

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](http://www.expresslogic.com/)

**©1997-2018 by Express Logic, Inc.**

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

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

**Trademarks**

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

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

**Warranty Limitations**

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

Part Number: 000-0021

Revision 5.8.2

# Contents

[Contents 3](#_TOC_250032)

Chapter 1 Overview 4

Chapter 2 Module Requirements 6

[Module Sources 7](#_TOC_250031)

[Module Preamble 9](#_TOC_250030)

[Module Properties Bit Map 10](#_TOC_250029)

[Module Linker Control File 10](#_TOC_250028)

[Module ThreadX Library 11](#_TOC_250027)

[Module Example 11](#_TOC_250026)

[Building Modules in IAR 17](#_TOC_250025)

Chapter 3 Module Manager Requirements 20

[Module Manager Sources 22](#_TOC_250024)

[Module Manager Initialization 24](#_TOC_250023)

[Module Manager Loading 24](#_TOC_250022)

[Module Manager Starting 24](#_TOC_250021)

[Module Manager Stopping 24](#_TOC_250020)

[Module Manager Unloading 25](#_TOC_250019)

[Module Manager Requests 25](#_TOC_250018)

[Module Manager Example 25](#_TOC_250017)

[Building Module Manager in IAR 27](#_TOC_250016)

[Chapter 4 Module APIs 30](#_TOC_250015)

[txm\_module\_application\_request 31](#_TOC_250014)

[txm\_module\_object\_allocate 33](#_TOC_250013)

[txm\_module\_object\_deallocate 35](#_TOC_250012)

[txm\_module\_object\_pointer\_get 37](#_TOC_250011)

Chapter 5 Module Manager APIs 39

[txm\_module\_manager\_external\_memory\_enable 40](#_TOC_250010)

[txm\_module\_manager\_file\_load 42](#_TOC_250009)

[txm\_module\_manager\_in\_place\_load 44](#_TOC_250008)

[txm\_module\_manager\_initialize 46](#_TOC_250007)

[txm\_module\_manager\_maximum\_module\_priority\_set 48](#_TOC_250006)

[txm\_module\_manager\_memory\_fault\_notify 50](#_TOC_250005)

[txm\_module\_manager\_memory\_load 52](#_TOC_250004)

[txm\_module\_manager\_object\_pool\_create 54](#_TOC_250003)

[txm\_module\_manager\_start 56](#_TOC_250002)

[txm\_module\_manager\_stop 58](#_TOC_250001)

[txm\_module\_manager\_unload 60](#_TOC_250000)

**Chapter 1**

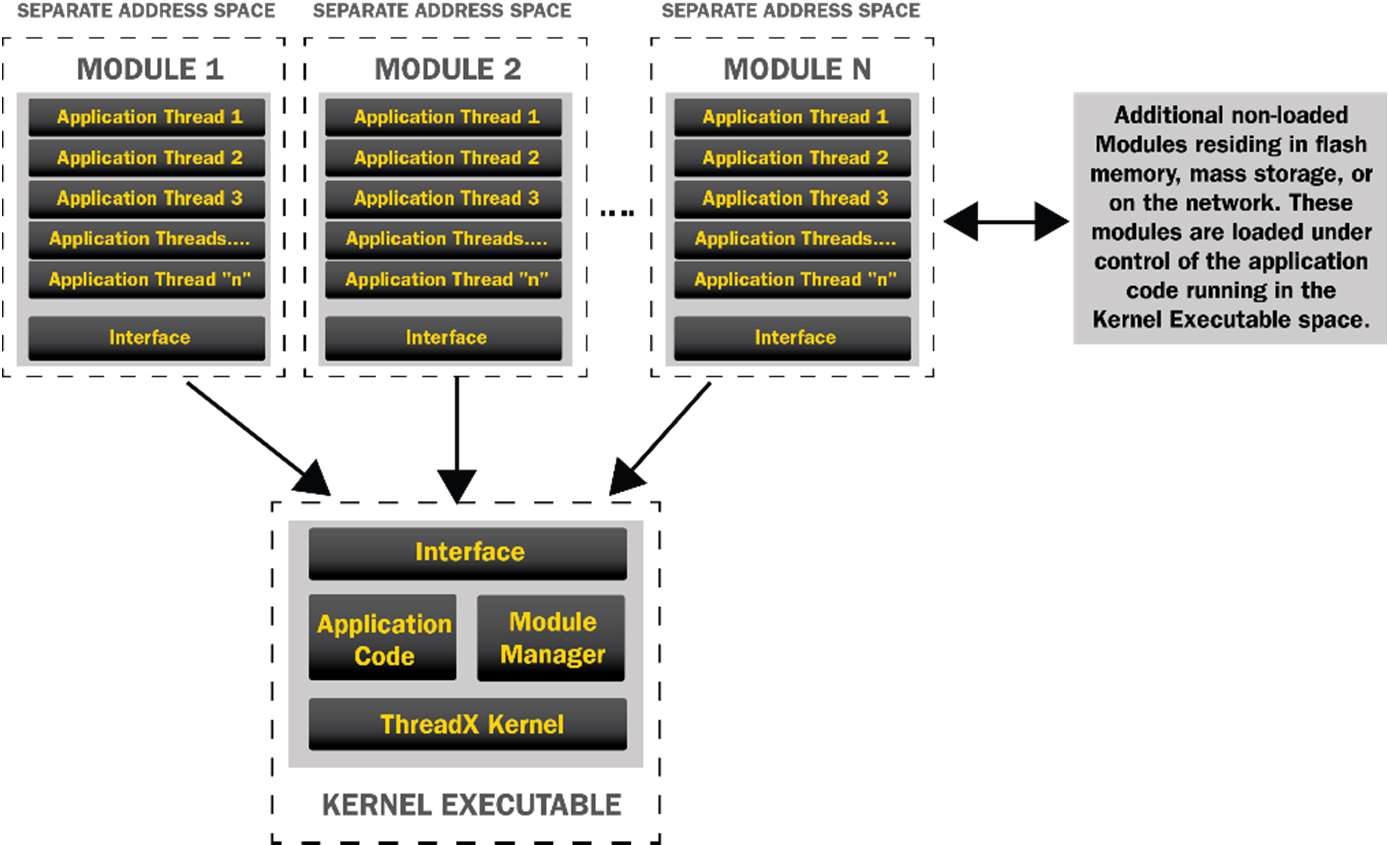
**Overview**

ThreadX模块组件提供了基础设施，用于应用程序动态加载与驻留部分独立构建的模块。这在应用程序代码大小超过可用内存的情况下特别有用;或者在核心映像部署后需要添加新模块时;或者当需要部分固件更新时。

基于模块preamble中指定的属性，模块的内存保护是可选的。此外，模块管理器可以配置为只接受内存保护的模块。当内存保护被指定时，Cortex-M7 MPU限制模块的所有线程只允许访问模块的代码和数据内存。任何额外的内存访问或执行都将导致内存故障，并且违规的模块线程将被终止。如果应用程序注册一个内存错误通知回调，它也将被调用来提醒应用程序内存错误。

ThreadX模块组件依赖于应用程序提供可以加载模块的内存区域。每个模块的指令区可以就地执行，也可以复制到RAM模块内存区执行。在所有情况下，模块内存需求都是从模块内存区域分配的。

对于同时可以加载的模块的数量没有限制(除了可用的内存量之外)，而模块管理器的代码只有一个副本。下面说明模块管理器与模块本身的关系:



**Figure 1.1**

每个模块必须有自己的指令区和数据区，由应用程序负责定义。模块有预先定义请求id，对应着模块请求的服务，模块管理器有软件分派机制，根据请求id和模块交互。此外，模块需要提供一个线程入口点，以及所需的堆栈大小、优先级、模块ID、回调线程堆栈大小/优先级等。这些信息在每个模块的preamble中定义。

模块管理器负责创建并执行模块的初始线程。模块的初始线程执行后，模块管理器负责处理模块发出的所有ThreadX API请求。模块具有对ThreadX API的完全访问权，包括在模块中创建额外线程的能力。

模块源代码的命名约定很简单:所有模块管理器源文件的名称是txm\_module\_manager\_\*，所有专属于模块的文件都省略了名称中的“Manager”部分。模块管理器和模块源代码共用一个主头：txm\_module.h。

**Chapter 2**

**模块要求**

ThreadX模块包含一个preamble，它定义了模块的基本特征。序言后面是模块的指令区。模块可以在适当的地方执行，也可以在执行之前由模块管理器加载到模块内存区域。唯一的要求是序言总是位于模块的第一个地址。图2.1给出了一个基本的模块布局.

[module preamble] [module instruction area] [module RAM area] **Figure 2.1**

注意:模块必须以ropi和rwpi的方式构建。这使得模块可以在任何内存区域中执行。

在创建模块线程时，将分配一个单独的堆栈空间，以便在内存保护的内核中使用。这个堆栈空间的大小是用户使用TXM\_MODULE\_KERNEL\_STACK\_SIZE在txm\_module\_port.h中配置的。这允许在创建模块线程时使用更小的堆栈大小，因为用户在调用tx\_thread\_create时指定的堆栈只在模块中使用。

注意:模块线程堆栈的顶部包含线程入口信息结构(TXM\_MODULE\_THREAD\_ENTRY\_INFO)，因此堆栈的大小会根据该结构的大小来减小。在模块中创建线程时，至少将其堆栈大小增加这个量。

创建/构建ThreadX模块需要以下步骤(每个步骤在下面详细描述):

1. 模块中的所有C文件必须在包含txm\_module.h之前包含TX\_MODULE。这可以在正在编译的源文件中完成，也可以作为项目设置的一部分完成。以便将ThreadX API调用重新映射到调用常驻模块管理器中的分派函数，以执行各API的特定于模块的版本。

2. 每个模块在其第一个指令区地址处必须有一个序言，它定义模块的特征和资源需求。

3. 每个模块必须使用一个特定的模块链接器控制文件。此链接器控制文件必须在模块指令区域的开始处找到序言。

4. 每个模块必须链接到一个模块库(txm.a)，该库包含用于与ThreadX交互的特定于模块的函数。

## 模块源文件

ThreadX模块有自己的源文件集，这些文件被设计成与模块源代码直接链接和定位。这些文件提供了独立模块和核心驻留模块管理器之间的桥梁。重要模块文件如下:

**File Name Contents**

***txm\_module.h*** defines module request information.

***txm\_module\_port.h*** port-specific module information.

***txm\_module\_preamble.s*** 模块preamble汇编文件，定义了各种特定于模块的属性，通常与模块应用程序代码链接。

***txm\_module\_application\_request.c1***

模块应用程序请求功能，发送应用相关的请教到常驻代码。

***txm\_module\_callback\_request\_thread\_entry.c1***

模块回调线程，负责处理模块请求的回调，包括计时器和通知回调

***txm\_\*.c1,2*** 标准ThreadX API服务，调用内核dispatcher.

***txm\_module\_object\_allocate.c1*** 在模块管理器的内存池中为模块分配内存。

***txm\_module\_object\_deallocate.c1*** 释放模块分配的内存

***txm\_module\_object\_pointer\_get.c1*** 获取指向系统对象的指针.

***txm\_module\_thread\_shell\_entry.c1*** 模块线程主入口。

***txm\_module\_thread\_system\_suspend.c1*** 挂起一个线程

txm\_\*.c编译成库”txm.a”。它们的名字与ThreadX API一一对应。内部调用一个转接函数。

## 模块Preamble

模块序言定义了模块的特征和资源。诸如初始线程入口函数和与线程关联的初始内存区域等信息在序言中定义。

图2.2显示了使用IAR开发工具对Cortex-M7目标的ThreadX模块的序言(粗体显示的值通常是由应用程序修改的值):

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 DC32 DC32  DC32 | 2046  ROPI$$Length RWPI$$Length  0 |  | ; Module Callback Thread Stack Size  ; Module Code Size  ; Module Data Size  ; 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 END | 0 |  | ; Reserved 15 |
|  |  | **Figure 2.2** |  |

在大多数情况下，开发人员只需要定义模块的起始线程(偏移0x1C)、模块ID(偏移0x10)、启动/停止线程优先级(偏移0x24)和启动/停止线程堆栈大小(偏移0x28)。上面的演示是这样设置的:模块的启动线程为demo\_module\_start，模块ID为0x12345678，启动线程的优先级为1，堆栈大小为2046字节。

一些应用程序可以选择定义一个停止线程，该线程在模块管理器停止模块时执行。此外，一些应用程序可能使用模块属性字段，定义如下:

## :

**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 define symbol define symbol define symbol

/\*-Sizes-\*/

ICFEDIT\_region\_ROM\_start ICFEDIT\_region\_ROM\_end ICFEDIT\_region\_RAM\_start ICFEDIT\_region\_RAM\_end

= 0x080f0000;

= 0x080fffff;

= 0x64010000;

= 0x64020000;

define symbol define symbol define symbol define symbol define symbol define symbol define symbol

ICFEDIT\_size\_cstack ICFEDIT\_size\_svcstack ICFEDIT\_size\_irqstack ICFEDIT\_size\_fiqstack ICFEDIT\_size\_undstack ICFEDIT\_size\_abtstack ICFEDIT\_size\_heap

= 0;

= 0;

= 0;

= 0;

= 0;

= 0;

= 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 =

//define block SVC\_STACK with alignment = 8, size =

//define block IRQ\_STACK with alignment = 8, size =

ICFEDIT\_size\_cstack ICFEDIT\_size\_svcstack ICFEDIT\_size\_irqstack

{ };

{ };

{ };

//define block FIQ\_STACK with alignment = 8, size =

//define block UND\_STACK with alignment = 8, size =

//define block ABT\_STACK with alignment = 8, size = define block HEAP with alignment = 8, size =

initialize by copy { readwrite };

do not initialize { section .noinit };

ICFEDIT\_size\_fiqstack ICFEDIT\_size\_undstack ICFEDIT\_size\_abtstack

ICFEDIT\_size\_heap

{ };

{ };

{ };

{ };

//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***

* 1. 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. \*/

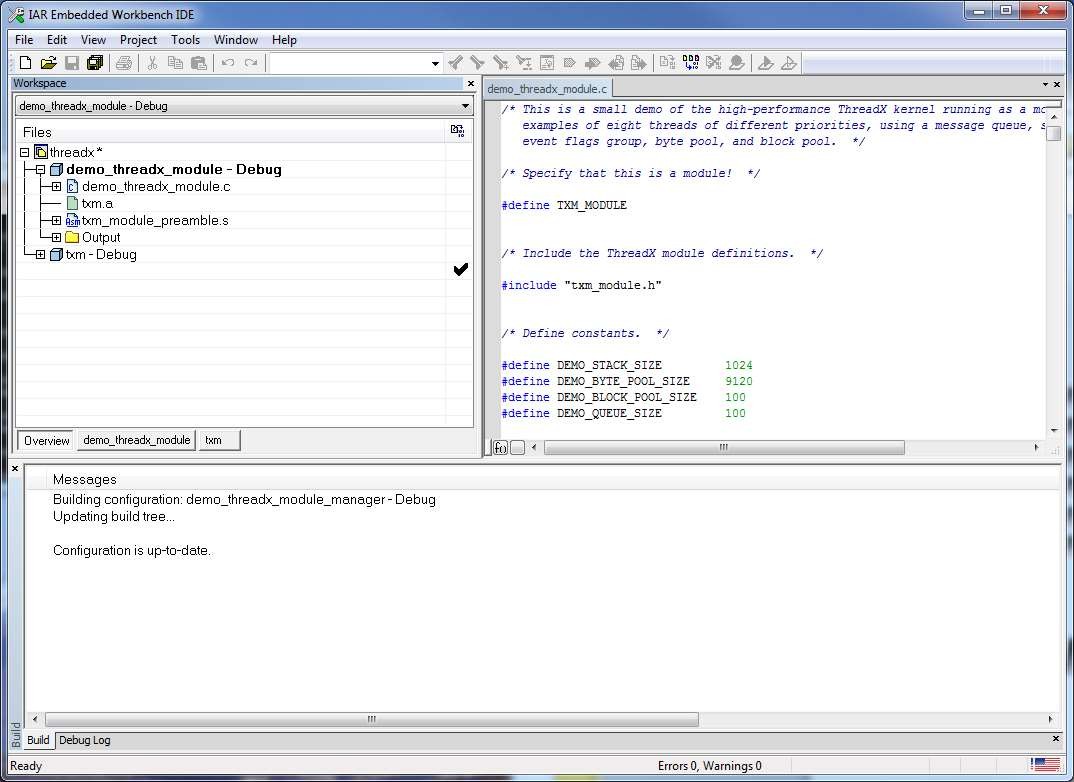
if (status != TX\_SUCCESS) break;

}

}

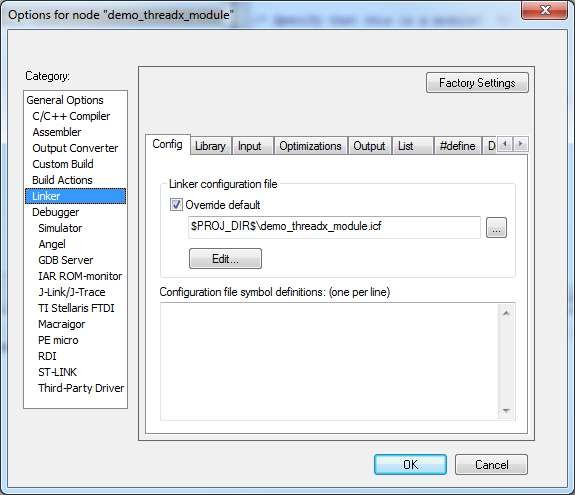
## Building Modules in IAR

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.



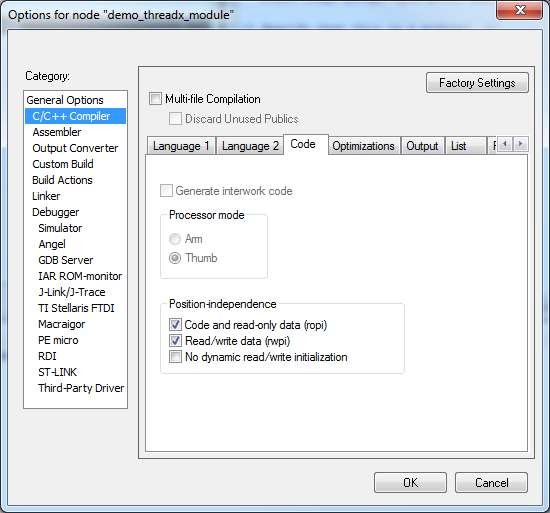
**Figure 2.3**

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**

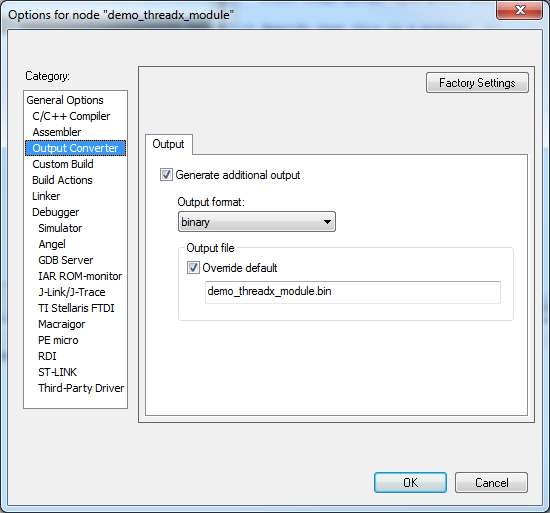
This example shows that the linker control file ***demo\_threadx\_module.icf*** is setup for this module. The linker control file is discussed later in this document but basically must ensure the module preamble is loaded first and the necessary information for position independent code and data is present. 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  VOID | \*tx\_queue\_module\_instance; \  (\*tx\_queue\_send\_module\_notify)(struct |
| TX\_QUEUE\_STRUCT \*queue\_ptr); |  |  |
| #define TX\_SEMAPHORE\_EXTENSION | VOID | \*tx\_semaphore\_module\_instance; \ |
| TX\_SEMAPHORE\_STRUCT \*semaphore\_ptr); | VOID | (\*tx\_semaphore\_put\_module\_notify)(struct |
| #define TX\_TIMER\_EXTENSION | VOID | \*tx\_timer\_module\_instance; \ |
| id); | VOID | (\*tx\_timer\_module\_expiration\_function)(ULONG |

1. Add all the ***txm\_module\_manager\*.c***, ***tx\_thread\_schedule.s***, and ***txm\_module\_manager\*.s*** files to the ThreadX library project ***tx***.
2. 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);

*/\* 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. \*/

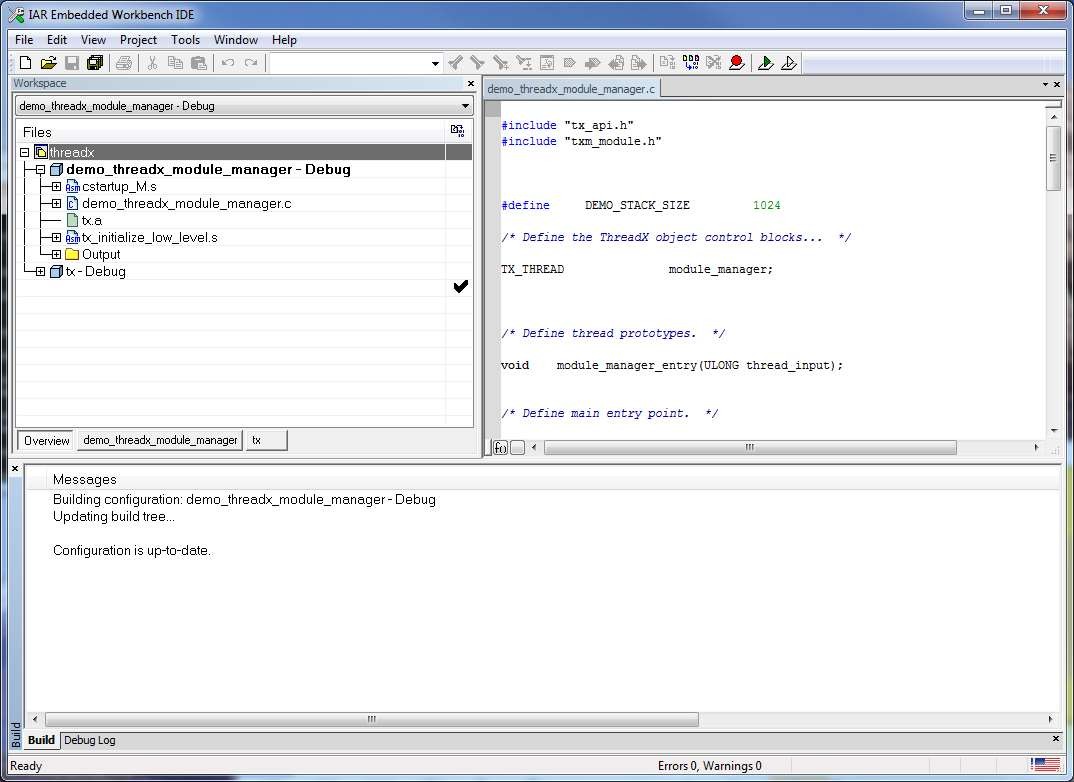
**txm\_module\_manager\_unload**(&my\_module);

}

}

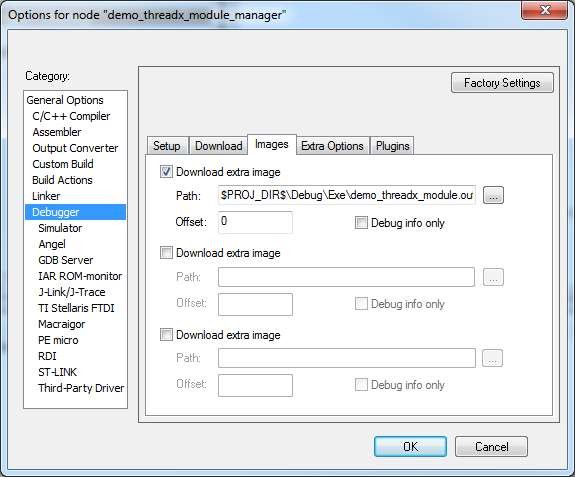
## Building Module Manager in IAR

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.



**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));

**See Also**

*/\* 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. \*/*

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);

**See Also**

*/\* If status is TX\_SUCCESS the object memory assocated ith queue\_pointer is deallocated. \*/*

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);

**See Also**

*/\* 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." \*/*

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