT
                    *tx_byte_pool_created_next,
                    *tx_byte_pool_created_previous;

#ifdef TX_BYTE_POOL_ENABLE_PERFORMANCE_INFO
    ULONG tx_byte_pool_performance_allocate_count;
    ULONG tx_byte_pool_performance_release_count;
    ULONG tx_byte_pool_performance_merge_count;
    ULONG tx_byte_pool_performance_split_count;
    ULONG tx_byte_pool_performance_search_count;
    ULONG tx_byte_pool_performance_suspension_count;
    ULONG tx_byte_pool_performance_timeout_count;

#endif

    TX_BYTE_POOL_EXTENSION /* Port defined    */

} TX_BYTE_POOL;
```

3.3 TX_EVENT_FLAGS_GROUP

```
typedef struct TX_EVENT_FLAGS_GROUP_STRUCT

{

    ULONG tx_event_flags_group_id;
    CHAR *tx_event_flags_group_name;
    ULONG tx_event_flags_group_current;
    UINT tx_event_flags_group_reset_search;
    struct TX_THREAD_STRUCT
                          *tx_event_flags_group_suspension_list;
     ULONG tx_event_flags_group_suspended_count;

     struct TX_EVENT_FLAGS_GROUP_STRUCT
                          *tx_event_flags_group_created_next,
                          *tx_event_flags_group_created_previous;
     ULONG tx_event_flags_group_delayed_clear;

#ifdef TX_EVENT_FLAGS_ENABLE_PERFORMANCE_INFO

    ULONG tx_event_flags_group_performance_set_count; ULONG
    tx_event_flags_group_performance_get_count;
    ULONG tx_event_flags_group_performance_suspension_count; ULONG
    tx_event_flags_group_performance_timeout_count;

#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
     VOID (*tx_event_flags_group_set_notify)
                    (struct TX_EVENT_FLAGS_GROUP_STRUCT);

#endif

     TX_EVENT_FLAGS_GROUP_EXTENSION /* Port defined */
} TX_EVENT_FLAGS_GROUP;
```

3.4 TX_MUTEX

```
typedef struct TX_MUTEX_STRUCT

{
    ULONG tx_mutex_id;
    CHAR *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;

    ULONG tx_mutex_highest_priority_waiting;
    struct TX_MUTEX_STRUCT
                      *tx_mutex_owned_next,
                      *tx_mutex_owned_previous;

#ifdef TX_ MUTEX_ENABLE_PERFORMANCE_INFO
    ULONG tx_mutex_performance_put_count;
    ULONG tx_mutex_performance_get_count;
    ULONG tx_mutex_performance_suspension_count;
    ULONG tx_mutex_performance_timeout_count;
    ULONG tx_mutex_performance_priority_inversion_count;
    ULONG tx_mutex_performance_priority_inheritance_count;
#endif

    TX_MUTEX_EXTENSION /* Port defined     */

} TX_MUTEX;
```

3.5 TX_QUEUE

```
typedef struct TX_QUEUE_STRUCT

{

    ULONG tx_queue_id;
    CHAR *tx_queue_name;
    UINT tx_queue_message_size;
    ULONG tx_queue_capacity;
    ULONG tx_queue_enqueued;

    ULONG tx_queue_available_storage;
    ULONG *tx_queue_start;

    ULONG *tx_queue_end;
    ULONG *tx_queue_read;
    ULONG *tx_queue_write;
    struct TX_THREAD_STRUCT
                     *tx_queue_suspension_list;
    ULONG tx_queue_suspended_count;

    struct TX_QUEUE_STRUCT
                     *tx_queue_created_next,
                     *tx_queue_created_previous;

#ifdef TX_QUEUE_ENABLE_PERFORMANCE_INFO

    ULONG tx_queue_performance_messages_sent_count;
    ULONG tx_queue_performance_messages_received_count;
    ULONG tx_queue_performance_empty_suspension_count;
    ULONG tx_queue_performance_full_suspension_count;
    ULONG tx_queue_performance_full_error_count;

    ULONG tx_queue_performance_timeout_count;
    #endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

    VOID *tx_queue_send_notify)(struct TX_QUEUE_STRUCT *);
#endif

    TX_QUEUE_EXTENSION /* Port defined      */

} TX_QUEUE;
```

3.6 TX_SEMAPHORE

```
typedef struct TX_SEMAPHORE_STRUCT

{
    ULONG tx_semaphore_id;
    CHAR *tx_semaphore_name;
    ULONG tx_semaphore_count;
    struct TX_THREAD_STRUCT

                        *tx_semaphore_suspension_list;
    ULONG tx_semaphore_suspended_count;

    struct TX_SEMAPHORE_STRUCT
                        *tx_semaphore_created_next,
                        *tx_semaphore_created_previous;

#ifdef TX_ SEMAPHORE_ENABLE_PERFORMANCE_INFO
    ULONG tx_semaphore_performance_put_count;
    ULONG tx_semaphore_performance_get_count;
    ULONG tx_semaphore_performance_suspension_count;
    ULONG tx_semaphore_performance_timeout_count;
#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS

    VOID (*tx_semaphore_put_notify)(struct TX_SEMAPHORE_STRUCT *);
#endif

    TX_SEMAPHORE_EXTENSION /* Port defined    */

} TX_SEMAPHORE;
```

3.7 TX_THREAD

```
typedef struct TX_THREAD_STRUCT

{

    ULONG tx_thread_id;

    ULONG tx_thread_run_count;
    VOID *tx_thread_stack_ptr;
    VOID *tx_thread_stack_start;
    VOID *tx_thread_stack_end;
    ULONG tx_thread_stack_size;
    ULONG tx_thread_time_slice;
```

```
        ULONG tx_thread_new_time_slice;
        struct TX_THREAD_STRUCT
                            *tx_thread_ready_next,
                            *tx_thread_ready_previous;

        TX_THREAD_EXTENSION_0 /* Port defined*/

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

        TX_THREAD_EXTENSION_1 /* Port defined */

        struct TX_THREAD_STRUCT
                            *tx_thread_created_next,
                            *tx_thread_created_previous;

        TX_THREAD_EXTENSION_2 /* Port defined */

        VOID *tx_thread_filex_ptr;
        UINT tx_thread_original_priority;
        UINT tx_thread_original_preempt_threshold;
        ULONG tx_thread_owned_mutex_count;

        struct TX_MUTEX_STRUCT
                            *tx_thread_owned_mutex_list;

#ifdef TX_ THREAD_ENABLE_PERFORMANCE_INFO
        ULONG tx_thread_performance_resume_count;
        ULONG tx_thread_performance_suspend_count;
```

```
    ULONG tx_thread_performance_solicited_preemption_count;
    ULONG tx_thread_performance_interrupt_preemption_count;
    ULONG tx_thread_performance_priority_inversion_count;
    struct TX_THREAD_STRUCT

                      *tx_thread_performance_last_preempting_thread;
    ULONG tx_thread_performance_time_slice_count;

    ULONG tx_thread_performance_relinquish_count; ULONG
    tx_thread_performance_timeout_count;
    ULONG tx_thread_performance_wait_abort_count;

#endif

#ifndef TX_DISABLE_NOTIFY_CALLBACKS
    VOID (*tx_thread_entry_exit_notify)

                      (struct TX_THREAD_STRUCT *, UINT);

#endif

    TX_THREAD_EXTENSION_3 /* Port defined */

    TX_THREAD_USER_EXTENSION

} TX_THREAD;
```

3.8 TX_TIMER

```
typedef struct TX_TIMER_STRUCT

{

    ULONG tx_timer_id;
    CHAR *tx_timer_name;
    TX_TIMER_INTERNAL tx_ timer_internal;
    struct TX_TIMER_STRUCT

                      *tx_timer_created_next, *tx_timer_created_previous;

#ifdef TX_TIMER_ENABLE_PERFORMANCE_INFO

    ULONG tx_timer_performance_activate_count;
    ULONG tx_timer_performance_reactivate_count;
    ULONG tx_timer_performance_deactivate_count;
    ULONG tx_timer_performance_expiration_count;

    ULONG tx_timer_performance_expiration_adjust_count;
#endif

} TX_TIMER;
```

3.9 TX_TIMER_INTERNAL

```
typedef struct TX_TIMER_INTERNAL_STRUCT

{

    ULONG tx_timer_internal_remaining_ticks;
    ULONG tx_timer_internal_re_initialize_ticks;
    VOID (*tx_timer_internal_timeout_function)(ULONG);
    ULONG tx_timer_internal_timeout_param;
     struct TX_TIMER_INTERNAL_STRUCT
                        *tx_timer_internal_active_next,

                        *tx_timer_internal_active_previous;
    struct TX_TIMER_INTERNAL_STRUCT
                        *tx_timer_internal_list_head;

} TX_TIMER_INTERNAL;
```

4 ThreadX API Services

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

4.1 Block Memory Services

4.1.1 tx_block_allocate()

Allocate fixed-size block of memory.

Prototype

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

Description

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

Input Parameters

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

	The wait options are defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless if it was successful or not. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR. Selecting TX_ WAIT_FOREVER causes the calling thread to suspend indefinitely until a memory block is available. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for a memory block.

Returns

TX_SUCCESS	(0x00)	Successful memory block allocation.
TX_DELETED	(0x01)	Memory block pool was deleted while thread was suspended.
TX_NO_MEMORY	(0x10)	Service was unable to allocate a block of memory within the specified time to wait.
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. */
```

4.1.2 tx_block_pool_create()

Create pool of fixed-size memory blocks.

Prototype

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

Description

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

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

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

Input Parameters

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

Returns

TX_SUCCESS	(0x00)	Successful memory block pool creation.
TX_POOL_ERROR	(0x02)	Invalid memory block pool pointer.Either the pointer is NULL or the pool is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the pool.

TX_SIZE_ERROR	(0x05)	Size of pool is invalid.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_BLOCK_POOLmy_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. */
```

4.1.3 tx_block_pool_delete ()

Delete memory block pool.

Prototype

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

Description

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

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

Input Parameters

pool_ptr	Pointer to a previously created memory block pool.

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_BLOCK_POOLmy_pool;
UINT            status;
 /* Delete entire memory block pool. Assume that the pool
       has already been created with a call to

       tx_block_pool_create.  */

    status =   tx_block_pool_delete(&my_pool);
    /*  If status equals TX_SUCCESS, the memory block pool is deleted. */
```

4.1.4 tx_block_pool_info_get()

Retrieve information about block pool.

Prototype

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

Description

This service retrieves information about the specified block memory pool.

Input Parameters

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_BLOCK_POOLmy_pool;
CHAR            *name;
```

```
ULONG           available;
ULONG           total_blocks;
TX_THREAD       *first_suspended;
ULONG           suspended_count;
TX_BLOCK_POOL*next_pool;
UINT            status;
/* Retrieve information about the previously created block pool "my_pool." */

status = tx_block_pool_info_get(&my_pool, &name, &available,&total_blocks,
                &first_suspended, &suspended_count, &next_pool);
/* If status equals TX_SUCCESS, the information requested is valid. */
```

4.1.5 tx_block_pool_performance_info_get()

Get block pool performance information.

Prototype

UINT **tx_block _pool_performance_info_get**(TX_BLOCK_POOL *__pool_ptr__, ULONG *__allocates__, ULONG *__releases__,ULONG *__suspensions__, ULONG *__timeouts__))

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

Description

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

Input Parameters

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_BLOCK_POOLmy_pool;
ULONG           allocates;
ULONG           releases;
ULONG           suspensions;
ULONG           timeouts;
/* Retrieve performance information on the previously created block pool. */

status    =    tx_block_pool_performance_info_get(&my_pool,    &allocates,    &releases,
                                         &suspensions, &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.1.6 tx_block_pool_performance_system_info_get()

Get block pool system performance information.

Prototype

UINT **tx_block _pool_performance _system_info_get**(ULONG ***allocates**, ULONG ***releases**, ULONG ***suspensions**, ULONG ***timeouts**);

Description

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

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

Input Parameters

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG           allocates;
ULONG           releases;

ULONG           suspensions;

ULONG           timeouts;
/* Retrieve performance information on all the block pools in the system. */

status          =           tx_block_pool_performance_system_info_get(&allocates,
                &releases,&suspensions, &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.1.7 tx_block_pool_prioritize()

Prioritize block pool suspension list.

Prototype

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

Description

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

Input Parameters

pool_ptr	Pointer to a memory block pool control block.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BLOCK_POOLmy_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. */
```

4.1.8 tx_block_release()

Release fixed-size block of memory.

Prototype

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

Description

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

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

Input Parameters

pool_ptr	Pointer to the previously allocated memory block.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_BLOCK_POOL      my_pool;

unsigned char      *memory_ptr;

UINT               status;
/*  Release a memory block back to my_pool. Assume that the pool has been created and
    the memory block has been allocated. */
status =   tx_block_release((VOID *) memory_ptr);
/*  If status equals TX_SUCCESS, the block of memory pointed to by memory_ptr has
    been returned to the pool. */
```

4.2 Byte Memory Services

4.2.1 tx_byte_allocate()

Allocate bytes of memory.

Prototype

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

Description

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

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

Input Parameters

pool_ptr	Pointer to a previously created memory pool.
memory_ptr	Pointer to a destination memory pointer. On successful allocation, the address of the allocated memory area is placed where this parameter points to.
memory_size	Number of bytes requested.
wait_option	Defines how the service behaves if there is not enough memory available. The wait options are defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from initialization.Selecting TX_ WAIT_FOREVER causes the calling thread to suspend indefinitely until enough memory is available.Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the memory.

Returns

TX_SUCCESS	(0x00)	Successful memory allocation.
TX_DELETED	(0x01)	Memory pool was deleted while thread was suspended.
TX_NO_MEMORY	(0x10)	Service was unable to allocate the memory within the specified time to wait.
TX_WAIT_ABORTED	(0x1A)	Suspension was aborted by another thread, timer, or ISR.
TX_POOL_ERROR	(0x02)	Invalid memory pool pointer.
TX_PTR_ERROR	(0x03)	Invalid pointer to destination pointer.
TX_SIZE_ERROR	(0X05)	Requested size is zero or larger than the pool.
TX_WAIT_ERROR	(0x04)	A wait option other than TX_NO_WAIT was specified on a call from a non-thread.

TX_CALLER_ERROR (0x13) Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

TX_BYTE_POOL my_pool;
unsigned char*memory_ptr;
UINT status;
/* Allocate a 112 byte memory area from my_pool. Assume that the pool has already been created with a call to tx_byte_pool_create. */
status = **tx_byte_allocate**(&my_pool, (VOID **) &memory_ptr, 112, TX_NO_WAIT);
/* If status equals TX_SUCCESS, memory_ptr contains the address of the allocated memory area. */

4.2.2 tx_byte_pool_create()

Create memory pool of bytes.

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_BYTE_POOL my_pool;
UINT          status;
/* Create a memory pool whose total size is 2000 bytes starting at address 0x500000.
   */
status = tx_byte_pool _create(&my_pool, "my_pool_name", (VOID *) 0x500000,
              2000);
/* If status equals TX_SUCCESS, my_pool is available for allocating memory. */
```

4.2.3 tx_byte_pool_delete()

Delete memory byte pool.

Prototype

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

Description

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

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

Input Parameters

pool_ptr	Pointer to a previously created memory pool.

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

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

4.2.4 tx_byte_pool_info_get()

Retrieve information about byte pool.

Prototype

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

Description

This service retrieves information about the specified memory byte pool.

Input Parameters

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BYTE_POOL my_pool;
CHAR         *name;
ULONG        available;
ULONG        fragments;

TX_THREAD    *first_suspended;
ULONG        suspended_count;

TX_BYTE_POOL *next_pool;
```

```
UINT            status;
/*  Retrieve information about the previously created block pool "my_pool." */

status = tx_byte_pool_info_get(&my_pool, &name, &available, &fragments,
                               &first_suspended, &suspended_count, &next_pool);
/*  If status equals TX_SUCCESS, the information requested is valid. */
```

4.2.5 tx_byte_pool_performance_info_get()

Get byte pool performance information.

Prototype

UINT **tx_byte_ pool_performance_info_get**(TX_ BYTE_POOL *__pool_ptr__, ULONG *__allocates__, ULONG *__releases__,ULONG *__fragments_searched__ , ULONG *__merges__, ULONG *__splits__, ULONG *__suspensions__, ULONG *__timeouts__);

Description

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

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

Input Parameters

pool_ptr	Pointer to previously created memory byte pool.
allocates	Pointer to destination for the number of allocate requests performed on this pool.
releases	Pointer to destination for the number of release requests performed on this pool.
fragments_searched	Pointer to destination for the number of internal memory fragments searched during allocation requests on this pool.
merges	Pointer to destination for the number of internal memory blocks merged during allocation requests on this pool.
splits	Pointer to destination for the number of internal memory blocks split (fragments) created during allocation requests on this pool.
suspensions	Pointer to destination for the number of thread allocation suspensions on this pool.
timeouts	Pointer to destination for the number of allocate suspension timeouts on this pool.

Note:Supplying a TX_NULL for any parameter indicates the parameter is not required

Returns

TX_SUCCESS	(0x00)	Successful byte pool performance get.

TX_PTR_ERROR	(0x03)	Invalid byte pool pointer.
TX_FEATURE_NOT_ENABLED	(0xFF)	The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_BYTE_POOL   my_pool;
ULONG          fragments_searched;

ULONG          merges;

ULONG          splits;

ULONG          allocates;

ULONG          releases;

ULONG          suspensions;

ULONG          timeouts;
/* Retrieve performance information on the previously created byte pool. */

status = tx_byte_pool_performance_info_get(&my_pool,
                        &fragments_searched,&merges, &splits, &allocates,
                        &releases, &suspensions,&timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.2.6 tx_byte_pool_performance_system_info_get()

Get byte pool system performance information.

Prototype

UINT **tx_byte_ pool_performance_ system_info_get**(ULONG ***allocates**, ULONG ***releases**, ULONG ***fragments_searched**, ULONG ***merges**, ULONG ***splits**, ULONG ***suspensions**, ULONG ***timeouts**);

Description

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

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

Input Parameters

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

releases	Pointer to destination for the number of release requests performed on this pool.
fragments_searched	Pointer to destination for the total number of internal memory fragments searched during allocation requests on all byte pools.
merges	Pointer to destination for the total number of internal memory blocks merged during allocation requests on all byte pools.
splits	Pointer to destination for the total number of internal memory blocks split (fragments) created during allocation requests on all byte pools.
suspensions	Pointer to destination for the total number of thread allocation suspensions on all byte pools.
timeouts	Pointer to destination for the total number of allocate suspension timeouts on all byte pools.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG          fragments_searched;
ULONG          merges;
ULONG          splits;
ULONG          allocates;
ULONG          releases;
ULONG          suspensions;
ULONG          timeouts;
/* Retrieve performance information on all byte pools in the system. */

status = tx_byte_pool_performance_system_info_get(&fragments_searched,

               &merges, &splits, &allocates, &releases, &suspensions, &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.2.7 tx_byte_pool_prioritize()

Prioritize byte pool suspension list.

Prototype

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

Description

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

Input Parameters

pool_ptr	Pointer to a memory pool control block.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_BYTE_POOL my_pool;
UINT          status;
/* Ensure that the highest priority thread will receive the next free memory from this
   pool. */
status = tx_byte_pool_prioritize(&my_pool);
/* If status equals TX_SUCCESS, the highest priority suspended thread is at the front
   of the list. The next tx_byte_release call will wake up this thread, if there is enough
   memory to satisfy its request. */
```

4.2.8 tx_byte_release()

Release bytes back to memory pool.

Prototype

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

Description

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

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

Input Parameters

memory_ptr	Pointer to the previously allocated memory area.

Returns

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

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
unsigned char    *memory_ptr;
UINT             status;
/* Release a memory back to my_pool. Assume that the memory area was previously
   allocated from my_pool. */

status =   tx_byte_release((VOID *) memory_ptr);
/* If status equals TX_ SUCCESS, the memory pointed to by memory_ptr has been
   returned to the pool. */
```

4.3 Event Flags Services

4.3.1 tx_event_flags_create()

Create event flags group.

Prototype

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

Description

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

Input Parameters

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

Returns

TX_SUCCESS	(0x00)	Successful event group creation.

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

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_EVENT_FLAGS_GROUP      my_event_group;
UINT                      status;
/* Create an event flags group.    */
status =   tx_event_flags_create(&my_event_group,
                      "my_event_group_name");
/*  If status equals TX_SUCCESS, my_event_group is ready for get and set services.
    */
```

4.3.2 tx_event_flags_delete()

Delete event flags group.

Prototype

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

Description

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

Note:The application must prevent use of a deleted event flags group.

Input Parameters

group_ptr	Pointer to a previously created event flags group.

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_EVENT_FLAGS_GROUP my_event_flags_group;
UINT                 status;
/* Delete event flags group. Assume that the group has
   already been created with a call to
```

```
        tx_event_flags_create.   */
    status =   tx_event_flags_delete(&my_event_flags_group);

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

4.3.3 tx_event_flags_get()

Get event flags from event flags group.

Prototype

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

Description

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

Input Parameters

group_ptr	Pointer to a previously created event flags group.
requested_flags	32-bit unsigned variable that represents the requested event flags.
get_option	Specifies whether all or any of the requested event flags are required. The following are valid selections: TX_AND (0x02) TX_AND_CLEAR (0x03) TX_OR (0x00) TX_OR_CLEAR (0x01) Selecting TX_AND or TX_AND_CLEAR specifies that all event flags must be present in the group. Selecting TX_OR or TX_OR_CLEAR specifies that any event flag is satisfactory. Event flags that satisfy the request are cleared (set to zero) if TX_AND_CLEAR or TX_OR_CLEAR are specified.
actual_flags_ptr	Pointer to destination of where the retrieved event flags are placed. Note that the actual flags obtained may contain flags that were not requested.
wait_option	Defines how the service behaves if the selected event flags are not set. The wait options are defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g.,Initialization, timer, or ISR. Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until the event flags are available. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of

	timer-ticks to stay suspended while waiting for the event flags.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

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

status   =   tx_event_flags_get(&my_event_flags_group,   0x111,   TX_AND_CLEAR,
                                &actual_events, 20);
/* If status equals TX_SUCCESS, actual_events contains the actual events obtained. */
```

4.3.4 tx_event_flags_info_get()

Retrieve information about event flags group.

Prototype

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

Description

This service retrieves information about the specified event flags group.

Input Parameters

group_ptr	Pointer to an event flags group control block.

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

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

Returns

TX_SUCCESS	(0x00)	Successful event group information retrieval.
TX_GROUP_ERROR	(0x06)	Invalid event group pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_EVENT_FLAGS_GROUP     my_event_group;
CHAR                     *name;

ULONG                    current_flags;

TX_THREAD                *first_suspended;

ULONG                    suspended_count;

TX_EVENT_FLAGS_GROUP     *next_group;

UINT                     status;
/* Retrieve information about the previously created event flags group
   "my_event_group." */

status = tx_event_flags_info_get(&my_event_group, &name, &current_flags,

                         &first_suspended, &suspended_count, &next_group);
/* If status equals TX_SUCCESS, the information requested is valid. */
```

4.3.5 tx_event_flags_performance info_get()

Get event flags group performance information

Prototype

```
UINT tx_event_flags_performance_info_ get(TX _EVENT_ FLAGS_GROUP *group_ptr,
                                ULONG *sets,
                                ULONG *gets, ULONG *suspensions,
                                ULONG *timeouts);
```

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

Description

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

Input Parameters

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

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

Returns

TX_SUCCESS	(0x00)	Successful event flags group performance get.
TX_PTR_ERROR	(0x03)	Invalid event flags group pointer.
TX_FEATURE_NOT_ENABLED	(0xFF)	The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_EVENT_FLAGS_GROUP    my_event_flag_group;
ULONG                   sets;

ULONG                   gets;

ULONG                   suspensions;

ULONG                   timeouts;
/* Retrieve performance information on the previously created event flag group. */

status = tx_event_flags_performance_info_get(&my_event_flag_group, &sets, &gets,
                                &suspensions, &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.3.6 tx_event_flags_performance_system_info_get()

Retrieve performance system information.

Prototype

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

Description

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

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

Input Parameters

sets	Pointer to destination for the total number of event flags set requests performed on all groups.
gets	Pointer to destination for the total number of event flags get requests performed on all groups.
suspensions	Pointer to destination for the total number of thread event flags get suspensions on all groups.
timeouts	Pointer to destination for the total number of event flags get suspension timeouts on all groups.

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

Returns

TX_SUCCESS	(0x00)	Successful event flags system performance get.
TX_FEATURE_NOT_ENABLED	(0xFF)	The system was not compiled with performance information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG          sets;
ULONG          gets;

ULONG          suspensions;

ULONG          timeouts;
/* Retrieve performance information on all previously created event flag groups. */

status    =    tx_event_flags_performance_system_info_get(&sets,    &gets,    &suspensions,
                  &timeouts);
```

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

4.3.7 tx_event_flags_set()

Set event flags in an event flags group

Prototype

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

Description

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

Input Parameters

group_ptr	Pointer to the previously created event flags group control block.
flags_to_set	Specifies the event flags to set or clear based upon the set option selected.
set_option	Specifies whether the event flags specified are ANDed or ORed into the current event flags of the group. The following are valid selections: TX_AND (0x02) TX_OR (0x00) Selecting TX_AND specifies that the specified event flags are ANDed into the current event flags in the group. This option is often used to clear event flags in a group. Otherwise, if TX_OR is specified, the specified event flags are ORed with the current event in the group.

Returns

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

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. */
```

4.3.8 tx_event_flags_set_notify()

Set event flags in an event flags group

Prototype

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

Description

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

Input Parameters

group_ptr	Pointer to previously created event flags group.
events_set_notify	Pointer to application's event flags set notification function. If this value is TX_NULL, notification is disabled.

Returns

TX_SUCCESS	(0x00)	Successful registration of event flags set notification.
TX_GROUP_ERROR	(0x06)	Invalid event flags group pointer.
TX_FEATURE_NOT_ENABLED	(0xFF)	The system was compiled with notification capabilities disabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_EVENT_FLAGS_GROUP    my_group;
/* Register the "my _event_ flags_ set_notify" function for monitoring event flags set in the event
    flags group "my_group." */

status=tx_event_flags_set_notify(&my_group, my_event_flags_set_notify);
/* If status is TX_SUCCESS the event flags set notification function was successfully registered. */
void my_event_flags_set_notify(TX_EVENT_FLAGS_GROUP *group_ptr)
    /* One or more event flags was set in this group!    */
```

4.4 Interrupt Control

4.4.1 tx_interrupt_control()

Enable and disable interrupts

Prototype

UINT **tx_interrupt_control**(UINT **new_posture**)

Description

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

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

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

Input Parameters

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

Returns

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

Allowed From

Threads, timers, and ISRs

Preemption Possible

No

Example

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

4.5 Interrupt Control

4.5.1 tx_mutex_create()

Create mutual exclusion mutex

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_MUTEX  my_mutex;
UINT      status;
/* Create a mutex to provide protection over a
   common resource.    */

status =   tx_mutex_create(&my_mutex,"my_mutex_name",

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

4.5.2 tx_mutex_delete()

Delete mutual exclusion mutex

Prototype

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

Description

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

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

Input Parameters

mutex_ptr	Pointer to a previously created mutex.

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

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

4.5.3 tx_mutex_get()

Obtain ownership of mutex

Prototype

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

Description

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

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

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

Input Parameters

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

	TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from Initialization. Selecting TX_ WAIT_FOREVER causes the calling thread to suspend indefinitely until the mutex is available. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for the mutex.

Returns

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

Allowed From

Initialization and threads and timers

Preemption Possible

Yes

Example

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

4.5.4 tx_mutex_info_get()

Retrieve information about mutex

Prototype

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

Description

This service retrieves information from the specified mutex.

Input Parameters

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

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

Returns

TX_SUCCESS	(0x00)	Successful mutex information retrieval.
TX_MUTEX_ERROR	(0x1C)	Invalid mutex 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 the previously created mutex "my_mutex." */

status = tx_mutex_info_get(&my_mutex, &name, &count, &owner,

                              &first_suspended, &suspended_count, &next_mutex);
/*  If status equals TX_SUCCESS, the information requested is valid. */
```

4.5.5 tx_mutex_performance_info_get()

Get mutex performance information

Prototype

UINT **tx_mutex _performance_ info_get**(TX_MUTEX ***mutex_ptr**, ULONG ***puts**, ULONG
***gets**, ULONG ***suspensions**, ULONG ***timeouts**,
ULONG ***inversions**, ULONG ***inheritances**);

Description

This service retrieves performance information about the specified mutex.

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

Input Parameters

mutex_ptr	Pointer to previously created mutex.
puts	Pointer to destination for the number of put requests performed on this mutex.
gets	Pointer to destination for the number of get requests performed on this mutex.
suspensions	Pointer to destination for the number of thread mutex get suspensions on this mutex.
timeouts	Pointer to destination for the number of mutex get suspension timeouts on this mutex.
inversions	Pointer to destination for the number of thread priority inversions on this mutex.
inheritances	Pointer to destination for the number of thread priority inheritance operations on this mutex.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_MUTEX        my_mutex;
ULONG           puts;

ULONG           gets;

ULONG           suspensions;

ULONG           timeouts;

ULONG           inversions;

ULONG           inheritances;
/* Retrieve performance information on the previously created mutex. */
```

status = **tx_mutex_performance _info_get**(&my_mutex_ptr, &puts, &gets, &suspensions, &timeouts, &inversions, &inheritances);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */

4.5.6 tx_mutex_performance_system_info_get()

Get mutex system performance information

Prototype

UINT tx_mutex _performance_system_ info_get(ULONG ***puts**, ULONG ***gets**, ULONG ***suspensions**, ULONG ***timeouts**, ULONG ***inversions**, ULONG ***inheritances**);

Description

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

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

Input Parameters

puts	Pointer to destination for the total number of put requests performed on all mutexes.
gets	Pointer to destination for the total number of get requests performed on all mutexes.
suspensions	Pointer to destination for the total number of thread mutex get suspensions on all mutexes.
timeouts	Pointer to destination for the total number of mutex get suspension timeouts on all mutexes.
inversions	Pointer to destination for the total number of thread priority inversions on all mutexes.
inheritances	Pointer to destination for the total number of thread priority inheritance operations on all mutexes.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG          puts;
ULONG          gets;

ULONG          suspensions;

ULONG          timeouts;

ULONG          inversions;

ULONG          inheritances;
/* Retrieve performance information on all previously created mutexes. */
status = tx_mutex_performance _system_info_get(&puts, &gets, &suspensions,
                    &timeouts,

                    &inversions, &inheritances);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.5.7 tx_mutex_prioritize()

Prioritize mutex suspension list.

Prototype

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

Description

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

Input Parameters

mutex_ptr	Pointer to the previously created mutex.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_MUTEX  my_mutex;
UINT      status;
/* Ensure that the highest priority thread will receive ownership of the mutex when it
   becomes available. */
status = tx_mutex_prioritize(&my_mutex);
/* If status equals TX_SUCCESS, the highest priority suspended thread is at the front
   of the list. The next tx_mutex_put call that releases ownership of the mutex will give
```

```
            ownership to this thread and wake it up. */
```

4.5.8 tx_mutex_put()

Release ownership of mutex

Prototype

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

Description

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

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

Input Parameters

mutex_ptr	Pointer to the previously created mutex.

Returns

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

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
TX_MUTEX  my_mutex;
UINT      status;
/* Release ownership of "my_mutex." */
status = tx_mutex_put(&my_mutex);
/*  If status equals TX_SUCCESS, the mutex ownership count has been decremented
    and if zero, released. */
```

4.6 Queue Services

4.6.1 tx_queue_create()

Create message queue

Prototype

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

Description

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

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

Input Parameters

queue_ptr	Pointer to a message queue control block.
name_ptr	Pointer to the name of the message queue.
message_size	Specifies the size of each message in the queue. Message sizes range from 1 32-bit word to 16 32-bit words. Valid message size options are numerical values from 1 through 16, inclusive.
queue_start	Starting address of the message queue.
queue_size	Total number of bytes available for the message queue.

Returns

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

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_QUEUE  my_queue;
UINT      status;
/* Create a message queue whose total size is 2000 bytes starting at address
   0x300000. Each message in this queue is defined to be 4 32-bit words long. */
status = tx_queue_create(&my _queue, "my_queue_name", 4, (VOID *) 0x300000,
                         2000);
/* If status equals TX_SUCCESS, my_queue contains room for storing 125
   messages (2000 bytes/ 16 bytes per message). */
```

4.6.2 tx_queue_delete()

Delete message queue

Prototype

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

Description

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

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

Input Parameters

queue_ptr	Pointer to a previously created message queue.

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_QUEUE  my_queue;
UINT      status;
/*  Delete entire message queue. Assume that the queue has already been created
    with a call to tx_queue_create. */
status =   tx_queue_delete(&my_queue);
/*  If status equals TX_SUCCESS, the message queue is deleted. */
```

4.6.3 tx_queue_flush()

Empty messages in message queue

Prototype

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

Description

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

Input Parameters

queue_ptr	Pointer to a previously created message queue.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_QUEUE  my_queue;
UINT      status;
/* Flush out all pending messages in the specified message queue. Assume that the
   queue has already been created with a call to tx_queue_create. */
status =  tx_queue_flush(&my_queue);
/* If status equals TX_SUCCESS, the message queue is empty. */
```

4.6.4 tx_queue_front_send()

Send message to the front of queue

Prototype

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

Description

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

Input Parameters

queue_ptr	Pointer to a message queue control block.
source_ptr	Pointer to the message.
wait_option	Defines how the service behaves if the message queue is full. The wait options are defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR. Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until there is room in the queue. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
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. */
```

4.6.5 tx_queue_info_get()

Retrieve information about queue

Prototype

UINT **tx_queue_info_get**(TX _QUEUE *__queue_ptr__, CHAR **__name__, ULONG *__enqueued__, ULONG *__available_storage__ TX_THREAD **__first_suspended__, ULONG *__suspended_count__, TX_QUEUE **__next_queue__)

Description

This service retrieves information about the specified message queue.

Input Parameters

queue_ptr	Pointer to a previously created message queue.
name	Pointer to destination for the pointer to the queue's name.
enqueued	Pointer to destination for the number of messages currently in the queue.

available_storage	Pointer to destination for the number of messages the queue currently has space for.
first_suspended	Pointer to destination for the pointer to the thread that is first on the suspension list of this queue.
suspended_count	Pointer to destination for the number of threads currently suspended on this queue.
next_queue	Pointer to destination for the pointer of the next created queue.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_QUEUE  my_queue;
CHAR      *name;

ULONG     enqueued;

ULONG     available_storage;

TX_THREAD   *first_suspended;

ULONG     suspended_count;

TX_QUEUE  *next_queue;

UINT      status;
/*  Retrieve information about the previously created message queue "my_queue."
   */

status =   tx_queue_info_get(&my_queue, &name,

                    &enqueued,       &available_storage,       &first_suspended,
                    &suspended_count, &next_queue);
/*  If status equals TX_SUCCESS, the information requested is valid. */
```

4.6.6 tx_queue_performance_info_get()

Get queue performance information

Prototype

UINT **tx_queue_performance_info_get**(TX_QUEUE *__queue_ptr__, ULONG *__messages_sent__, ULONG *__messages_received__, ULONG *__empty_suspensions__, ULONG *__full_suspensions__, ULONG *__full_errors__, ULONG *__timeouts__);

Description

This service retrieves performance information about the specified queue.

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

Input Parameters

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_QUEUE        my_queue;
ULONG           messages_sent;

ULONG           messages_received;

ULONG           empty_suspensions;
ULONG           full_suspensions;

ULONG           full_errors;

ULONG           timeouts;
/* Retrieve performance information on the previously created queue. */

status     =       tx_queue_performance_info_get(&my_queue,        &messages_sent,
             &messages_received,   &empty_suspensions,   &full_suspensions,   &full_errors,
             &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.6.7 tx_queue_performance_system_info_get()

Get queue system performance information

Prototype

UINT **tx_queue_performance_system_info_get**(ULONG *__messages_sent__, ULONG *__messages_received__, ULONG *__empty_suspensions__,
ULONG *__full_suspensions__, ULONG *__full_errors__, ULONG *__timeouts__);

Description

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

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

Input Parameters

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
    ULONG          messages_sent;
    ULONG          messages_received;

    ULONG          empty_suspensions;

    ULONG          full_suspensions;

    ULONG          full_errors;

    ULONG          timeouts;

    /* Retrieve performance information on all previously created queues. */

    status = tx_queue_performance_system_info_get(&messages _sent,
                   &messages_received, &empty_suspensions, &full_suspensions,
                   &full_errors, &timeouts);
    /* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.6.8 tx_queue_prioritize()

Prioritize queue suspension list

Prototype

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

Description

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

Input Parameters

queue_ptr	Pointer to a previously created message queue.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_QUEUE  my_queue;
UINT      status;
/*  Ensure that the highest priority thread will receive the next message placed on this
    queue. */
status =   tx_queue_prioritize(&my_queue);
/*  If status equals TX_SUCCESS, the highest priority suspended thread is at the front
    of the list. The next tx_queue_send or tx_queue_front_send call made to this
    queue will wake up this thread. */
```

4.6.9 tx_queue_receive()

Get message from message queue

Prototype

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

Description

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

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

Input Parameters

queue_ptr	Pointer to a previously created message queue.
destination_ptr	Location of where to copy the message.
wait_option	Defines how the service behaves if the message queue is empty. The wait options are defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR. Selecting TX_ WAIT_FOREVER causes the calling thread to suspend indefinitely until a message is available. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for a message.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_QUEUE    my_queue;
UINT        status;

ULONG       my_message[4];
/*  Retrieve a message from "my_queue." If the queue is empty, suspend until a message is
    present. Note that this suspension is only possible from application threads. */

status = tx_queue_receive(&my_queue, my_message, TX_WAIT_FOREVER);
/*  If status equals TX_SUCCESS, the message is in "my_message." */
```

4.6.10 tx_queue_send()

Send message to message queue

Prototype

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

Description

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

Input Parameters

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

	defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., Initialization, timer, or ISR. Selecting TX_WAIT_FOREVER causes the calling thread to suspend indefinitely until there is room in the queue. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for room in the queue.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
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. */
```

4.6.11 tx_queue_send_notify()

Notify application when message is sent to queue

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_QUEUE      my_queue;
/* Register the "my_ queue_send _notify" function for monitoring messages sent to the queue
   "my_queue." */

status =   tx_queue_send_notify(&my_queue, my_queue_send_notify);
/* If status is TX_SUCCESS the queue send notification function was successfully registered. */

void my_queue_send_notify(TX_QUEUE *queue_ptr)
{
    /* A message was just sent to this queue!     */
  }
```

4.7 Semaphore Services

4.7.1 tx_semaphore_ceiling_put()

Place an instance in counting semaphore with ceiling

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_SEMAPHORE my_semaphore;
/* Increment the counting semaphore "my_ semaphore" but make sure that it never exceeds 7
   as specified in the call. */

status =   tx_semaphore_ceiling_put(&my_semaphore, 7);
/* If status is TX_SUCCESS the semaphore count has been incremented. */
```

4.7.2 tx_semaphore_create()

Create counting semaphore

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization and threads

Preemption Possible

No

Example

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

Status = tx_semaphore_create(&my_semaphore, "my_semaphore_name", 1);
/*  If status equals TX_SUCCESS, my_semaphore is ready for use. */
```

4.7.3 tx_semaphore_delete()

Delete counting semaphore

Prototype

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

Description

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

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

Input Parameters

semaphore_ptr	Pointer to a previously created semaphore.

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

```
TX_SEMAPHORE  my_semaphore;
UINT          status;
```

```
/* Delete counting semaphore. Assume that the counting semaphore has already
   been created. */
status =   tx_semaphore_delete(&my_semaphore);
/* If status equals TX_SUCCESS, the counting semaphore is deleted. */
```

4.7.4 tx_semaphore_get()

Get instance from counting semaphore

Prototype

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

Description

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

Input Parameters

semaphore_ptr	Pointer to a previously created counting semaphore.
wait_option	Defines how the service behaves if there are no instances of the semaphore available; i.e., the semaphore count is zero. The wait options are defined as follows: TX_NO_WAIT (0x00000000) TX_WAIT_FOREVER (0xFFFFFFFF) timeout value (0x00000001 through 0xFFFFFFFE) Selecting TX_NO_WAIT results in an immediate return from this service regardless of whether or not it was successful. This is the only valid option if the service is called from a non-thread; e.g., initialization, timer, or ISR. Selecting TX_ WAIT_FOREVER causes the calling thread to suspend indefinitely until a semaphore instance is available. Selecting a numeric value (1-0xFFFFFFFE) specifies the maximum number of timer-ticks to stay suspended while waiting for a semaphore instance.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_SEMAPHORE my_semaphore;
UINT          status;
/* Get a semaphore instance from the semaphore
   "my_semaphore." If the semaphore count is zero,

   suspend until an instance becomes available.

   Note that this suspension is only possible from

   application threads.   */

status =  tx_semaphore_get(&my_semaphore, TX_WAIT_FOREVER);
/* If status equals TX_SUCCESS, the thread has obtained an instance of the
   semaphore. */
```

4.7.5 tx_semaphore_info_get()

Retrieve information about semaphore

Prototype

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

Description

This service retrieves information about the specified semaphore.

Input Parameters

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

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

Returns

TX_SUCCESS	(0x00)	Successful semaphore information retrieval.
TX_SEMAPHORE_ERROR	(0x0C)	Invalid semaphore pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_SEMAPHORE my_semaphore;
CHAR         *name;

ULONG        current_value;

TX_THREAD    *first_suspended;

ULONG        suspended_count;

TX_SEMAPHORE *next_semaphore;

UINT         status;
/*  Retrieve information about the previously created semaphore "my_semaphore."
    */

status = tx_semaphore_info_get(&my_semaphore, &name, &current_value,

                               &first_suspended, &suspended_count, &next_semaphore);
/*  If status equals TX_SUCCESS, the information requested is valid. */
```

4.7.6 tx_semaphore_performance_info_get()

Get semaphore performance information

Prototype

UINT **tx_semaphore_performance_info_get**(TX_SEMAPHORE *__semaphore_ptr__, ULONG *__puts__, ULONG *__gets__, ULONG *__suspensions__, ULONG *__timeouts__);

Description

This service retrieves performance information about the specified semaphore.

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

Input Parameters

semaphore_ptr	Pointer to previously created semaphore.

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

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_SEMAPHORE my_semaphore;
ULONG          puts;

ULONG          gets;

ULONG          suspensions;

ULONG          timeouts;
/* Retrieve performance information on the previously created semaphore. */
status = tx_semaphore_ performance_info_get(&my_semaphore, &puts, &gets, &suspensions,
                  &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.7.7 tx_semaphore_performance_system_info_get()

Get semaphore system performance information

Prototype

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

Description

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

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

Input Parameters

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

	semaphores.
gets	Pointer to destination for the total number of get requests performed on all semaphores.
suspensions	Pointer to destination for the total number of thread suspensions on all semaphores.
timeouts	Pointer to destination for the total number of thread suspension timeouts on all semaphores.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG          puts;
ULONG          gets;

ULONG          suspensions;

ULONG          timeouts;
/* Retrieve performance information on all previously created semaphores. */

status = tx_semaphore_performance_system_info_get(&puts, &gets, &suspensions,
               &timeouts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.7.8 tx_semaphore_prioritize()

Prioritize semaphore suspension list

Prototype

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

Description

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

Input Parameters

semaphore_ptr	Pointer to a previously created semaphore.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_SEMAPHORE my_semaphore;
UINT          status;
/* Ensure that the highest priority thread will receive the next instance of this
   semaphore. */
status =   tx_semaphore_prioritize(&my_semaphore);
/* If status equals TX_SUCCESS, the highest priority suspended thread is at the
   front of the list. The
   next tx_semaphore_put call made to this semaphore will wake up this thread. */
```

4.7.9 tx_semaphore_put()

Place an instance in counting semaphore

Prototype

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

Description

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

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

Input Parameters

semaphore_ptr	Pointer to the previously created counting semaphore control block.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_SEMAPHORE  my_semaphore;
UINT          status;
```

```
/* Increment the counting semaphore "my_semaphore." */
status = tx_semaphore_put(&my_semaphore);
/* If status equals TX_SUCCESS, the semaphore count has been incremented. Of
   course, if a thread was waiting, it was given the semaphore instance and resumed. */
```

4.7.10 tx_semaphore_put_notify()

Notify application when semaphore is put

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_SEMAPHORE my_semaphore;
/* Register the "my_semaphore _put_notify" function for monitoring the put operations on the
   semaphore "my_semaphore." */

status = tx_semaphore_put_notify(&my_semaphore,
                my_semaphore_put_notify);
/* If status is TX_SUCCESS the semaphore put notification function was successfully registered.
   */
void my_semaphore_put_notify(TX_SEMAPHORE *semaphore_ptr)
{
        /* The semaphore was just put!   */
    }
```

4.8 Thread Control Services

4.8.1 tx_thread_create()

Create application thread

Prototype

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

Description

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

Input Parameters

thread_ptr	Pointer to a thread control block.
name_ptr	Pointer to the name of the thread.
entry_function	Specifies the initial C function for thread execution. When a thread returns from this entry function, it is placed in a completed state and suspended indefinitely.
entry_input	A 32-bit value that is passed to the thread's entry function when it first executes. The use for this input is determined exclusively by the application.
stack_start	Starting address of the stack's memory area.
stack_size	Number bytes in the stack memory area. The thread's stack area must be large enough to handle its worst-case function call nesting and local variable usage.
priority	Numerical priority of thread. Legal values range from 0 through (TX_MAX_PRIORITES-1), where a value of 0 represents the highest priority.
preempt_threshold	Highest priority level (0 through (TX_MAX_PRIORITIES-1)) of disabled preemption. Only priorities higher than this level are allowed to preempt this thread. This value must be less than or equal to the specified priority. A value equal to the thread priority disables preemption-threshold.
time_slice	Number of timer-ticks this thread is allowed to run before other ready threads of the same priority are given a chance to run. Note that using preemption-threshold disables time-slicing.Legal time-slice values range from 1

	to 0xFFFFFFFF (inclusive). A value of TX_NO_TIME_SLICE (a value of 0) disables time-slicing of this thread. *Using time-slicing results in a slight amount of system overhead. Since time-slicing is only useful in cases where multiple threads share the same priority, threads having a unique priority should not be assigned a time-slice.*
auto_start	Specifies whether the thread starts immediately or is placed in a suspended state. Legal options are TX_AUTO_START (0x01) and TX_DONT_START (0x00). If TX_DONT_START is specified, the application must later call tx_thread_resume in order for the thread to run.

Returns

TX_SUCCESS	(0x00)	Successful thread creation.
TX_THREAD_ERROR	(0x0E)	Invalid thread control pointer. Either the pointer is NULL or the thread is already created.
TX_PTR_ERROR	(0x03)	Invalid starting address of the entry point or the stack area is invalid, usually NULL.
TX_SIZE_ERROR	(0x05)	Size of stack area is invalid.Threads must have at least **TX_MINIMUM_STACK** bytes to execute.
TX_PRIORITY_ERROR	(0x0F)	Invalid thread priority, which is a value outside the range of (0 through (TX_MAX_PRIORITIES-1)).
TX_THRESH_ERROR	(0x18)	Invalid preemption-threshold specified. This value must be a valid priority less than or equal to the initial priority of the thread.
TX_START_ERROR	(0x10)	Invalid auto-start selection.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Initialization and threads

Preemption Possible

Yes

Example

```
TX_THREAD     my_thread;
UINT          status;
/* Create a thread of priority 15 whose entry point is "my_thread_entry". This thread's
   stack area is 1000 bytes in size, starting at address 0x400000. The

   preemption-threshold is setup to allow preemption of threads with priorities ranging from 0
   through 14. Time-slicing is disabled. This thread is automatically put into a ready condition.
   */
status = tx_thread_create(&my_thread, "my_thread_name", my_thread_entry, 0x1234,

              (VOID     *)     0x400000,     1000,     15,     15,
              TX_NO_TIME_SLICE, TX_AUTO_START);
/* If status equals TX_SUCCESS, my_thread is ready for execution! */
```

```
...

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

{
     /* When we get here, the value of initial_input is 0x1234. See how this was
        specified during creation. */
     /* The real work of the thread, including calls to other function should be called
        from here! */

     /* When this function returns, the corresponding thread is placed into a
        "completed" state. */
}
```

4.8.2 tx_thread_delete()

Delete application thread

Prototype

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

Description

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

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

Input Parameters

thread_ptr	Pointer to the previously created counting semaphore control block.

Returns

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

Allowed From

Threads and timers

Preemption Possible

No

Example

```
TX_THREAD    my_thread;
```

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

4.8.3 tx_thread_entry_exit_notify()

Notify application upon thread entry and exit

Prototype

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

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_THREAD     my_thread;
/* Register the "my_entry _exit_notify" function for monitoring the entry/exit of the thread
   "my_thread." */

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

                /* Thread exit!     */
    }
```

4.8.4 tx_thread_identify()

Retrieves pointer to currently executing thread

Prototype

TX_THREAD* **tx_thread_identify**(VOID)

Description

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

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

Input Parameters

None

Returns

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. */
```

4.8.5 tx_thread_info_get()

Retrieve information about thread

Prototype

UINT **tx_thread_info_get**(TX_THREAD ***thread_ptr** , CHAR ****name**, UINT ***state**,
ULONG ***run_count**,
UINT ***priority**,UINT ***preemption_threshold**, ULONG
***time_slice**,

TX_THREAD ****next_thread**,
TX_THREAD ****suspended_thread**)

Description

This service retrieves information about the specified thread.

Input Parameters

thread_ptr	Pointer to thread control block.
name	Pointer to destination for the pointer to the thread's name.
state	Pointer to destination for the thread's current execution state. Possible values are as follows: TX_READY (0x00) TX_COMPLETED (0x01) TX_TERMINATED (0x02) TX_SUSPENDED (0x03) TX_SLEEP (0x04) TX_QUEUE_SUSP (0x05) TX_SEMAPHORE_SUSP (0x06) TX_EVENT_FLAG (0x07) TX_BLOCK_MEMORY (0x08) TX_BYTE_MEMORY (0x09) TX_MUTEX_SUSP (0x0D)
run_count	Pointer to destination for the thread's run count.
priority	Pointer to destination for the thread's priority.
preemption_threshold	Pointer to destination for the thread's preemption-threshold.
time_slice	Pointer to destination for the thread's time-slice.
next_thread	Pointer to destination for next created thread pointer.
suspended_thread	Pointer to destination for pointer to next thread in suspension list.

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

Returns

TX_SUCCESS	(0x00)	Successful thread information retrieval.
TX_THREAD_ERROR	(0x0E)	Invalid thread control pointer.

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_THREAD    my_thread;
CHAR         *name;

UINT         state;

ULONG        run_count;
```

```
    UINT            priority;

    UINT            preemption_threshold;

    UINT            time_slice;

    TX_THREAD       *next_thread;

    TX_THREAD       *suspended_thread;

    UINT            status;
        /*  Retrieve information about the previously created thread "my_thread." */

        status = tx_thread_info_get(&my_thread, &name, &state, &run_count,

                                    &priority, &preemption_threshold, &time_slice,
                                    &next_thread,&suspended_thread);

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

4.8.6 tx_thread_performance_info_get()

Get thread performance information

Prototype

```
UINT tx_thread_performance_info_get(
        TX_THREAD   *thread_ptr,
        ULONG *resumptions, ULONG *suspensions,

       ULONG *solicited_preemptions ,
       ULONG *interrupt_preemptions,
       ULONG *priority_inversions, ULONG *time_slices,
       ULONG *relinquishes, ULONG *timeouts,
       ULONG *wait_aborts, TX_THREAD **last_preempted_by);
```

Description

This service retrieves performance information about the specified thread.

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

Input Parameters

thread_ptr	Pointer to previously created thread.
resumptions	Pointer to destination for the number of resumptions of this thread.

suspensions	Pointer to destination for the number of suspensions of this thread
solicited_preemptions	Pointer to destination for the number of preemptions as a result of a ThreadX API service call made by this thread.
interrupt_preemptions	Pointer to destination for the number of preemptions of this thread as a result of interrupt processing.
priority_inversions	Pointer to destination for the number of priority inversions of this thread.
time_slices	Pointer to destination for the number of time-slices of this thread.
relinquishes	Pointer to destination for the number of thread relinquishes performed by this thread.
timeouts	Pointer to destination for the number of suspension timeouts on this thread.
wait_aborts	Pointer to destination for the number of wait aborts performed on this thread.
last_preempted_by	Pointer to destination for the thread pointer that last preempted this thread.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_THREAD    my_thread;
ULONG        resumptions;

ULONG        suspensions;

ULONG        solicited_preemptions;

ULONG        interrupt_preemptions;

ULONG        priority_inversions;

ULONG        time_slices;

ULONG        relinquishes;

ULONG        timeouts;

ULONG        wait_aborts;

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

                    &wait_aborts, &last_preempted_by);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.8.7 tx_thread_performance_system_info_get()

Get thread system performance information

Prototype

UINT **tx_thread_performance_system_info_get**(ULONG *__resumptions__, ULONG *__suspensions__, ULONG *__solicited_preemptions__, ULONG *__interrupt_preemptions__, ULONG *__priority_inversions__, ULONG *__time_slices__, ULONG *__relinquishes__, ULONG *__timeouts__, ULONG *__wait_aborts__, ULONG *__non_idle_returns__,

ULONG *__idle_returns__);

Description

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

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

Input Parameters

resumptions	Pointer to destination for the total number of thread resumptions.
suspensions	Pointer to destination for the total number of thread suspensions.
solicited_preemptions	Pointer to destination for the total number of thread preemptions as a result of a thread calling a ThreadX API service.
interrupt_preemptions	Pointer to destination for the total number of thread preemptions as a result of interrupt processing.
priority_inversions	Pointer to destination for the total number of thread priority inversions.
time_slices	Pointer to destination for the total number of thread time-slices.
relinquishes	Pointer to destination for the total number of thread relinquishes.
timeouts	Pointer to destination for the total number of thread suspension timeouts.
wait_aborts	Pointer to destination for the total number of thread wait aborts.
non_idle_returns	Pointer to destination for the number of times a thread returns to the system when another thread is ready to execute.
idle_returns	Pointer to destination for the number of times a thread returns to the system when no other thread is ready to execute (idle system).

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG       resumptions;
ULONG       suspensions;

ULONG       solicited_preemptions;

ULONG       interrupt_preemptions;

ULONG       priority_inversions;

ULONG       time_slices;

ULONG       relinquishes;

ULONG       timeouts;

ULONG       wait_aborts;

ULONG       non_idle_returns;

ULONG       idle_returns;
/* Retrieve performance information on all previously created thread. */

status = tx_thread_performance_system_info_get(&resumptions, &suspensions,

            &solicited_preemptions,     &interrupt_preemptions,     &priority_inversions,
            &time_slices,   &relinquishes,   &timeouts,   &wait_aborts,   &non_idle_returns,
            &idle_returns);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.8.8 tx_thread_preemption_change()

Change preemption-threshold of application thread

Prototype

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

Description

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

Note:Using preemption-threshold disables time-slicing for the specified thread.

Input Parameters

thread_ptr	Pointer to a previously created application thread.
new_threshold	New preemption-threshold priority level (0 through (TX_MAX_PRIORITIES-1)).
old_threshold	Pointer to a location to return the previous preemption-threshold.

Returns

TX_SUCCESS	(0x00)	Successful preemption-threshold change.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_THRESH_ERROR	(0x18)	Specified new preemption-threshold is not a valid thread priority (a value other than (0 through (TX_MAX_PRIORITIES-1)) or is greater than (lower priority) than the current thread priority.
TX_PTR_ERROR	(0x03)	Invalid pointer to previous preemption-threshold storage location.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

Allowed From

Threads and timers

Preemption Possible

Yes

Example

```
TX_THREAD     my_thread;
UINT          my_old_threshold;

UINT          status;
/* Disable all preemption of the specified thread. The current preemption-threshold is
   returned in "my_old_threshold". Assume that "my_thread" has already been
   created. */

status =  tx_thread_preemption_change(&my_thread,

                                      0, &my_old_threshold);
/* If status equals TX _SUCCESS, the application thread is non-preemptable by another
   thread. Note that ISRs are not prevented by preemption disabling. */
```

4.8.9 tx_thread_priority_change()

Change priority of application thread

Prototype

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

Description

This service changes the priority of the specified thread. Valid priorities range from 0 through (TX_MAX_PRIORITES-1), where 0 represents the highest priority level.

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

Input Parameters

thread_ptr	Pointer to a previously created application thread.
new_priority	New thread priority level (0 through (TX_MAX_PRIORITIES-1)).
old_priority	Pointer to a location to return the thread's previous priority.

Returns

TX_SUCCESS	(0x00)	Successful priority change.
TX_THREAD_ERROR	(0x0E)	Invalid application thread pointer.
TX_PRIORITY_ERROR	(0x0F)	Specified new priority is not valid (a value other than (0 through (TX_MAX_PRIORITIES-1)).
TX_PTR_ERROR	(0x03)	Invalid pointer to previous priority storage location.
TX_CALLER_ERROR	(0x13)	Invalid caller of this service.

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. */
```

4.8.10 tx_thread_relinquish()

Relinquish control to other application threads

Prototype

VOID **tx_thread_relinquish**(VOID)

Description

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

Input Parameters

None

Return Values

None

Allowed From

Threads

Preemption Possible

Yes

Example

```
ULONG run_counter_1 =     0;
ULONG run_counter_2 =     0;
/* Example of two threads relinquishing control to each other in an infinite loop.
   Assume that

   both of these threads are ready and have the same priority. The run counters will
   always stay within one of each other. */
VOID my_first_thread(ULONG thread_input)
{
    /* Endless loop of relinquish. */ while(1)

    {
        /* Increment the run counter. */ run_counter_1++;
        /* Relinquish control to other thread. */ tx_thread_relinquish();

    }

}
VOID my_second_thread(ULONG thread_input)
{
    /* Endless loop of relinquish. */ while(1)
    {
        /* Increment the run counter. */ run_counter_2++;

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

    }

}
```

4.8.11 tx_thread_reset()

Reset thread

Prototype

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

Description

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

Note:The thread must be resumed for it to execute again.

Input Parameters

thread_ptr	Pointer to a previously created thread.

Returns

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

Allowed From

Threads

Example

```
TX_THREAD     my_thread;
/* Reset the previously created thread "my_thread." */
status = tx_thread_reset(&my_thread);
/* If status is TX_SUCCESS the thread is reset.      */
```

4.8.12 tx_thread_resume()

Resume suspended application thread

Prototype

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

Description

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

Input Parameters

thread_ptr	Pointer to a suspended application thread.

Returns

TX_SUCCESS	(0x00)	Successful thread resume.

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_THREAD    my_thread;
UINT         status;
/* Resume the thread represented by "my_thread". */
status = tx_thread_resume(&my_thread);
/* If status equals TX_SUCCESS, the application thread is now ready to execute. */
```

4.8.13 tx_thread_sleep()

Suspend current thread for specified time

Prototype

UINT **tx_thread_sleep**(ULONG **timer_ticks**)

Description

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

Input Parameters

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

Returns

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

Allowed From

Threads

Preemption Possible

Yes

Example

```
UINT status;
/*  Make the calling thread sleep for 100 timer-ticks. */

status =   tx_thread_sleep(100);
```

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

4.8.14 tx_thread_stack_error_notify()

Register thread stack error notification callback

Prototype

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

Description

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

Note:The ThreadX library must be built with TX_ENABLE_STACK_CHECKING defined in order for this service to return performance information.

Input Parameters

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

void my_stack_error_handler(TX_THREAD *thread_ptr);
/* Register the "my_stack_error_handler" function with ThreadX
so that thread stack errors can be handled by the application. */ status = **tx_thread_stack_error_notify**(my_stack_error_handler);
/* If status is TX_SUCCESS the stack error handler is registered.*/

4.8.15 tx_thread_suspend()

Suspend application thread

Prototype

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

Description

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

Note:If the specified thread is already suspended for another reason, this suspension is held

internally until the prior suspension is lifted. When that happens, this unconditional suspension of the specified thread is performed. Further unconditional suspension requests have no effect. After being suspended, the thread must be resumed by tx_thread_resume to execute again.

Input Parameters

thread_ptr	Pointer to an application thread.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_THREAD     my_thread;
UINT          status;
/* Suspend the thread represented by "my_thread". */
status = tx_thread_suspend(&my_thread);
/* If status equals TX_SUCCESS, the application thread is unconditionally suspended. */
```

4.8.16 tx_thread_terminate()

Terminates application thread

Prototype

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

Description

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

Note:After being terminated, the thread must be reset for it to execute again.

Input Parameters

thread_ptr	Pointer to application thread.

Returns

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

Allowed From

Threads and timers

Preemption Possible

Yes

Example

```
TX_THREAD     my_thread;
UINT          status;
/* Terminate the thread represented by "my_thread". */ status =
tx_thread_terminate(&my_thread);
/* If status equals TX _SUCCESS, the thread is terminated and cannot execute again
   until it is reset. */
```

4.8.17 tx_thread_time_slice_change

Changes time-slice of application thread

Prototype

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

Description

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

Note:Using preemption-threshold disables time-slicing for the specified thread.

Input Parameters

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

Returns

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

Allowed From

Threads and timers

Preemption Possible

No

Example

```
TX_THREAD     my_thread;
ULONG         my_old_time_slice;

UINT          status;
```

```
/* Change the time -slice of the thread associated with "my_thread" to 20. This will mean
   that "my_thread" can only run for 20 timer-ticks consecutively before other threads of
   equal or higher priority get a chance to run. */
status = tx_thread_time_slice_change(&my_thread, 20, &my_old_time_slice);
/* If status equals TX_ SUCCESS, the thread's time-slice has been changed to 20 and
   the previous time-slice is in "my_old_time_slice." */
```

4.8.18 tx_thread_wait_abort()

Abort suspension of specified thread

Prototype

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

Description

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

Note:This service does not release explicit suspension that is made by the tx_thread_suspend *service.*

Input Parameters

thread_ptr	Pointer to a previously created application thread.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

Yes

Example

```
TX_THREAD     my_thread;
UINT          status;
/* Abort the suspension condition of "my_thread." */
status = tx_thread_wait_abort(&my_thread);
/* If status equals TX_SUCCESS, the thread is now ready again, with a return value
   showing its suspension was aborted (TX_WAIT_ABORTED). */
```

4.9 Time Services

4.9.1 tx_time_get()

Retrieves the current time

Prototype

ULONG **tx_time_get**(VOID)

Description

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

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

Input Parameters

None

Return Values

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
ULONG current_time;
/* Pickup the current system time, in timer-ticks. */
current_time = tx_time_get();
/* Current time now contains a copy of the internal system clock. */
```

4.9.2 tx_time_set()

Sets the current time

Prototype

VOID **tx_time_set**(ULONG **new_time**)

Description

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

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

Input Parameters

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

Returns

None

Allowed From

Threads, timers, and ISRs

Preemption Possible

No

Example

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

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

4.10 Timer Services

4.10.1 tx_timer_activate()

Activate application timer

Prototype

UINT tx_timer_activate(TX_TIMER *timer_ptr)

Description

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

Input Parameters

timer_ptr	Pointer to a previously created application timer.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER        my_timer;
UINT            status;
/* Activate an application timer. Assume that the application timer has already
    been created. */
status =   tx_timer_activate(&my_timer);
/* If status equals TX_SUCCESS, the application timer is now active. */
```

4.10.2 tx_timer_change()

Change application timer

Prototype

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

Description

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

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

Input Parameters

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

Returns

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

Allowed From

Threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER        my_timer;
UINT            status;
/* Change a previously created and now deactivated timer to expire every 50 timer ticks,
   including the initial expiration. */
status =   tx_timer_change(&my_timer,50, 50);
/* If status equals TX_SUCCESS, the specified timer is changed to expire every 50
   ticks. */

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

4.10.3 tx_timer_create()

Create application timer

Prototype

UINT **tx_timer_create**(TX_TIMER * **timer_ptr**, CHAR ***name_ptr**,
VOID (***expiration_function**)(ULONG),

ULONG **expiration_input**, ULONG **initial_ticks**,
ULONG **reschedule_ticks**, UINT **auto_activate**)

Description

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

Input Parameters

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

Returns

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

Allowed From

Initialization and threads

Preemption Possible

No

Example

```
TX_TIMER        my_timer;
UINT            status;
```

```
/* Create an application timer that executes "my_timer_function" after 100 ticks initially
   and then after every 25 ticks. This timer is specified to start immediately! */

status = tx_timer_create(&my_timer,"my_timer_name", my_timer_function, 0x1234, 100, 25,
                         TX_AUTO_ACTIVATE);
/* If status equals TX_ SUCCESS, my_ timer_function will be called 100 timer ticks later
   and then called every
   25 timer ticks. Note that the value 0x1234 is passed to my_timer_function every time it
   is called. */
```

4.10.4 tx_timer_deactivate()

Deactivate application timer

Prototype

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

Description

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

Input Parameters

timer_ptr	Pointer to a previously created application timer.

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER      my_timer;
UINT          status;
/* Deactivate an application timer. Assume that the application timer has already
   been created. */
status =  tx_timer_deactivate(&my_timer);
/* If status equals TX_SUCCESS, the application timer is now deactivated. */
```

4.10.5 tx_timer_delete()

Delete application timer

Prototype

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

Description

This service deletes the specified application timer.

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

Input Parameters

timer_ptr	Pointer to a previously created application timer.

Returns

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

Allowed From

Threads

Preemption Possible

No

Example

```
TX_TIMER        my_timer;
UINT            status;
/* Delete application timer. Assume that the application timer has already been created.
   */
status =   tx_timer_delete(&my_timer);
/* If status equals TX_SUCCESS, the application timer is deleted. */
```

4.10.6 tx_timer_info_get()

Retrieve information about an application timer

Prototype

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

Description

This service retrieves information about the specified application timer.

Input Parameters

timer_ptr	Pointer to a previously created application timer.
name	Pointer to destination for the pointer to the timer's name.
active	Pointer to destination for the timer active indication. If the timer is inactive or this service is called from the timer itself, a TX_FALSE value is returned. Otherwise, if the timer is active, a TX_TRUE value is returned.
remaining_ticks	Pointer to destination for the number of timer ticks left before the timer expires.
reschedule_ticks	Pointer to destination for the number of timer ticks that will be used to automatically

	reschedule this timer. If the value is zero, then the timer is a one-shot and won't be rescheduled.
next_timer	Pointer to destination for the pointer of the next created application timer.

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Preemption Possible

No

Example

```
TX_TIMER        my_timer;
CHAR            *name;

UINT            active;

ULONG           remaining_ticks;

ULONG           reschedule_ticks;

TX_TIMER        *next_timer;

UINT            status;
/* Retrieve information about the previously created application timer "my_timer." */

status   =   tx_timer_info_get(&my_timer,   &name,   &active,&remaining_ticks,
                      &reschedule_ticks, &next_timer);
/* If status equals TX_SUCCESS, the information requested is valid. */
```

4.10.7 tx_timer_performance_info_get()

Get timer performance information

Prototype

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

Description

This service retrieves performance information about the specified application timer.

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

Input Parameters

timer_ptr	Pointer to previously created timer.
activates	Pointer to destination for the number of activation requests performed on this timer.
reactivates	Pointer to destination for the number of automatic reactivations performed on this periodic timer.
deactivates	Pointer to destination for the number of deactivation requests performed on this timer.
expirations	Pointer to destination for the number of expirations of this timer.
expiration_adjusts	Pointer to destination for the number of internal expiration adjustments performed on this timer. These adjustments are done in the timer interrupt processing for timers that are larger than the default timer list size (by default timers with expirations greater than 32 ticks).

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

Returns

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

Allowed From

Initialization, threads, timers, and ISRs

Example

```
TX_TIMER      my_timer;
ULONG         activates;

ULONG         reactivates;

ULONG         deactivates;

ULONG         expirations;

ULONG         expiration_adjusts;
/* Retrieve performance information on the previously created timer. */

status = tx_timer_performance_info_get(&my _timer, &activates, &reactivates,&deactivates,
              &expirations, &expiration_adjusts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```

4.10.8 tx_timer_performance_system_info_get()

Get timer system performance information

Prototype

UINT **tx_timer_performance_system_info_get**(ULONG *activates, ULONG *reactivates, ULONG *deactivates,

ULONG *expirations, ULONG *expiration_adjusts);

Description

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

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

Input Parameters

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

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

Returns

TX_SUCCESS	(0x00) Successful timer system performance get.
TX_FEATURE_NOT_ENABLED	(0xFF)The system was not compiled with performance Information enabled.

Allowed From

Initialization, threads, timers, and ISRs

Example

```
ULONG          activates;
ULONG          reactivates;

ULONG          deactivates;

ULONG          expirations;

ULONG          expiration_adjusts;
```

```
/* Retrieve performance information on all previously created timers. */

status = tx_timer_performance _system_info_get(&activates, &reactivates,
                    &deactivates, &expirations, &expiration_adjusts);
/* If status is TX_SUCCESS the performance information was successfully retrieved. */
```