Modules

Cortex-M7/IAR

with Memory Protection

User Guide

Express Logic, Inc.

858.613.6640
Toll Free 888.THREADX
FAX 858.521.4259

www.expresslogic.com

Part Number: 000-0021
Revision 5.8.2

Contents

Chapter 1

Overview

The ThreadX Module component provides an infrastructure for applications to dynamically load modules that are built separately from the resident portion of the application. This is especially useful in situations where the application code size exceeds available memory; or when new modules are required to be added after the core image is deployed; or when partial firmware updates are required.

Memory protection for the module is optional, based on the properties specified in the module preamble. In addition, the Module Manager can be configured to only accept memory protected modules. When memory protection is specified, the Cortex-M7 MPU hardware is configured such that all threads of the module are only allowed access to the module's code and data memory. Any extraneous memory access or execution will result in a memory fault and the offending module thread will be terminated. If the application registers a memory fault notification callback, this will also be called to alert the application of the memory fault.

The ThreadX Module component relies on the application to provide a memory area where modules can be loaded. The instruction area of each module may execute in place or be copied into the RAM module memory area for execution. In all cases, the module memory requirements are allocated from the module memory area.

There are no limits on the number of modules that can be loaded at the same time (aside from the amount of memory available), while there is only one copy of the core resident Module Manager code. The following illustrates the relationship of the Module Manager and the modules themselves:

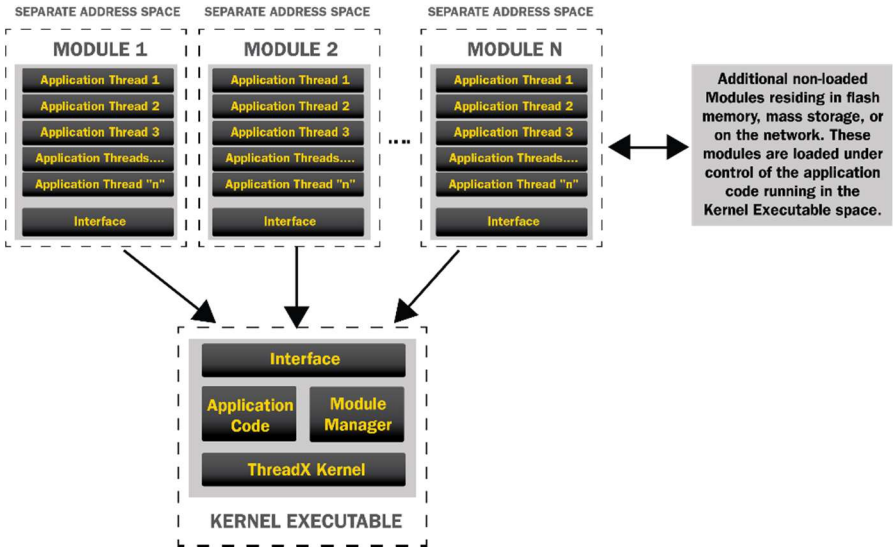

Figure 1.1

Each module must have its own instruction area and data area, which are the application's responsibility to define. The Module and the Module Manager interact through a software dispatch function via pre-defined request IDs that correspond to services requested by the module. In addition, the module is required to provide a single thread entry point as well as the required stack size, priority, module ID, callback thread stack size/priority, etc. This information is defined in each module's preamble.

The Module Manager is responsible for creating the initial module thread and initiating its execution. Once the module's initial thread is executing, the Module Manager is responsible for fielding all ThreadX API requests made by the module. A module has full access to the ThreadX API, including the ability to create additional threads within the module.

The module source code naming conventions are straightforward: all Module Manager source files are names ***txm_module_manager_**** and all files associated exclusively with the module omit the "***manager***" portion of the name. The main include file ***txm_module.h*** is shared by the manager and module source code.

Chapter 2

Module Requirements

ThreadX Module contains a preamble, which defines the basic characteristics of the module. The preamble is followed by the instruction area of the module. Modules may be executed in place or they may be loaded into the module memory area by the Module Manager prior to execution. The only requirement is that the preamble is always located at the first address of the module. Figure 2.1 illustrates a basic module layout.

[module preamble]

[module instruction area]

[module RAM area]

Figure 2.1

Note: *Modules must be built with the appropriate position independent code and data compiler/linker options. This enables execution of the module in any memory area.*

When a Module thread is created, a separate stack space is allocated for use when in the memory-protected kernel. The size of this stack space is user-configurable using ***TXM_MODULE_KERNEL_STACK_SIZE*** in txm_module_port.h. This allows a smaller stack size to be used when creating a Module thread, as the stack specified by the user when calling tx_thread_create is only used in the module.

Note: *The top of a module thread stack contains the thread entry information structure (TXM_MODULE_THREAD_ENTRY_INFO), so the stack size is decreased by the size of this structure. When creating a thread in a module, increase its stack size by at least this amount.*

The following steps are required for creating/building a ThreadX Module (each step is described in greater detail below):

1. All C files in a module must include ***TX_MODULE*** prior to including ***txm_module.h***. This can be accomplished in the source file being compiled or as part of the project settings.

Doing so remaps the ThreadX API calls to the module-specific version of the API that invokes the dispatch function in the resident Module Manager to perform the actual API.

2. Each module must have—at its first instruction area address—a preamble, which defines the characteristics and the resource needs of the module.

3. Each module must use a specific module linker control file. This linker control file must locate the preamble at the beginning of the module instruction area.

4. Each module must link against a module library (***txm.a***), which contains module-specific functions used to interact with ThreadX.

Module Sources

ThreadX modules have their own set of source files that are designed to be linked and located directly with the module source code. These files provide the bridge between the separate module and core resident Module Manager. The important module files are as follows:

File Name	Contents
txm_module.h	Include file that defines module request information.
txm_module_port.h	Include file that defines port-specific module information.
txm_module_preamble.s	Module preamble assembly file. This file defines various module-specific attributes and is generally linked with the module application code.
txm_module_application_request.c[1]	Module application request function sends an application-specific request to the resident code.
txm_module_callback_request_thread_entry.c[1]	Module callback thread that is responsible for processing callbacks requested by the module, including timers and notification callbacks.
txm_*.c[1,2]	The standard ThreadX API services, these call the kernel dispatcher.
txm_module_object_allocate.c[1]	Module function to allocate memory for module objects located in the manager memory pool.
txm_module_object_deallocate.c[1]	Module function to deallocate memory for module objects located in the manager memory pool.
txm_module_object_pointer_get.c[1]	Module function to retrieve a pointer to a system object.
txm_module_thread_shell_entry.c[1]	Module thread entry function.

***txm_module_thread_system_suspend.c*[1]** Module function to suspend a thread.

1 located in library txm.a.

2 these files have the same name as the ThreadX API files, with txm_ prefix instead of tx_ prefix.

Module Preamble

The Module Preamble defines characteristics and resources of the module. Information such as the initial thread entry function and the initial memory area associated with the thread are defined in the preamble. Figure 2.2 shows the ThreadX module preamble for a Cortex-M7 target using the IAR development tools (the values in **BOLD** are values typically modified by the application):

```
    SECTION .text:CODE

        AAPCS INTERWORK, ROPI, RWPI_COMPATIBLE,  VFP_COMPATIBLE
        PRESERVE8

    PUBLIC __txm_module_preamble

    EXTERN demo_module_start

    EXTERN  _txm_module_thread_shell_entry
    EXTERN  _txm_module_callback_request_thread_entry
    EXTERN  ROPI$$Length
    EXTERN  RWPI$$Length

    DATA
__txm_module_preamble:
    DC32      0x4D4F4455                                    ; Module ID
    DC32      0x5                                           ; Module Major Version
    DC32      0x8                                           ; Module Minor Version
    DC32      32                                            ; Module Preamble Size in 32-bit words
    DC32      0x12345678                                    ; Module ID (application defined)
    DC32      0x00000007                                    ; Module Properties where:
                                                            ;   Bits 31-24: Compiler ID
                                                            ;           0 -> IAR
                                                            ;           1 -> RVDS
                                                            ;           2 -> GNU
                                                            ;   Bits 23-3: Reserved
                                                            ;   Bit 2:  0 -> Disable shared/external memory access
                                                            ;           1 -> Enable shared/external memory access
                                                            ;   Bit 1:  0 -> No MPU protection
                                                            ;           1 -> MPU protection (must have user
                                                            ;                mode selected - bit 0 set)
                                                            ;   Bit 0:  0 -> Privileged mode execution
                                                            ;           1 -> User mode execution

    DC32      _txm_module_thread_shell_entry - . - 0        ; Module Shell Entry Point
    DC32      demo_module_start - . - 0                     ; Module Start Thread Entry Point
    DC32      0                                             ; Module Stop Thread Entry Point
    DC32      1                                             ; Module Start/Stop Thread Priority
    DC32      2046                                          ; Module Start/Stop Thread Stack Size
    DC32      _txm_module_callback_request_thread_entry - . - 0 ; Module Callback Thread Entry
    DC32      1                                             ; Module Callback Thread Priority
    DC32      2046                                          ; Module Callback Thread Stack Size
    DC32      ROPI$$Length                                  ; Module Code Size
    DC32      RWPI$$Length                                  ; Module Data Size
    DC32      0                                             ; Reserved 0
    DC32      0                                             ; Reserved 1
    DC32      0                                             ; Reserved 2
    DC32      0                                             ; Reserved 3
    DC32      0                                             ; Reserved 4
    DC32      0                                             ; Reserved 5
    DC32      0                                             ; Reserved 6
    DC32      0                                             ; Reserved 7
    DC32      0                                             ; Reserved 8
    DC32      0                                             ; Reserved 9
    DC32      0                                             ; Reserved 10
    DC32      0                                             ; Reserved 11
    DC32      0                                             ; Reserved 12
    DC32      0                                             ; Reserved 13
    DC32      0                                             ; Reserved 14
    DC32      0                                             ; Reserved 15
    END
```

Figure 2.2

In most cases, the developer only needs to define the module's starting thread (offset 0x1C)**,** module ID (offset 0x10), start/stop thread priority (offset 0x24), and start/stop thread stack size (offset 0x28). The demonstration above is setup such that the starting thread of the module

is ***demo_module_start***, the module ID is ***0x12345678***, and the starting thread has a priority of ***1***, and a stack size of ***2046*** bytes.

Some applications may optionally define a stopping thread, which is executed as the Module Manager stops the module. In addition, some applications might utilize the Module Properties field, defined as follows:

Module Properties Bit Map:

Bit	Meaning
0	0: Privileged mode execution 1: User mode execution
1	0: No MPU protection 1: MPU protection (must have user mode selected)
2	0: Disable shared/external memory access 1: Enable shared/external memory access
[23-3]	Reserved
[31-24]	Compiler ID 0: IAR 1: RVDS 2: GNU

Module Linker Control File

Building the module requires a special linker control file, which is simpler than the core resident linker control file. The module linker control file only needs to define where in memory the module must reside and ensure that the module preamble is the first item in the code area. The following is an example linker control file for a Cortex-M7 target using the IAR development tools:

```
/*###ICF### Section handled by ICF editor, don't touch! ****/
/*-Editor annotation file-*/
/* IcfEditorFile="$TOOLKIT_DIR$\config\ide\IcfEditor\a_v1_0.xml" */
/*-Specials-*/
define symbol __ICFEDIT_intvec_start__ = 0x0;
/*-Memory Regions-*/
define symbol __ICFEDIT_region_ROM_start__ = 0x080f0000;
define symbol __ICFEDIT_region_ROM_end__   = 0x080fffff;
define symbol __ICFEDIT_region_RAM_start__ = 0x64010000;
define symbol __ICFEDIT_region_RAM_end__   = 0x64020000;
/*-Sizes-*/
define symbol __ICFEDIT_size_cstack__   = 0;
define symbol __ICFEDIT_size_svcstack__ = 0;
define symbol __ICFEDIT_size_irqstack__ = 0;
define symbol __ICFEDIT_size_fiqstack__ = 0;
define symbol __ICFEDIT_size_undstack__ = 0;
define symbol __ICFEDIT_size_abtstack__ = 0;
define symbol __ICFEDIT_size_heap__     = 0x1000;
/**** End of ICF editor section. ###ICF###*/

define memory mem with size = 4G;
define region ROM_region   = mem:[from __ICFEDIT_region_ROM_start__   to
__ICFEDIT_region_ROM_end__];
define region RAM_region   = mem:[from __ICFEDIT_region_RAM_start__   to
__ICFEDIT_region_RAM_end__];

//define block CSTACK    with alignment = 8, size = __ICFEDIT_size_cstack__   { };
//define block SVC_STACK with alignment = 8, size = __ICFEDIT_size_svcstack__ { };
//define block IRQ_STACK with alignment = 8, size = __ICFEDIT_size_irqstack__ { };
```

```
//define block FIQ_STACK with alignment = 8, size = __ICFEDIT_size_fiqstack__ { };
//define block UND_STACK with alignment = 8, size = __ICFEDIT_size_undstack__ { };
//define block ABT_STACK with alignment = 8, size = __ICFEDIT_size_abtstack__ { };
define block HEAP      with alignment = 8, size = __ICFEDIT_size_heap__     { };

initialize by copy { readwrite };
do not initialize  { section .noinit };

//place at address mem:__ICFEDIT_intvec_start__ { readonly section .intvec };

define movable block ROPI with alignment = 4, fixed order
{
  ro object txm_module_preamble.o,
  ro,
  ro data
};

define movable block RWPI with alignment = 8, fixed order, static base
{
  rw,
  block HEAP
};

place in ROM_region   { block ROPI };
place in RAM_region   { block RWPI };
```

This linker control produces an ELF image for this module, with the instruction area starting at address 0x080F0000 and the data area starting at address 0x64010000. The module preamble, defined in the section ***txm_module_preamble*** is the first element located in the code area, which is at address 0x080F0000. Note that the addresses are arbitrary since the code and data access is done in a position independent manner.

Module ThreadX Library

Each module must link against a special, module-centric ThreadX library. This library provides access to ThreadX services in the resident code. Most of the access is accomplished via the txm_*.c files. The following is an example of the module access call for the ThreadX API ***tx_thread_relinquish*** (in txm_thread_relinquish.c):

```
(_txm_module_kernel_call_dispatcher)(TXM_THREAD_RELINQUISH_CALL, 0, 0, 0);
```

In this example, the function pointer supplied by the Module Manager is used to call the Module Manager dispatch function with the ID associated with the ***tx_thread_relinquish*** service.

Module Example

The following is an example of the standard ThreadX demonstration in the form of a module. The main differences between the standard ThreadX demonstration and the module demonstration are:

1. Replacement of ***tx_api.h*** with ***txm_module.h***
2. Addition of ***TXM_MODULE*** define prior to ***txm_module.h***
3. Replacement of ***main*** and ***tx_application_define*** with ***demo_module_start***

4. Declaring pointers to ThreadX objects rather than the objects themselves.

```
#define TXM_MODULE
#include "txm_module.h"

#define DEMO_STACK_SIZE         1024
#define DEMO_BYTE_POOL_SIZE     9120
#define DEMO_BLOCK_POOL_SIZE    100
#define DEMO_QUEUE_SIZE         100

/* Define the pool space in the bss section of the module. ULONG is used to
   get the word alignment.  */

ULONG                   demo_module_pool_space[DEMO_BYTE_POOL_SIZE / 4];

/* Define the ThreadX object control blocks...  */
TX_THREAD               *thread_0;
TX_THREAD               *thread_1;
TX_THREAD               *thread_2;
TX_THREAD               *thread_3;
TX_THREAD               *thread_4;
TX_THREAD               *thread_5;
TX_THREAD               *thread_6;
TX_THREAD               *thread_7;
TX_QUEUE                *queue_0;
TX_SEMAPHORE            *semaphore_0;
TX_MUTEX                *mutex_0;
TX_EVENT_FLAGS_GROUP    *event_flags_0;
TX_BYTE_POOL            *byte_pool_0;
TX_BLOCK_POOL           *block_pool_0;


/* Define the counters used in the demo application...  */
ULONG           thread_0_counter;
ULONG           thread_1_counter;
ULONG           thread_1_messages_sent;
ULONG           thread_2_counter;
ULONG           thread_2_messages_received;
ULONG           thread_3_counter;
ULONG           thread_4_counter;
ULONG           thread_5_counter;
ULONG           thread_6_counter;
ULONG           thread_7_counter;
ULONG           semaphore_0_puts;
ULONG           event_0_sets;
ULONG           queue_0_sends;

/* Define thread prototypes.  */

void    thread_0_entry(ULONG thread_input);
void    thread_1_entry(ULONG thread_input);
void    thread_2_entry(ULONG thread_input);
void    thread_3_and_4_entry(ULONG thread_input);
void    thread_5_entry(ULONG thread_input);
void    thread_6_and_7_entry(ULONG thread_input);

void  semaphore_0_notify(TX_SEMAPHORE *semaphore_ptr)
{
    if (semaphore_ptr == semaphore_0)
        semaphore_0_puts++;
}


void  event_0_notify(TX_EVENT_FLAGS_GROUP *event_flag_group_ptr)
{
    if (event_flag_group_ptr == event_flags_0)
        event_0_sets++;
}


void  queue_0_notify(TX_QUEUE *queue_ptr)
{
    if (queue_ptr == queue_0)
        queue_0_sends++;
}


/* Define the module start function.  */

void    demo_module_start(ULONG id)
{

CHAR    *pointer;

    /* Allocate all the objects. In MPU mode, modules cannot allocate control blocks within
       their own memory area so they cannot corrupt the resident portion of ThreadX by overwriting
       the control block(s).  */
    txm_module_object_allocate((void*)&thread_0, sizeof(TX_THREAD));
    txm_module_object_allocate((void*)&thread_1, sizeof(TX_THREAD));
    txm_module_object_allocate((void*)&thread_2, sizeof(TX_THREAD));
    txm_module_object_allocate((void*)&thread_3, sizeof(TX_THREAD));
```

```
txm_module_object_allocate((void*)&thread_4, sizeof(TX_THREAD));
txm_module_object_allocate((void*)&thread_5, sizeof(TX_THREAD));
txm_module_object_allocate((void*)&thread_6, sizeof(TX_THREAD));
txm_module_object_allocate((void*)&thread_7, sizeof(TX_THREAD));
txm_module_object_allocate((void*)&queue_0, sizeof(TX_QUEUE));
txm_module_object_allocate((void*)&semaphore_0, sizeof(TX_SEMAPHORE));
txm_module_object_allocate((void*)&mutex_0, sizeof(TX_MUTEX));
txm_module_object_allocate((void*)&event_flags_0, sizeof(TX_EVENT_FLAGS_GROUP));
txm_module_object_allocate((void*)&byte_pool_0, sizeof(TX_BYTE_POOL));
txm_module_object_allocate((void*)&block_pool_0, sizeof(TX_BLOCK_POOL));


/* Create a byte memory pool from which to allocate the thread stacks.  */
tx_byte_pool_create(byte_pool_0, "module byte pool 0", demo_module_pool_space, DEMO_BYTE_POOL_SIZE);

/* Put system definition stuff in here, e.g. thread creates and other assorted
   create information.  */

/* Allocate the stack for thread 0.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

/* Create the main thread.  */
tx_thread_create(thread_0, "module thread 0", thread_0_entry, 0,
        pointer, DEMO_STACK_SIZE,
        1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);


/* Allocate the stack for thread 1.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

/* Create threads 1 and 2. These threads pass information through a ThreadX
   message queue.  It is also interesting to note that these threads have a time
   slice.  */
tx_thread_create(thread_1, "module thread 1", thread_1_entry, 1,
        pointer, DEMO_STACK_SIZE,
        16, 16, 4, TX_AUTO_START);

/* Allocate the stack for thread 2.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

tx_thread_create(thread_2, "module thread 2", thread_2_entry, 2,
        pointer, DEMO_STACK_SIZE,
        16, 16, 4, TX_AUTO_START);

/* Allocate the stack for thread 3.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

/* Create threads 3 and 4.  These threads compete for a ThreadX counting semaphore.
   An interesting thing here is that both threads share the same instruction area.  */
tx_thread_create(thread_3, "module thread 3", thread_3_and_4_entry, 3,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

/* Allocate the stack for thread 4.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

tx_thread_create(thread_4, "module thread 4", thread_3_and_4_entry, 4,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

/* Allocate the stack for thread 5.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

/* Create thread 5.  This thread simply pends on an event flag which will be set
   by thread_0.  */
tx_thread_create(thread_5, "module thread 5", thread_5_entry, 5,
        pointer, DEMO_STACK_SIZE,
        4, 4, TX_NO_TIME_SLICE, TX_AUTO_START);

/* Allocate the stack for thread 6.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

/* Create threads 6 and 7.  These threads compete for a ThreadX mutex.  */
tx_thread_create(thread_6, "module thread 6", thread_6_and_7_entry, 6,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

/* Allocate the stack for thread 7.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_STACK_SIZE, TX_NO_WAIT);

tx_thread_create(thread_7, "module thread 7", thread_6_and_7_entry, 7,
        pointer, DEMO_STACK_SIZE,
        8, 8, TX_NO_TIME_SLICE, TX_AUTO_START);

/* Allocate the message queue.  */
tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_QUEUE_SIZE*sizeof(ULONG), TX_NO_WAIT);

/* Create the message queue shared by threads 1 and 2.  */
tx_queue_create(queue_0, "module queue 0", TX_1_ULONG, pointer, DEMO_QUEUE_SIZE*sizeof(ULONG));

tx_queue_send_notify(queue_0, queue_0_notify);

/* Create the semaphore used by threads 3 and 4.  */
tx_semaphore_create(semaphore_0, "module semaphore 0", 1);
```

```
    tx_semaphore_put_notify(semaphore_0, semaphore_0_notify);

    /* Create the event flags group used by threads 1 and 5.  */
    tx_event_flags_create(event_flags_0, "module event flags 0");

    tx_event_flags_set_notify(event_flags_0, event_0_notify);

    /* Create the mutex used by thread 6 and 7 without priority inheritance.  */
    tx_mutex_create(mutex_0, "module mutex 0", TX_NO_INHERIT);

    /* Allocate the memory for a small block pool.  */
    tx_byte_allocate(byte_pool_0, (VOID **) &pointer, DEMO_BLOCK_POOL_SIZE, TX_NO_WAIT);

    /* Create a block memory pool to allocate a message buffer from.  */
    tx_block_pool_create(block_pool_0, "module block pool 0", sizeof(ULONG), pointer, DEMO_BLOCK_POOL_SIZE);

    /* Allocate a block and release the block memory.  */
    tx_block_allocate(block_pool_0, (VOID **) &pointer, TX_NO_WAIT);

    /* Release the block back to the pool.  */
    tx_block_release(pointer);
}


/* Define the test threads.  */

void    thread_0_entry(ULONG thread_input)
{

UINT    status;


    /* This thread simply sits in while-forever-sleep loop.  */
    while(1)
    {

        /* Increment the thread counter.  */
        thread_0_counter++;

        /* Sleep for 10 ticks.  */
        tx_thread_sleep(10);

        /* Set event flag 0 to wakeup thread 5.  */
        status =  tx_event_flags_set(event_flags_0, 0x1, TX_OR);

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


void    thread_1_entry(ULONG thread_input)
{

UINT    status;


    /* This thread simply sends messages to a queue shared by thread 2.  */
    while(1)
    {

        /* Test memory handler - read protected memory.  */
        thread_1_counter = *(ULONG *)0x08000000;

        /* Increment the thread counter.  */
        thread_1_counter++;

        /* Send message to queue 0.  */
        status =  tx_queue_send(queue_0, &thread_1_messages_sent, TX_WAIT_FOREVER);

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

        /* Increment the message sent.  */
        thread_1_messages_sent++;
    }
}


void    thread_2_entry(ULONG thread_input)
{

ULONG   received_message;
UINT    status;

    /* This thread retrieves messages placed on the queue by thread 1.  */
    while(1)
    {
        /* Increment the thread counter.  */
        thread_2_counter++;

        /* Retrieve a message from the queue.  */
        status = tx_queue_receive(queue_0, &received_message, TX_WAIT_FOREVER);
```

```
        /* Check completion status and make sure the message is what we
           expected.  */
        if ((status != TX_SUCCESS) || (received_message != thread_2_messages_received))
            break;

        /* Otherwise, all is okay.  Increment the received message count.  */
        thread_2_messages_received++;
    }
}


void    thread_3_and_4_entry(ULONG thread_input)
{

UINT    status;


    /* This function is executed from thread 3 and thread 4.  As the loop
       below shows, these function compete for ownership of semaphore_0.  */
    while(1)
    {

        /* Increment the thread counter.  */
        if (thread_input == 3)
            thread_3_counter++;
        else
            thread_4_counter++;

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

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

        /* Sleep for 2 ticks to hold the semaphore.  */
        tx_thread_sleep(2);

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

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


void    thread_5_entry(ULONG thread_input)
{

UINT    status;
ULONG   actual_flags;


    /* This thread simply waits for an event in a forever loop.  */
    while(1)
    {
        /* Increment the thread counter.  */
        thread_5_counter++;

        /* Wait for event flag 0.  */
        status =  tx_event_flags_get(event_flags_0, 0x1, TX_OR_CLEAR,
                                            &actual_flags, TX_WAIT_FOREVER);

        /* Check status.  */
        if ((status != TX_SUCCESS) || (actual_flags != 0x1))
            break;
    }
}


void    thread_6_and_7_entry(ULONG thread_input)
{

UINT    status;


    /* This function is executed from thread 6 and thread 7.  As the loop
       below shows, these function compete for ownership of mutex_0.  */
    while(1)
    {

        /* Increment the thread counter.  */
        if (thread_input == 6)
            thread_6_counter++;
        else
            thread_7_counter++;

        /* Get the mutex with suspension.  */
        status =  tx_mutex_get(mutex_0, TX_WAIT_FOREVER);

        /* Check status.  */
        if (status != TX_SUCCESS)
```

```
            break;

        /* Get the mutex again with suspension.  This shows
           that an owning thread may retrieve the mutex it
           owns multiple times.  */
        status =  tx_mutex_get(mutex_0, TX_WAIT_FOREVER);

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

        /* Sleep for 2 ticks to hold the mutex.  */
        tx_thread_sleep(2);

        /* Release the mutex.  */
        status =  tx_mutex_put(mutex_0);

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

        /* Release the mutex again.  This will actually
           release ownership since it was obtained twice.  */
        status =  tx_mutex_put(mutex_0);

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

Building Modules in IAR

ThreadX modules have principally two project files for building in the IAR development environment, one for the module itself and one for the module library, which contains all the interface functions to the resident portion of the code. Figure 2.3 below shows an IAR workspace with a module project named ***demo_threadx_module*** and the module ThreadX library ***txm*** project.

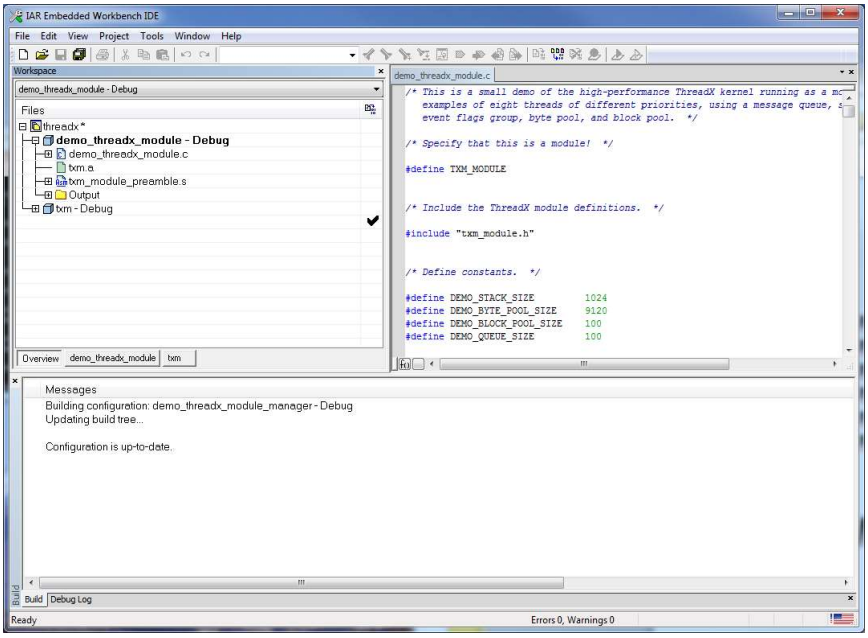

Figure 2.3

The first item to note is the contents of a typical ThreadX module, as shown within the ***demo_threadx_module*** project. Each module project is required to have the ***txm_module_preamble*** (setup specifically for the module) and the module library ***txm.a***. In addition, the module specific linker control file is defined in the linker dialog “Project -> Options -> Linker”. Figure 2.4 below shows the linker dialog for this example project.

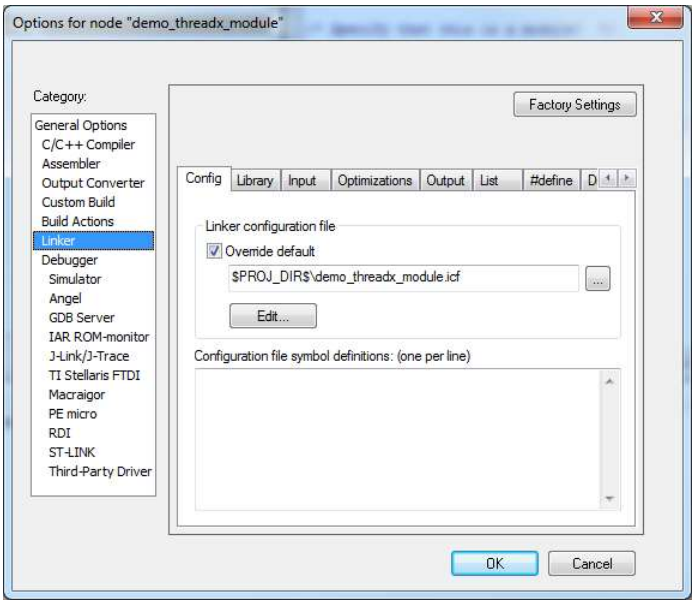

Figure 2.4

This example shows that the linker control file ***demo_threadx_module.icf*** is setup for this module. The linker control file is discussed later in this document but basically must ensure the module preamble is loaded first and the necessary information for position independent code and data is present. In addition to the linker control file, the module's C code must be complied in with the necessary position independent options, as shown in Figure 2.5.

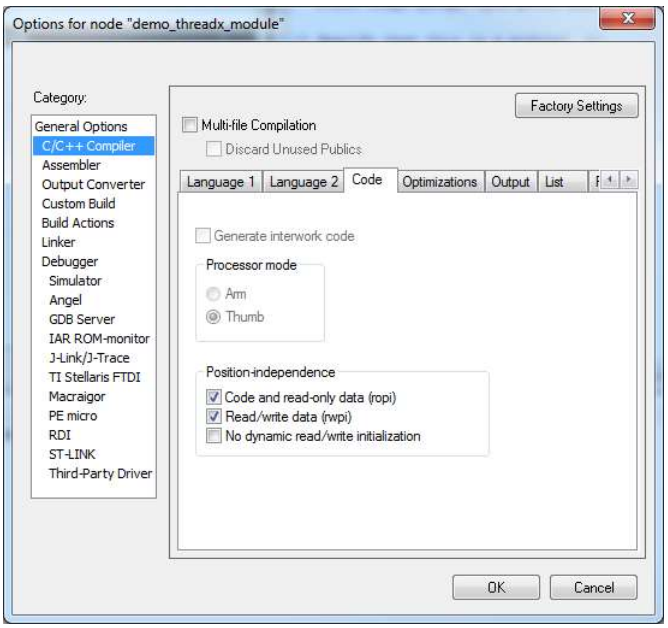

Figure 2.5

This dialog is selected via “Project -> Options -> C/C++ Compiler -> Code”. The two checked selections instruct the compiler to generate position independent code and data. Note that this should be selected for both the module project and the ***txm*** library project.

The final option instructs the IAR linker to create a binary image of the module. This image can be loaded by the Module Manager via FileX and the ***tmx_module_manager_file_load*** API. Figure 2.6 below shows the dialog to generate a binary image of the module.

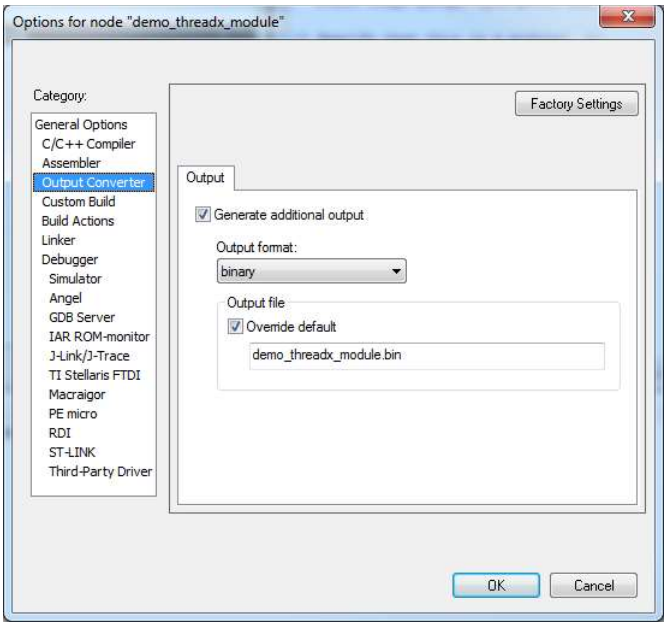

Figure 2.6

This dialog is selected via “Project -> Options -> Output Converter”. The selection of “binary” and the output file name ***demo_threadx_module.bin*** generates a binary image of the module that can be loaded by the Module Manager.

Chapter 3

Module Manager Requirements

The ThreadX Module Manager resides in the core resident portion of the application along with the ThreadX RTOS. It is responsible for starting the module as well as fielding and dispatching all module requests for ThreadX API services.

Note: *The ThreadX Module Manager source files (C and assembly) should be added to the ThreadX library project "**tx**".*

The following steps are required for building the ThreadX Module Manager (each step is described in greater detail below):

1. The ***TX_THREAD*** control block must be extended to include module information. The easiest way to accomplish this is to replace the definition of ***TX_THREAD_EXTENSION_2*** in the ***tx_port.h*** file, as follows:

```
#define TX_THREAD_EXTENSION_2               VOID    *tx_thread_module_instance_ptr;         \
                                            VOID    *tx_thread_module_entry_info_ptr;       \
                                            ULONG   tx_thread_module_current_user_mode;     \
                                            ULONG   tx_thread_module_user_mode;             \
                                            ULONG   tx_thread_module_saved_lr;              \
                                            VOID    *tx_thread_module_kernel_stack_start;   \
                                            VOID    *tx_thread_module_kernel_stack_end;     \
                                            ULONG   tx_thread_module_kernel_stack_size;     \
                                            VOID    *tx_thread_module_stack_ptr;            \
                                            VOID    *tx_thread_module_stack_start;          \
                                            VOID    *tx_thread_module_stack_end;            \
                                            ULONG   tx_thread_module_stack_size;            \
                                            VOID    *tx_thread_module_reserved;             \
                                            VOID    *tx_thread_iar_tls_pointer;
```

   The following extensions must also be defined in ***tx_port.h***:

```
#define TX_EVENT_FLAGS_GROUP_EXTENSION  VOID    *tx_event_flags_group_module_instance; \
                                        VOID
(*tx_event_flags_group_set_module_notify)(struct TX_EVENT_FLAGS_GROUP_STRUCT *group_ptr);

#define TX_QUEUE_EXTENSION              VOID    *tx_queue_module_instance; \
                                        VOID   (*tx_queue_send_module_notify)(struct
TX_QUEUE_STRUCT *queue_ptr);

#define TX_SEMAPHORE_EXTENSION          VOID    *tx_semaphore_module_instance; \
                                        VOID   (*tx_semaphore_put_module_notify)(struct
TX_SEMAPHORE_STRUCT *semaphore_ptr);

#define TX_TIMER_EXTENSION              VOID    *tx_timer_module_instance; \
                                        VOID   (*tx_timer_module_expiration_function)(ULONG
id);
```

2. Add all the ***txm_module_manager*.c***, ***tx_thread_schedule.s***, and ***txm_module_manager*.s*** files to the ThreadX library project ***tx***.

3. Rebuild all libraries and executable projects. Note that if NetX Duo is required, all Module and Module Manager C code should be built with ***TXM_MODULE_ENABLE_NETX_DUO*** defined.

   Also note the following MPU configuration constants in ***txm_module_port.h***:

```
#define TXM_MODULE_MPU_TOTAL_ENTRIES        16
#define TXM_MODULE_MPU_CODE_ENTRIES         4
#define TXM_MODULE_MPU_DATA_ENTRIES         4
#define TXM_MODULE_MPU_SHARED_ENTRIES       3
```

   There are 16 MPU entries for the Cortex-M7. One entry is used for the module to enter kernel mode. Four MPU entries are used for protecting the module code memory, and four MPU entries are used to protect the module data memory. Three entries are available for shared memory access. The remaining four entries are unused.

Module Manager Sources

The ThreadX Module Manager has a set of source files that are designed to be linked and located directly with the resident ThreadX code. These files provide the ability to launch the module and field subsequent ThreadX API requests from the module. The important module files are as follows:

File Name	Contents
txm_module.h	Include file that defines module request information (also included in the module source code).
txm_module_manager_dispatch.h	Include file that defines dispatch helper functions.
txm_module_port.h	Include file that defines port-specific module information (also included in the module source code).
tx_thread_schedule.s	Contains the trap processing code, which in this case is used to switch to privileged mode.
txm_module_manager_thread_stack_build.s	Builds all module initial stacks, includes setup for position independent data access
txm_module_manager_alignment_adjust.c	Handles port-specific alignment requirements.
txm_module_manager_application_request.c	Handles the application-specific requests of the resident code
txm_module_manager_callback_request.c	Sends a callback request to a module.
txm_module_manager_event_flags_notify_trampoline.c	Processes the event flags set notification call from ThreadX
txm_module_manager_external_memory_enable.c	Creates an entry in the MPU table for a shared memory space the module can access
txm_module_manager_file_load.c	Allocates and loads a binary module file into the module memory area and prepares it for execution
txm_module_manager_in_place_load.c	Allocates the module data area and prepares for module execution from the supplied code address
txm_module_manager_initialize.c	Initializes the Module Manager, including specification of the module memory area available for loading and running modules
txm_module_manager_kernel_dispatch.c	Handles the module API requests, based on the request ID
txm_module_manager_maximum_module_priority_set.c	Sets the maximum thread priority allowed in a module
txm_module_manager_memory_fault_handler.c	Handles memory faults detected

	in a thread of an executing module
txm_module_manager_memory_fault_notify.c	Registers an application notification callback whenever a memory fault occurs
txm_module_manager_memory_load.c	Allocates and loads the modules code and data and prepares the module for execution
txm_module_manager_object_pointer_get.c	Searches for the supplied object type and name, and if found, returns the object pointer
txm_module_manager_object_pool_create.c	Creates a pool of objects that application can allocate from that is outside the module's data area.
txm_module_manager_queue_notify_trampoline.c	Processes the queue notification call from ThreadX
txm_module_manager_semaphore_notify_trampoline.c	Processes the semaphore put notification call from ThreadX
txm_module_manager_setup_mpu_registers.c	Sets up MPU registers for the module based on where the code and data are loaded
txm_module_manager_start.c	Starts execution of a module
txm_module_manager_stop.c	Stops execution of a module
txm_module_manager_thread_create.c	Creates all module threads
txm_module_manager_thread_notify_trampoline.c	Processes the thread entry/exit notification call from ThreadX
txm_module_manager_thread_reset.c	Reset a module thread
txm_module_manager_timer_notify_trampoline.c	Processes timer expirations from ThreadX
txm_module_manager_unload.c	Unloads the module from the module memory area

Module Manager Initialization

The resident portion of the application is responsible for calling the Module Manager initialization function ***txm_module_manager_initialize***. This function sets up the internal structures for loading and unloading modules, including setting up the memory area used for allocating module memory.

Module Manager Loading

The Module Manager can load modules dynamically into the module memory from binary module files or from a module code section that is already present in the resident code area. In addition, the module manager can execute code in place, i.e., only the module data is allocated in the module memory and the code execution is done in place. The following Module Manager load APIs are available:

txm_module_manager_file_load
txm_module_manager_in_place_load
txm_module_manager_memory_load

The memory protected version of the Module Manager also makes sure that the module is loaded with the proper alignment and the Cortex-M7 MPU registers are setup properly for each module. When memory protection is enabled via the module preamble options, module memory access is restricted to the module code and data areas.

Module Manager Starting

The Module Manager initiates execution of a previously loaded module via the ***txm_module_manager_start*** API. To initiate module execution, this API creates a thread that enters the module at the starting location specified in the module preamble. The priority and stack size of this thread is also specified in the module preamble.

Module Manager Stopping

The Module Manager terminates execution of a previously loaded and executing module via the ***txm_module_manager_stop*** API. This API first terminates and deletes the initial starting thread. If the module preamble specifies a stop thread, this thread is created and executed. The Module Manager waits for a fixed period of time for the stop thread to complete. Once complete, all system resources created by the module are deleted

and the module is placed in a dormant state, from which it can be either restarted or unloaded.

Module Manager Unloading

The Module Manager unloads a previously loaded but not executing module via the ***txm_module_manager_unload*** API. This API releases all memory associated with the module, freeing it for use with another module in the future.

Module Manager Requests

Requests made by modules to the Module Manager are done via macros in ***txm_module.h*** that map all ThreadX calls to call the Module Manager dispatch function via a function pointer supplied to the module by the Module Manager.

Additional application specific services made via the module calling ***txm_module_application_request*** are handled by the same macro mechanism used for the ThreadX API. By default, this handling function in the Module Manager is empty and designed such that the application adds the necessary code to process the application-specific requests.

If the request is not implemented by the Module Manager, a value of ***TX_NOT_AVAILABLE*** error status is returned by the Module Manager. This error code is also returned if the module requests an operation that is outside the scope of the module's access. For example, a module is not allowed to create a timer with the timer control block or callback address outside of the module's code area.

Module Manager Example

The following is an example of Module Manager code that launches the example module previously defined in Chapter 2. It is assumed that the module is already loaded, presumably by the debugger, at ROM address 0x080F0000.

```
#include "tx_api.h"
#include "txm_module.h"

#define     DEMO_STACK_SIZE         1024

/* Define the ThreadX object control blocks...  */
TX_THREAD               module_manager;
TXM_MODULE_INSTANCE     my_module;

/* Define thread prototypes.  */
void    module_manager_entry(ULONG thread_input);
```

```c
/* Define main entry point.  */
int main()
{

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


/* Define what the initial system looks like.  */
void    tx_application_define(void *first_unused_memory)
{

CHAR    *pointer = (CHAR*)first_unused_memory;

    /* Create module manager thread.  */
    tx_thread_create(&module_manager, "Module Manager Thread", module_manager_entry, 0,
                     pointer, DEMO_STACK_SIZE,
                     1, 1, TX_NO_TIME_SLICE, TX_AUTO_START);

}

/* Define the module manager thread.  */
void  module_manager_entry(ULONG thread_input)
{

    /* Initialize the module manager with 64KB of RAM starting at address 0x64010000.  */
    txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

    /* Loop to let load, start, stop, unload the module repetitively.  */
    while (1)
    {

        /* Load the module that has its code area at address 0x080F0000.  */
        txm_module_manager_in_place_load(&my_module, "my module", (VOID *) 0x080F0000);

        /* Start the module.  */
        txm_module_manager_start(&my_module);

        /* Let the module run for a while....  */
        tx_thread_sleep(20000);

        /* Stop the module.  */
        txm_module_manager_stop(&my_module);

        /* Unload the module.  */
        txm_module_manager_unload(&my_module);
    }
}
```

Building Module Manager in IAR

The ThreadX Module Manager is effectively the same as a standard ThreadX workspace, which is effectively one or more application files linked together with the ThreadX library ***tx.a***. Figure 3.1 shows an example Module Manager project along with the ThreadX library project.

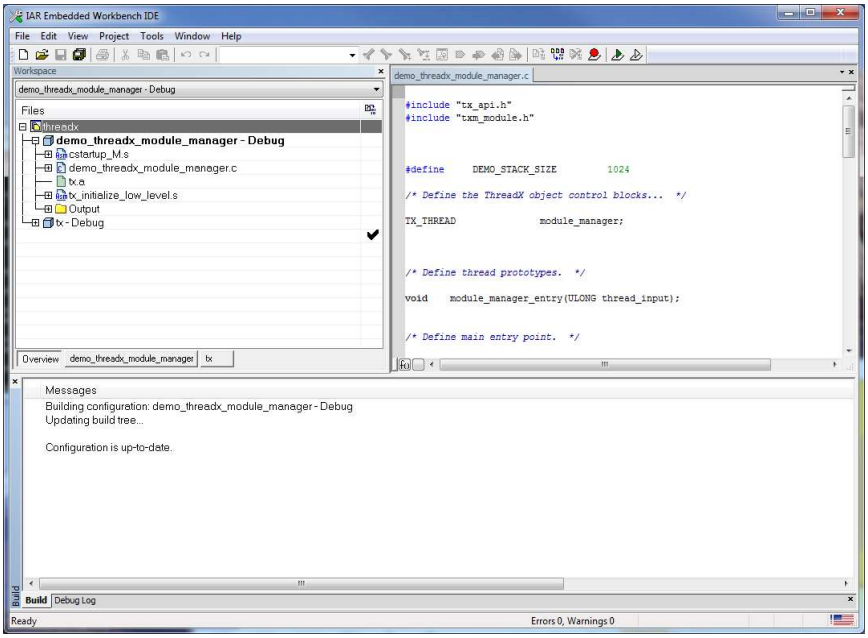

Figure 3.1

This example shows the Module Manager demonstration contained in the ***demo_threadx_module_manager*** project and is effectively just like any standard ThreadX application. The only additional option that is applicable to modules is the ability to load additional images into the debugger. The dialog for this is shown below in Figure 3.2.

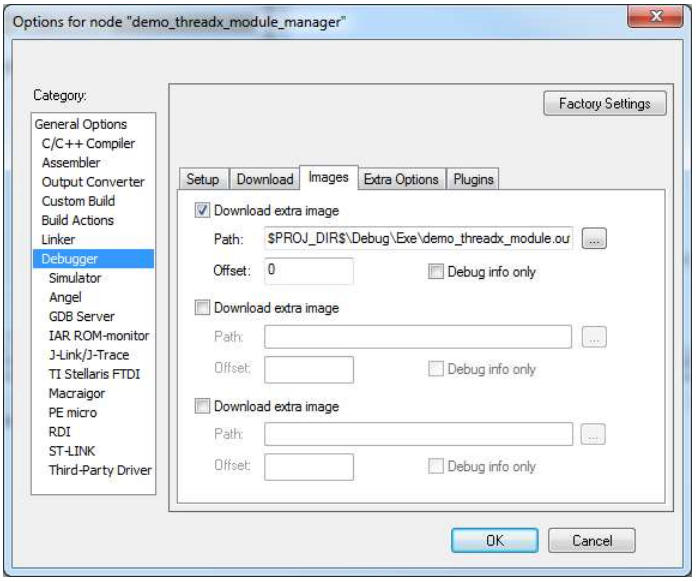

Figure 3.2

This dialog shows that the previously built module in the file ***demo_threadx_module.out*** is loaded at the same time as the Module Manager file, and because of the module's linker control file it will be loaded at address 0x080F0000. Note that this is also the same address specified in the previous Module Manager example.

The only other difference in Module Manager projects is the addition of the ***txm_module_manager_**** source files in the ***tx*** library project. These files should be added to the ***tx*** library project via the "***Add Files***" dialog, selected by right-clicking on the project. Figure 3.3 shows the Module Manager files added into the ***tx*** library project.


txm_module_manager_alignment_adjust.c
txm_module_manager_application_request.c
txm_module_manager_callback_request.c
txm_module_manager_event_flags_notify_trampoline.c
txm_module_manager_external_memory_enable.c
txm_module_manager_file_load.c
txm_module_manager_in_place_load.c
txm_module_manager_initialize.c
txm_module_manager_kernel_dispatch.c
txm_module_manager_maximum_module_priority_set.c
txm_module_manager_memory_fault_handler.c
txm_module_manager_memory_fault_notify.c
txm_module_manager_memory_load.c
txm_module_manager_object_pointer_get.c
txm_module_manager_object_pool_create.c
txm_module_manager_queue_notify_trampoline.c
txm_module_manager_semaphore_notify_trampoline.c
txm_module_manager_setup_mpu_registers.c
txm_module_manager_start.c
txm_module_manager_stop.c
txm_module_manager_thread_create.c
txm_module_manager_thread_notify_trampoline.c
txm_module_manager_thread_reset.c
txm_module_manager_thread_stack_build.s
txm_module_manager_timer_notify_trampoline.c
txm_module_manager_unload.c

Figure 3.3

Chapter 4

Module APIs

There are several additional APIs available to the module, as follows:

txm_module_application_request
Application-specific request to resident code

txm_module_object_allocate
Allocate memory outside of module for object

txm_module_object_deallocate
Deallocate previously allocated object memory

txm_module_object_pointer_get
Find system object and retrieve object pointer

Note that additional error codes are returned for some ThreadX and NetX APIs. These additional error codes are defined as follows:

TXM_MODULE_INVALID_MEMORY

(0xF4) Indicates the memory supplied by the module is invalid or is in an invalid location. For example, in memory protected versions, system control blocks are not allowed to be located in memory the module can access.

TXM_MODULE_INVALID_CALLBACK

(0xF5) Callback specified in the API is outside the range of the module's code and therefore is invalid.

txm_module_application_request

Application-specific request to resident code

Prototype

```
UINT txm_module_application_request(ULONG request, ULONG param_1,
                                    ULONG param_2, ULONG param_3);
```

Description

This service makes the specified request to the resident portion of the application. It is assumed that the request structure is prepared prior to the call. The actual processing of the request takes place in the resident code in the function ***_txm_module_manager_application_request***. By default, this function is left empty and is designed for the application to modify.

Input Parameters

Request	Request ID (application defined)
param_1	First parameter
param_2	Second parameter
param_3	Third parameter

Return Values

TX_SUCCESS	(0x00)	Successful request.
TX_NOT_AVAILABLE	(0xID)	Request not supported by resident code.

Allowed From

Module threads

Example

```
/* Call application resident code with ID=77 and the
   Parameters set to 1, 2, 3.  */
status =  txm_module_application_request(77, 1, 2, 3);

/* If status is TX_SUCCESS the request was successful.  */
```

See Also

txm_module_object_allocate, txm_module_object_deallocate, txm_module_object_pointer_get

txm_module_object_allocate

Allocate memory outside of module for object

Prototype

```
UINT txm_module_object_allocate(VOID **object_ptr, ULONG object_size);
```

Description

This service allocates memory for a module object from memory outside of the module, which helps prevent corruption of the object control block by the module's code. In memory protected systems, all object control blocks must be allocated with this API before they can be created.

Input Parameters

object_ptr	Destination of object pointer on successful allocation.
object_size	Size in bytes of the object to be allocated.

Return Values

TX_SUCCESS	(0x00)	Successful object allocate.
TX_NO_MEMORY	(0x10)	Not enough memory.
TX_NOT_AVAILABLE	(0x1D)	Module manager has not created an object pool to allocate from.

Allowed From

Module threads

Example

```
TX_QUEUE    *queue_pointer;

      /* Allocate a control block for a module message queue.  */
      status =  txm_module_object_allocate(&queue_pointer,
                                           sizeof(TX_QUEUE));

      /* If status is TX_SUCCESS the queue_pointer points to
            memory allocated outside of the module and can be
            supplied to tx_queue_create to create a queue for
            the module.  */
```

See Also

txm_module_application_request, txm_module_object_deallocate, txm_module_object_pointer_get

txm_module_object_deallocate

Deallocate previously allocated object memory

Prototype

```
UINT txm_module_object_deallocate(VOID *object_ptr);
```

Description

This service has been deprecated because it is no longer needed.

The memory that was previously allocated via ***txm_module_object_allocate*** is deallocated in the tx_*_delete service.

Input Parameters

object_ptr	Object pointer to deallocate.

Return Values

TX_SUCCESS	(0x00)	Successful object allocate.

Allowed From

Module threads

Example

```
TX_QUEUE    *queue_pointer;

      /* Deallocate control block for a module message queue.  */
      status =  txm_module_object_deallocate(queue_pointer);

      /* If status is TX_SUCCESS the object memory assocated ith
            queue_pointer is deallocated. */
```

See Also

txm_module_application_request, txm_module_object_allocate, txm_module_object_pointer_get

txm_module_object_pointer_get

Find system object and retrieve object pointer

Prototype

```
UINT txm_module_object_pointer_get(UINT object_type, CHAR *name,
                                   VOID **object_ptr);
```

Description

This service retrieves the object pointer of a particular type with a particular name. If the object is not found, an error is returned. Otherwise, if the object is found, the address of that object is placed in “object_ptr”. This pointer can then be used to make system service calls with in order to interact with the resident code and/or other loaded modules in the system.

Input Parameters

object_type Type of ThreadX object requested. Valid types are as follows:

TXM_BLOCK_POOL_OBJECT
TXM_BYTE_POOL_OBJECT
TXM_EVENT_FLAGS_OBJECT
TXM_MUTEX_OBJECT
TXM_QUEUE_OBJECT
TXM_SEMAPHORE_OBJECT
TXM_THREAD_OBJECT
TXM_TIMER_OBJECT
TXM_IP_OBJECT
TXM_PACKET_POOL_OBJECT
TXM_UDP_SOCKET_OBJECT
TXM_TCP_SOCKET_OBJECT

name Application-specific object name as defined when the object was created.

object_ptr Destination for object pointer.

Return Values

TX_SUCCESS	(0x00)	Successful object get.
TX_OPTION_ERROR	(0x08)	Invalid object type.
TX_PTR_ERROR	(0x03)	Invalid destination.
TX_SIZE_ERROR	(0x05)	Invalid size.
TX_NOT_DONE	(0x20)	Object not found.

Allowed From

Module threads

Example

```
TX_QUEUE    *queue_pointer;

      /* Find the pointer for "fft_queue" in the resident part
         of the application.  */
      status =  txm_module_object_pointer_get(TXM_QUEUE_OBJECT,
                                    "fft_queue", &queue_pointer);

      /* If status is TX_SUCCESS the found queue pointer is in
      "queue_pointer". This queue pointer can then be used to
      send messages to the "fft_queue." */
```

See Also

txm_module_application_request, txm_module_object_allocate, txm_module_object_deallocate

Chapter 5

Module Manager APIs

There are several additional APIs available to the resident portion of the application, as follows:

txm_module_manager_external_memory_enable
Enable module to access a shared memory space

txm_module_manager_file_load
Load module from file via FileX

txm_module_manager_in_place_load
Load module data only

txm_module_manager_initialize
Initialize the module manager

txm_module_manager_maximum_module_priority_set
Set the maximum thread priority allowed in a module

txm_module_manager_memory_fault_notify
Register an application callback on memory fault

txm_module_manager_memory_load
Load the module from memory

txm_module_manager_object_pool_create
Create an object pool for modules

txm_module_manager_start
Start execution of the specified module

txm_module_manager_stop
Stop execution of the specified module

txm_module_manager_unload
Unload the module

txm_module_manager_external_memory_enable

Enable module to access a shared memory space

Prototype

```
UINT txm_module_manager_external_memory_enable(
                                        TXM_MODULE_INSTANCE *module_instance,
                                        VOID *start_address, ULONG length,
                                        UINT attributes);
```

Description

This service creates an entry in the MPU table for a shared memory region that the module can access.

Input Parameters

module_instance Pointer to the instance of the module.

start_address Starting address of shared memory region.

length Length of shared memory region.

attributes Attributes of memory region (read only, write, etc).

Return Values

TX_SUCCESS (0x00) Successful MPU entry created.
TX_NOT_AVAILABLE (0x1D) Manager not initialized.
TX_PTR_ERROR (0x03) Invalid module instance.
TX_START_ERROR (0x10) Module not in loaded state.
TXM_MODULE_ALIGNMENT_ERROR
(0xF0) Invalid start address alignment.
TXM_MODULE_INVALID_PROPERTIES
(0xF3) Incompatible properties.

Allowed From

Initialization and threads

Example

```
  TXM_MODULE_INSTANCE    my_module;

  /* Initialize the module manager with 64KB of RAM starting at
     address 0x64010000.  */
  txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

  /* Load the module that has its code area at address
     0x080F0000.  */
  txm_module_manager_in_place_load(&my_module, "my module",
                                   (VOID *) 0x080F0000);

  /* Create a shared memory space 256 bytes long at address 0x64005000
     with read and write attributes.  */
  txm_module_manager_external_memory_enable(&my_module, (VOID *)
0x64005000, 256, TXM_MODULE_MANAGER_SHARED_ATTRIBUTE_WRITE);
```

See Also

txm_module_manager_file_load, txm_module_manager_initialize, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_file_load

Load module from file via FileX

Prototype

```
UINT txm_module_manager_file_load(TXM_MODULE_INSTANCE *module_instance,
            CHAR *module_name, FX_MEDIA *media_ptr, CHAR *file_name);
```

Description

This service loads the binary image of the module contained in the specified file into the module memory area and prepares it for execution. It is assumed that the supplied media is already opened.

Note: The FileX system is utilized to load the file. In order to enable FileX access, the module, module library, Module Manager and the ThreadX library (with the Module Manager sources) must be built with ***FX_FILEX_PRESENT*** *defined in the projects.*

Input Parameters

module_instance	Pointer to the instance of the module.
module_name	Name of the module.
media_ptr	Pointer to already opened FileX media.
file_name	Name of module's binary file.

Return Values

TX_SUCCESS	(0x00)	Successful module load.
TX_CALLER_ERROR	(0x13)	Invalid caller.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_NO_MEMORY	(0x10)	Not enough memory to load module.
TX_NOT_DONE	(0x20)	Media not open, file not found or file is invalid.
TX_PTR_ERROR	(0x03)	Invalid module pointer.
TXM_MODULE_ALIGNMENT_ERROR	(0xF0)	Invalid alignment.
TXM_MODULE_ALREADY_LOADED	(0xF1)	Module already loaded.
TXM_MODULE_INVALID	(0xF2)	Invalid module preamble.
TXM_MODULE_INVALID_PROPERTIES	(0xF3)	Incompatible properties.

Allowed From

threads

Example

```
TXM_MODULE_INSTANCE my_module;

/* Initialize the module manager.  */
status = txm_module_manager_initialize((VOID*)0x64010000,0x10000);

/* Load the module from a binary file.  */
status = txm_module_manager_file_load(&my_module, "my module",
                    &sdio_disk, "demo_thread_module.bin");

/* Start the module.  */
status = txm_module_manager_start(&my_module);
```

See Also

txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_in_place_load

Load module data only

Prototype

```
UINT txm_module_manager_in_place_load(TXM_MODULE_INSTANCE *module_instance,
                                      CHAR *module_name, VOID *location);
```

Description

This service loads the module's data area only into the module memory area and prepares it for execution. Module code execution will be in-place, i.e., from the address offset specified by the module preamble at the supplied location.

Input Parameters

module_instance	Pointer to the instance of the module.
module_name	Name of the module.
location	Pointer to module's code area, preamble first.

Return Values

TX_SUCCESS	(0x00)	Successful module load.
TX_CALLER_ERROR	(0x13)	Invalid caller.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_NO_MEMORY	(0x10)	Not enough memory to load module.
TX_PTR_ERROR	(0x03)	Invalid pointer, module instance, or module preamble.
TXM_MODULE_ALIGNMENT_ERROR	(0xF0)	Invalid alignment.
TXM_MODULE_ALREADY_LOADED	(0xF1)	Module already loaded.
TXM_MODULE_INVALID	(0xF2)	Invalid module preamble.
TXM_MODULE_INVALID_PROPERTIES	(0xF3)	Incompatible properties.

Allowed From

Initialization and threads

Example

```
TXM_MODULE_INSTANCE     my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Loop to let load, start, stop, unload the module repetitively.  */
while (1)
{

    /* Load the module that has its code area at address
       0x080F0000.  */
    txm_module_manager_in_place_load(&my_module, "my module",
                                     (VOID *) 0x080F0000);

    /* Start the module.  */
    txm_module_manager_start(&my_module);

    /* Let the module run for a while....  */
    tx_thread_sleep(20000);

    /* Stop the module.  */
    txm_module_manager_stop(&my_module);

    /* Unload the module.  */
    txm_module_manager_unload(&my_module);
 }
```

See Also

txm_module_manager_file_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_initialize

Initialize the module manager

Prototype

```
UINT txm_module_manager_initialize(VOID *module_memory_start,
                                   ULONG module_memory_size);
```

Description

This service initializes the Module Manager's internal resources, including the memory area used for loading modules.

Input Parameters

module_memory_start	Pointer to the start of module memory.
module_memory_size	Size in bytes of the module memory.

Return Values

TX_SUCCESS	(0x00)	Successful initialization.
TX_CALLER_ERROR	(0x13)	Invalid caller.

Allowed From

Initialization and Threads

Example

```
TXM_MODULE_INSTANCE     my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Loop to let load, start, stop, unload the module repetitively.  */
while (1)
{

    /* Load the module that has its code area at address
       0x080F0000.  */
    txm_module_manager_in_place_load(&my_module, "my module",
                                          (VOID *) 0x080F0000);

    /* Start the module.  */
    txm_module_manager_start(&my_module);

    /* Let the module run for a while....  */
    tx_thread_sleep(20000);

    /* Stop the module.  */
    txm_module_manager_stop(&my_module);

    /* Unload the module.  */
    txm_module_manager_unload(&my_module);
 }
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_maximum_module_priority_set

Set the maximum thread priority allowed in a module

Prototype

```
UINT txm_module_manager_maximum_module_priority_set(
                                    TXM_MODULE_INSTANCE *module_instance,
                                    UINT priority);
```

Description

This service sets the maximum thread priority allowed in a module.

Input Parameters

module_instance	Pointer to the instance of the module.
priority	Maximum thread priority.

Return Values

TX_SUCCESS	(0x00)	Successful initialization.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_PTR_ERROR	(0x03)	Invalid module instance.
TX_START_ERROR	(0x10)	Module not in loaded state.

Allowed From

Initialization and Threads

Example

```
TXM_MODULE_INSTANCE    my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Load the module that has its code area at address
   0x080F0000.  */
txm_module_manager_in_place_load(&my_module, "my module",
                                    (VOID *) 0x080F0000);

/* Set the maximum thread priority in my_module to 5.  */
txm_module_manager_maximum_module_priority_set(&my_module, 5);
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_memory_load, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_memory_fault_notify

Register an application callback on memory fault

Prototype

```
UINT txm_module_manager_memory_fault_notify(
            VOID (*notify_function)(TX_THREAD *, MODULE_INSTANCE *));
```

Description

This service registers the specified application memory fault notification callback function with the Module Manager. If a memory fault occurs, this function is called with a pointer to the offending thread and the module instance corresponding to the offending thread. The Module Manger processing automatically terminates the offending thread, but leaves any other threads in the module untouched. It is up to the application to decide what to do with the module associated with the memory fault.

Please see the internal **_txm_module_manager_memory_fault_info** for specific information on the memory fault itself.

Note that the memory fault notification callback function is executed directly from the memory fault exception, so only ThreadX APIs allowed from interrupt service routines can be called. Thus, in order to stop and unload the offending module, the application notification callback must send a signal to an application task so that the module can be stopped and unloaded.

Input Parameters

notify_function	Function pointer to the application's memory fault notification callback. Supplying a NULL, disables memory fault notification.

Return Values

TX_SUCCESS	(0x00)	Successful module load.

Allowed From

Initialization and threads

Example

```
/* Register a memory fault callback.  */
txm_module_manager_memory_fault_notify(my_memory_fault_handler);
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_memory_load

Load module from memory

Prototype

```
UINT txm_module_manager_memory_load(TXM_MODULE_INSTANCE *module_instance,
                                    CHAR *module_name, VOID *location);
```

Description

This service loads the module's code and data area only into the module memory area and prepares it for execution.

Input Parameters

module_instance	Pointer to the instance of the module.
module_name	Name of the module.
location	Pointer to module's code area, preamble first.

Return Values

TX_SUCCESS	(0x00)	Successful module load.
TX_CALLER_ERROR	(0x13)	Invalid caller.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_NO_MEMORY	(0x10)	Not enough memory to load module.
TX_PTR_ERROR	(0x03)	Invalid pointer, module instance, or module preamble.
TXM_MODULE_ALIGNMENT_ERROR	(0xF0)	Invalid alignment.
TXM_MODULE_ALREADY_LOADED	(0xF1)	Module already loaded.
TXM_MODULE_INVALID	(0xF2)	Invalid module preamble.
TXM_MODULE_INVALID_PROPERTIES	(0xF3)	Incompatible properties.

Allowed From

Initialization and threads

Example

```
TXM_MODULE_INSTANCE     my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Loop to let load, start, stop, unload the module repetitively.  */
while (1)
{

    /* Load the module that has its code area at address
       0x080F0000.  */
    txm_module_manager_memory_load(&my_module, "my module",
                                           (VOID *) 0x080F0000);

    /* Start the module.  */
    txm_module_manager_start(&my_module);

    /* Let the module run for a while....  */
    tx_thread_sleep(20000);

    /* Stop the module.  */
    txm_module_manager_stop(&my_module);

    /* Unload the module.  */
    txm_module_manager_unload(&my_module);
 }
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_object_pool_create

Create an object pool for modules

Prototype

```
UINT txm_module_manager_object_pool_create(VOID *pool_memory_start,
                                           ULONG pool_memory_size);
```

Description

This service creates a Module Manager object memory pool that the modules can allocate ThreadX/NetX object from, thereby keeping the system object out of the module's memory area.

Input Parameters

pool_memory_start	Pointer to the start of object memory.
pool_memory_size	Size in bytes of the object memory pool.

Return Values

TX_SUCCESS	(0x00)	Successful initialization.
TX_CALLER_ERROR	(0x13)	Invalid caller.

Allowed From

Initialization and Threads

Example

```
TXM_MODULE_INSTANCE    my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Create an object memory pool in the next 64KB of memory. */
txm_module_manager_object_pool_create((VOID *) 0x64020000, 0x10000);

/* Loop to let load, start, stop, unload the module repetitively.  */
while (1)
{

    /* Load the module that has its code area at address
       0x080F0000.  */
    txm_module_manager_in_place_load(&my_module, "my module",
                                     (VOID *) 0x080F0000);

    /* Start the module.  */
    txm_module_manager_start(&my_module);

    /* Let the module run for a while....  */
    tx_thread_sleep(20000);

    /* Stop the module.  */
    txm_module_manager_stop(&my_module);

    /* Unload the module.  */
    txm_module_manager_unload(&my_module);
 }
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_start, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_start

Start execution of the module

Prototype

```
UINT txm_module_manager_start(TXM_MODULE_INSTANCE *module_instance);
```

Description

This service starts execution of the specified, already loaded module.

Input Parameters

module_instance Pointer to previously loaded module instance.

Return Values

TX_SUCCESS	(0x00)	Successful module start.
TX_CALLER_ERROR	(0x13)	Invalid caller.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_PTR_ERROR	(0x03)	Invalid pointer or module instance.
TX_START_ERROR	(0x10)	Module already started.

Allowed From

Initialization and threads

Example

```
TXM_MODULE_INSTANCE     my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Loop to let load, start, stop, unload the module repetitively.  */
while (1)
{

    /* Load the module that has its code area at address
       0x080F0000.  */
    txm_module_manager_memory_load(&my_module, "my module",
                                        (VOID *) 0x080F0000);

    /* Start the module.  */
    txm_module_manager_start(&my_module);

    /* Let the module run for a while....  */
    tx_thread_sleep(20000);

    /* Stop the module.  */
    txm_module_manager_stop(&my_module);

    /* Unload the module.  */
    txm_module_manager_unload(&my_module);
 }
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_stop, txm_module_manager_unload

txm_module_manager_stop

Stop execution of the module

Prototype

```
UINT txm_module_manager_stop(TXM_MODULE_INSTANCE *module_instance);
```

Description

This service stops a module that was previously loaded and started. Stopping a module includes executing the module's optional stop thread, terminating all threads and deleting all resources associated with the module.

Input Parameters

module_instance	Pointer to module instance

Return Values

TX_SUCCESS	(0x00)	Successful module stop.
TX_CALLER_ERROR	(0x13)	Invalid caller.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_PTR_ERROR	(0x03)	Invalid pointer or module instance.
TX_START_ERROR	(0x10)	Module not started.

Allowed From

Threads

Example

```
TXM_MODULE_INSTANCE    my_module;

/* Initialize the module manager with 64KB of RAM starting at
   address 0x64010000.  */
txm_module_manager_initialize((VOID *) 0x64010000, 0x10000);

/* Loop to let load, start, stop, unload the module repetitively.  */
while (1)
{

    /* Load the module that has its code area at address
       0x080F0000.  */
    txm_module_manager_memory_load(&my_module, "my module",
                                        (VOID *) 0x080F0000);

    /* Start the module.  */
    txm_module_manager_start(&my_module);

    /* Let the module run for a while....  */
    tx_thread_sleep(20000);

    /* Stop the module.  */
    txm_module_manager_stop(&my_module);

    /* Unload the module.  */
    txm_module_manager_unload(&my_module);
 }
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_unload

txm_module_manager_unload

Unload the module

Prototype

```
UINT txm_module_manager_unload(TXM_MODULE_INSTANCE *module_instance);
```

Description

This service unloads the previously loaded and stopped module, freeing all the associated module memory resources.

Input Parameters

module_instance Pointer to the instance of the module.

Return Values

TX_SUCCESS	(0x00)	Successful module unload.
TX_CALLER_ERROR	(0x13)	Invalid caller.
TX_NOT_AVAILABLE	(0x1D)	Manager not initialized.
TX_NOT_DONE	(0x20)	Invalid module or module not stopped.
TX_PTR_ERROR	(0x03)	Invalid pointer or module instance.

Allowed From

Initialization and threads

Example

```
TXM_MODULE_INSTANCE my_module;

/* Unload the module.  */
status = txm_module_manager_unload(&my_module);

/* If status is TX_SUCCESS, the module is unloaded.  */
```

See Also

txm_module_manager_file_load, txm_module_manager_in_place_load, txm_module_manager_initialize, txm_module_manager_memory_fault_notify, txm_module_manager_memory_load, txm_module_manager_object_pool_create, txm_module_manager_start, txm_module_manager_stop