Freescale MQX™ RTOS 用户指南

Document Number: MQXUG
Rev 12, 02/2014

freescale™

内容

小节编号 标题 页

第 1 章
开始之前

第 2 章
MQX RTOS 概览

第 3 章
使用 MQX RTOS

第 4 章
重新编译 MQX RTOS

第 5 章
开发一个新的 BSP

小节编号 标题 页

第 6 章
FAQ

第 1 章
开始之前

1.1 关于 MQX™实时操作系统（RTOS）

MQX™实时操作系统是为采用单处理器、多处理器和分布式处理器的嵌入式实时系统而设计的。

为借力 MQX 操作系统的成功，飞思卡尔半导体将这一软件平台应用到其微处理器中。与原始 MQX RTOS 发行版相比，Freescale MQX RTOS 发行版更易于配置和使用。现在，单一发布版本即包含了 MQX 操作系统以及支持指定微处理器器件的所有其他的软件组件。本文档以如下方式标记 Freescale MQX RTOS 版本特有的章节。

表 1-1. 附注格式

附注	本文档中特定于 Freescale MQX RTOS 版本的注释采用此方式标记。

MQX RTOS 提供运行实时函数库，各种程序使用此库成为实时多任务处理应用程序。MQX RTOS 的主要特点是规模可扩展、面向组件的架构和易用性。

MQX 实时操作系统支持多处理器应用程序，可与灵活的嵌入式 I/O 产品配合，用于联网、数据通信和文件管理。

在本书中，我们使用 MQX RTOS 作为 Message Queue Executive 的缩写。

表 1-2. 相对路径

<KSDK_DIR>	Kinetis SDK 软件包在您的硬件上的安装目录。
<MQX_DIR>	MQX RTOS 在 KSDK 内所在的目录。确切地说，即<KSDK_DIR>/rtos/mqx。
<board>	替换电路板名称（即 TWR-K64F120M）。
<mcu>	替换处理器名称（即 MK64F120M）。
<tool>	替换工具链名称（即 IAR）。
<target>	替换项目目标名称（即 Debug）。
<library>	替换库名称（即 PSP）。

1.2 关于本书

请将本书与以下内容结合使用:

- MQX RTOS 参考手册 - 包含 MQX RTOS 简单与复杂数据类型，以及按字母顺序排列的 MQX RTOS 函数原型。

表 1-3. 版本内容

附注	Freescale MQX RTOS 版本还包含其它基于 MQX 操作系统的软件产品。另请参见 RTCS TCP/IP 协议栈、USB 主机开发套件、USB 设备开发套件、MFS 文件系统和其它用户指南和参考手册。

1.3 约定

下面说明了 MQX RTOS 文档中约定的提示、附注和注意用法。

1.3.1 提示

提示指出有用信息。

表 1-4. 通用提示格式

提示	从 ISR 分配消息的最有效方式是使用_msg_alloc()。

1.3.2 附注

附注指出重要信息。

表 1-5. 通用附注格式

附注	非严格信号量没有优先级继承。

1.3.3 注意

“注意”向您提示可能产生意外效果或不利影响、或者可能给您的文件或硬件带来危险的命令或步骤。

表 1-6. 通用注意格式

注意	如果您修改 MQX 数据类型，MQX Embedded 的某些 MQX 主机工具可能无法正常运行。

第 2 章
MQX RTOS 概览

2.1 MQX RTOS 的组织结构

MQX RTOS 由核心组件（必选）和可选组件组成。对于核心组件，只有那些 MQX 或应用程序调用的函数包含在映像中。为了符合应用程序要求，可以通过添加可选组件扩展应用程序。

在下图中，核心组件处于中央，可选组件在外围。

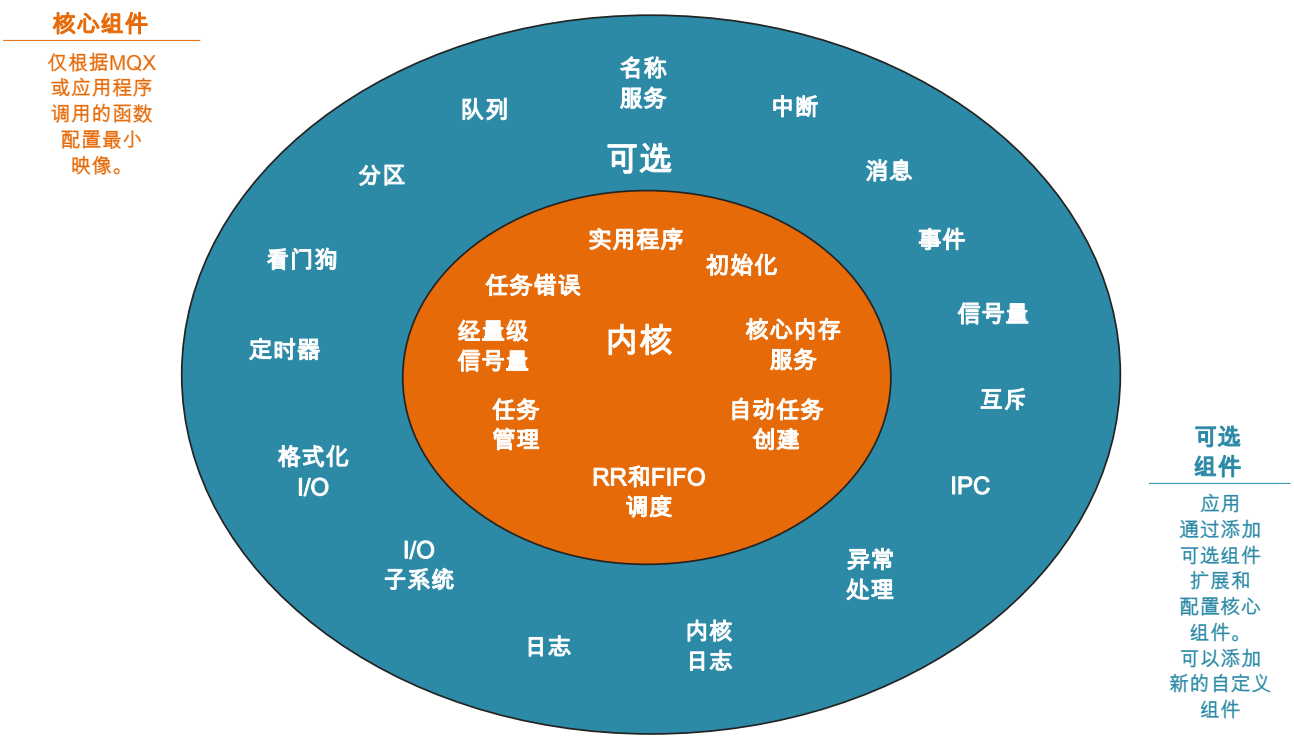

图 2-1. 核心和可选组件

下表汇总了核心和可选组件，且这章的后续小节中简要介绍了每一种组件。

表 2-1. 核心组件和可选组件

组件	包括	类型
初始化	初始化和自动任务创建	核心
任务管理	动态任务管理	核心
调度	循环和 FIFO	核心
	显式使用任务队列	可选
任务同步和通信	经量级信号量	核心
	信号量	可选
	轻量级事件	可选
	事件	可选
	互斥	可选
	轻量级消息队列	可选
	消息	可选
	任务队列	可选
处理器间通信		可选
定时	时间组件	可选（BSP）
	轻量级定时器	可选
	定时器	可选
	看门狗	可选
内存管理	可变大小的存储块	核心
	固定大小的存储块（区块）	可选
	MMU、高速缓存和虚拟存储器	可选
	轻量级存储器	可选
中断处理		可选(BSP)
I/O 驱动器	I/O 子系统	可选(BSP)
	格式化 I/O	可选(BSP)
检测工具	堆栈使用	核心
	内核日志	可选
	日志	可选
	轻量级日志	可选
错误处理	任务错误代码、异常处理、运行时测试	核心
队列操作		核心
名称组件		可选

2.2 初始化

初始化是一个核心组件。应用程序在_mqx()运行时启动。该函数初始化硬件并启动 MQX RTOS。当 MQX RTOS 启动时，它会创建并运行被应用程序定义为自启动的任务。

2.3 任务管理

任务管理是一个核心组件。

除了它在 MQX RTOS 启动时自动创建任务，应用程序还可以在运行时创建、管理和终止任务。它可以创建同一任务的多个实例,并且应用程序中的总任务数无限制。应用程序可以动态地更改任何任务的属性。MQX RTOS 在终止任务时释放任务资源。

此外，您可以对每个任务指定:

- 退出函数，MQX RTOS 在终止任务时调用。
- 异常处理程序，MQX RTOS 在任务处于活动状态下发生异常时调用。

2.4 调度

调度遵从 POSIX.4（实时扩展）标准，并支持以下策略:

- FIFO（也称为基于优先级的抢占）调度是核心组件 - 活动任务是就绪时间最长的优先级最高的任务。
- 循环（也称为时间片）调度是核心组件 - 活动任务是就绪时间最长而未消耗其时间片的优先级最高的任务。
- 显式调度（使用任务队列）是可选组件 - 您可以使用任务队列显式地调度任务，或创建更复杂的同步机制。由于任务队列提供尽可能少的功能，因此执行速度很快。在创建任务队列时，应用程序可以指定 FIFO 或循环调度策略。

2.5 管理块大小可变的存储器

为分配和释放大小可变的内存片 (称为内存块)，MQX RTOS 提供类似于 malloc()和 free() (大多数 C 语言运行时库提供) 的核心服务。您也可以在默认存储器池之内和之外的存储器池分配内存块。您可以将内存块分配给任务或系统。分配给任务的内存是任务的资源，如果分配内存的任务终止时，MQX RTOS 会释放该内存。

2.6 管理块大小固定的存储器（分区）

分区是可选组件。您可以分配和管理大小固定的内存片(称为分区块)。分区组件支持快速、固定大小的内存分配，可减少内存碎片，节省内存资源。分区可以位于默认存储器池中（动态分区），也可以在默认存储器池之外（静态分区）。您可以将分区块分配给任务或系统。分配给任务的分区块是任务的资源，如果分配内存的任务终止时，MQX RTOS 会释放这些分区块。

2.7 控制高速缓存

MQX RTOS 函数让您可以控制某些 CPU 具有的指令高速缓存和数据高速缓存。

2.8 控制 MMU

对某些 CPU,您必须在启用高速缓存之前初始化内存管理单元(MMU)。MQX RTOS 函数可用于初始化、启用和禁用 MMU，以及向其添加存储器区域。您可以使用 MMU 分页表控制 MMU。

2.9 轻量级存储器管理

如果应用程序受数据或代码大小要求限制，可以使用轻量级存储管理器。它的接口函数较少，代码和数据大小也较小。因此，某些方面不够稳健可靠（删除了标头校验和），而且速度较慢（销毁任务时释放所属内存时间较长）。

如果您修改了编译时配置选项，在分配存储空间时 MQX RTOS 会使用轻量级存储管理器组件。欲了解更多信息，请参见在编译时配置 MQX RTOS。

2.10 轻量级事件

轻量级事件（LWEvent）是可选组件。这些事件是任务利用位状态变化，以低开销进行同步的方式。轻量级事件需要的内存量少，运行快速。

2.11 事件

事件是可选组件。它们可用于动态地管理格式化为位字段的对象。任务和中断服务程序可以使用事件以位状态变化的形式同步和传送简单信息。有命名组和快速事件组两种。事件组可以具有自动清除的事件位, MQX RTOS 通过这些位在置位后立即清除。应用程序可以将远程处理器上的事件组中的事件位置位。

2.12 轻量级信号量

轻量级信号量 (LWSem) 是核心组件。它们是任务以低开销同步共享资源访问的方式。LWSem 需要存储空间很少，运行速度快。LWSem 计数无优先级继承的 FIFO 信号量。

2.13 信号量

信号量是可选组件。它们是计数信号量。您可以使用信号量来同步任务。您可以使用信号量来保护对共享资源的访问, 或实行生产者/消费者信号发送机制。信号量提供 FIFO 排队、优先级排队和优先级继承。信号量可以是严谨和非严谨的。有命名和快速信号量两种信号量。

2.14 互斥

互斥是可选组件。当任务访问共享资源时，互斥在任务之间提供相互排斥。互斥提供轮询、FIFO 排队、优先级排队、仅自旋及有限自旋排队、优先级继承和优先级保护。互斥是严谨的；即除非任务已先锁定互斥，否则不能对互斥解锁。

2.15 轻量级消息队列

轻量级消息队列是可选组件。它用于以低开销处理标准 MQX RTOS 消息。任务将消息发送至轻量级消息队列，并从轻量级消息队列接收消息。消息池中的消息大小固定，是 32 位的倍数。提供阻塞读取和阻塞写入。

2.16 消息

消息是可选组件。任务可以通过向其它任务打开的消息队列发送消息来彼此通信。每个任务都会打开其自己的输入消息队列。消息队列由创建队列时 MQX RTOS 分配的队列 ID 唯一地标识。只有打开了消息队列的任务才能接收来自队列的消息。任何任务只要知道先前已打开队列的队列 ID，就可以向此队列发送消息。

任务从消息池分配消息。消息池分为系统消息池和私有消息池。任何任务都可以从系统消息池分配消息（系统消息）。具有消息池 ID 的任何任务都可以从私有消息池分配消息（私有消息）。

2.17 任务队列

除了提供调度机制外，任务队列还提供简单有效的方法实现任务同步。您可以挂起任务队列中的任务，并将它们从任务队列中移除。

2.18 处理器间通信

处理器间通信（IPC）是一个可选组件。

应用程序可以在多个处理器上并发运行，在每个处理器上运行一个 MQX 可执行映象。这些映象通过内存传输的消息或通过处理器间的通信链路进行通信与协作。每个映象中的应用程序任务不必相同，实际上，它们通常是不相同的。

2.19 时间组件

时间是可选组件，您可以在 BSP 级别启用和禁用。有流逝时间和绝对时间两种。您可以更改绝对时间。时间分辨率取决于 MQX RTOS 启动时为目标硬件设置的应用程序定义的分辨率。

2.20 轻量级定时器

轻量级定时器是可选组件，提供低开销的机制来周期性地调用应用程序函数。轻量级定时器的实现方法是创建定期队列，然后添加一个在周期开始后有一定偏移时到期的定时器。

当您向队列添加定时器时，您可以指定通知函数，当定时器到期时，MQX RTOS 滴答中断 ISR 将调用该函数。由于定时器是在 ISR 运行，并非所有 MQX RTOS 函数都可以在该定时器调用。

2.21 定时器

定时器是可选组件。它们周期性地执行某个应用程序函数。MQX RTOS 支持一次性定时器 (一次到期) 和周期性定时器 (在给定间隔内重复到期)。您可以将定时器设置为在指定时间或在指定时间段之后启动。

当您设置定时器时，指定当定时器到期时，定时器任务调用的通知函数。通知函数可通过发送消息、设置事件或使用其他某种 MQX RTOS 同步机制来同步任务。

2.22 看门狗

看门狗是可选组件，供用户在任务级别检测任务崩溃和死锁情况。

2.23 中断和异常处理

中断和异常处理在 PSP 级别是可选的。MQX RTOS 在 BSP 定义的范围内提供所有硬件中断，并为活动任务保存最基本的上下文。如果 CPU 支持嵌套中断，则 MQX RTOS 也支持完全嵌套中断。一旦进入中断服务程序 (ISR)，应用程序可以重新启用任务中断级别。为进一步减少中断延迟，MQX RTOS 会延迟任务调度，直至所有 ISR 均已运行。此外，仅当 ISR 已将新任务准备就绪后，MQX RTOS 才会重新调度。为减小堆栈大小，MQX RTOS 支持单独的中断堆栈。

ISR 不是任务；它是一种对硬件中断迅速作出响应的小型高速例程。ISR 通常以 C 语言编写。其职责包括重置设备、获取其数据和给相应的任务发送信号。ISR 可用于给包含任何非阻塞 MQX RTOS 函数的任务发送信号。

2.24 I/O 驱动器

I/O 驱动器是 BSP 级别的可选组件。包括格式化的 I/O 和 I/O 子系统。本书中未介绍 I/O 驱动器。

2.24.1 格式化 I/O

MQX RTOS 提供 函数库，该库是连接 I/O 子系统的 API。

2.24.2 I/O 子系统

您可以动态地安装 I/O 器件驱动程序，安装后，任何任务都可以打开它们。

2.25 日志

日志组件是可选组件，用于存储和检索应用程序特定信息。每个日志条目都有时间戳和序号。您可以使用这些信息来测试、调试、验证和分析性能。

2.26 轻量级日志

轻量级日志与日志类似，但仅使用大小固定的条目。这些日志比传统应用程序日志速度更快，供内核日志使用。

2.27 内核日志

内核日志是用于记录 MQX RTOS 活动的可选组件。您可以在特定位置创建内核日志，也可以让 MQX RTOS 选择位置。您可以将内核日志配置为记录所有 MQX RTOS 函数调用、上下文切换或中断响应。性能工具使用内核日志。

2.28 堆栈使用

MQX RTOS 具有一些核心功能，支持动态地检查中断堆栈和所有任务的堆栈使用，从而可以确定是否分配了足够的堆栈空间。

2.29 任务错误代码

每个任务都有一个与该任务的上下文相关联的任务错误代码。特定的 MQX 函数可以读取和更新任务错误代码。

2.30 异常处理

您可以指定对所有未处理中断运行的默认 ISR，以及在 ISR 产生异常时运行的特定的 ISR 异常处理程序。

2.31 运行时测试

MQX RTOS 提供核心运行时测试函数，应用程序可在其正常运行期间调用这些函数。下列组件有测试函数:

- 事件和轻量级事件
- 内核日志和轻量级日志
- 块大小固定的内存（分区）
- 块大小可变的存储器和轻量级存储器
- 消息池和消息队列
- 互斥
- 名称组件
- 队列（应用程序定义）
- 信号量与轻量级信号量
- 任务队列
- 定时器和轻量级定时器
- 看门狗

2.32 队列操作

一个核心组件用于设置队列元素的双链表。您可以初始化队列，添加、删除和读取元素。

2.33 名称组件

名称组件是可选的。它提供名称数据库，将字符串动态地映射到定义的标量（如队列 ID）。

第 3 章
使用 MQX RTOS

3.1 开始之前

本章介绍如何使用 MQX RTOS。包含您可以编译并运行的示例。

表 3-1. 参考

如需此信息	请参见
本章中提到的每个函数的原型。	MQX RTOS 参考手册
本章中提到的数据类型。	MQX RTOS 参考手册

3.2 初始化并启动 MQX RTOS

MQX RTOS 以**_mqx()**函数开始运行，该函数以 MQX RTOS 初始化结构体作为参数。基于这一结构体中的值，MQX RTOS 执行以下操作:

- 设置并初始化 MQX RTOS 内部使用的数据，包括默认的存储器池、就绪队列、中断堆栈和任务堆栈。
- 初始化硬件（如：片选)。
- 启用定时器。
- 设置默认的时间片值。
- 创建空闲任务，如果没有其他任务就绪则会将其激活。
- 创建任务模板列表中定义为自动启动的任务。
- 开始调度任务。

3.2.1 MQX RTOS 初始化结构体

MQX RTOS 初始化结构体定义了应用程序和目标硬件的参数。

```
typedef struct mqx_initialization_struct
{
```

```
  _mqx_uint                   PROCESSOR_NUMBER;
  void *                          START_OF_KERNEL_MEMORY;
  void *                          END_OF_KERNEL_MEMORY;
  _mqx_uint                   INTERRUPT_STACK_SIZE;
  TASK_TEMPLATE_STRUCT_PTR    TASK_TEMPLATE_LIST;
  _mqx_uint                   MQX_HARDWARE_INTERRUPT_LEVEL_MAX;
  _mqx_uint                   MAX_MSGPOOLS;
  _mqx_uint                   MAX_MSGQS;
  char  *                         IO_CHANNEL;
  char  *                         IO_OPEN_MODE;
  _mqx_uint                   RESERVED[2];
} MQX_INITIALIZATION_STRUCT, * MQX_INITIALIZATION_STRUCT_PTR;
```

关于每个字段的说明，请参见《*Freescale MQX™ RTOS* 参考手册》。

3.2.1.1 默认 MQX RTOS 初始化结构体

您可以定义自己的 MQX RTOS 初始化结构体的初始化值，也可以使用每个 BSP 提供的默认初始化。默认初始化变量由 **MQX_init_struct** 调用，位于相应的 BSP 目录的 mqx_init.c 中。该函数已编译并与 MQX RTOS 链接。

附注	为保证任务感知的调试主机工具正常工作，MQX RTOS 初始化结构变量必须命名为 **MQX_init_struct**。

本章中的示例使用下面的 **MQX_init_struct**。

```
MQX_INITIALIZATION_STRUCT  MQX_init_struct =
{
/* PROCESSOR_NUMBER         */ BSP_DEFAULT_PROCESSOR_NUMBER,
/* START_OF_KERNEL_MEMORY   */ BSP_DEFAULT_START_OF_KERNEL_MEMORY,
/* END_OF_KERNEL_MEMORY     */ BSP_DEFAULT_END_OF_KERNEL_MEMORY,
/* INTERRUPT_STACK_SIZE     */ BSP_DEFAULT_INTERRUPT_STACK_SIZE,
/* TASK_TEMPLATE_LIST       */ (void *)MQX_template_list,
/* MQX_HARDWARE_INTERRUPT_LEVEL_MAX*/
                        BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
/* MAX_MSGPOOLS             */ BSP_DEFAULT_MAX_MSGPOOLS,
/* MAX_MSGQS                */ BSP_DEFAULT_MAX_MSGQS,
/* IO_CHANNEL               */ BSP_DEFAULT_IO_CHANNEL,
/* IO_OPEN_MODE             */ BSP_DEFAULT_IO_OPEN_MODE
};
```

附注	将 **RESERVED** 字段的两个元素均初始化为 0。

3.2.2 任务模板列表

任务模板列表（**TASK_TEMPLATE_STRUCT**）定义了一组初始化模板，从而可在处理器上创建任务。

初始化时，MQX RTOS 为每个任务创建一个实例，模板将其定义为自启动任务。此外，当运行应用程序时，它可以动态地创建使用任务模板列表定义或应用程序定义作为任务模板的其他任务。任务模板列表的结尾是一个填入全零的任务模板。

```
typedef struct task_template_struct
{
  _mqx_uint  TASK_TEMPLATE_INDEX;
  TASK_FPTR  TASK_ADDRESS;
  _mem_size  TASK_STACKSIZE;
  _mqx_uint  TASK_PRIORITY;
  char * TASK_NAME;
  _mqx_uint  TASK_ATTRIBUTES;
  uint32_t    CREATION_PARAMETER;
  _mqx_uint  DEFAULT_TIME_SLICE;
} TASK_TEMPLATE_STRUCT,  *  TASK_TEMPLATE_STRUCT_PTR;
```

关于每个字段的说明，请参见《*MQX™ RTOS* 参考手册》。

3.2.2.1 分配任务优先级

附注	如果您对某个任务分配优先级 0，该任务将在禁用中断的状态下运行。 在有些目标处理器平台（如 ColdFire）上，某些任务优先级会保留，并与处理器中断任务优先级对应。以这种特殊优先级运行的任务可以防止响应更低优先级的中断。请参阅章节处理中断和异常 中关于中断处理的更多详细信息。

当您在任务模板列表中分配任务优先级时，请注意:

- MQX RTOS 为每个优先级创建一个就绪队列，直至最低优先级（最高编号)。
- 当应用程序运行时，它不能创建优先级低于任务模板列表中的最低优先级的任务。

3.2.2.2 分配任务属性

您可以对任务分配以下属性的任意组合:

- 自动启动 - 当 MQX RTOS 启动时，会创建任务的一个实例。
- DSP - MQX RTOS 在任务的上下文中保存 DSP 协处理器寄存器。
- 浮点 - MQX RTOS 在任务的上下文中保存浮点寄存器。
- 时间片 - MQX RTOS 对任务使用循环调度（默认为 FIFO 调度)。

3.2.2.3 默认任务模板列表

您可以初始化自己的任务模板列表，也可以使用名为 **MQX_template_list** 的默认任务模板列表。

3.2.2.4 示例：任务模板列表

```
TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 { MAIN_TASK, world_task, 0x2000, 5, "world_task",
     MQX_AUTO_START_TASK,                        0L, 0},
 { HELLO,     hello_task, 0x2000, 5, "hello_task",
     MQX_TIME_SLICE_TASK,                        0L, 100},
 { FLOAT,     float_task, 0x2000, 5, "Float_task",
     MQX_AUTO_START_TASK | MQX_FLOATING_POINT_TASK, 0L, 0},
 { 0, 0, 0, 0, 0, 0, 0L, 0 }
};
```

world_task

world_task 是一项自动启用任务。因此，在初始化时，MQX RTOS 会使用创建参数 0 创建一个该任务的实例。应用程序定义任务模板索引(MAIN_TASK)。该任务的优先级为 5。函数 **world_task()**是该任务的代码入口点。堆栈大小为 0x2000 个单一可寻址单位。

hello_task

hello_task 任务是一个时间片任务，时间片为 100，单位为毫秒（如果使用默认的编译时配置选项)。有关这些选项的信息，请参阅在编译时配置 MQX RTOS 页面。

Float_task

Float_task 任务是一个浮点型自动启动任务。

3.2.2.5 示例：创建自动启动任务

单个任务输出 Hello World 并终止。

```
/* hello.c */
#include <mqx.h>
#include <fio.h>
/* Task IDs */
#define HELLO_TASK     5
extern void hello_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,    Stack, Priority, Name,    Attributes,            Param, Time
Slice */
  { HELLO_TASK, hello_task, 1500,  8,       "hello", MQX_AUTO_START_TASK, 0,     0 },
  { 0 }
};
void hello_task(uint32_t initial_data)
{
   printf("\n Hello World \n");
   _mqx_exit(0);
}
```

3.2.2.5.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录：

```
mqx\examples\hello
```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档中的说明运行该应用程序。 请参阅以获取有关支持的工具链的更多详细信息。

输出设备上将显示以下内容:

```
Hello World
```

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》以获取有关支持的工具链的更多详细信息。

3.3 管理任务

从同一个任务模板中创建的多个任务可以共存，每个任务都有唯一的实例。MQX RTOS 通过保存实例上下文对每个实例进行维护，即程序计数器、寄存器和堆栈。每个任务均具有一个应用程序唯一的 32 位任务 ID，MQX RTOS 及其他任务通过该 ID 对任务进行识别。

初始化章节 (初始化并启动 MQX RTOS 页) 显示了当初始化 MQX RTOS 时如何自动启动任务。在应用程序运行时，您也可以创建、管理和终止任务。

表 3-2. 汇总：管理任务

_task_abort	在运行任务退出句柄并释放资源后终止任务。
_task_check_stack	判断任务堆栈是否溢出。
_task_create	分配和启动（使其就绪）一个新任务。
_task_create_blocked	分配一个处于阻塞状态的任务。
_task_create_at	创建一个指定堆栈位置的新任务。
_task_destroy	释放资源后终止任务。
_task_disable_fp	如果是浮点任务，禁止该任务的浮点上下文切换。
_task_enable_fp	启用任务的浮点上下文切换。
_task_errno	获取用于激活任务的任务错误代码。
_task_get_creator	获取创建任务的任务 ID。
_task_get_environment	获取任务的环境数据指针。
_task_get_error	获取任务错误代码。
_task_get_error_ptr	获取任务错误代码的指针。
_task_get_exit_handler	获取任务的退出句柄。
_task_get_id	获取任务 ID。
_task_get_id_from_name	获取任务模板中首个使用此名称的任务的 ID。
_task_get_index_from_id	获取任务 ID 的任务模板索引。

下一页继续介绍此表...

表 3-2. 汇总：管理任务 (继续)

_task_get_parameter	获取任务创建参数。
_task_get_parameter_for	获取用于任务的任务创建参数。
_task_get_processor	获取任务所在的处理器编号。
_task_get_td	将任务 ID 转换成指向任务描述符的指针。
_task_get_template_index	获取任务名称的任务模板索引。
_task_get_template_ptr	获取用于任务 ID 的指向任务模板的指针。
_task_restart	在任务函数开始处重启任务；保留相同的任务描述符、任务 ID 和任务堆栈。
_task_set_environment	设置任务的环境数据指针。
_task_set_error	设置任务错误代码。
_task_set_exit_handler	设置任务的退出句柄。
_task_set_parameter	设置任务创建参数。
_task_set_parameter_for	设置用于任务的任务创建参数。

3.3.1 创建任务

通过调用**_task_create()**、**_task_create_at()**或**_task_create_blocked()**并传递处理器编号、任务模板索引和任务创建参数，任何任务（创建者）都可以创建另一任务（子任务)。应用程序定义一个创建参数,通常用于向子任务提供初始化信息。通过指定模板索引 0，任务还可以创建任务模板列表中未定义的任务。在此情况下，MQX RTOS 会将任务创建参数解释为任务模板的指针。

函数用来初始化子任务的堆栈。函数**_task_create()**将子任务置于就绪队列以获得任务的优先级。如果子任务的优先级高于创建者，则子任务成为活动任务，因为它是优先级最高的就绪任务。如果创建者优先级较高或相同，则它保持为活动任务。

函数**_task_create_blocked()**创建被阻塞的任务。在另一任务调用**_task_ready()**前，该任务不会进入运行就绪状态。

函数**_task_create_at()**使用指定的堆栈位置创建任务，也就是说，任务堆栈不是动态分配的，而是必须在执行**_task_create_at()**函数前分配。

3.3.2 获取任务 ID

通过**_task_get_id()**函数，任务可直接获取其任务 ID。如果一个函数将任务 ID 作为参数，您可指定 **MQX_NULL_TASK_ID** 标识当前活动任务的任务 ID。

通过**_task_get_creator()**函数，任务可直接获取其创建者的任务 ID。函数**_task_create()**将向创建者返回子任务 ID。

还可由创建任务的任务模板中的任务名称来确定任务 ID。这通过 **_task_get_id_from_name()**函数实现，此函数返回任务模板列表中第一个名称相匹配的任务的任务 ID。

3.3.3 设置任务环境

通过**_task_set_environment()**函数，任务可以保存应用程序专用的环境指针。其他任务可通过**_task_get_environment()**函数访问环境指针。

3.3.4 管理任务错误

每项任务都有一个与该任务的上下文相关联的错误代码（任务错误代码）。一些 MQX 函数在检测到错误时，会更新任务错误代码。

如果一个 MQX 函数检测到错误，而应用程序忽略了该错误，则可能仍会发生其他错误。通常第一个错误最能说明问题；而后续错误可能会产生误导。为了给诊断问题提供可靠机会，在 MQX RTOS 将任务错误代码设置为与 **MQX_OK** 不同的值后，MQX RTOS 将不会再改变任务错误代码，直到任务显式将其复位为 **MQX_OK** 为止。

任务可通过以下函数获取其任务错误代码:

- **_task_get_error()**
- **_task_errno**

任务通过调用**_task_set_error()**函数将任务错误代码复位为 **MQX_OK**。该函数返回先前的任务错误代码并能够将任务错误代码设置为 **MQX_OK**。

利用_task_set_error()函数，任务可将其任务错误代码设置为不同于 **MQX_OK** 的值。然而，只有当前的任务错误代码为 **MQX_OK**，MQX RTOS 才会将任务错误代码修改为新值。

如果 **MQX_CHECK_ERRORS** 设置为 0（见 MQX RTOS 编译时配置选项），则不会返回针对特定函数列出的所有错误代码。

3.3.5 重启任务

应用程序可以通过调用**_task_restart()**来重启任务，在其函数的开始处重启任务，保留相同的任务描述符、任务 ID 和任务堆栈。

3.3.6 终止任务

任务可以终止其自身，或任何其他已知任务 ID 的任务。当任务终止时，其子任务不会终止。当任务终止时，MQX RTOS 会释放任务所有的 MQX RTOS 资源。这些资源包括:

- 动态分配的内存区和区块
- 消息队列
- 消息
- 互斥
- 非严谨信号量
- 传递后的严谨信号量
- 排队的连接将会出列
- 任务描述符

附注	用户负责在终止任务前销毁所有轻量级对象（轻量级信号量、轻量级事件和轻量级定时器等），因为 MQX RTOS 任务终止函数不会处理这些对象！

应用程序可以通过**_task_destroy()**函数直接终止任务 (在 MQX RTOS 释放任务资源后) 或通过**_task_abort()**函数“礼貌地”终止。**_task_destroy()**将引起任务在调用者的上下文处销毁，并立即执行；**_task_abort()**将受影响任务从其阻塞的队列中移除，PC 会立即设置任务退出句柄，受影响任务将添加到就绪运行的队列中。通常会应用任务调度和优先级规则，因而实际任务销毁可能被无限推迟 (或一段较长时间)。这意味着当从**_task_abort()**返回时，不能保证受影响任务被销毁。

当将要终止的任务变为活动状态时，将运行一个应用程序定义的任务退出句柄。该退出句柄能够清除不由 MQX RTOS 管理的资源。

任务退出句柄通过**_task_set_exit_handler()**函数设置，也可通过**_task_get_exit_handler()**函数获取。

如果任务从其任务主体返回，则 MQX RTOS 也会调用任务退出句柄。

3.3.7 示例：创建任务

此示例向示例: 创建自动启动任务 页上的示例添加另一个任务(world_task)。我们修改该示例的任务模板列表，以包括有关 world_task 的信息，以及更改 hello_task，使其成为非自动启动任务。world_task 任务是一项自动启动任务。

当 MQX RTOS 启动时，它会创建 world_task。然后，world_task 使用 hello_task 作为参数调用**_task_create()**，创建 hello_task。MQX RTOS 使用 hello_task 模板创建 hello_task 的实例。

如果**_task_create()**成功，它返回新的子任务的任务 ID；否则返回 **MQX_NULL_TASK_ID**。

新的 hello_task 任务置于就绪队列，获取任务的优先级。由于其优先级高于 world_task，它成为活动任务。该活动任务输出 Hello。然后 world_task 任务成为活动任务，并检查是否已成功创建 hello_task。如果成功，world_task 输出 World；否则，world_task 输出一则错误消息。最后，MQX RTOS 退出。

如果您将 world_task 的优先级更改为与 hello_task 相同，则仅输出 World。world_task 在 hello_task 之前运行，因为 world_task 具有相同的优先级，并且不使用阻塞函数让出控制权。由于 world_task 在输出 World 后调用**_mqx_exit()**，无法再输出其它任何内容，因为 hello_task 没有机会再次运行。

3.3.7.1 创建任务的代码示例

```
        /*  hello2.c */

#include <mqx.h>
#include <fio.h>
/* Task IDs */
#define HELLO_TASK     5
#define WORLD_TASK     6
extern void hello_task(uint32_t);
extern void world_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,    Stack, Priority, Name,    Attributes,            Param, Time
Slice */
  { WORLD_TASK, world_task, 1000,  9,       "world", MQX_AUTO_START_TASK, 0,     0 },
  { HELLO_TASK, hello_task, 1000,  8,       "hello", 0,                   0,     0 },
  { 0 }
};
/*TASK*-----------------------------------------------------
*
* Task Name    : world_task
* Comments     :
*    This task creates hello_task and then prints "World".
*
*END*-----------------------------------------------------*/
void world_task(uint32_t initial_data)
{
   _task_id        hello_task_id;
   hello_task_id = _task_create(0, HELLO_TASK, 0);
   if (hello_task_id == MQX_NULL_TASK_ID) {
      printf("\n Could not create hello_task\n");
   } else {
      printf(" World \n");
   }
   _mqx_exit(0);
}
/*TASK*-----------------------------------------------------
*
* Task Name    : hello_task
* Comments     :
*    This task prints "Hello".
*
*END*-----------------------------------------------------*/
void hello_task(uint32_t initial_data)
{
```

```
   printf("  Hello \n");
   _task_block();
}
```

3.3.7.2 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\hello2
   ```
2. 请参阅 《Freescale MQX™ RTOS for Kinetis SDK 入门》(文档 MQXKSDKGSUG) 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》 中的说明运行该应用程序。

输出设备上将显示以下消息:

```
Hello

World
```

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档 。

3.4 调度任务

MQX RTOS 提供以下这些任务调度策略:

- FIFO
- 循环
- 显式使用任务队列 (在后续章节轻量级消息队列 中说明)。

您可以为处理器和每个任务分别设置调度策略为 FIFO 或循环方式。从而，应用程序可能由采用任意 FIFO 或循环调度组合的多个任务组成。

3.4.1 FIFO 调度

FIFO 是默认的调度策略。使用 FIFO 调度，下一个要运行 (变为活动状态) 的任务是等待了最长时间具有最高优先级的任务。活动任务将保持运行，直到发生以下情况:

- 活动任务自愿放弃处理器，因为其调用了一条阻塞 MQX 函数。
- 产生一个优先级高于活动任务的中断。
- 一个优先级高于活动任务的任务就绪。

您可以通过**_task_set_priority()**函数改变任务的优先级。

3.4.2 循环调度

循环调度与 FIFO 调度类似，但是增加了每个循环任务都分配了其处于活动状态的最长时间（时间片）的限制。

只有当任务的 **MQX_TIME_SLICE_TASK** 属性在任务模板中置位时，任务才会使用循环调度。任务的时间片由模板中 **DEFAULT_TIME_SLICE** 的值决定。不过，如果值为 0，任务的时间片默认为处理器的时间片。最初，处理器的默认时间片为周期性定时器中断时间间隔的十倍。由于大多数 BSP 的时间间隔为 5 ms, 处理器的初始默认时间片通常为 50 ms。您可以通过**_sched_set_rr_interval()**函数或**_sched_set_rr_interval_ticks()**函数来改变处理器的默认时间片，将任务 ID 设为 **MQX_DEFAULT_TASK_ID** 参数。

在活动循环任务的时间片到期后，MQX RTOS 会保存任务上下文。然后 MQX RTOS 会执行切换操作，检查就绪队列以确定应变为活动状态的任务。MQX RTOS 将到期任务移到任务就绪队列的末尾，该动作会在就绪队列中将控制交给下一个任务。如果就绪队列中没有其他任务，则会继续运行到期任务。

利用循环调度，具有相同优先级的任务可以等时的方式共用处理器。

表 3-3. 汇总：获取和设置调度信息

_sched_get_max_priority	或许任务允许的最高优先级；总是返回 0。
_sched_get_min_priority	获取任务允许的最低优先级。
_sched_get_policy	获取调度策略。
_sched_get_rr_interval	获取时间片（以毫秒为单位）。
_sched_get_rr_interval_ticks	获取时间片（以滴答时间为单位）。
_sched_set_policy	设置调度策略。
_sched_set_rr_interval	设置时间片（以毫秒为单位）。
_sched_set_rr_interval_ticks	设置时间片（以滴答时间为单位）。

表 3-4. 汇总：调度任务

_sched_yield	将活动任务移至其就绪队列末尾，从而将处理器交给相同优先级的下一个就绪任务。
_task_block	阻塞任务。
_task_get_priority	获取任务优先级。
_task_ready	使任务就绪。
_task_set_priority	设置任务优先级。
_task_start_preemption	重新启用任务抢占。
_task_stop_preemption	禁止任务抢占。

每个任务都处于以下逻辑状态之一:

- 阻塞—任务未就绪未变为活动状态，因为其在等待某个情况发生；当该情况发生时，任务变为就绪。
- 就绪—任务就绪变为活动状态，但由于其与活动任务具有相同优先级或较低优先级而未进入活动状态。
- 活动—任务正在运行。

如果活动任务变为阻塞或抢占，MQX RTOS 会执行切换操作，检查就绪队列以确定应变为活动状态的任务。MQX RTOS 将具有最高优先级的就绪任务变为活动任务。如果有不止一个相同优先级的任务就绪，则位于就绪队列开始处的任务变为活动任务。即，每个就绪序列按 FIFO 顺序执行。

3.4.2.1 抢占

活动任务可以被抢占。当较高优先级任务变为就绪时，会发生抢占并且该任务会成为活动任务。先前的活动任务仍为就绪状态，但不再是活动任务。当中断句柄引起较高优先级任务变为就绪，或活动任务使较高优先级任务就绪时，会发生抢占。

3.5 管理块大小可变的存储器

默认情况下，MQX RTOS 从其默认存储器池分配存储器块。任务也可以在默认存储器池之外创建存储器池，并从中分配内存区。

两种分配流程相似，均使用 **malloc()**和 **free()**函数，这两个函数位于大多数 C 语言运行时软件库中。

附注	您不能将存储器块作为消息使用。必须从消息池分配消息（见消息 ）。

内存块可以是私有内存块（属于分配它的任务所有的资源）或是系统内存块（不属于任何任务）。当任务终止时，MQX RTOS 将任务的私有内存块返回给存储器。

当 MQX RTOS 分配内存块时，其至少要分配所需大小的块（也可以是更大的块）。

一个任务可以将内存块的所有权转移给另一个任务（**_mem_transfer()**）。

表 3-5. 汇总：管理块大小可变的存储器

_mem_alloc	从默认存储器池中分配私有内存块。
_mem_alloc_from	从指定存储器池中分配私有内存块。
_mem_alloc_zero	从默认存储器池中分配全零填充的私有内存块。
_mem_alloc_zero_from	从指定存储器池中分配全零填充的私有内存块。

下一页继续介绍此表...

表 3-5. 汇总：管理块大小可变的存储器 (继续)

_mem_alloc_system	从默认存储器池中分配系统内存块。
_mem_alloc_system_from	从指定存储器池中分配系统内存块。
_mem_alloc_system_zero	从默认存储器池中分配全零填充的系统内存块。
_mem_alloc_system_zero_from	从指定存储器池中分配全零填充的系统内存块。
_mem_alloc_align	从默认存储器池中分配对齐的私有内存块。
_mem_alloc_align_from	从指定存储器池中分配对齐的私有内存块。
_mem_alloc_system_align	从默认存储器池中分配对齐的系统内存块。
_mem_alloc_system_align_from	从指定存储器池中分配对齐的系统内存块。
_mem_alloc_at	在定义的初始地址处分配私有内存块。
_mem_copy	将数据从一个存储器位置复制到另一个存储器位置。
_mem_create_pool	创建默认存储器池以外的存储器池。
_mem_extend	向默认存储器池增添附加存储器；附加存储器必须位于当前默认存储器池之外，但不必与之相邻。
_mem_extend_pool	向默认存储器池之外的存储器池增添附加存储器；附加存储器必须位于该存储器池之外，但不必与之相邻。
_mem_free	释放位于默认存储器池内部或外部的内存块。
_mem_free_part	释放部分内存块（当内存块比要求的块大或比所需容量大时使用）。
_mem_get_error	获取使用**_mem_test()**函数指示错误时指向内存块的指针。
_mem_get_error_pool	获取使用**_mem_test_pool()**函数指示错误时指向最后一个内存块的指针。
_mem_get_highwater	获取在默认存储器池中已分配的最高位存储器地址（可能已经被释放）。
_mem_get_highwater_pool	获取已分配的最高位存储器池地址（可能已经被释放）。
_mem_get_size	获取内存块大小，尺寸可能大于要求尺寸。
_mem_swap_endian	转换为其他字节格式。
_mem_test	测试默认存储器池；即，检查内部校验和以确定存储器的完整性是否被破坏（失败原因通常是应用程序写入超出内存块边界）。
_mem_test_and_set	测试和设置存储器位置。
_mem_test_pool	测试存储器池的错误，见对于**_mem_test()**的说明。
_mem_transfer	将内存块所有权转移给另一个任务。
_mem_zero	将整个或部分内存块设为零。

3.5.1 管理块大小可变的轻量级存储器

轻量级存储器函数与管理块大小可变的存储器 中描述的常规存储器函数相类似。但是，它们在数据和代码方面具有更低的开销。

如果您修改了 MQX RTOS 编译时配置选项，在分配存储器时 MQX RTOS 会使用轻量级存储器组件。 欲了解更多信息，请参见在编译时配置 MQX RTOS 。

表 3-6. 汇总：管理块大小可变的轻量级存储器

轻量级存储器使用某些在 *lwmem.h* 中定义的结构和常数。	轻量级存储器使用某些在 *lwmem.h* 中定义的结构和常数。

下一页继续介绍此表...

表 3-6. 汇总：管理块大小可变的轻量级存储器 (继续)

_lwmem_alloc	从默认轻量级存储器池中分配专用轻量级存储器块。
_lwmem_alloc_from	从指定轻量级存储器池中分配专用轻量级存储器块。
_lwmem_alloc_zero	从默认轻量级存储器池中分配全零填充的专用轻量级存储器块。
_lwmem_alloc_zero_from	从指定轻量级存储器池中分配全零填充的专用轻量级存储器块。
_lwmem_alloc_system	从默认轻量级存储器池中分配系统轻量级存储器块。
_lwmem_alloc_system_from	从指定轻量级存储器池中分配系统轻量级存储器块。
_lwmem_alloc_system_zero	从默认轻量级存储器池中分配全零填充的系统轻量级存储器块。
_lwmem_alloc_system_zero_from	从指定轻量级存储器池中分配全零填充的系统存储器块。
_lwmem_alloc_align	从默认轻量级存储器池中分配对齐的私有轻量级存储器块。
_lwmem_alloc_align_from	从指定轻量级存储器池中分配对齐的私有轻量级存储器块。
_lwmem_alloc_system_align	从默认轻量级存储器池中分配对齐的系统轻量级存储器块。
_lwmem_alloc_system_align_from	从指定轻量级存储器池中分配对齐的系统轻量级存储器块。
_lwmem_alloc_at	在定义的初始地址处分配专用轻量级存储器块。
_lwmem_create_pool	创建一个轻量级存储器池。
_lwmem_free	释放一个轻量级存储器块。
_lwmem_get_size	获取轻量级存储器块大小，尺寸可能大于要求尺寸。
_lwmem_set_default_pool	将默认轻量级存储器池置位。
_lwmem_test	测试所有轻量级存储器池。
_lwmem_transfer	将轻量级存储器块所有权转移给另一个任务。

3.5.2 管理块大小固定的存储器（分区）

利用分区组件，您可以管理固定大小内存区的分区，其大小在创建分区时由任务指定。具有可以扩展的动态分区（在默认存储器池内）和不可扩展的静态分区（在默认存储器池之外)。

3.5.2.1 为动态分区创建分区组件

您可以使用_partition_create_component()显式地创建分区组件。如果不显式地创建，MQX RTOS 会在应用程序首次创建分区时创建。没有参数。

3.5.2.2 创建分区

有两种类型的分区。

表 3-7. 静态和动态分区

分区类型:	创建自:	通过调用:
动态	默认内存池	**_partition_create()**
静态	默认内存池外	**_partition_create_at()**

如果您创建静态分区，则必须确保内存不会覆盖您的应用程序所使用的代码或数据空间。

3.5.2.3 分配和释放分区块

应用程序可以从动态或静态分区分配两种类型的分区块。

表 3-8. 专用和系统分区块

分区块类型:	通过调用以下函数分配:	是其资源:	可由其释放:
专用	**_partition_alloc()**	分配它的任务	仅所有者
系统	**_partition_alloc_system()**	非任一任务	任何任务

如果任务终止，其专用分区将被释放。

3.5.2.4 销毁动态分区

如果动态分区中的所有分区块均已释放，任何任务都可以通过调用 **_partition_destroy()**销毁该分区。不能销毁静态分区。

3.5.2.5 示例：两个分区

下图显示一个静态分区和一个动态分区。

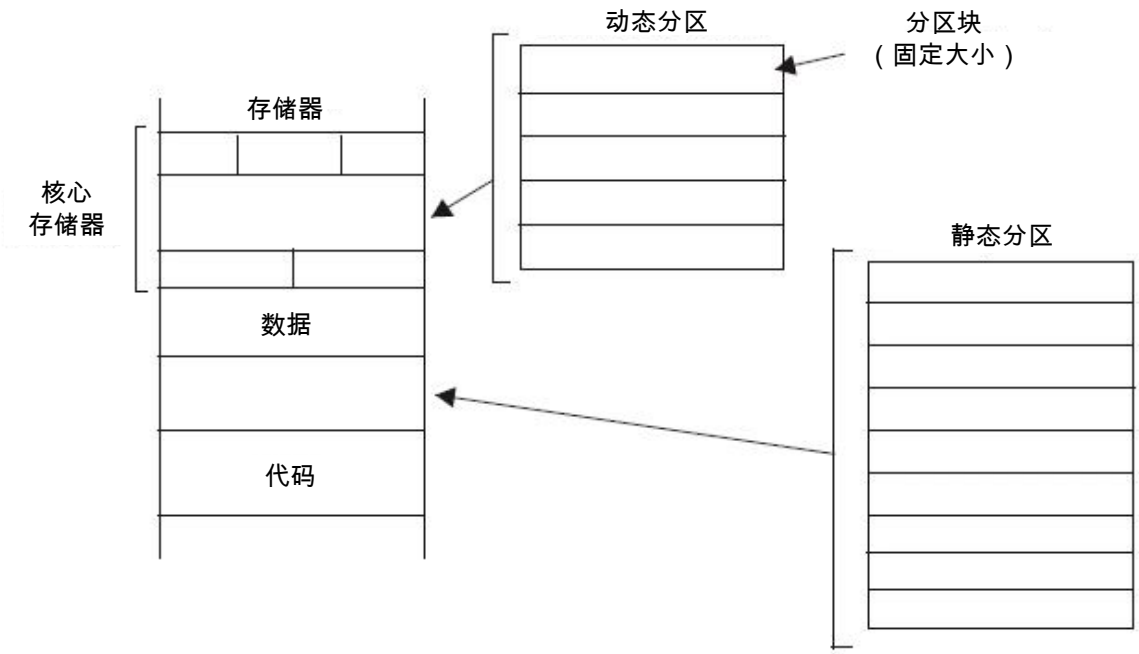

图 3-1. 示例：两个分区

表 3-9. 汇总：管理块大小固定的内存（分区）

_partition_alloc	从分区分配专用分区块。
_partition_alloc_system	从分区分配系统分区块。
_partition_alloc_system_zero	从分区分配填充了 0 的系统分区块。
_partition_alloc_zero	从分区分配填充了 0 的专用分区块。
_partition_calculate_blocks	从分区块大小和分区大小（静态分区）计算分区块数。
_partition_calculate_size	从分区块大小和分区数计算分区大小。
_partition_create	从默认内存池创建分区（动态分区）。
_partition_create_at	在默认内存池外的特定位置创建分区（静态分区）。
_partition_create_component	创建分区组件。
_partition_destroy	销毁无已分配分区块的动态分区。
_partition_extend	添加内存至静态分区；添加的内存划分为与分区中其他块大小相同的分区块。
_partition_free	将分区块释放回分区。
_partition_get_block_size	获取分区中的分区块大小。
_partition_get_free_blocks	获取分区中的可用分区块数。
_partition_get_max_used_blocks	获取分区中的已分配分区块数；即表示已同时分配的最大数的高水位，不一定是当前分配的数量。
_partition_get_total_blocks	获取分区中的分区块数。
_partition_get_total_size	获取分区大小，包括扩展。
_partition_test	测试分区组件。

下一页继续介绍此表...

表 3-9. 汇总：管理块大小固定的内存（分区） (继续)

_partition_transfer	将分区块的所有权转给另一任务（包括系统）；只有新所有者可以释放分区块。

3.5.3 控制高速缓存

MQX 函数让您可以控制某些 CPU 具有的指令高速缓存和数据高速缓存。

这样, 您可以编写同时应用于缓存和非缓存系统的应用程序。MQX RTOS 将函数打包在宏中。对于没有高速缓存的 CPU，宏不映射任何函数。某些 CPU 采用统一的高速缓存 (数据和代码使用同一高速缓存), 在此情况下, **_DCACHE_**和**_ICACHE_**宏映射到同一函数。

3.5.3.1 刷新数据高速缓存

MQX RTOS 使用术语刷新来表示刷新整个数据高速缓存。在缓存中未写入的数据会被写入到物理存储器中。

3.5.3.2 使数据或指令缓存失效

MQX RTOS 使用术语“失效”来表示使所有缓存条目失效。遗留在缓存中未写入存储器的数据或指令将丢失。后续访问会重新将物理存储器中的数据或指令加载到缓存中。

表 3-10. 汇总：控制数据缓存

_DCACHE_DISABLE	禁止数据缓存。
_DCACHE_ENABLE	启用数据缓存。
_DCACHE_FLUSH	刷新整个数据缓存。
_DCACHE_FLUSH_LINE	刷新包含指定地址的数据缓存行。
_DCACHE_FLUSH_ MLINES	刷新包含指定存储器区域的数据缓存行。
_DCACHE_INVALIDATE	使数据缓存失效。
_DCACHE_INVALIDATE_ LINE	使包含指定地址的数据缓存行失效。
_DCACHE_INVALIDATE_ MLINES	使包含指定存储器区域的数据缓存行失效。

表 3-11. 汇总：控制指令缓存

_ICACHE_DISABLE	禁止指令缓存。
_ICACHE_ENABLE	启用指令缓存。

下一页继续介绍此表...

表 3-11. 汇总：控制指令缓存 (继续)

_ICACHE_INVALIDATE	使指令缓存失效。
_ICACHE_INVALIDATE_ LINE	使包含指定地址的指令缓存行失效。
_ICACHE_INVALIDATE_MLINES	使包含指定存储器区域的指令缓存行失效。

附注	刷新和失效函数总是对整个缓存行进行操作。如果数据条目未与缓存行大小对齐，则这些操作会影响当前正刷新/失效的数据区域前后的数据。 MQX RTOS 存储器分配器默认将数据条目与缓存行大小对齐。如果条目进行静态申明，则不能保证其与缓存行大小对齐（除非使用对齐指令）。

3.5.4 控制 MMU（虚拟内存）

对某些 CPU，您必须在启用高速缓存之前初始化内存管理单元（MMU)。MQX 函数可用于初始化、启用和禁用 MMU，以及向其添加存储器区域。并非所有架构都支持 MMU 函数。

您可以使用 MMU 分页表控制 MMU。

虚拟内存组件使应用程序可以控制 MMU 分页表。

下图显示了虚拟地址、MMU 分页表、MMU 分页、物理分页以及物理地址之间的关系。

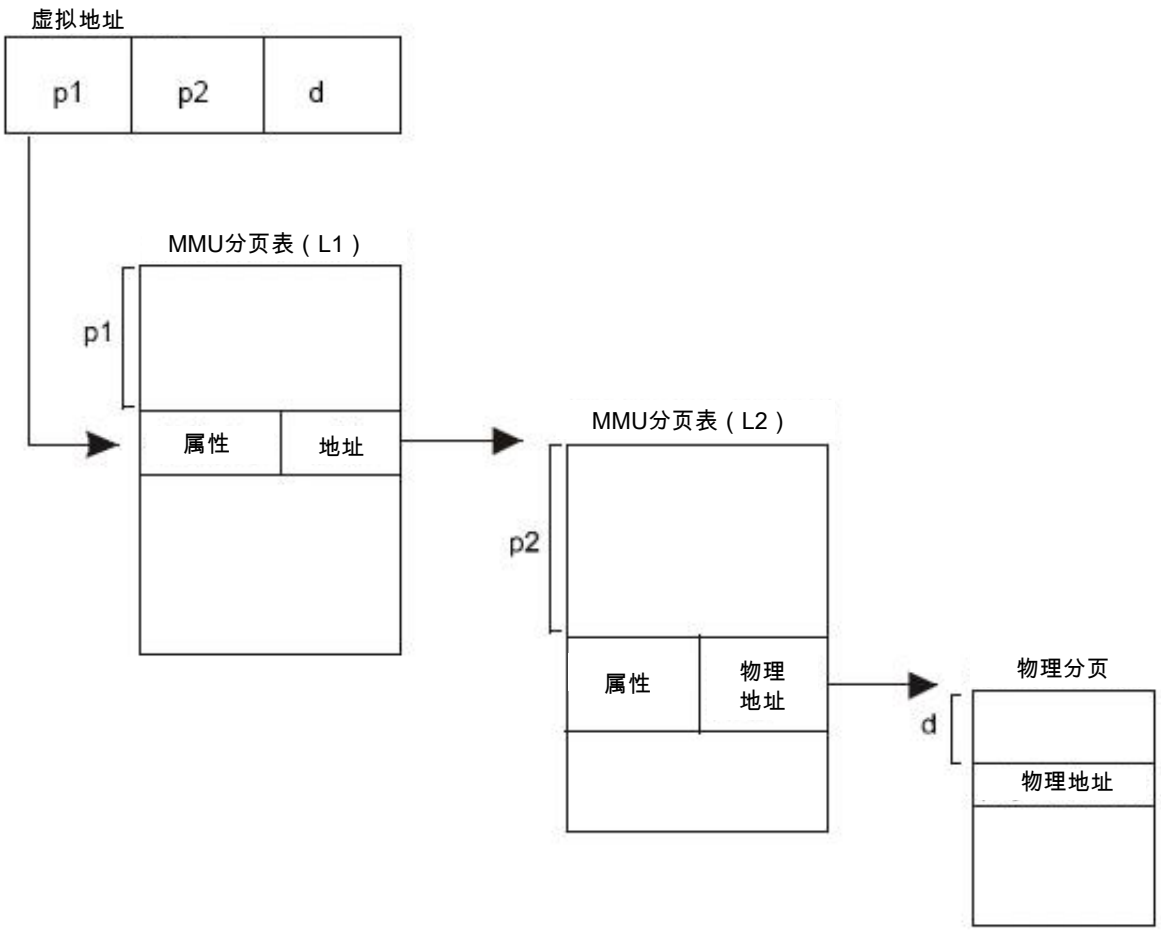

图 3-2. 虚拟和物理地址。

通过虚拟内存组件，应用程序可以管理映射到物理地址的虚拟内存。

应用程序可以使用虚拟内存组件来为任务创建虚拟上下文。虚拟上下文提供任务专用的内存，且仅当任务为活动任务时才可用。

这些函数在 BSP 初始化时调用。

表 3-12. 汇总：管理虚拟内存

_mmu_add_vcontext	添加内存区域到虚拟上下文。
_mmu_add_vregion	添加内存区域到所有任务和 MQX RTOS 都可以使用的 MMU 分页表。
_mmu_create_vcontext	为任务创建虚拟上下文。
_mmu_create_vtask	使用已初始化的虚拟上下文创建任务。
_mmu_destroy_vcontext	为任务销毁虚拟上下文。
_mmu_get_vmem_attributes	获取 MMU 分页的虚拟内存属性。
_mmu_get_vpage_size	获取 MMU 分页的大小。
_mmu_set_vmem_attributes	修改 MMU 分页的虚拟内存属性。
_mmu_vdisable	禁用虚拟内存。

下一页继续介绍此表...

表 3-12. 汇总：管理虚拟内存 (继续)

_mmu_venable	启用虚拟内存。
_mmu_vinit	初始化 MMU 以使用 MMU 分页表。
_mmu_vtop	获取与虚拟地址对应的物理地址。

3.5.4.1 示例：使用虚拟存储器初始化 MMU

添加一些内存区域来支持指令缓存和数据缓存。所有任务均可访问这些区域。

```
_mqx_uint _bsp_enable_operation(void)
{
   ...
   _mmu_vinit(MPC860_MMU_PAGE_SIZE_4K, NULL);
   /* Set up and initialize the instruction cache: */
   _mmu_add_vregion(BSP_FLASH_BASE, BSP_FLASH_BASE,
      BSP_FLASH_SIZE, PSP_MMU_CODE_CACHE | PSP_MMU_CACHED);
   _mmu_add_vregion(BSP_DIMM_BASE, BSP_DIMM_BASE, BSP_DIMM_SIZE,
      PSP_MMU_CODE_CACHE | PSP_MMU_CACHED);
   _mmu_add_vregion(BSP_RAM_BASE, BSP_RAM_BASE, BSP_RAM_SIZE,
      PSP_MMU_CODE_CACHE | PSP_MMU_CACHED);
   /* Set up and initialize the data cache: */
   _mmu_add_vregion(BSP_FLASH_BASE, BSP_FLASH_BASE,
      BSP_FLASH_SIZE, PSP_MMU_DATA_CACHE |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_PCI_MEMORY_BASE, BSP_PCI_MEMORY_BASE,
      BSP_PCI_MEMORY_SIZE, PSP_MMU_DATA_CACHE |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_PCI_IO_BASE, BSP_PCI_IO_BASE,
      BSP_PCI_IO_SIZE, PSP_MMU_DATA_CACHE |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_DIMM_BASE, BSP_DIMM_BASE, BSP_DIMM_SIZE,
      PSP_MMU_DATA_CACHE | PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_RAM_BASE, BSP_RAM_BASE,
      BSP_COMMON_RAM_SIZE, PSP_MMU_DATA_CACHE |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_venable();
   _ICACHE_ENABLE(0);
   _DCACHE_ENABLE(0);
   ...
}
```

3.5.4.2 示例：设置虚拟上下文

设置活动任务以访问位于 0xA0000000 的 64 KB 专用内存。

```
...
{
void *   virtual_mem_ptr;
uint32_t   size;
virtual_mem_ptr = (void *)0xA0000000;
size = 0x10000L;
...
result = _mmu_create_vcontext(MQX_NULL_TASK_ID);
if (result != MQX_OK) {
}
result = _mmu_add_vcontext(MQX_NULL_TASK_ID,
   virtual_mem_ptr, size, 0);
```

```
if (result != MQX_OK) {
}
...
```

3.5.4.3 示例：使用虚拟上下文创建任务

使用虚拟上下文和通用数据副本创建任务。

```
...
/* Task template number for the virtual-context task: */
#define VMEM_TTN         10
/* Global variable: */
unsigned char * data_to_duplicate[0x10000] = { 0x1, 0x2, 0x3 };
...
{
void *   virtual_mem_ptr;
virtual_mem_ptr = (void *)0xA0000000;
...
result = _mmu_create_vtask(VMEM_TTN, 0, &data_to_duplicate,
   virtual_mem_ptr, sizeof(data_to_duplicate), 0);
if (result == MQX_NULL_TASK_ID) {
}
result = _mmu_create_vtask(VMEM_TTN, 0, &data_to_duplicate,
   virtual_mem_ptr, sizeof(data_to_duplicate), 0);
if (result == MQX_NULL_TASK_ID) {
  }
...
}
```

3.6 任务的同步

您可以通过以下一种或多种机制对任务进行同步，这些机制将在后续章节中说明:

- 事件—任务可以等待一组事件位被置位。一个任务可以置位或清除该组事件位。
- 轻量级事件—事件的简化实现。
- 信号量—任务可以等待信号量从非零开始递增。任务可以发出 (递增) 信号量。MQX 信号量通过优先级继承防止优先级倒置。关于优先级倒置的讨论，请参见优先级倒置 。
- 轻量级信号量—简单计数信号量。
- 互斥—任务可以通过互斥来确保同一时刻仅有一个任务访问共用数据。为了访问共用数据，任务会锁定互斥量，如果该互斥量已被锁定，任务则等待。当任务完成对共用数据的访问之后，其会解锁该互斥量。互斥通过优先级继承和优先级保护来防止优先级倒置。欲了解详细信息，请参见互斥量 。
- 消息传递—允许在任务之间传输数据。任务在消息中填充数据，并将其发送到指定的消息队列。另一个任务等待消息到达消息队列 (接收消息)。
- 轻量级消息队列—消息的简化实现。
- 任务队列—允许应用程序挂起并恢复任务。

3.6.1 事件

事件可用于与另一任务或与 ISR 进行同步。

事件组件由事件组组成，事件组是事件位的集合。事件组中的事件位数量为 **_mqx_uint** 中的位数量。

任何任务都可以等待事件组中的事件位。如果事件位没有置位，则任务阻塞。另一个任务或 ISR 可以将该事件位置位。在事件位置位时，MQX RTOS 将符合等待条件的所有等待任务放入任务就绪队列。如果事件组在创建时带有自动清除事件位功能，则 MQX RTOS 会在这些事件位置位后立即清除，并将任务设为就绪状态。

附注	为了在某些目标平台上优化代码和数据存储器要求，事件组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

有两种事件组：命名事件组，其通过唯一的字符串名称标识，以及快速事件组，其通过唯一的编号标识。

应用程序可以打开远程处理器上的事件组，方法是在用于打开该事件组的字符串中指定处理器编号。打开远程处理器事件组后，应用程序可以将该事件组中的任何事件位置位。然而应用程序不能等待远程事件组中的事件位。

表 3-13. 汇总：使用事件组件

事件 [1]	描述
_event_clear	将事件组中的指定事件位清除。
_event_close	关闭与事件组的连接。
_event_create	创建命名事件组。
_event_create_auto_clear	创建一个支持自动清除事件位的事件组。
_event_create_component	创建事件组件。
_event_create_fast	创建快速事件组。
_event_create_fast_auto_clear	创建一个支持自动清除事件位的快速事件组。
_event_destroy	销毁命名事件组。
_event_destroy_fast	销毁快速事件组。
_event_get_value	获取事件组的值。
_event_get_wait_count	获取等待事件组中的事件位的任务数。
_event_open	打开与命名事件组的连接。
_event_open_fast	打开与快速名事件组的连接。
_event_set	将本地处理器或远程处理器上的事件组中的指定事件位置位。
_event_test	测试事件组件。
_event_wait_all	在指定的毫秒数内，等待事件组中的所有指定的事件位被置位。

下一页继续介绍此表...

表 3-13. 汇总：使用事件组件 (继续)

_event_wait_all_for	在指定的滴答时间周期内（包括硬件滴答时间），等待事件组中的所有指定的事件位。
_event_wait_all_ticks	在指定的滴答数内，等待事件组中的所有指定的事件位被置位。
_event_wait_all_until	等待事件组中的所有指定的事件位，直到指定的滴答时间结束为止。
_event_wait_any	在指定的毫秒数内，等待事件组中的任意指定的事件位被置位。
_event_wait_any_for	在指定的滴答时间周期内，等待事件组中的任意指定的事件位被置位。
_event_wait_any_ticks	在指定的滴答数内，等待事件组中的任意指定的事件位被置位。
_event_wait_any_until	等待事件组中的任意指定的事件位被置位，直到指定的滴答时间结束为止。

1. 事件使用某些在 event.h 中定义的结构和常数。

3.6.1.1 创建事件组件

您可以使用**_event_create_component()**显式地创建事件组件。如果您不显式地创建该组件，MQX RTOS 会使用应用程序首次创建事件组时的默认值创建。

表 3-14. 事件组件默认值

参数	含义	默认值
初始数量	可以创建的事件组的初始数量	8
增加数量	创建所有事件组后，达到最大数量之前可以创建的额外事件组数	8
最大数量	增加数量不为 0 时可以创建的最大事件组数	0（无限）

3.6.1.2 创建事件组

要使用事件组件，任务必须先创建一个事件组。

表 3-15. 事件组创建

要创建此类型的事件组:	调用:	使用:
快速（带自动清除事件位）	**_event_create_fast()** **_event_create_fast_ auto_clear()**	索引（必须在创建事件组件时指定的限制范围内）
命名（带自动清除事件位）	**_event_create()** **_event_create_auto_ clear()**	字符串名称

如果事件组在创建时带有自动清除事件位，则 MQX RTOS 会在这些位置位后立即清除。此操作将使等待这些位的所有任务就绪，不需要任务去清除这些事件位。

3.6.1.3 打开与事件组的连接

在任务可以使用事件组件之前，其必须打开一个与已创建事件组的连接。

表 3-16. 事件组打开

为了打开与此类型事件组的连接:	调用:	使用:
快速	**_event_open_fast()**	索引，必须在创建事件组件时指定的限制范围内。
命名	**_event_open()**	字符串名称

两个函数均会返回这个事件组的唯一句柄。

3.6.1.4 等待事件位（多个事件）

任务可以使用 **_event_wait_all()**或**_event_wait_any()**等待事件组中的一组事件位被置位（一个掩码)。当一个位被置位时，MQX RTOS 使等待该位的任务处于就绪状态。如果事件组在创建时带有自动清除事件位功能（**_event_create_auto_clear()**或**_event_create_fast_auto_clear()**），则 MQX RTOS 会清除该位，而等待任务无需对该位执行清除。

3.6.1.5 设置事件位

任务可以使用**_event_set()**将事件组中的一组事件位模式（通过掩码）置位。事件组可位于本地或远程处理器上。当事件位置位时，等待该位的任务变为就绪状态。如果事件组在创建时带有自动清除事件位功能，则 MQX RTOS 会在这些位置位后立即清除。

3.6.1.6 清除事件位

任务可以使用**_event_clear()**将事件组中的一组事件位（通过掩码）清除。但如果事件组在创建时带有自动清除事件位功能，则 MQX RTOS 会在这些位置位后立即清除它们。

3.6.1.7 关闭与事件组的连接

当任务不再需要使用事件组时，可以使用**_event_close()**关闭与该事件组的连接。

3.6.1.8 销毁事件组

如果任务被阻塞，并且正在等待即将销毁的事件组中的事件位，MQX RTOS 会将它们移至其就绪队列。

3.6.1.9 示例：使用事件

Simulated_tick ISR 在每次运行时将某个事件位置位。服务任务在每次发生滴答信号时执行特定操作，及等待 Simulated_tick 将该事件位置位。

3.6.1.9.1 使用事件的代码示例

```
/* event.c */
#include <mqx.h>
#include <fio.h>
#include <event.h>
/* Task IDs */
#define SERVICE_TASK 5
#define ISR_TASK     6
/* Function Prototypes */
extern void simulated_ISR_task(uint32_t);
extern void service_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index,  Function,          Stack,Prio, Name,           Attributes,          Param,
TS */
 { SERVICE_TASK, service_task,      2000,  8, "service",      MQX_AUTO_START_TASK, 0, 0},
 { ISR_TASK,    simulated_ISR_task, 2000,  8, "simulated_ISR", 0,                   0, 0},
 { 0 }
};
/*TASK*-----------------------------------------------------
*
* Task Name    : simulated_ISR_task
* Comments     :
*    This task opens a connection to the event. After

         *    delaying the event bits are set.

         *END*-----------------------------------------------------*/

         void simulated_ISR_task(uint32_t  initial_data)

         {

            void *    event_ptr;

            /* open event connection */

            if (_event_open("global", &event_ptr) != MQX_OK) {

               printf("\nOpen Event failed");

               _mqx_exit(0);

            }

            while (TRUE) {

               _time_delay(1000);

               if (_event_set(event_ptr, 0x01) != MQX_OK) {
```

```
                printf("\nSet Event failed");

                _mqx_exit(0);
            }
        }
    }
    /*TASK*-----------------------------------------------------
    *
    * Task Name    : service_task
    * Comments     :
    *    This task creates an event and the simulated_ISR_task
    *    task. It opens a connection to the event and waits.
    *    After all bits have been set "Tick" is printed and
    *    the event is cleared.
    *END*-----------------------------------------------------*/
    void service_task(uint32_t initial_data)
    {
       void *   event_ptr;
       _task_id  second_task_id;
       /* setup event */
       if (_event_create("global") != MQX_OK) {
          printf("\nMake event failed");
          _mqx_exit(0);
       }
       if (_event_open("global", &event_ptr) != MQX_OK) {
          printf("\nOpen event failed");
          _mqx_exit(0);
       }
       /* create task */
       second_task_id = _task_create(0, ISR_TASK, 0);
          if (second_task_id == MQX_NULL_TASK_ID) {
             printf("Could not create simulated_ISR_task \n");
             _mqx_exit(0);
          }
       while (TRUE) {
          if (_event_wait_all(event_ptr, 0x01, 0) != MQX_OK) {
             printf("\nEvent Wait failed");
             _mqx_exit(0);
          }
          if (_event_clear(event_ptr, 0x01) != MQX_OK) {
             printf("\nEvent Clear Failed");
             _mqx_exit(0);
          }
          printf(" Tick \n");
       }
    }
```

3.6.1.9.2 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\event
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档《Freescale MQX™ RTOS for Kinetis SDK 入门》(文档 MQXKSDKGSUG) 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序

每次将事件位置位，事件任务便输出一条消息。

附注	借助 Freescale MQX，CodeWarrior Development Studio 成为了 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。

3.6.2 轻量级事件

轻量级事件是事件的简单化、低开销实现。

轻量级事件组件由轻量级事件组组成，轻量级事件组是事件位的集合。轻量级事件组中的事件位数量为**_mqx_uint** 中的位数量。

任何任务可以等待轻量级事件组中的事件位。如果事件位没有置位，则任务阻塞。另一个任务或 ISR 可以将事件位置位。在事件位置位时，MQX RTOS 将符合等待条件的所有等待任务放入任务就绪队列。如果轻量级事件组在创建时带有自动清除事件位，则 MQX RTOS 会在这些事件位置位后立即清除，并将任务设为就绪状态。

轻量级事件组由静态数据结构创建，并且不支持多处理器。

表 3-17. 汇总：使用轻量级事件组件

事件 [1]	说明
_lwevent_clear	将轻量级事件组中的指定事件位清除。
_lwevent_create	创建轻量级事件组，指示其是否具有自动清除事件位。
_lwevent_destroy	销毁轻量级事件组。
_lwevent_set	将轻量级事件组中的指定事件位置位。
_lwevent_test	测试轻量级事件组件。
_lwevent_wait_for	在指定的滴答时间周期内，等待轻量级事件组中的所有或任何指定的事件位被置位。
_lwevent_wait_ticks	在指定的滴答数内，等待轻量级事件组中的所有或任何指定的事件位被置位。
_lwevent_wait_until	等待轻量级事件组中的所有或任何指定的事件位被置位，直到指定的滴答时间结束为止。

1. 轻量级事件使用某些在 lwevent.h 中定义的结构和常数。

3.6.2.1 创建轻量级事件组

要创建轻量级事件组，应用程序要声明类型为 **LWEVENT_STRUCT** 的变量，并通过使用指向该变量的指针和指示事件组是否具有自动清除事件位的标志调用 **_lwevent_create()**来初始化该变量。

3.6.2.2 等待事件位

任务通过**_lwevent_wait** 函数等待轻量级事件组中的某些事件位（通过掩码）被置位。如果不符合等待条件，函数会等待一段指定时间后结束。

3.6.2.3 设置事件位

任务可以使用**_lwevent_set()**将轻量级事件组中的某些事件位（通过掩码）被置位。如果任务等到了合适的位, MQX RTOS 会使其就绪。如果事件组具有自动清除事件位功能，则 MQX RTOS 仅使第一个等待的任务就绪。

3.6.2.4 清除事件位

任务可以使用**_lwevent_clear()**将轻量级事件组中的某些事件位（通过掩码）清除。但如果轻量级事件组在创建时带有自动清除事件位功能，则 MQX RTOS 会在这些位置位后立即清除它们。

3.6.2.5 销毁轻量级事件组

当任务不再需要轻量级事件组时，可以使用**_lwevent_destroy()**销毁该事件组。

3.6.3 关于信号量类型对象

MQX RTOS 提供轻量级信号量（LWSem）、信号量和互斥功能。

您可以使用不同类型的信号量实现任务同步和互斥操作。任务等待信号量。如果信号量计数为零，则 MQX RTOS 阻塞该任务；否则，MQX RTOS 降低信号量计数，并为该任务提供信号量，该任务继续运行。如果有任务在使用此信号量的情况下完成，它会传递该信号量；任务保持就绪状态。如果有任务正在等待信号量，MQX RTOS 将该任务置入任务就绪队列；否则，MQX RTOS 增加信号量计数。

您可以使用互斥量实现互斥操作。互斥量有时也被称为二进制信号量，因为它的计数只能是 0 或 1 。

3.6.3.1 严谨性

如果信号量类型的对象是严谨的，则任务在释放对象前必须先等待并获取该对象。如果对象为非严谨，则任务在释放该对象前无需获取。

3.6.3.2 优先级倒置

任务的优先级倒置是一种典型情况，此时相关任务的优先级会出现倒置。当任务使用信号量或互斥操作来获取访问共享资源时，可能会发生优先级倒置。

3.6.3.3 示例：优先级倒置

有三种不同优先级的任务。中等优先级任务可阻止最高优先级任务运行。

表 3-18. 优先级倒置示例

序号	Task_1（最高优先级 P1）	Task_2（中等优先级 P2）	Task_3（最低优先级 P3）
1 2			• 运行 • 获取信号量
3		• 置于就绪状态	
4		• 抢占 Task_3 并运行	
5	• 置于就绪状态		
6	• 抢占 Task_2 并运行		
7	• 尝试获取 Task_3 持有的信号量		
8	• 阻塞，等待信号量		
9		• 运行并保持运行	

3.6.3.4 使用优先级继承避免优先级倒置

在您创建 MQX 信号量或互斥量时，您所能够指定的属性之一是防止优先级倒置的优先级继承。

如果您指定优先级继承，在任务锁定信号量或互斥量期间，该任务的优先级永远不会低于等待该信号量或互斥量的任何任务的优先级。如果较高优先级任务在等待该信号量或互斥量，MQX RTOS 会将持有该信号量或互斥量的任务的优先级临时提高到正在等待的任务的优先级。

表 3-19. 优先级继承属性

序号	Task_1（最高优先级 P1）	Task_2（中等优先级 P2）	Task_3（最低优先级 P3）
1			• 运行
2			• 获取信号量
3		• 置于就绪状态	
4		• 抢占 Task_3 并运行	
5	• 置于就绪状态		
6	• 抢占 Task_2 并运行		
7	• 尝试获取 Task_3 持有的信号量		

下一页继续介绍此表...

表 3-19. 优先级继承属性 (继续)

8	• 将 Task_3 的优先级提升到 P1 并阻塞		
9			• 抢占 Task_1 并运行
10			• 完成工作并传递信号量
11			• 优先级降至 P3
12	• 抢占 Task_3 和 Task_2 并运行		
13	• 获取信号量		

3.6.3.5 使用优先级保护避免优先级倒置

创建 MQX 互斥时，可以指定优先级保护的互斥属性和互斥优先级。这些属性可防止优先级倒置。

如果请求锁定互斥的任务的优先级不如互斥优先级高，则只要任务锁定了互斥，MQX RTOS 会将该任务的优先级临时提高至互斥优先级。

表 3-20. 互斥属性

序号	Task_1（最高优先级 P1）	Task_2（中等优先级 P2）	Task_3（最低优先级 P3）
1			• 运行
2			• 锁定互斥（优先级 P1）；优先级提升至 P1
3		• 置于就绪状态	
4		• 不抢占 Task_3	
5	• 置于就绪状态		
6	• 不抢占 Task_3		
7			• 以互斥方式完成并解锁
8			• 优先级降至 P3
9	• 抢占 Task_3 并运行		
10	• 锁定互斥		

表 3-21. 轻量级信号量、信号量和互斥量的比较

特性	LWSem	信号量	互斥量
超时	是	是	否
排队	FIFO	FIFO 优先级	FIFO 优先级仅自旋并且为限定自旋
严谨	否	否或是	是
优先级继承	否	是	是
优先级保护	否	否	是
大小	最小	最大	介于轻量级信号量与信号量之间

下一页继续介绍此表...

表 3-21. 轻量级信号量、信号量和互斥量的比较 (继续)

速度	最快	最慢	介于轻量级信号量与信号量之间

3.6.4 轻量级信号量

轻量级信号量是信号量的简单化、低开销实现。

轻量级信号量由静态数据结构创建，并且不支持多处理器。

表 3-22. 汇总：使用轻量级信号量

_lwsem_create	创建轻量级信号量。
_lwsem_destroy	销毁轻量级信号量。
_lwsem_poll	轮询轻量级信号量（非阻塞）。
_lwsem_post	发送轻量级信号量。
_lwsem_test	测试轻量级信号量组件。
_lwsem_wait	等待轻量级信号量。
_lwsem_wait_for	在指定滴答时间周期内等待轻量级信号量。
_lwsem_wait_ticks	在指定滴答数内等待轻量级信号量。
_lwsem_wait_until	等待轻量级信号量，直到指定滴答数结束为止。

3.6.4.1 创建轻量级信号量

要创建轻量级信号量，要声明类型为 **LWSEM_STRUCT** 的变量，并通过使用该变量的指针和初始信号量值调用_lwsem_create()来进行初始化。信号量的值表示可以被这个信号量同时服务的请求的个数，设置为初始计数值。

3.6.4.2 等待并传递轻量级信号量

任务通过**_lwsem_wait()**函数等待轻量级信号量。如果信号量计数大于零，则 MQX RTOS 递减该计数，任务继续运行。如果计数为零，则 MQX RTOS 阻塞该任务，直到某个其他任务传递了该轻量级信号量。

为了释放轻量级信号量，任务通过**_lwsem_post()**函数进行传递。如果没有任务在等待该轻量级信号量，MQX RTOS 会递增该信号量的计数。

由于轻量级信号量为非严谨，任务无需等待便能进行传递，因此信号量计数没有限制，并且可以递增超出其初始计数。

3.6.4.3 销毁轻量级信号量

当任务不再需要轻量级信号量时，可以使用**_lwsem_destroy()**销毁它。

3.6.4.4 示例：生产者和消费者

生产者和消费者任务使用轻量级信号量彼此同步。

1. 读取任务创建:
   - 多个写入任务，并给每个任务分配一个唯一的字符。
   - 一个写入 LWSem。
   - 一个读取 LWSem。
2. 每个写入任务在写入字符到缓冲区之前都必须等待写入 LWSem。当字符被写入时，每个写入任务传递读取 LWSem，表示字符现在可供读取任务读取。
3. 读取任务在读取该字符前必须等待读取 LWSem。在读取该字符后，读取任务会传递写信号量，表示缓冲区现在已经可以写入其它字符。

3.6.4.4.1 示例的定义和结构

```
/* read.h */
/* Number of Writer Tasks */
#define NUM_WRITERS  3
/* Task IDs */
#define WRITE_TASK   5
#define READ_TASK    6
/* Global data structure accessible by read and write tasks.
** Contains two lightweight semaphores that govern access to the
** data variable.
*/
typedef struct sw_fifo
{
   LWSEM_STRUCT   READ_SEM;
   LWSEM_STRUCT   WRITE_SEM;
   uchar          DATA;
} SW_FIFO, _PTR_ SW_FIFO_PTR;
/* Function prototypes */
extern void write_task(uint32_t initial_data);
extern void read_task(uint32_t initial_data);
extern SW_FIFO fifo;
```

3.6.4.4.2 用于生产者与消费者的任务模板示例

```
/*  ttl.c */
#include <mqx.h>
#include <bsp.h>
#include "read.h"
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,    Stack, Priority, Name,    Attributes,            Param, Time
Slice */
  { WRITE_TASK, write_task, 1000,  8,       "write", 0,                    0,     0 },
  { READ_TASK,  read_task,  1000,  8,       "read",  MQX_AUTO_START_TASK, 0,     0 },
```

```
  { 0 }
};
```

3.6.4.4.3 写入任务的代码

```
/* write.c */
#include <mqx.h>
#include <bsp.h>
#include "read.h"
/*TASK*-------------------------------------------------------
*
* Task Name : write_task
* Comments  : This task waits for the write semaphore,
**            then writes a character to "data" and posts a
*            read semaphore.
*END*-------------------------------------------------------*/
void write_task(uint32_t initial_data)
{
   printf("\nWrite task created: 0x%lX", initial_data);
   while (TRUE) {
      if (_lwsem_wait(&fifo.WRITE_SEM) != MQX_OK) {
         printf("\n_lwsem_wait failed");
         _mqx_exit(0);
      }
      fifo.DATA = (uchar)initial_data;
      _lwsem_post(&fifo.READ_SEM);
   }
}
```

3.6.4.4.4 读取任务的代码

```
/* read.c */
#include <mqx.h>
#include <bsp.h>
#include "read.h"
SW_FIFO     fifo;
/*TASK*-------------------------------------------------------
*
* Task Name : read_task
* Comments  : This task creates two semaphores and
*             NUM_WRITER write_tasks. Then it waits
*             on the read_sem and finally outputs the
*             "data" variable.
*END*-------------------------------------------------------*/
void read_task(uint32_t initial_data)
{
   _task_id    task_id;
   _mqx_uint   result;
   _mqx_uint   i;
   /* Create the lightweight semaphores */
   result = _lwsem_create(&fifo.READ_SEM, 0);
   if (result != MQX_OK) {
      printf("\nCreating read_sem failed: 0x%X", result);
      _mqx_exit(0);
   }
   result = _lwsem_create(&fifo.WRITE_SEM, 1);
   if (result != MQX_OK) {
      printf("\nCreating write_sem failed: 0x%X", result);
      _mqx_exit(0);
   }
   /* Create write tasks */
   for (i = 0; i < NUM_WRITERS; i++) {
      task_id = _task_create(0, WRITE_TASK, (uint32_t)('A' + i));
      printf("\nwrite_task created, id 0x%lX", task_id);
   }
   while (TRUE) {
```

```
      result = _lwsem_wait(&fifo.READ_SEM);
      if (result != MQX_OK) {
         printf("\n_lwsem_wait failed: 0x%X", result);
         _mqx_exit(0);
      }
      putchar('\n');
      putchar(fifo.DATA);
      _lwsem_post(&fifo.WRITE_SEM);
   }
}
```

3.6.4.4.5 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\lwsem
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序。

输出设备上将显示以下内容:

```
A

A

B

C

A

B

...
```

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 成为了 MQX RTOS 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。

3.6.5 信号量

信号量可用于实现任务同步和互斥操作。任务对信号量的主要操作是等待并传递信号量。

附注	为了在某些目标平台上优化代码和数据存储器要求，信号量组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

表 3-23. 汇总：使用信号量

信号量 [1]	说明
_sem_close	关闭与信号量的连接。
_sem_create	创建信号量。
_sem_create_component	创建信号量组件。
_sem_create_fast	创建快速信号量。
_sem_destroy	销毁已命名的信号量。
_sem_destroy_fast	销毁快速信号量。
_sem_get_value	获取当前信号量计数。
_sem_get_wait_count	获取等待信号量的任务数量。
_sem_open	打开与已命名信号量的连接。
_sem_open_fast	打开与快速信号量的连接。
_sem_post	传递（释放）信号量。
_sem_test	测试信号量组件。
_sem_wait	在一定毫秒数内等待信号量。
_sem_wait_for	在滴答时间周期内等待信号量。
_sem_wait_ticks	在一定滴答数内等待信号量。
_sem_wait_until	等待信号量，直到一段时间（以滴答时间为单位）为止。

1. 信号量使用某些在 sem.h 中定义的结构和常数。

3.6.5.1 使用信号量

为了使用信号量，任务需执行以下步骤，每个步骤都将在后续章节中进行说明。

1. 创建信号量组件（可选）。
2. 创建信号量。
3. 打开与信号量的连接。
4. 如果信号量是严谨的，任务会等待该信号量。
5. 当任务暂时完成使用该信号量后，它会传递该信号量。
6. 如果任务不再需要该信号量，它会关闭与该信号量的连接。
7. 如果信号量正在保护一个共享资源，而该共享资源已不存在或不能访问时，任务会销毁该信号量。

3.6.5.2 创建信号量组件

您可以使用**_sem_create_component()**显式地创建信号量组件。如果您不显式地创建该组件，MQX RTOS 会使用应用程序首次创建信号量时的默认值创建。

参数及默认值与事件组件的相同，如第创建事件组件 页所述。

3.6.5.3 创建信号量

要使用信号量，任务必须先创建该信号量。

表 3-24. 信号量创建

信号量类型	调用	使用
快速	**_sem_create_fast()**	索引，必须在创建信号量组件时指定的限制范围内。
命名	**_sem_create()**	字符串名称

当任务创建信号量时，还会指定:

- 初始计数值 - 信号量计数的初始值代表该信号量具有的锁定数。(一个任务可以获取多个锁定)。
- 优先级排序 - 如果指定了优先级排序，等待信号量的任务队列按优先级排序，MQX RTOS 将信号量给优先级最高的等待任务。
- 如果未指定优先级排序，则队列按 FIFO 顺序，MQX RTOS 将信号量给等待最久的任务。
- 优先级继承 - 如果指定了优先级继承，且较高优先级的任务在等待信号量，则 MQX RTOS 会将持有该信号量的任务的优先级提升至等待任务的优先级。有关详细信息，请参阅第使用优先级继承避免优先级倒置 页关于优先级继承的讨论。要使用优先级继承，信号量必须为严谨信号量。
- 严谨性 - 如果指定了严谨性，任务必须在等待信号量之后才能传递该信号量。如果信号量是严谨信号量，则初始计数值为信号量计数的最大值。如果信号量不是严谨信号量，则计数不受限制。

3.6.5.4 打开与信号量的连接

在任务可以使用信号量之前，必须打开一个与该信号量的连接。

表 3-25. 打开与信号量的连接

信号量类型	调用	使用
快速	**_sem_open_fast()**	索引，必须在创建信号量组件时指定的限制范围内。
命名	**_sem_open()**	字符串名称

两个函数均会返回信号量的唯一句柄。

3.6.5.5 等待信号量与传递信号量

任务调用函数系列中的**_sem_wait_**函数用以等待信号量。如果信号量计数为零，MQX RTOS 阻塞该任务，直到另一个任务传递（使用**_sem_post()**函数）该信号量或任务指定等待时间超时。如果计数不等于零，则 MQX RTOS 递减该计数，任务继续运行。

当任务传递信号量的同时还有其他任务等待该信号量，MQX RTOS 会将这些任务置入就绪队列。如果没有任务在等待，MQX RTOS 会递增该信号量的计数。无论如何，传递任务保持就绪状态。

3.6.5.6 关闭与信号量的连接

当任务不再需要使用信号量时，可以使用_sem_close()关闭它与信号量的连接。

3.6.5.7 销毁信号量

当任务不再需要信号量时，可以销毁它。

表 3-26. 信号量销毁

信号量类型	调用	使用
快速	**_sem_destroy_fast()**	索引，必须在创建信号量组件时指定的限制范围内。
命名	**_sem_destroy()**	字符串名称

任务也可以指定是否强制销毁。如果强制销毁，MQX RTOS 会为等待信号量的任务做好准备，并在所有具有该信号量的任务传递信号量后销毁它。

如果不是强制销毁，MQX RTOS 会在最后等的任务获得并传递信号量后销毁该信号量。(这是严谨信号量的默认操作)。

3.6.5.8 示例：任务同步和互斥

此示例在第示例: 生产者和消费者 页上的轻量级信号量基础上构建。其中显示了如何将信号量用于实现任务同步和互斥操作。

该示例管理一个 FIFO，多个任务可以写入并从中读取。互斥是访问 FIFO 数据结构所必需的。要在 FIFO 满时阻止写入任务以及在 FIFO 空时阻止读取任务，需要任务同步。使用三种信号量:

- 用于 FIFO 互斥的索引信号量。

- 同步读取函数的读取信号量。
- 同步写入函数的写入信号量。

示例包含三种任务：主要任务、读取任务和写入任务。主任务初始化信号量并创建读取任务和写入任务。

3.6.5.8.1 示例的定义和结构

```
/* main.h
** This file contains definitions for the semaphore example.
*/
#define MAIN_TASK    5
#define WRITE_TASK   6
#define READ_TASK    7
#define ARRAY_SIZE   5
#define NUM_WRITERS  2
/* Global data structure accessible by read and write tasks.
** Contains a DATA array that simulates a FIFO. READ_INDEX
** and WRITE_INDEX mark the location in the array that the read
** and write tasks are accessing. All data is protected by
** semaphores.
*/
typedef struct
{
   _task_id  DATA[ARRAY_SIZE];
   uint32_t   READ_INDEX;
   uint32_t   WRITE_INDEX;
} SW_FIFO, * SW_FIFO_PTR;
/* Function prototypes */
extern void main_task(uint32_t initial_data);
extern void write_task(uint32_t initial_data);
extern void read_task(uint32_t initial_data);
extern   SW_FIFO fifo;
```

3.6.5.8.2 用于任务同步和互斥操作的任务模板示例

```
/*  ttl.c */
#include <mqx.h>
#include "main.h"
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice */
  { MAIN_TASK,  main_task,  2000,  8,       "main",  MQX_AUTO_START_TASK,0,     0 },
  { WRITE_TASK, write_task, 2000,  8,       "write", 0,                  0,     0 },
  { READ_TASK,  read_task,  2000,  8,       "read",  0,                  0,     0 },
  { 0 }
};
```

3.6.5.8.3 主任务代码

主任务创建：

- 信号量组件
- 索引、读取和写入信号量
- 读取和写入任务

```
/* main.c */
#include <mqx.h>
```

```
#include <bsp.h>
#include <sem.h>
#include "main.h"
SW_FIFO   fifo;
/*TASK*-------------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*     This task initializes three semaphores, creates NUM_WRITERS
*     write_tasks, and creates one read_task.
*END*-------------------------------------------------------------*/
void main_task(uint32_t initial_data)
{
   _task_id   task_id;
   _mqx_uint  i;
   fifo.READ_INDEX = 0;
   fifo.WRITE_INDEX = 0;
   /* Create semaphores: */
   if (_sem_create_component(3, 1, 6) != MQX_OK) {
      printf("\nCreating semaphore component failed");
      _mqx_exit(0);
   }
   if (_sem_create("write", ARRAY_SIZE, 0) != MQX_OK) {
      printf("\nCreating write semaphore failed");
      _mqx_exit(0);
   }
   if (_sem_create("read", 0, 0) != MQX_OK) {
      printf("\nCreating read semaphore failed");
      _mqx_exit(0);
   }
   if (_sem_create("index", 1, 0) != MQX_OK) {
      printf("\nCreating index semaphore failed");
      _mqx_exit(0);
   }
   /* Create tasks: */
   for (i = 0; i < NUM_WRITERS; i++) {
      task_id = _task_create(0, WRITE_TASK, i);
      printf("\nwrite_task created, id 0x%lx", task_id);
   }
   task_id = _task_create(0, READ_TASK, 0);
   printf("\nread_task created, id 0x%lx", task_id);
}
```

3.6.5.8.4 读取任务的代码

```
/* read.c */
#include <mqx.h>
#include <bsp.h>
#include <sem.h>
#include "main.h"
/*TASK*-------------------------------------------------------------
* Task Name : read_task
* Comments  :
*     This task opens a connection to all three semaphores, then
*     waits to lock a read semaphore and an index semaphore. One
      element in the DATA array is displayed. The index and write
      semaphores are then posted.
*END*-------------------------------------------------------------*/
void read_task(uint32_t initial_data)
{
   void *  write_sem;
   void *  read_sem;
   void *  index_sem;
   /* Open connections to all semaphores: */
   if (_sem_open("write", &write_sem) != MQX_OK) {
      printf("\nOpening write semaphore failed");
      _mqx_exit(0);
   }
```

```
   if (_sem_open("index", &index_sem) != MQX_OK) {
      printf("\nOpening index semaphore failed");
      _mqx_exit(0);
   }
   if (_sem_open("read", &read_sem) != MQX_OK) {
      printf("\nOpening read semaphore failed");
      _mqx_exit(0);
   }
   while (TRUE) {
      /* Wait for the semaphores: */
      if (_sem_wait(read_sem, 0) != MQX_OK) {
         printf("\nWaiting for read semaphore failed");
         _mqx_exit(0);
      }
      if (_sem_wait(index_sem, 0) != MQX_OK) {
        printf("\nWaiting for index semaphore failed");
        _mqx_exit(0);
      }
      printf("\n 0x%lx", fifo.DATA[fifo.READ_INDEX++]);
      if (fifo.READ_INDEX >=ARRAY_SIZE) {
         fifo.READ_INDEX = 0;
      }
      /* Post the semaphores: */
      _sem_post(index_sem);
      _sem_post(write_sem);
   }
}
```

3.6.5.8.5 写入任务的代码

```
/* write.c */
#include <mqx.h>
#include <bsp.h>
#include <sem.h>
#include "main.h"
/*TASK*-------------------------------------------------------------
* Task Name : write_task
* Comments  :
*     This task opens a connection to all three semaphores, then
*     waits to lock a write and an index semaphore. One element
*     in the DATA array is written to. The index and read
      semaphores are then posted.
*END*-------------------------------------------------------------*/
void write_task(uint32_t initial_data)
{
   void *  write_sem;
   void *  read_sem;
   void *  index_sem;
   /* Open connections to all semaphores: */
   if (_sem_open("write", &write_sem) != MQX_OK) {
      printf("\nOpening write semaphore failed");
      _mqx_exit(0);
   }
   if (_sem_open("index", &index_sem) != MQX_OK) {
      printf("\nOpening index semaphore failed");
      _mqx_exit(0);
   }
   if (_sem_open("read", &read_sem) != MQX_OK) {
      printf("\nOpening read semaphore failed");
      _mqx_exit(0);
   }
   while (TRUE) {
      /* Wait for the semaphores: */
      if (_sem_wait(write_sem, 0) != MQX_OK) {
         printf("\nWaiting for write semaphore failed");
         _mqx_exit(0);
      }
      if (_sem_wait(index_sem, 0) != MQX_OK) {
```

```
            printf("\nWaiting for index semaphore failed");
            _mqx_exit(0);
         }
         fifo.DATA[fifo.WRITE_INDEX++] = _task_get_id();
         if (fifo.WRITE_INDEX >=ARRAY_SIZE) {
            fifo.WRITE_INDEX = 0;
         }
         /* Post the semaphores: */
         _sem_post(index_sem);
         _sem_post(read_sem);
      }
   }
```

3.6.5.8.6 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   \mqx\examples\sem
   ```

2. 请参阅《MQX RTOS 版本注释》获取有关如何构建和运行该应用程序的说明。
3. 按照《MQX RTOS 版本注释》中的说明运行该应用程序

读任务输出写入到 FIFO 的数据。

修改程序以删除优先级继承，然后再次运行应用程序。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS 入门*》文档以获取有关支持的工具链的更多详细信息。

3.6.6 互斥量

互斥量用来实现互斥操作，从而保证同一时刻只有一个任务使用共享资源，如数据或器件。要访问共享资源,任务需要锁定与资源相关的互斥量。任务将拥有互斥量,直到其解锁为止。

附注	为了在某些目标平台上优化代码和数据存储器要求，互斥组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

互斥量通过优先级继承和优先级保护来防止优先级倒置。

表 3-27. 汇总：使用互斥

互斥 [1]	说明
_mutex_create_component	创建互斥组件。
_mutex_destroy	销毁互斥。
_mutex_get_priority_ceiling	获取互斥的优先级。
_mutex_get_wait_count	获取等待互斥的任务数量。

下一页继续介绍此表...

表 3-27. 汇总：使用互斥 (继续)

_mutex_init	初始化互斥。
_mutex_lock	锁定互斥。
_mutex_set_priority_ceiling	设置互斥的优先级。
_mutex_test	测试互斥组件。
_mutex_try_lock	尝试锁定互斥。
_mutex_unlock	解锁互斥。

1. 互斥使用某些在 mutex.h 中定义的结构和常数。

3.6.6.1 创建互斥量组件

您可以使用**_mutex_create_component()**显式地创建互斥量组件。如果不显式地创建，MQX RTOS 会在应用程序首次初始化互斥时创建。没有参数。

3.6.6.2 互斥量属性

互斥量可以根据其等待和调度协议而具有相应属性。

3.6.6.3 等待策略

一个互斥量可以支持多个等待策略中的一种，这将影响请求锁定一个已锁定互斥量的任务。

表 3-28. 互斥量等待策略

等待策略 [1]	说明
排队（默认）	阻塞，直到另一个任务解锁该互斥量。当互斥量解锁后，请求锁定的第一个任务（无论优先级）会锁定该互斥量。
优先级排队	阻塞，直到另一个任务解锁该互斥量。当互斥量解锁后，请求锁定的优先级最高的任务会锁定该互斥量。
仅自旋	无限期自旋（在时间片内），直到另一个任务解锁该互斥量。这意味着 MQX RTOS 保存请求任务的上下文，并执行相同优先级就绪队列中的下一个任务。当就绪队列中的所有任务均运行后，请求任务再次激活。如果互斥量仍然被锁定，则重复自旋。
有限自旋	自旋指定次数，如果另一个任务先解锁该互斥量，则自旋次数更少。

1. 如果互斥量已锁定，请求任务将执行如下操作。

仅自旋协议仅在共享互斥量的任务属于以下情况时才正常工作:

- 时间片任务
- 相同优先级

如果具有不同优先级的非时间片任务尝试共享仅自旋互斥量时，较高优先级任务将不能锁定已被较低优先级任务锁定的互斥（除非较低优先级任务阻塞）。

仅自旋协议容易导致死锁，因而不建议使用。

3.6.6.4 调度策略

为了避免优先级倒置，互斥可以具有特殊的调度策略。这些规则可能在任务已锁定互斥期间影响到任务的优先级。默认是两个策略都不起作用。

表 3-29. 互斥调度策略

调度策略	含义
优先级继承	如果已锁定互斥任务（task_A）的优先级不比等待锁定互斥任务（task_B）的优先级高，MQX RTOS 会将 task_A 的优先级提升至与 task_B 一致，而 task_A 仍拥有互斥。
优先级保护	互斥量可以具有优先级。如果要求锁定互斥任务（task_A）的优先级不比互斥优先级高，MQX RTOS 会在 task_A 锁定互斥期间将 task_A 的优先级提升至与互斥优先级一致。

3.6.6.5 创建和初始化互斥

任务通过先定义变量类型 **MUTEX_STRUCT** 创建互斥量。

当使用默认的队列等待策略和非特定的调度策略去初始化互斥量时，任务可以使用指向互斥量的指针和 NULL 指针调用**_mutex_init()**。

但是，要初始化具有默认属性以外的其他属性的互斥量，任务需要执行以下操作:

1. 定义类型为 **MUTEX_ATTR_STRUCT** 的互斥量属性结构体。
2. 使用**_mutatr_init()**初始化属性结构体。
3. 调用各种函数来设置相应的属性，包括:
4. • **_mutatr_set_priority_ceiling()**
   • **_mutatr_set_sched_protocol()**
   • **_mutatr_set_spin_limit()**
   • **_mutatr_set_wait_protocol()**
5. 通过使用互斥量和属性结构体的指针调用**_mutex_init()**来初始化互斥量。互斥量初始化之后，任何任务都可以使用它。
6. 使用**_mutatr_destroy()**销毁互斥量属性结构体。

表 3-30. 汇总：使用互斥量属性结构体

_mutatr_destroy	销毁互斥量属性结构体。
_mutatr_get_priority_ceiling	获取互斥量属性结构体的优先级。
_mutatr_get_sched_protocol	获取互斥量属性结构体的调度策略。

下一页继续介绍此表...

表 3-30. 汇总：使用互斥量属性结构体 (继续)

_mutatr_get_spin_limit	获取互斥量属性结构体的有限旋转计数。
_mutatr_get_wait_protocol	获取互斥量属性结构体的等待策略。
_mutatr_init	初始化互斥量属性结构体。
_mutatr_set_priority_ceiling	设置互斥量属性结构体中的优先级值。
_mutatr_set_sched_protocol	设置互斥量属性结构体的调度策略。
_mutatr_set_spin_limit	设置互斥量属性结构体的有限旋转计数。
_mutatr_set_wait_protocol	设置互斥量属性结构体的等待策略。

3.6.6.6 锁定互斥量

要访问共享资源，任务可以通过调用**_mutex_lock()**函数来锁定与资源相关的互斥量。如果互斥量未被锁定，任务将对其锁定并继续运行。如果互斥量已锁定，则根据等待策略 中描述的互斥量等待策略，可以阻塞任务直到互斥量解锁为止。

为了确定任务未被阻塞，可以通过**_mutex_trylock()**函数尝试锁定互斥量。如果互斥量未被锁定，任务将对其锁定并继续运行。如果互斥量已被锁定，则任务不会获得互斥量，但仍将继续运行。

3.6.6.7 解锁互斥量

只有当任务已锁定互斥量，才能对其解锁（**_mutex_unlock()**）。

3.6.6.8 销毁互斥量

如果不再需要某个互斥量，任务可以使用**_mutex_destroy()**销毁它。如果有任务在等待互斥量，MQX RTOS 会将它们移至其就绪队列。

3.6.6.9 示例：使用互斥量

互斥量用于互相排斥。有两种时间片任务，这两种任务都输出到同一器件。互斥量可防止输出交叉。

3.6.6.9.1 使用互斥量的代码示例

```
/* main.c */
#include <mqx.h>
#include <bsp.h>
#include <mutex.h>
```

```
/* Task IDs */
#define MAIN_TASK   5
#define PRINT_TASK  6
extern void main_task(uint32_t initial_data);
extern void print_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice */
  { MAIN_TASK,  main_task,  1000,  8,        "main", MQX_AUTO_START_TASK,0,     0 },
  { PRINT_TASK, print_task, 1000,  9,        "print",0,                  0,     3 },
  { 0 }
};
MUTEX_STRUCT   print_mutex;
/*TASK*-----------------------------------------------------
*
* Task Name : main_task
* Comments  : This task creates a mutex, and then two
*             instances of the print task.
*END*-----------------------------------------------------*/
void main_task(uint32_t initial_data)
{
    MUTEX_ATTR_STRUCT   mutexattr;
    char* string1 = "Hello from Print task 1\n";
    char* string2 = "Print task 2 is alive\n";

   /* Initialize mutex attributes: */
   if (_mutatr_init(&mutexattr) != MQX_OK) {
      printf("Initializing mutex attributes failed.\n");
      _mqx_exit(0);
   }

   /* Initialize the mutex: */
   if (_mutex_init(&print_mutex, &mutexattr) != MQX_OK) {
      printf("Initializing print mutex failed.\n");
      _mqx_exit(0);
   }
   /* Create the print tasks */
   _task_create(0, PRINT_TASK, (uint32_t)string1);
   _task_create(0, PRINT_TASK, (uint32_t)string2);
}
/*TASK*-----------------------------------------------------
*
* Task Name : print_task
* Comments  : This task prints a message. It uses a mutex to
*             ensure I/O is not interleaved.
*END*-----------------------------------------------------*/
void print_task(uint32_t initial_data)
{
   while(TRUE) {
      if (_mutex_lock(&print_mutex) != MQX_OK) {
         printf("Mutex lock failed.\n");
         _mqx_exit(0);
     }
      _io_puts((char *) initial_data);
      _mutex_unlock(&print_mutex);
   }
}
```

3.6.6.9.2 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\mutex
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。

3. 按照《*MQX™ RTOS* 版本注释》文档中的说明运行该应用程序。 请参阅以获取有关支持的工具链的更多详细信息。

3.6.7 消息

任务可以通过交换消息相互通信。任务从消息池分配消息。任务将消息发送至消息队列，并从消息队列接收消息。可以将优先级分配给消息或将消息标注为紧急。任务可以发送广播消息。

附注	为了在某些目标平台上优化代码和数据存储器要求，消息组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

表 3-31. 汇总：使用消息

消息使用某些在 *message.h* 中定义的结构体和常数。	消息使用某些在 *message.h* 中定义的结构体和常数。
_msg_alloc	从私有消息池分配消息。
_msg_alloc_system	从系统消息池分配消息。
_msg_available	获取消息池中可用消息的数量。
_msg_create_component	创建消息组件。
_msg_free	释放消息。
_msg_swap_endian_data	将消息中由应用程序定义的数据转换为其他字节存储格式。
_msg_swap_endian_header	将消息头转换为其他字节存储格式。
_msgpool_create	创建私有消息池。
_msgpool_create_system	创建系统消息池。
_msgpool_destroy	销毁占用消息池。
_msgpool_test	测试所有消息池。
_msgq_close	关闭消息队列。
_msgq_get_count	获取消息队列中的消息数量。
_msgq_get_id	将队列编号和处理器编号转换为队列 ID。
_msgq_get_notification_function	获取与消息队列相关的通知函数。
_msgq_get_owner	获取拥有消息队列的任务的任务 ID。
_msgq_open	打开私有消息队列。
_msgq_open_system	打开系统消息队列。
_msgq_peek	获取位于消息队列头部的消息指针（不会使消息出列）。
_msgq_poll	在消息队列中轮询（非阻塞）消息。
_msgq_receive	在指定的毫秒数内，从消息队列中接收消息。
_msgq_receive_for	在指定的滴答周期内，从消息队列中接收消息。
_msgq_receive_ticks	在指定的滴答数内，从消息队列中接收消息。
_msgq_receive_until	从消息队列中接收消息，并等待指定的滴答时间。

下一页继续介绍此表...

表 3-31. 汇总：使用消息 (继续)

_msgq_send	向消息队列发送消息。
_msgq_send_broadcast	向多个消息队列发送消息。
_msgq_send_priority	向消息队列发送优先级消息。
_msgq_send_queue	向消息队列直接发送消息（无需经过处理器）
_msgq_send_urgent	向消息队列发送紧急消息。
_msgq_set_notification_function	设置用于消息队列的通知函数。
_msgq_test	测试消息队列。

3.6.7.1 创建消息组件

您可以使用**_msg_create_component()**显式地创建消息组件。如果不显式地创建，MQX RTOS 会在应用程序首次创建消息池或打开消息队列时创建。

3.6.7.2 使用消息池

任务必须先创建消息池，然后从消息池分配消息。任务可创建私有消息池（**_msgpool_create()**）或系统消息池（**_msgpool_create_system()**）。

当任务创建消息池时，会指定以下信息：

- 池中的消息大小。
- 池中消息的初始数量。
- 增长因子：MQX RTOS 添加到池中的额外消息数量（如果任务已分配了所有消息）。
- 池中的最大消息数（如果增长因子为非零，如果此值为零，则表示消息池可容纳不限数量的消息）。

可多次调用**_msgpool_create_system()**函数来创建多个系统消息池，每个消息池具有不同的特征。

函数**_msgpool_create()**返回消息池 ID，任何任务可使用该 ID 来访问私有消息池。

表 3-32. 使用消息池

	系统消息池	私有消息池
创建消息池	**_msgpool_create_system()**	**_msgpool_create()**
分配消息	**_msg_alloc_system()** （MQX RTOS 搜索所有系统消息池。）	**_msg_alloc()** （MQX RTOS 仅搜索指定的私有消息池。）
释放消息（仅限消息拥有者）	**_msg_free()**	**_msg_free()**
销毁消息池	不能销毁系统消息池。	**_msgpool_destroy()**

表 3-32. 使用消息池

		（由任何任务通过消息池 ID 实现，之后池中的所有消息被释放。）

3.6.7.3 分配和释放消息

在任务发送消息前，任务会通过**_msg_alloc_system()**或**_msg_alloc()**函数从系统或私有消息池分配适当大小的消息。

系统消息池不是任何任务的资源，任何任务都可以从中分配消息。具有池 ID 的任何任务都可以从私有消息池分配消息。

当任务从任一类型的池分配消息时，消息成为该任务的资源，直到任务释放该消息(**_msg_free()**)或将其置于消息队列（**_msgq_send** 系列函数）。当任务从消息队列获取消息（**_msgq_poll()**或**_msgq_receive** 系列）时，消息成为该任务的资源。只有持有消息作为资源的任务才能释放该消息。

消息以消息头(**MESSAGE_HEADER_STRUCT**)开始，消息头定义 MQX RTOS 路由该消息所需的信息。消息头后跟随应用程序定义的数据。

```
typedef struct message_header_struct
{
  _msg_size  SIZE;
#if MQX_USE_32BIT_MESSAGE_QIDS
  uint16_t    PAD;
#endif
  _queue_id  TARGET_QID;
  _queue_id  SOURCE_QID;
  uchar      CONTROL;
#if MQX_USE_32BIT_MESSAGE_QIDS
  uchar      RESERVED[3];
#else
  uchar      RESERVED;
#endif
} MESSAGE_HEADER_STRUCT, * MESSAGE_HEADER_STRUCT_PTR;
```

关于每个字段的说明，请参见《MQX RTOS 参考手册》。

3.6.7.4 发送消息

在任务分配消息并填写消息头字段和任意数据字段后，通过**_msgq_send()**函数发送消息，将消息发送到消息头中指定的目标消息队列。发送消息不属于阻塞操作。

3.6.7.5 消息队列

任务通过消息队列来交换消息。消息队列分为私有消息队列和系统消息队列。当任务打开一个消息队列（由消息队列编号指定），MQX RTOS 返回一个应用程序唯一的队列 ID，之后任务使用这个 ID 来访问消息队列。

任务可以通过**_msgq_get_id()**函数将队列编号转换为队列 ID。

3.6.7.5.1 16 位队列 ID

16 位队列 ID 中的最高有效字节包含处理器编号，最低有效字节则包含队列编号。

表 3-33. 16 位队列 ID

位位置	15 8	7 0
队列 ID	处理器编号	队列编号

3.6.7.5.2 32 位队列 ID

32 位队列 ID 中的最高有效字节包含处理器编号，最低有效字节则包含队列编号。

表 3-34. 32 位队列 ID

位位置	31	16	15 0
队列 ID	处理器编号		队列编号

3.6.7.6 使用私有消息队列来接收消息

任务可以发送消息到任意私有消息队列，但仅有打开私有消息队列的任务才能接收来自该队列的消息。同一时刻只有一个任务可以打开该私有消息队列。

任务通过指定队列编号打开私有消息队列（**_msgq_open()**），该编号是介于 8 和由 MQX 初始化结构体指定的最大队列数之间的值。(队列编号 0-7 保留。) 如果任务用队列编号零来调用**_msgq_open()**, MQX RTOS 会打开任务中任意一个未打开的私有消息队列。

打开私有消息队列的任务可以通过**_msgq_close()**函数将该队列关闭，这会删除消息队列中的所有消息并将其释放。

任务通过**_msgq_receive** 系列函数从其私有消息队列中接收消息，这会删除在指定队列中的第一条消息，并返回指向该消息的指针。如果任务指定队列的 ID 为零，它将接收来自任意打开消息队列中的消息。从私有消息队列中接收消息是一个阻塞动作，除非任务指定超时，即任务等待消息的最长时间。

3.6.7.7 使用系统消息队列来接收消息

系统消息队列不归任务所有，任务在等待接收消息时不会阻塞。由于等待来自系统消息队列的消息不会阻塞，因此 ISR 可以使用系统队列。任务或 ISR 通过 **_msgq_open_system()**函数打开系统消息。

任务或 ISR 通过**_msgq_poll()**函数从系统消息队列中接收消息。如果系统消息队列中没有消息，则函数返回 NULL。

3.6.7.8 确定挂起的消息数

任务可以使用**_msgq_get_count()**确定系统队列或其中一个私有消息队列中有多少消息。

3.6.7.9 通知函数

对于系统和私有消息队列，任务可以指定一个通知函数在发送消息给队列时运行。对于系统消息队列，任务在其打开队列时指定通知函数。对于私有消息队列，任务在其打开队列后，通过**_msgq_set_notification_function()**函数设置通知函数。应用程序可使用通知函数将另一个同步服务（如事件或信号量）与消息队列配对。

3.6.7.10 示例：客户端/服务器模式

此客户端/服务器模式使用消息传递显示通信和任务同步。

服务器任务阻止等待来自客户端任务的请求消息。当服务器收到请求后，执行该请求并将消息返回到客户端。使用双向消息，以便在服务器运行时阻止客户端。

服务器打开将用于从客户端任务接收请求的输入消息队列，并创建消息池，它将从该池分配请求消息。然后，服务器创建一些客户端任务。在实际应用中，客户任务大多数情况下不是由服务器创建的。

当服务器打开其消息队列并创建其消息池后，会进入一个循环，从消息队列接收消息、对这些消息执行操作（在本例中为输出数据），然后将消息返回到客户端。

客户端也打开一个消息队列。它从消息池分配一条消息，填充消息字段，将该消息发送至服务器，并等待服务器的响应。

3.6.7.10.1 消息定义

```
/* server.h */
#include <mqx.h>
#include <message.h>
/* Number of clients */
#define NUM_CLIENTS 3
/* Task IDs */
#define SERVER_TASK  5
#define CLIENT_TASK  6
/* Queue IDs */
#define SERVER_QUEUE       8
#define CLIENT_QUEUE_BASE  9
/* This struct contains a data field and a message struct. */
typedef struct {
   MESSAGE_HEADER_STRUCT   HEADER;
   uchar                   DATA[5];
} SERVER_MESSAGE, * SERVER_MESSAGE_PTR;
/* Function prototypes */
extern void server_task(uint32_t initial_data);
extern void client_task(uint32_t initial_data);
extern _pool_id message_pool;
```

3.6.7.10.2 用于客户端/服务器模型的任务模板示例

```
/*  ttl.c */
#include <mqx.h>
#include <bsp.h>
#include "server.h"
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index,   Function,    Stack, Priority, Name,      Attributes,           Param, Time
Slice */
  { SERVER_TASK, server_task, 1000, 8,       "server", MQX_AUTO_START_TASK, 0,     0 },
  { CLIENT_TASK, client_task, 1000, 8,       "client", 0,                   0,     0 },
  { 0 }
};
```

3.6.7.10.3 服务器任务代码

```
/* server.c */
#include <mqx.h>
#include <bsp.h>
#include "server.h"
/* Declaration of a global message pool: */
_pool_id            message_pool;
/*TASK*-------------------------------------------------------
*
* Task Name   : server_task
* Comments  : This task creates a message queue for itself,
* allocates a message pool, creates three client tasks, and
  then waits for a message. After receiving a message, the
  task returns the message to the sender.
*END*-------------------------------------------------------*/
void server_task(uint32_t param)
{
   SERVER_MESSAGE_PTR   msg_ptr;
   uint32_t              i;
   _queue_id            server_qid;
   /* Open a message queue: */
   server_qid = _msgq_open(SERVER_QUEUE, 0);
   /* Create a message pool: */
   message_pool = _msgpool_create(sizeof(SERVER_MESSAGE),
      NUM_CLIENTS, 0, 0);
   /* Create clients: */
```

```
   for (i = 0; i < NUM_CLIENTS; i++) {
      _task_create(0, CLIENT_TASK, i);
   }
   while (TRUE) {
      msg_ptr = _msgq_receive(server_qid, 0);
      printf(" %c \n", msg_ptr->DATA[0]);
      /* Return the message: */
      msg_ptr->HEADER.TARGET_QID = msg_ptr->HEADER.SOURCE_QID;
      msg_ptr->HEADER.SOURCE_QID = server_qid;
      _msgq_send(msg_ptr);
   }
}
```

3.6.7.10.4 客户端任务代码

```
/* client.c */
#include <string.h>
#include <mqx.h>
#include <bsp.h>
#include "server.h"
/*TASK*-----------------------------------------------------------
*
* Task Name    : client_task
* Comments      This task creates a message queue and allocates
   a message in the message pool. It sends the message to the
   server_task and waits for a reply. It then frees the message.
*END*-----------------------------------------------------------*/
void client_task(uint32_t index)
{

   SERVER_MESSAGE_PTR   msg_ptr;
   _queue_id            client_qid;

   client_qid  = _msgq_open((_queue_number)(CLIENT_QUEUE_BASE +
      index), 0);

   while (TRUE) {
   /* Allocate a message: */
      msg_ptr = (SERVER_MESSAGE_PTR) _msg_alloc(message_pool);
      if(msg_ptr == NULL){
         printf("\nCould not allocate a message\n");
         _mqx_exit(0);
      }/* if */
      msg_ptr->HEADER.SOURCE_QID = client_qid;
      msg_ptr->HEADER.TARGET_QID = _msgq_get_id(0, SERVER_QUEUE);
      msg_ptr->HEADER.SIZE = sizeof(MESSAGE_HEADER_STRUCT) +
         strlen((char *)msg_ptr->DATA) + 1;
      msg_ptr->DATA[0] = ('A'+ index);

      printf("Client Task %d\n",  index);
      _msgq_send(msg_ptr);
      /* Wait for the return message: */
      msg_ptr = _msgq_receive(client_qid, 0);

      /* Free the message: */
      _msg_free(msg_ptr);
   }
}
```

3.6.7.10.5 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

```
mqx\examples\lwmsgq
```

2. 请参阅 《Freescale MQX™ RTOS for Kinetis SDK 入门》(文档 MQXKSDKGSUG) 获取有关如何构建和运行该应用程序的说明。
3. 运行应用程序。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。

3.6.8 轻量级消息队列

轻量级消息队列是标准 MQX 消息的简单化、低开销实现。任务将消息发送至轻量级消息队列，并从轻量级消息队列接收消息。消息池中的消息大小固定，是 32 位的倍数。提供阻塞读取和阻塞写入。

附注	为了在某些目标平台上优化代码和数据存储器要求，轻量级消息队列组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

表 3-35. 汇总：使用轻量级消息队列组件

轻量级消息队列组件使用某些在 *lwmsgq.h* 中定义的结构和常数。	轻量级消息队列组件使用某些在 *lwmsgq.h* 中定义的结构和常数。
_lwmsgq_init	创建轻量级消息队列。
_lwmsgq_receive	从轻量级消息队列中获取消息。
_lwmsgq_send	在轻量级消息队列中放入一条消息。

3.6.8.1 轻量级消息队列的初始化

通过调用**_lwmsgq_init()**函数来初始化轻量级消息队列。

在初始化该组件之前，必须静态分配消息池。当任务初始化轻量级消息队列时，需要指定要创建的消息数量和每条消息的大小。

3.6.8.2 发送消息

任务通过**_lwmsgq_send()**函数向轻量级消息队列发送消息。无需特殊结构的消息，然而消息大小必须与_lwmsgq_init()函数中指定的消息大小相匹配。

如果队列已满，任务会被阻塞并等待，或是返回错误代码。也可能在消息发送后阻塞任务。

3.6.8.3 接收消息

任务通过**_lwmsgq_receive()**函数从轻量级消息队列获取消息。该函数从队列中删除第一条消息，并将该消息复制到用户缓冲区。该消息变为任务的一个资源。

如果队列为空，读任务将会超时。如果轻量级消息队列为空，也可能将会阻塞读取任务。

3.6.8.4 示例：客户端/服务器模式

此示例是示例：客户端/服务器模式中介绍的客户端/服务器示例的修改版。消息组件被替换为轻量级消息队列组件。

服务器任务初化消息队列，创建 3 个客户端任务，然后等待消息。收到消息后，任务将消息返回给发送者。客户端任务向服务器任务发送一条消息，然后等待答复。

3.6.8.4.1 消息定义

```
/* server.h */
#include <mqx.h>
/* Number of clients */
#define NUM_CLIENTS 3
/* Task IDs */
#define SERVER_TASK       5
#define CLIENT_TASK       6
/* This structure contains a data field and a message header structure */
#define NUM_MESSAGES  3
#define MSG_SIZE      1
extern uint32_t server_queue[];
extern uint32_t client_queue[];
/* Function prototypes */
extern void server_task (uint32_t initial_data);
extern void client_task (uint32_t initial_data);
```

3.6.8.4.2 用于客户端/服务器模型的任务模板

```
/*  ttl.c */
#include <mqx.h>
#include <bsp.h>
#include <lwmsgq.h>
#include "server.h"
uint32_t server_queue[sizeof(LWMSGQ_STRUCT)/sizeof(uint32_t) + NUM_MESSAGES * MSG_SIZE];
uint32_t client_queue[sizeof(LWMSGQ_STRUCT)/sizeof(uint32_t) + NUM_MESSAGES * MSG_SIZE];
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index,   Function,    Stack, Priority, Name,      Attributes,          Param, Time
Slice */
  { SERVER_TASK, server_task, 2000, 8,       "server", MQX_AUTO_START_TASK, 0,     0 },
  { CLIENT_TASK, client_task, 1000, 8,       "client", 0,                   0,     0 },
```

```
  { 0 }
};
```

3.6.8.4.3 服务器任务代码

```
/* server.c */
#include <mqx.h>
#include <bsp.h>
#include <lwmsgq.h>
#include "server.h"
/*TASK*-----------------------------------------------------------
*
* Task Name : server_task
* Comments  : This task initializes the message queues,
*  creates three client tasks, and then waits for a message.
*  After recieving a message, the task returns the message to
*  the sender.
*END*-----------------------------------------------------------*/
void server_task
   (
      uint32_t param
   )
{
   _mqx_uint          msg[MSG_SIZE];
   _mqx_uint          i;
   _mqx_uint          result;
   result = _lwmsgq_init((void *)server_queue, NUM_MESSAGES, MSG_SIZE);
   if (result != MQX_OK) {
      // lwmsgq_init failed
   } /* Endif */
   result = _lwmsgq_init((void *)client_queue, NUM_MESSAGES, MSG_SIZE);
   if (result != MQX_OK) {
      // lwmsgq_init failed
   } /* Endif */

   /* create the client tasks */
   for (i = 0; i < NUM_CLIENTS; i++) {
      _task_create(0, CLIENT_TASK, (uint32_t)i);
   }

   while (TRUE) {
      _lwmsgq_receive((void *)server_queue, msg, LWMSGQ_RECEIVE_BLOCK_ON_EMPTY, 0, 0);
      printf(" %c \n", msg[0]);

      _lwmsgq_send((void *)client_queue, msg, LWMSGQ_SEND_BLOCK_ON_FULL);
   }
}
```

3.6.8.4.4 客户端任务代码

```
/* client.c */
#include <string.h>
#include <mqx.h>
#include <bsp.h>
#include <lwmsgq.h>
#include "server.h"
/*TASK*-----------------------------------------------------------
*
* Task Name : client_task
* Comments  : This task sends a message to the server_task and
*   then waits for a reply.
*END*-----------------------------------------------------------*/
void client_task
   (
      uint32_t index
   )
```

```
{
   _mqx_uint           msg[MSG_SIZE];

   while (TRUE) {
      msg[0] = ('A'+ index);

      printf("Client Task %ld\n", index);
      _lwmsgq_send((void *)server_queue, msg, LWMSGQ_SEND_BLOCK_ON_FULL);
      _time_delay_ticks(1);

      /* wait for a return message */
      _lwmsgq_receive((void *)client_queue, msg, LWMSGQ_RECEIVE_BLOCK_ON_EMPTY, 0, 0);
   }
}
```

3.6.8.4.5 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   \mqx\examples\msg
   ```

2. 请参阅《MQX RTOS 版本注释》获取有关如何构建和运行该应用程序的说明。
3. 运行应用程序。

附注	借助 Freescale MQX，CodeWarrior Development Studio 成为了 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。 feature="ksdk_integrated">请参阅《Freescale MQX™ RTOS for Kinetis SDK 入门》。

3.6.9 任务队列

您可使用任务队列来:

- 从 ISR 调度任务。
- 执行显式任务调度。
- 实现自定义同步机制。

表 3-36. 汇总：使用任务队列

_taskq_create	通过指定队列规则（FIFO 或优先级）来创建任务队列。
_taskq_destroy	销毁任务队列（并将任何等待任务放入合适的就绪队列）。
_taskq_get_value	获取任务队列大小。
_taskq_resume	重启在任务队列中被挂起的任务，或重启任务队列中的所有任务（并将其放入就绪队列）。
_taskq_suspend	挂起任务并将其放入指定任务队列（或从任务就绪队列中删除）。
_taskq_suspend_task	挂起非阻塞任务并将其放入指定任务队列（或从任务就绪队列中删除）。
_taskq_test	测试所有任务序列。

3.6.9.1 创建和销毁任务队列

应用程序要能够执行显式任务调度，必须先通过使用任务队列的排队策略调用**_taskq_create()**来初始化该任务。MQX RTOS 创建任务队列并返回队列 ID，任务随后使用该队列 ID 来访问任务队列。

任务队列不是创建任务队列的任务的资源。它是系统资源，并不会在创建它的任务终止时销毁。

任务可以使用**_taskq_destroy()**显式地销毁任务队列。如果任务队列中存在任务，MQX RTOS 会将它们移至其就绪队列。

3.6.9.2 挂起任务

任务可通过**_taskq_suspend()**函数在指定任务队列中将自己挂起。MQX RTOS 将根据任务队列的排队规则将该任务放入队列（阻塞该任务）。

3.6.9.3 恢复任务

任务调用**_taskq_resume()**函数从指定任务队列中删除一个或所有任务。MQX RTOS 将其放入就绪队列。

3.6.9.4 示例：同步任务

任务使用 ISR 同步。另一个任务模拟该中断。

service_task 任务等待周期性中断，并在每次发生中断时输出一条消息。任务首先创建任务队列，然后将自身暂挂在队列中。simulated_ISR_task 任务使用**_time_delay()**模拟周期性中断，当超时到期时调度 service_task。

3.6.9.4.1 代码示例

```
/* taskq.c */
#include <mqx.h>
#include <fio.h>
/* Task IDs */
#define SERVICE_TASK 5
#define ISR_TASK     6
extern void simulated_ISR_task(uint32_t);
extern void service_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index,  Function,         Stack,Prio,Name,           Attributes,         Param, TS
*/
 { SERVICE_TASK,service_task,      2000, 8,   "service",      MQX_AUTO_START_TASK,0,     0},
 { ISR_TASK,    simulated_ISR_task,2000, 8,   "simulated_ISR",0,                  0,     0},
 { 0 }
```

```
};
void *   my_task_queue;
/*TASK*-----------------------------------------------------
*
* Task Name    : simulated_ISR_task
* Comments     :
*    This task pauses and then resumes the task queue.
*END*-----------------------------------------------------*/
void simulated_ISR_task(uint32_t initial_data)
{
   while (TRUE) {
      _time_delay(200);
      _taskq_resume(my_task_queue, FALSE);
   }
}
/*TASK*-----------------------------------------------------
*
* Task Name    : service_task
* Comments     :
*    This task creates a task queue and the simulated_ISR_task
*    task. Then it enters an infinite loop, printing "Tick" and
*    suspending the task queue.
*END*-----------------------------------------------------*/
void service_task(uint32_t initial_data)
{
   _task_id second_task_id;
   /* Create a task queue: */
   my_task_queue = _taskq_create(MQX_TASK_QUEUE_FIFO);
   if (my_task_queue == NULL) {
      _mqx_exit(0);
   }
   /* Create the task: */
   second_task_id = _task_create(0, ISR_TASK, 0);
      if (second_task_id == MQX_NULL_TASK_ID) {
         printf("\n Could not create simulated_ISR_task\n");
         _mqx_exit(0);
      }

   while (TRUE) {
         printf(" Tick \n");
        _taskq_suspend(my_task_queue);
   }
}
```

3.6.9.4.2 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\lwmsgq
   ```

2. 请参阅 《Freescale MQX™ RTOS for Kinetis SDK 入门》(文档 MQXKSDKGSUG) 获取有关如何构建和运行该应用程序的说明。
3. 运行应用程序。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。

3.7 处理器之间的通信

通过处理器间通信（IPC）组件，任务可以在远程处理器上执行以下操作:

- 交换消息
- 创建任务（阻塞或未阻塞）
- 销毁任务
- 打开和关闭指定的事件组
- 在指定的事件组中置位事件位

所有处理器不必直接连接或属于同一类型。IPC 组件通过中间处理器路由消息，并将它们转换为适当的端格式。IPC 组件通过数据包控制块（PCB）设备驱动器通信。

当具有 IPC 组件的 MQX RTOS 初始化时，它会初始化 IPC 消息驱动器，并构建消息路由表，该表定义了消息在处理器间的传输路径。具体硬件相关的信息，请参阅您的 MQX RTOS 版本附带的版本说明。

表 3-37. 汇总：设置处理器间通信

_ipc_add_ipc_handler	为 MQX 组件添加 IPC 处理程序。
_ipc_add_io_ipc_handler	为 I/O 组件添加 IPC 处理程序。
_ipc_msg_route_add	向消息路由表添加消息。
_ipc_msg_route_remove	从消息路由表删除消息。
_ipc_pcb_init	初始化 PCB 驱动器的 IPC。
_ipc_task	初始化 IPC 和处理远程服务请求的任务。

3.7.1 向远程处理器发送消息

除了具有消息路由表, 每个处理器具有一个或多个 IPC, 每个 IPC 由以下内容组成:

- 输入函数
- 输出函数
- 输出队列

当任务向消息队列发送消息时, MQX RTOS 会检验目标处理器编号，该编号为队列 ID 的一部分。如果目标处理器不是本地的，MQX RTOS 会检查路由表。

如果有路由，路由表会显示 IPC 队列到达目标处理器使用的输出队列。然后 MQX 引导消息到达输出队列。输出函数开始运行，并传输 IPC 上的消息。

当 IPC 接收消息时，输入函数开始运行。输入函数会对消息进行编译，并调用 **_msgq_send()**函数。输入函数无需确定该输入消息是否用于本地处理器。如果消息不是用于本地处理器，MQX RTOS 会将该消息路由到目标处理器。

3.7.1.1 示例：四核处理器应用程序

此图显示一个简单的四核处理器应用程序。表中的编号是随机（但对处理器而言是唯一的）输出队列编号。

每个处理器有两个 IPC。每个处理器都有两条可能的路由；例如，处理器 1 有一个 IPC 到处理器 2，一个到处理器 4。路由表支持一条路由，因此应选择最佳路由。该表显示每个处理器的路由表的一种可能性。

3.7.1.1.1 处理器 1 的路由表

表 3-38. 路由表

源处理器	目标处理器 1	目标处理器 2	目标处理器 3	目标处理器 4
1	-	10	10	11
2	21	-	20	20
3	31	31	-	30
4	40	41	41	-

如表所示，当处理器 1 上的任务要向处理器 3 上的消息队列发送消息时，MQX RTOS 通过队列 10 将处理器 1 上的消息发送到处理器 2，然后再通过队列 20 将消息从处理器 2 发送到处理器 3。当处理器 3 上的 IPC 收到消息时，MQX RTOS 将该消息发送到目标消息队列。

3.7.2 在远程处理器上创建和销毁任务

通过 IPC 组件，任务可以向远程处理器上的 IPC 任务发送服务请求，从而在该处理器上创建和销毁任务。IPC 任务处理该请求并响应发出请求的处理器。

3.7.3 访问远程处理器上的事件组

采用 IPC 组件后，任务可以打开并关闭远程处理器上的指定名称的事件组，并在该事件组中设置事件位。但是，任务不能等待远程处理器上的事件位。

通过在事件名称中指定处理器编号（后接冒号）即可在远程处理器上打开事件组。以下示例可以打开处理器编号 4 上的事件 Fred:

```
_event_open("4:fred", &handle);
```

3.7.4 创建和初始化 IPC

对于需要在处理器间通信的任务，应用程序必须在每个处理器上创建并初始化 IPC 组件，以下步骤对此进行了总结。每个步骤将会在后续章节中详细说明，其中路由表参照前面的例子路由。

1. 构建 IPC 路由表。
2. 构建 IPC 协议初始化表。
3. 提供 IPC 协议初始化函数和数据。
4. 创建 IPC 任务（**_ipc_task()**）。

3.7.4.1 构建 IPC 路由表

IPC 路由表定义处理器间消息的路由。每个处理器一个路由表，被称为 _ipc_routing_table。在上一示例中的处理器 2 上，发送给处理器 1 的消息路由到队列编号 20；发送给处理器 3 和 4 的消息路由到队列编号 21。

路由表是一个路由结构数组，以填充了 0 的条目结束。

```
typedef struct ipc_routing_struct
{
  _processor_number  MIN_PROC_NUMBER;
  _processor_number  MAX_PROC_NUMBER;
  _queue_number      QUEUE;
} IPC_ROUTING_STRUCT, * IPC_ROUTING_STRUCT_PTR;
```

《MQX 参考手册》中介绍了这些字段的详细意义。

3.7.4.1.1 处理器 1 的路由表

```
IPC_ROUTING_STRUCT _ipc_routing_table[] =
   { {2, 3, 10},
     {4, 4, 11},
     {0, 0, 0}};
```

3.7.4.1.2 处理器 2 的路由表

```
IPC_ROUTING_STRUCT _ipc_routing_table[] =
   { {1, 1, 21},
     {3, 4, 20},
     {0, 0, 0}};
```

3.7.4.1.3 处理器 3 的路由表

```
IPC_ROUTING_STRUCT _ipc_routing_table[] =
   { {1, 2, 31},
```

```
    {4, 4, 30},
    {0, 0, 0}};
```

3.7.4.1.4 处理器 4 的路由表

```
IPC_ROUTING_STRUCT _ipc_routing_table[] =
   { {1, 1, 40},
     {2, 3, 41},
     {0, 0, 0}};
```

3.7.4.2 构建 IPC 协议初始化表

IPC 协议初始化表定义并初始化设置 IPC 的协议。路由表中的每个 IPC 输出队列引用一个 IPC，该 IPC 必须在协议初始化表中有对应的条目，用来定义配置该 IPC 的协议和通信路径。

附注	IPC_PROTOCOL_INIT_STRUCT 中的 IPC_OUT_QUEUE 字段必须与 IPC_ROUTING_STRUCT 中的 QUEUE 字段匹配。

协议初始化表是一个协议初始化结构的数组，以填充了零的条目结束。

```
typedef struct ipc_protocol_init_struct
{
  IPC_INIT_FPTR  IPC_PROTOCOL_INIT
  void *         IPC_PROTOCOL_INIT_DATA;
  char *     IPC_NAME;
  _queue_number  IPC_OUT_QUEUE;
} IPC_PROTOCOL_INIT_STRUCT, * IPC_PROTOCOL_INIT_STRUCT_PTR;
```

《MQX 参考手册》中对这些字段进行了详细的描述。

当具有 IPC 组件的 MQX RTOS 初始化时，它会为表中的每个 IPC 调用 **IPC_PROTOCOL_INIT** 函数。它将包含了指定 IPC 初始化信息的 **IPC_PROTOCOL_INIT_DATA** 传给 IPC。

3.7.4.3 IPC 使用 I/O PCB 器件驱动程序

当开发特殊功能的 IPC 时，MQX RTOS 提供在 I/O 数据包控制块（PCB）器件驱动程序上编译的标准 IPC。

通过该 IPC，应用程序可以使用任何 I/O PCB 器件驱动程序来接收和发送消息（见 IPC 初始化信息 ）。

设置 **IPC_PROTOCL_INIT_STRUCT** 以在 PCB 器件驱动程序上使用标准 MQX RTOS IPC:

```
{ _ipc_pcb_init, &pcb_init, "Pcb_to_test2", QUEUE_TO_TEST2 },
   { NULL, NULL, NULL, 0}
```

3.7.4.4 启动 IPC 任务

IPC 任务将检查 IPC 协议初始化表，并启动 IPC 服务器来初始化各 IPC 驱动程序。IPC 服务器从其他处理器接收消息，以执行远程程序调用。

在各处理器的 MQX 初始化结构中，应用程序必须将 IPC 任务定义为自启动任务。必须将指向 IPC_INIT_STRUCT 类型的 IPC 初始化结构的指针作为创建参数传递给 IPC 任务。该结构包含 IPC 路由表和 IPC 初始化表指针。如果未提供，将使用默认的 IPC_INIT_STRUCT。IPC 任务的任务模板为:

```
{ IPC_TTN, _ipc_task, IPC_DEFAULT_STACK_SIZE, 6,
  "_ipc_task", MQX_AUTO_START_TASK, (uint32_t)&ipc_init, 0}
```

3.7.4.5 示例: IPC 初始化信息

在本示例中，两个处理器使用 MQX RTOS 附带的 PCB 器件驱动器，通过异步串行端口创建 IPC 通信。每个处理器都由中断驱动的异步字符器件驱动器"ittyb:"连接。IPC 使用 PCB_MQXA 驱动器发送和接收 MQX RTOS 定义的格式的数据包。

ipc_init_table 通过 PCB I/O 驱动器初始化函数**_ipc_pcb_init()**使用 MQX RTOS IPC 及其初始化 pcb_init 所需的数据结构，该初始化定义了以下几个部分:

- 打开驱动器时要使用的 PCB I/O 驱动器名称。
- 要调用的安装函数，在本例中为**_io_pcb_mqxa_install()**（如果先前已安装 PCB I/O 驱动器，则无需指定）。
- PCB I/O 驱动器特定的初始化 pcb_mqxa_init。

3.7.4.5.1 IPC 初始化信息

```
/* ipc_ex.h */
#define TEST_ID            1
#define IPC_TTN            9
#define MAIN_TTN          10
#define QUEUE_TO_TEST2    63
#define MAIN_QUEUE        64
#define TEST2_ID           2
#define RESPONDER_TTN     11
#define QUEUE_TO_TEST     67
#define RESPONDER_QUEUE   65
typedef struct the_message
{
   MESSAGE_HEADER_STRUCT  HEADER;
   uint32_t               DATA;
} THE_MESSAGE, * THE_MESSAGE_PTR;
```

3.7.4.5.2 处理器 1 的代码

```
/* ipc1.c */
#include <mqx.h>
```

```
#include <bsp.h>
#include <message.h>
#include <ipc.h>
#include <ipc_pcb.h>
#include <io_pcb.h>
#include <pcb_mqxa.h>
#include "..\ipc_ex.h"
extern void main_task(uint32_t);
IO_PCB_MQXA_INIT_STRUCT pcb_mqxa_init =
{
   /* IO_PORT_NAME */          "ittyb:", /* must be set by the user */
   /* BAUD_RATE    */          19200,
   /* IS POLLED */             FALSE,
   /* INPUT MAX LENGTH */      sizeof(THE_MESSAGE),
   /* INPUT TASK PRIORITY */   7,
   /* OUPUT TASK PRIORITY */   7
};
IPC_PCB_INIT_STRUCT pcb_init =
{
   /* IO_PORT_NAME */             "pcb_mqxa_ittyx:",
   /* DEVICE_INSTALL? */          _io_pcb_mqxa_install,
   /* DEVICE_INSTALL_PARAMETER*/  (void *)&pcb_mqxa_init,
   /* IN_MESSAGES_MAX_SIZE */     sizeof(THE_MESSAGE),
   /* IN MESSAGES_TO_ALLOCATE */  8,
   /* IN MESSAGES_TO_GROW */      8,
   /* IN_MESSAGES_MAX_ALLOCATE */ 16,
   /* OUT_PCBS_INITIAL */         8,
   /* OUT_PCBS_TO_GROW */         8,
   /* OUT_PCBS_MAX */             16
};
const IPC_ROUTING_STRUCT ipc_routing_table[] =
{
   { TEST2_ID, TEST2_ID, QUEUE_TO_TEST2 },
   { 0, 0, 0 }
};
const IPC_PROTOCOL_INIT_STRUCT ipc_init_table[] =
{
   { _ipc_pcb_init, &pcb_init, "Pcb_to_test2", QUEUE_TO_TEST2 },
   { NULL, NULL, NULL, 0}
};
static const IPC_INIT_STRUCT ipc_init = {
    ipc_routing_table,
    ipc_init_table
};
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
   /* Task Index, Function,  Stack,                  Priority, Name,
Attributes,
Param, Time Slice */
    { IPC_TTN,    _ipc_task, IPC_DEFAULT_STACK_SIZE, 8,        "_ipc_task",
MQX_AUTO_START_TASK,
(uint32_t)&ipc_init,     0 },
    { MAIN_TTN,   main_task, 2000,                   9,        "Main",
MQX_AUTO_START_TASK,
0,                       0 },
    { 0 }
};
MQX_INITIALIZATION_STRUCT  MQX_init_struct =
{
   /* PROCESSOR_NUMBER */                TEST_ID,
   /* START_OF_KERNEL_MEMORY */          BSP_DEFAULT_START_OF_KERNEL_MEMORY,
   /* END_OF_KERNEL_MEMORY */            BSP_DEFAULT_END_OF_KERNEL_MEMORY,
   /* INTERRUPT_STACK_SIZE */            BSP_DEFAULT_INTERRUPT_STACK_SIZE,
   /* TASK_TEMPLATE_LIST */              (void *)MQX_template_list,
   /* MQX_HARDWARE_INTERRUPT_LEVEL_MAX */ BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
   /* MAX_MSGPOOLS */                    8,
   /* MAX_MSGQS */                       16,
   /* IO_CHANNEL */                      BSP_DEFAULT_IO_CHANNEL,
   /* IO_OPEN_MODE */                    BSP_DEFAULT_IO_OPEN_MODE
};
```

```
/*TASK*-------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*     This task creates a message pool and a message queue then
*     sends a message to a queue on the second CPU.
*     It waits for a return message, validating the message before
*     sending a new message.
*END*-------------------------------------------------------*/
void main_task
   (
      uint32_t dummy
   )
{
   _pool_id        msgpool;
   THE_MESSAGE_PTR msg_ptr;
   _queue_id       qid;
   _queue_id       my_qid;
   uint32_t         test_number = 0;
   my_qid  = _msgq_open(MAIN_QUEUE,0);
   qid     = _msgq_get_id(TEST2_ID,RESPONDER_QUEUE);
   msgpool = _msgpool_create(sizeof(THE_MESSAGE), 8, 8, 16);
   while (test_number < 64) {
      msg_ptr = (THE_MESSAGE_PTR)_msg_alloc(msgpool);
      msg_ptr->HEADER.TARGET_QID = qid;
      msg_ptr->HEADER.SOURCE_QID = my_qid;
      msg_ptr->DATA = test_number++;
      putchar('-');
      _msgq_send(msg_ptr);
      msg_ptr = _msgq_receive(MSGQ_ANY_QUEUE, 10000);
      if (msg_ptr == NULL) {
         puts("Receive failed\n");
         _mqx_exit(1);
      } else if (msg_ptr->HEADER.SIZE != sizeof(THE_MESSAGE)) {
         puts("Message wrong size\n");
         _mqx_exit(1);
      } else if (msg_ptr->DATA != test_number) {
         puts("Message data incorrect\n");
         _mqx_exit(1);
      }
      _msg_free(msg_ptr);
   }
   puts("All complete\n");
   _mqx_exit(0);
}
```

3.7.4.5.3 处理器 2 的代码

```
/* ipc2.c */
#include <mqx.h>
#include <bsp.h>
#include <message.h>
#include <ipc.h>
#include <ipc_pcb.h>
#include <io_pcb.h>
#include <pcb_mqxa.h>
#include "ipc_ex.h"
extern void responder_task(uint32_t);
IO_PCB_MQXA_INIT_STRUCT pcb_mqxa_init =
{
   /* IO_PORT_NAME */         "ittyb:", /* must be set by the user */
   /* BAUD_RATE    */         19200,
   /* IS POLLED */            FALSE,
   /* INPUT MAX LENGTH */     sizeof(THE_MESSAGE),
   /* INPUT TASK PRIORITY */  7,
   /* OUPUT TASK PRIORITY */  7
};
IPC_PCB_INIT_STRUCT pcb_init =
```

```
{
   /* IO_PORT_NAME */             "pcb_mqxa_ittyx:",
   /* DEVICE_INSTALL? */          _io_pcb_mqxa_install,
   /* DEVICE_INSTALL_PARAMETER*/  (void *)&pcb_mqxa_init,
   /* IN_MESSAGES_MAX_SIZE */     sizeof(THE_MESSAGE),
   /* IN MESSAGES_TO_ALLOCATE */  8,
   /* IN MESSAGES_TO_GROW */      8,
   /* IN_MESSAGES_MAX_ALLOCATE */ 16,
   /* OUT_PCBS_INITIAL */         8,
   /* OUT_PCBS_TO_GROW */         8,
   /* OUT_PCBS_MAX */             16
};
const IPC_ROUTING_STRUCT ipc_routing_table[] =
{
   { TEST_ID, TEST_ID, QUEUE_TO_TEST },
   { 0, 0, 0 }
};
const IPC_PROTOCOL_INIT_STRUCT ipc_init_table[] =
{
   { _ipc_pcb_init, &pcb_init, "Pcb_to_test", QUEUE_TO_TEST },
   { NULL, NULL, NULL, 0}
};
static const IPC_INIT_STRUCT ipc_init = {
    ipc_routing_table,
    ipc_init_table
};
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
   /* Task Index,    Function,       Stack,                  Priority,  Name,
Attributes,
Param, Time Slice */
    { IPC_TTN,       _ipc_task,      IPC_DEFAULT_STACK_SIZE, 8,         "_ipc_task",
MQX_AUTO_START_TASK, (uint32_t)&ipc_init,     0 },
    { RESPONDER_TTN, responder_task, 2000,                   9,         "Responder",
MQX_AUTO_START_TASK, 0,                       0 },
    { 0 }
};
MQX_INITIALIZATION_STRUCT  MQX_init_struct =
{
   /* PROCESSOR_NUMBER */                 TEST2_ID,
   /* START_OF_KERNEL_MEMORY */           BSP_DEFAULT_START_OF_KERNEL_MEMORY,
   /* END_OF_KERNEL_MEMORY */             BSP_DEFAULT_END_OF_KERNEL_MEMORY,
   /* INTERRUPT_STACK_SIZE */             BSP_DEFAULT_INTERRUPT_STACK_SIZE,
   /* TASK_TEMPLATE_LIST */               (void *)MQX_template_list,
   /* MQX_HARDWARE_INTERRUPT_LEVEL_MAX */ BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
   /* MAX_MSGPOOLS */                     8,
   /* MAX_MSGQS */                        16,
   /* IO_CHANNEL */                       BSP_DEFAULT_IO_CHANNEL,
   /* IO_OPEN_MODE */                     BSP_DEFAULT_IO_OPEN_MODE
};
/*TASK*-------------------------------------------------------------
*
* Task Name : responder_task
* Comments  :
*     This task creates a message queue then waits for a message.
*     Upon receiving the message the data is incremented before
*     the message is returned to the sender.
*END*-------------------------------------------------------------*/
void responder_task(uint32_t dummy) {
   THE_MESSAGE_PTR    msg_ptr;
   _queue_id          qid;
   _queue_id          my_qid;
   puts("Receiver running...\n");
   my_qid = _msgq_open(RESPONDER_QUEUE, 0);
   while (TRUE) {
      msg_ptr = _msgq_receive(MSGQ_ANY_QUEUE, 0);
      if (msg_ptr != NULL) {
         qid      = msg_ptr->HEADER.SOURCE_QID;
         msg_ptr->HEADER.SOURCE_QID = my_qid;
         msg_ptr->HEADER.TARGET_QID = qid;
```

```
            msg_ptr->DATA++;
            putchar('+');
            _msgq_send(msg_ptr);
         } else {
            puts("RESPONDER RECEIVE ERROR\n");
            _mqx_exit(1);
         }
      }
}
```

3.7.4.5.4 编译应用程序并链接到 MQX RTOS

1. 有关如何构建和运行应用程序的说明，请参阅您的《MQX 版本注释》文档。
2. 转至此目录以编译处理器 1:

   ```
   mqx\examples\taskq
   ```
3. 构建项目。
4. 转至此目录以编译处理器 2:

   ```
   mqx\examples\ipc\cpu2\
   ```
5. 构建项目。
6. 将处理器 1 的 ttyb:连接至处理器 2 的 ttyb:。
7. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序。在处理器 1 之前启动处理器 2。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 成为了 MQX RTOS 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》以获取有关支持的工具链的更多详细信息。

3.7.5 消息头的端模式转换

当处理器从远程处理器收到消息时，IPC 输入函数检查消息头中的 **CONTROL** 字段，以确定消息是否来自使用其它端模式的处理器。如果是，输入函数将消息头转换为本地处理器自己的端模式，并设置 **CONTROL** 字段，以指定其端模式。

```
MESSAGE_HEADER_STRUCT  msg_ptr;
...
if (MSG_MUST_CONVERT_HDR_ENDIAN(msg_ptr->CONTROL)){
   _msg_swap_endian_header(msg_ptr);}
```

附注	IPC 不能将消息的数据部分转换为其他端模式，因为它不知道该数据的格式。 应用程序需要将接收到的消息的数据部分转换为其他端模式。要检查是否需要转换，请使用宏 **MSG_MUST_CONVERT_DATA_ENDIAN**。要转换消息数据，请使用 **_msg_swap_endian_data()**。两个函数都在 *message.h* 中定义。有关详细信息，请参阅《MQX 参考手册》。

3.8 定时

MQX RTOS 提供了内核时间组件，可用于扩展可选定时器和看门狗组件。

3.8.1 MQX RTOS 定时翻转

当应用程序开始运行后，MQX RTOS 在内部保持一个可作为滴答中断的 64 位计数器。这为 MQX RTOS 定时翻转提供了一段非常长的时间。例如，如果滴答速率为每纳秒一次，MQX RTOS 经将过 584 年才会翻转一次。

3.8.2 MQX RTOS 时间精度

MQX RTOS 在内部将时间保持为 64 位滴答中断数计数，但当应用程序请求滴答中断时，该时间还包括一个 32 位的数，表示自上次滴答中断以来发生的硬件“滴答”数。通常，MQX RTOS 从硬件计数器中获取时间。因此，应用程序会接收到所能确定的最精确时间。

3.8.3 时间组件

时间组件是一个内核组件，提供消逝时间和绝对时间，以秒和毫秒时间戳（秒/毫秒时间）、滴答（滴答时间）或日期（日期时间和扩展的日期时间）形式表示。

表 3-39. 汇总：使用时间组件

_ticks_to_time	将滴答时间转换为秒/毫秒时间。
_time_add_day_to_ticks	将天数添加到滴答时间。
_time_add_hour_to_ticks	将小时添加到滴答时间。
_time_add_min_to_ticks	将分钟添加到滴答时间。
_time_add_msec_to_ticks	将毫秒添加到滴答时间。
_time_add_nsec_to_ticks	将纳秒添加到滴答时间。
_time_add_psec_to_ticks	将皮秒添加到滴答时间。
_time_add_sec_to_ticks	将秒添加到滴答时间。
_time_add_usec_to_ticks	将微秒添加到滴答时间。
_time_delay	将活动任务挂起指定的毫秒数。
_time_delay_for	将活动任务挂起指定滴答时间周期（包括硬件滴答）。
_time_delay_ticks	将活动任务挂起指定的滴答数。
_time_delay_until	将活动任务挂起，直到指定滴答时间为止。
_time_dequeue	从超时队列中删除任务（由任务 ID 指定）。

下一页继续介绍此表...

表 3-39. 汇总：使用时间组件 (继续)

_time_dequeue_td	从超时队列中删除任务（由任务描述符指定）。
_time_diff	获取两个秒/毫秒时间结构之间的秒/毫秒时间差。
_time_diff_days	获取两个滴答时间之间的天数时间差。
_time_diff_hours	获取两个滴答时间之间的小时时间差。
_time_diff_microseconds	获取两个滴答时间之间的微秒时间差。
_time_diff_milliseconds	获取两个滴答时间之间的毫秒时间差。
_time_diff_minutes	获取两个滴答时间之间的分钟时间差。
_time_diff_nanoseconds	获取两个滴答时间之间的纳秒时间差。
_time_diff_picoseconds	获取两个滴答时间之间的皮秒时间差。
_time_diff_seconds	获取两个滴答时间之间的秒时间差。
_time_diff_ticks	获取两个滴答时间之间的滴答时间差。
_time_from_date	通过日期时间获取秒/毫秒时间。
_time_get	通过秒/毫秒时间获取绝对时间。
_time_get_ticks	通过滴答时间获取绝对时间（包括滴答和硬件滴答）。
_time_get_elapsed	获取从应用程序在该处理器上开始运行到现在的已消逝的秒/毫秒时间。
_time_get_elapsed_ticks	获取从应用程序在该处理器上开始运行到现在的已消逝的滴答时间。
_time_get_hwticks	获取从上一次滴答到现在的硬件滴答数。
_time_get_hwticks_ per_tick	获取每次滴答的硬件滴答数。
_time_get_microseconds	获取从上一周期性定时器中断到现在的计算得到的微秒数。
_time_get_nanoseconds	获取从上一周期性定时器中断到现在的计算得到的纳秒数。
_time_get_resolution	获取周期性定时器中断的分辨率。
_time_get_ticks_per_sec	获取时钟中断的频率（以滴答/秒为单位）。
_time_init_ticks	使用滴答数初始化滴答时间结构。
_time_notify_kernel	当发生周期性定时器中断时由 BSP 调用。
_time_set	设置秒/毫秒时间格式的绝对时间。
_time_set_hwticks_per_tick	设置每次滴答的硬件滴答数。
_time_set_ticks	设置滴答时间格式的绝对时间。
_time_set_resolution	设置周期性定时器中断的分辨率。
_time_set_timer_vector	设置 MQX RTOS 使用的周期性定时器中断向量。
_time_set_ticks_per_sec	设置时钟中断的频率（以滴答/秒为单位）。
_time_to_date	将秒/毫秒时间转换为日期时间。
_time_to_ticks	将秒/毫秒时间转换为滴答时间。
mktime	将分解时间值（本地时间形式）转换为日历时间表示。
gmtime_r	将日历时间转换为分解时间表示（国际协调时间（UTC）形式）。
timegm	将分解时间结构（UTC 时间形式）转换为日历时间表示。
localtime_r	将日历时间转换为分解时间表示（本地时间形式）。

3.8.3.1 秒/毫秒时间

时间可以表示为秒和毫秒形式。相对于处理滴答时间，处理秒/毫秒时间会更复杂，需要更多的 CPU 资源。

```
typedef struct time_struct
{
  uint32_t  SECONDS;
  uint32_t  MILLISECONDS;
} TIME_STRUCT, * TIME_STRUCT_PTR;
```

《MQX 参考手册》中介绍了这些字段。

3.8.3.2 时间戳

时间戳是描述时间瞬间的系统，定义为自新纪元 1970 年 1 月 1 日 UTC 00:00:00 以来经过的秒数。

```
typedef uint32_t time_t
```

3.8.3.3 滴答时间

时间可以表示为滴答时间形式。相对于处理秒/毫秒时间，处理滴答时间更简单，需要更少的 CPU 资源。

```
typedef struct mqx_tick_struct
{
  _mqx_uint  TICKS[MQX_NUM_TICK_FIELDS];
  uint32_t    HW_TICKS;
} MQX_TICK_STRUCT, * MQX_TICK_STRUCT_PTR;
```

《MQX 参考手册》中介绍了这些字段。

3.8.3.4 消逝时间

消逝时间是自 MQX RTOS 在处理器上启动后经过的时间量。任务使用 **_time_get_elapsed()**获取以秒/毫秒为单位的消逝时间，或使用 **_time_get_elapsed_ticks()**获取以滴答时间为单位的消逝时间。

3.8.3.5 时间分辨率

MQX RTOS 启动时会装载周期性定时器 ISR，可用于设置硬件的时间分辨率。分辨率定义了 MQX RTOS 更新时间的频率以及滴答发生的频率。分辨率通常为每秒 200 次滴答或 5 毫秒。任务通过**_time_get_resolution()**函数获取毫秒格式的分辨率，通过**_time_get_resolution_ticks()**获取每秒滴答数格式的分辨率。

任务可以通过调用**_time_get_elapsed()**函数获取以微秒分辨率格式的消逝时间，随后通过**_time_get_microseconds()**函数获取从上一次周期性定时器中断到现在的微秒数。

任务可以通过调用**_time_get_elapsed()**函数获取以纳秒分辨率格式的消逝时间，随后通过**_time_get_nanoseconds()**函数获取从上一次周期性定时器中断到现在的纳秒数。

任务还可以通过调用**_time_get_hwticks()**函数获取从上一次中断到现在的硬件滴答次数。任务可以通过调用**_time_get_hwticks_per_tick()**函数获取硬件滴答的分辨率。

3.8.3.6 绝对时间

为使不同处理器上的任务可以交换根据共同时间参考依据标记了时间戳的信息，时间组件提供了绝对时间。

作为初始值，绝对时间是自参考日期 1970 年 1 月 1 日 0:00:00.000 以来的时间。应用程序可以使用**_time_set()**更改以秒/毫秒为单位的参考时间，或使用**_time_set_ticks()**更改以滴答数为单位的参考日期，从而更改绝对时间。

任务使用**_time_get()**获取以秒/毫秒为单位的绝对时间，或使用**_time_get_ticks()**获取以滴答数为单位的绝对时间。

除非应用程序更改绝对时间，否则以下函数对返回相同的值:

- **_time_get()**与**_time_get_elapsed()**
- **_time_get_ticks()**与**_time_get_elapsed_ticks()**

附注	任务应使用消逝时间测量间隔或设置定时器。这样可防止测量结果受其他任务影响，因为其他任务可能调用**_time_set()**或**_time_set_ticks()**，从而更改绝对时间。

3.8.3.7 日期中的时间格式

为帮助设置或中断以秒/毫秒或滴答时间表示的绝对时间，时间组件提供日期格式或分解时间结构（tm 结构）的时间表示。

3.8.3.7.1 DATE_STRUCT

```
typedef struct date_struct
{
   int16_t       YEAR;
   int16_t       MONTH;
   int16_t       DAY;
   int16_t       HOUR;
   int16_t       MINUTE;
   int16_t       SECOND;
   int16_t       MILLISEC;
   int16_t       WDAY;
   int16_t       YDAY;
} DATE_STRUCT, * DATE_STRUCT_PTR;
```

《MQX 参考手册》中介绍了这些字段。

3.8.3.7.2 TM STRUCT

```
struct tm {
    int32_t         tm_sec;
    int32_t         tm_min;
    int32_t         tm_hour;
    int32_t         tm_mday;
    int32_t         tm_mon;
    int32_t         tm_year;
    int32_t         tm_wday;
    int32_t         tm_yday;
    int32_t         tm_isdst;
};
```

《MQX 参考手册》中介绍了该字段（第 8 页）。

3.8.3.8 超时

任务可将时间作为超时参数提供给多个 MQX 组件，例如在**_msgq_receive**、**_lwmsgq_receive**、**_sem_wait**、**_lwsem_wait**、**_event_wait** 和**_lwevent_wait** 系列中的函数。请注意，所有时间函数的分辨率始终为一次滴答。

_time_delay()、**_event_wait_all()**、**_event_wait_any()**、**_sem_wait()**、**msgq_receive()** 和**_sched_set_rr_interval()**函数至少等待指定时间（以毫秒为单位）。该时间通常大于所需时间，取决于滴答长度、调度事件和优先级。

_time_delay_ticks()函数至少等待所需的滴答中断数。

_time_delay_ticks(1)函数至少等待第一个滴答中断。

_time_delay(0)和**_time_delay_tick(0)**导致调用 **shed_yield()**函数。对于大于零的滴答次数，实际等待时间通常小于滴答次数与滴答时间（以毫秒为单位）的乘积。

任务还可以通过调用**_time_delay** 系列中的函数将其自身显性挂起。当时间超时后，MQX RTOS 将任务放入任务就绪队列。

3.8.4 定时器

定时器是可选组件，用于扩展内核时间组件。应用程序可以使用定时器进行如下操作:

- 使通知函数在特定时间运行——当 MQX RTOS 创建定时器组件时，它将启动定时器任务，用来维护定时器及应用程序指定的通知函数。当定时器超时后，定时器任务会调用合适的通知函数。
- 在时间周期超时后通信。

附注	为了在某些目标平台上优化代码和数据存储器要求，定时器组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

任务可以在任何指定时间或在当前时间基础上经过一段特定时间后启动定时器。定时器可使用消逝时间或绝对时间。

提供两种类型的定时器:

- 一次性定时器，仅超时一次。
- 周期性定时器，根据指定的时间间隔反复超时。当周期性定时器超时后，MQX RTOS 会复位该定时器。

表 3-40. 汇总：使用定时器

定时器使用某些在 *timer.h* 中定义的结构和常数。	
_timer_cancel	取消未完成的定时器请求。
_timer_create_component	创建定时器组件。
_timer_start_oneshot_after	启动定时器，在一段时间延时（以毫秒为单位）后超时一次。
_timer_start_oneshot_after_ticks	启动定时器，在一段时间延时（以滴答为单位）后超时一次。
_timer_start_oneshot_at	启动定时器，在一段指定时间（以秒/毫秒时间为单位）后超时一次。
_timer_start_oneshot_at_ticks	启动定时器，在一段指定时间（以滴答时间为单位）后超时一次。
_timer_start_periodic_at	启动周期性定时器，在一段指定时间（以秒/毫秒时间为单位）后超时。
_timer_start_periodic_at_ticks	启动周期性定时器，在一段指定时间（以滴答时间为单位）后超时。
_timer_start_periodic_every	每经过指定毫秒数后启动周期性定时器。
_timer_start_periodic_every_ticks	每经过指定滴答数后启动周期性定时器。
_timer_test	测试定时器组件。

3.8.4.1 创建定时器组件

MQX RTOS 在创建定时器组件时会创建定时器任务的优先级和堆栈大小, 您可以通过使用它们调用**_timer_create_component()**来显式地创建定时器组件。定时器任务管理定时器队列，并为通知函数提供上下文。

如果您不显式地创建定时器组件, MQX RTOS 则会在应用程序第一次启动定时器时利用默认值创建。

表 3-41. 默认定时器任务参数

参数	默认值
定时器任务优先级	1
定时器任务堆栈大小	500

3.8.4.2 启动定时器

任务通过以下函数启动定时器:

- **_timer_start_oneshot_after()**和**_timer_start_oneshot_after_ticks()**
- **_timer_start_oneshot_at()**和**_timer_start_oneshot_at_ticks()**
- **_timer_start_periodic_at()**和**_timer_start_periodic_at_ticks()**
- **_timer_start_periodic_every()**和**_timer_start_periodic_every_ticks()**

当任务调用这些函数时, MQX RTOS 会向未完成的定时器队列插入定时器请求。当定时器超时后，会运行通知函数。

附注	定时器任务的堆栈空间必须包括通知函数所需的堆栈空间。

3.8.4.3 取消未完成的定时器请求

任务可以使用一个**_timer_start** 系列函数返回的定时器处理程序，调用**_timer_cancel()**以取消未完成的定时器请求。

3.8.4.4 示例: 使用定时器

模拟 LED 每秒点亮和关闭。一个定时器点亮 LED，另一个用于关闭。定时器每两秒超时一次，偏移量为一秒。

3.8.4.4.1 定时器代码示例

```
/* main.c */
#include <mqx.h>
#include <bsp.h>
#include <fio.h>
#include <timer.h>
#define TIMER_TASK_PRIORITY  2
#define TIMER_STACK_SIZE     1000
#define MAIN_TASK   10
extern void main_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice
*/
  { MAIN_TASK,  main_task,  2000,  8,        "Main", MQX_AUTO_START_TASK, 0,     0},
  { 0 }
};
/*FUNCTION*-------------------------------------------------------
*
* Function Name  : LED_on
* Returned Value : none
* Comments       :
*     This timer function prints "ON"
*END*-------------------------------------------------------------*/
void LED_on
   (
      _timer_id id,
      void * data_ptr,
      MQX_TICK_STRUCT_PTR tick_ptr
   )
{
   printf("ON\n");
}
/*FUNCTION*-------------------------------------------------------
*
* Function Name  : LED_off
* Returned Value : none
* Comments       :
*     This timer function prints "OFF"
*END*-------------------------------------------------------------*/
void LED_off
   (
      _timer_id id,
      void * data_ptr,
      MQX_TICK_STRUCT_PTR tick_ptr
   )
{
   printf("OFF\n");
}
/*TASK*-----------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*     This task creates two timers, each of a period of 2 seconds,
*     the second timer offset by 1 second from the first.
*END*-------------------------------------------------------------*/
void main_task
   (
      uint32_t initial_data
   )
{
   MQX_TICK_STRUCT ticks;
   MQX_TICK_STRUCT dticks;
   _timer_id       on_timer;
   _timer_id       off_timer;
   /*
```

```
   ** Create the timer component with more stack than the default
   ** in order to handle printf() requirements:
   */
   _timer_create_component(TIMER_DEFAULT_TASK_PRIORITY, 1024);
   _time_init_ticks(&dticks, 0);
   _time_add_sec_to_ticks(&dticks, 2);
   _time_get_ticks(&ticks);
   _time_add_sec_to_ticks(&ticks, 1);
   on_timer = _timer_start_periodic_at_ticks(LED_on, 0,
      TIMER_ELAPSED_TIME_MODE, &ticks, &dticks);
   _time_add_sec_to_ticks(&ticks, 1);
   off_timer = _timer_start_periodic_at_ticks(LED_off, 0,
      TIMER_ELAPSED_TIME_MODE, &ticks, &dticks);
   _time_delay_ticks(600);
   printf("\nThe task is finished!");
   _timer_cancel(on_timer);
   _timer_cancel(off_timer);
   _mqx_exit(0);
}
```

3.8.4.4.2 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\timer
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序

定时器通知函数每运行一次输出一条消息。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。

3.8.5 轻量级定时器

轻量级定时器是可选组件，用于扩展内核时间组件。轻量级定时器为应用程序提供周期性通知。

任务可以创建一个周期性队列，并向其添加定时器。定时器超时与队列周期速率一致，但与周期的超时时间不同。

表 3-42. 汇总：使用轻量级定时器

轻量级定时器使用某些在 *lwtimer.h.*中定义的结构和常数。	轻量级定时器使用某些在 *lwtimer.h.*中定义的结构和常数。
_lwtimer_add_timer_to_queue	向周期性队列添加轻量级定时器。
_lwtimer_cancel_period	删除周期性队列中的所有定时器。
_lwtimer_cancel_timer	删除周期性队列中的一个定时器。

下一页继续介绍此表...

表 3-42. 汇总：使用轻量级定时器 (继续)

_lwtimer_create_periodic_queue	创建周期性队列（具有指定滴答数周期），向其添加轻量级定时器。
_lwtimer_test	测试所有周期性队列及其定时器。

3.8.5.1 启动轻量级定时器

任务要启动一个轻量级定时器，首先要通过调用**_lwtimer_create_periodic_queue()**函数创建一个周期性队列，以及指向变量类型 **LWTIMER_PERIOD_STRUCT** 的指针，用于指定队列周期（以滴答格式）。然后通过队列地址和 **LWTIMER_STRUCT** 调用**_lwtimer_add_timer_to_queue()**来添加定时器，当定时器溢出时，指定的函数将被调用。

当定时器超时后，会运行由定时器指定的通知函数。

附注	由于通知函数在内核定时器 ISR 环境下运行，其受制于 ISR 限制（请参见 ISR 限制。）。 MQX 中断堆栈大小必须包括通知函数所需的堆栈空间。

3.8.5.2 取消未完成的轻量级定时器请求

任务可以通过地址 **LWTIMER_STRUCT** 调用**_lwtimer_cancel_timer()**，从而取消未完成的轻量级定时器请求。

任务也可以通过地址 **LWTIMER_PERIOD_STRUCT** 调用**_lwtimer_cancel_period()**，从而取消轻量级定时器队列上的所有定时器。

3.8.6 看门狗

大多数嵌入式系统都具有硬件看门狗定时器。如果应用程序在特定时间内不复位定时器 (可能由于死锁或某些其他错误条件)，硬件会产生一个复位操作。因此，硬件看门狗定时器监视处理器上的所有应用程序；但不监视独立的任务。

附注	为了在某些目标平台上优化代码和数据存储器要求，看门狗组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

MQX 看门狗组件为每个任务提供软件看门狗。如果单个任务崩溃或者超时时，看门狗会提供检测问题的途径。首先，任务通过设置特定时间值启动看门狗，如果任务在超时前没有停止或重启看门狗, MQX RTOS 会调用一个处理器唯一且由应用程序提供的能够恢复错误的函数。

表 3-43. 汇总：使用看门狗

看门狗使用某些在 *watchdog.h* 中定义的结构和常数。	看门狗使用某些在 *watchdog.h* 中定义的结构和常数。
_watchdog_create_component	创建看门狗组件。
_watchdog_start	启动或重启看门狗（时间以毫秒为单位）。
_watchdog_start_ticks	启动或重启看门狗（时间以滴答为单位）。
_watchdog_stop	停止看门狗。
_watchdog_test	测试看门狗组件。

3.8.6.1 创建看门狗组件

要使任务能够使用看门狗组件，应用程序必须首先调用**_watchdog_create_component()**来显式地创建该组件，调用时需要提供周期性定时器器件的中断向量和看门狗到期时由 MQX RTOS 调用的函数的指针。

3.8.6.2 启动或复位看门狗

任务通过调用以下函数来启动或复位看门狗:

- 看门狗溢出之前若干毫秒之内调用**_watchdog_start()**。
- 看门狗溢出之前若干时钟嘀嗒之内调用**_watchdog_start_ticks()**。

如果任务在看门狗超时前未复位或停止看门狗，MQX RTOS 会调用终止函数。

3.8.6.3 停止看门狗

任务通过调用**_watchdog_stop()**函数停止看门狗。

3.8.6.4 示例：使用看门狗

任务在周期性定时器中断向量上创建看门狗组件并指定终止函数(**handle_watchdog_expiry()**)。然后启动将在 2 秒后超时的看门狗。为了防止看门狗超时，任务必须在 2 秒内停止或复位看门狗。

```
/*watchdog.c */
#include <mqx.h>
#include <bsp.h>
#include <watchdog.h>
#define MAIN_TASK       10
extern void    main_task(uint32_t);
extern void    handle_watchdog_expiry(void *);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice
*/
  { MAIN_TASK,  main_task,  2000,  8,        "Main", MQX_AUTO_START_TASK, 0,     0},
  { 0 }
};
/*FUNCTION*-------------------------------------------------------
*
* Function Name  : handle_watchdog_expiry
* Returned Value : none
* Comments       :
*     This function is called when a watchdog has expired.
*END*-------------------------------------------------------------*/
void handle_watchdog_expiry(void * td_ptr)
{
    printf("\nwatchdog expired for task: %p", td_ptr);
}
/*FUNCTION*-------------------------------------------------------
*
* Function Name  : waste_time
* Returned Value : input value times 10
* Comments       :
*     This function loops the specified number of times,
*     essentially wasting time.
*END*-------------------------------------------------------------*/
_mqx_uint waste_time
   (
      _mqx_uint n
   )
{
   _mqx_uint          i;
   volatile _mqx_uint result;
   result = 0;
   for (i = 0; i < n; i++) {
      result += 1;
   }
   return result*10;
}
/*TASK*-----------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*   This task creates a watchdog, then loops, performing
*   work for longer and longer periods until the watchdog fires.
*END*-------------------------------------------------------------*/
void main_task
   (
      uint32_t initial_data
   )
{
   MQX_TICK_STRUCT ticks;
   _mqx_uint       result;
   _mqx_uint       n;
   _time_init_ticks(&ticks, 10);

   result = _watchdog_create_component(BSP_TIMER_INTERRUPT_VECTOR,
      handle_watchdog_expiry);
   if (result != MQX_OK) {
      printf("\nError creating watchdog component");
      _mqx_exit(0);
   }
   n = 100;
```

```
   while (TRUE) {
      result = _watchdog_start_ticks(&ticks);
      n = waste_time(n);
      _watchdog_stop();
      printf("\n %d", n);
   }
}
```

3.8.6.4.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

```
mqx\examples\watchdog
```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序

当看门狗到期时，主任务会将消息输出至输出设备。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 将成为 MQX 开发和构建的首选环境。 请参阅《*Freescale MQX™ RTOS* 入门》文档以获取有关支持的工具链的更多详细信息。

3.9 处理中断和异常

MQX RTOS 通过中断服务例程（ISR）来处理硬件中断和异常。ISR 不是任务；它是一种对硬件中断或异常迅速作出响应的小型的快速响应程序。ISR 通常用 C 语言编写。ISR 的功能可能包括:

- 为器件提供服务
- 清除错误条件
- 向任务发信号

当 MQX RTOS 调用 ISR 时，它传递一个在应用程序装载 ISR 时由应用程序定义的参数。例如，参数可以是用来配置具体器件结构的指针。

附注	参数不应指向任务堆栈中的数据，因为 ISR 可能不能访问存储器。

可能会禁用一些 ISR 中的中断，具体取决于服务中断的优先级。因此，ISR 执行尽可能少量的函数。ISR 通常使任务就绪。然后根据任务的优先级确定处理来自中断器件信息的速度有多快。ISR 提供多种途径使任务就绪：通过轻量级事件、事件、轻量级信号量、信号量、消息、轻量级消息队列或任务队列。

MQX RTOS 提供一个内核 ISR, 以汇编语言编写。内核 ISR 在任何其他 ISR 之前运行，并执行以下操作:

- 保存活动任务的上下文。
- 切换到中断堆栈。
- 调用合适的 ISR。
- 在 ISR 返回后，恢复最高优先级就绪任务的上下文。

当 MQX RTOS 启动时，它会对所有可能中断装载默认内核 ISR (**_int_kernel_isr()**)。

当 ISR 返回到内核 ISR 时，如果 ISR 就绪的任务优先级高于中断时活动的任务优先级，则内核 ISR 会执行任务分配操作。这意味着之前活动任务的上下文被保存，较高优先级任务变为活动任务。

以下图表显示了 MQX RTOS 是如何处理中断的。

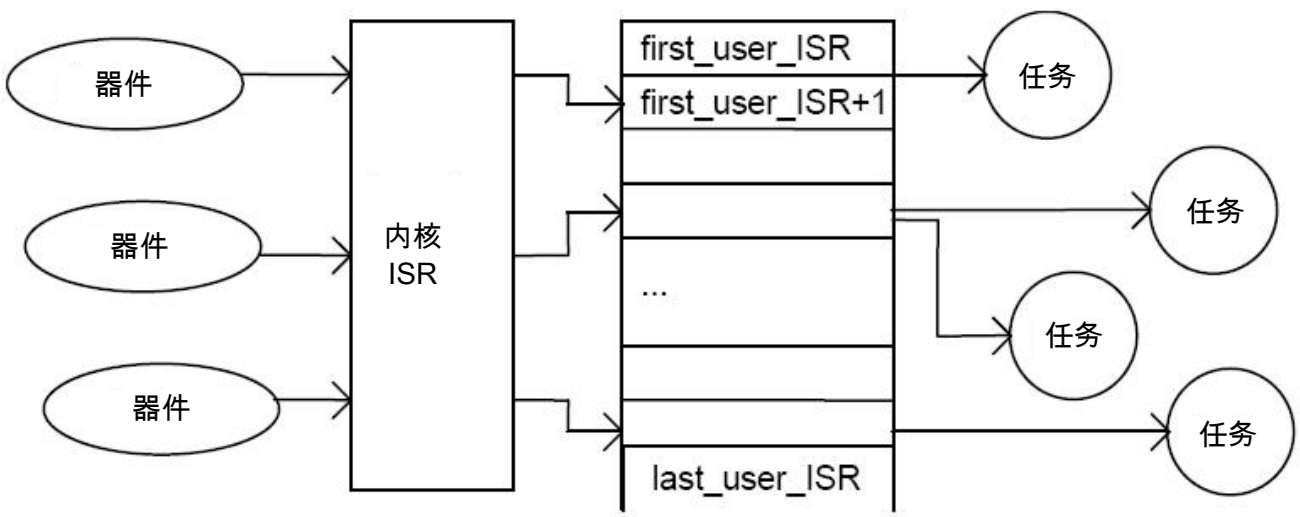

图 3-3. 处理中断

表 3-44. 汇总：处理中断和异常

_int_disable	禁止硬件中断。
_int_enable	启用硬件中断。
_int_get_isr	获取向量编号的 ISR。
_int_get_isr_data	获取中断相关的数据指针。
_int_get_isr_depth	获取当前 ISR 的嵌套深度。
_int_get_kernel_isr	获取中断的内核 ISR。
_int_get_previous_vector_table	获取当 MQX RTOS 启动时存储的中断向量表指针。
_int_get_vector_table	获取当前中断向量表的指针。
_int_install_isr	装载应用程序定义的 ISR。
_int_install_kernel_isr	装载内核 ISR。
_int_install_unexpected_isr	装载_int_unexpected_isr()作为默认 ISR。

下一页继续介绍此表...

表 3-44. 汇总：处理中断和异常 (继续)

_int_kernel_isr	默认内核 ISR。
_int_set_isr_data	设置指定中断相关的数据。
_int_set_vector_table	修改向量表位置。

3.9.1 初始化中断处理

当 MQX RTOS 启动时，它会初始化 ISR 表，其中包含对应于各中断编号的入口。每个入口包含:

- 用于调用 ISR 的指针。
- 作为参数传递到 ISR 的数据。
- 指向 ISR 异常句柄的指针。

最初，每个入口的 ISR 为默认 ISR **_int_default_isr()**，将阻塞活动的任务。

3.9.2 装载应用程序定义的 ISR

通过**_int_install_isr()**函数，应用程序可以用应用程序定义的、中断指定的 ISR 来取代中断发生时 MQX RTOS 调用的 ISR。在初始化器件前，应用程序不应执行该取代操作。

用于**_int_install_isr()**函数的参数包括:

- 中断编号
- ISR 函数指针
- ISR 数据
- 应用程序定义的 ISR 通常可以执行以下操作向任务发送信号:
- 设置事件位（**_event_set()**）。
- 发布轻量级信号量（**_lwsem_post()**）。
- 发布非严谨信号量（**_sem_post()**）。
- 向消息队列发送消息。ISR 还能从系统消息队列中接收消息（**_msgq_send** 系列）。

附注	从 ISR 分配消息的最有效方式是使用**_msg_alloc()**。

- 将某个任务从任务队列中排除，这将使任务放入任务就绪队列。任务队列使您可以设置专为应用程序定制的信号发送方法（**_taskq_resume()**）。

3.9.3 ISR 限制。

下表包含了关于 ISR 限制的信息。

3.9.3.1 ISR 不能调用的函数

如果 ISR 调用以下任意函数，MQX RTOS 将返回错误。

表 3-45. ISR 不能调用的函数

组件	函数
事件	**_event_close() _event_create() _event_create_auto_clear() _event_create_component() _event_create_fast() _event_create_fast_auto_clear() _event_destroy() _event_destroy_fast() _event_wait_all** 系列 **_event_wait_any** 系列
轻量级事件	**_lwevent_destroy() _lwevent_test() _lwevent_wait** 系列
轻量级日志	**_lwlog_create_component()**
轻量级消息队列	**_lwmsgq_send()** （当使用 LWMSGQ_SEND_BLOCK_ON_FULL 或 LWMSGQ_SEND_BLOCK_ON_SEND 标志时） _lwmsgq_receive()
经量级信号量	**_lwsem_test() _lwsem_wait()**
日志	**_log_create_component()**
消息	**_msg_create_component() _msgq_receive** 系列
互斥	**_mutex_create_component() _mutex_lock()**
命名	**_name_add() _name_create_component() _name_delete()**
分区	**_partition_create_component()**
信号量	**_sem_close() _sem_create() _sem_create_component() _sem_create_fast() _sem_destroy() _sem_destroy_fast() _sem_post()** （仅用于严谨的信号量） **_sem_wait** 系列
任务队列	**_taskq_create() _taskq_destroy() _taskq_suspend() _taskq_suspend_task() _taskq_test()**
定时器	**_timer_create_component() _timer_cancel()**
看门狗	**_watchdog_create_component()**

3.9.3.2 ISR 不应调用的函数

ISR 不应调用 MQX 函数，这可能会导致阻塞运行或消耗过长的时间。这些函数包括:

- **_io_**系列中的大部分函数
- **_event_wait** 系列

- **_int_default_isr()**
- **_int_unexpected_isr()**
- **_klog_display()**
- **_klog_show_stack_usage()**
- **_lwevent_wait** 系列
- **_lwmsgq_send()** （当使用 LWMSGQ_SEND_BLOCK_ON_FULL 或 LWMSGQ_SEND_BLOCK_ON_SEND 标志时）
- **_lwmsgq_receive()**
- **_lwsem_wait** 系列
- **_msgq_receive** 系列
- **_mutatr_set_wait_protocol()**
- **_mutex_lock()**
- **_partition_create_component()**
- **_task_block()**
- **_task_create()**和**_task_create_blocked()**
- **_task_destroy()**
- **_time_delay** 系列
- **_timer_start** 系列

3.9.3.3 不可屏蔽中断

不可屏蔽中断（NMI）可理解为不能由软件禁止（屏蔽）的中断。可以在 MQX 应用程序中使用这种中断，不过 NMI 服务例程必须作为内核 ISR 直接装载到向量表（使用**_int_install_kernel_isr()**函数而不是_int_install_isr()函数）。NMI 服务例程不允许调用任何 MQX API 函数。

请注意，只有当向量表位于 RAM 存储器中时，才会启用_int_install_kernel_isr()调用（请参见在编译时配置 MQX RTOS 章节中的 MQX_ROM_VECTORS 配置选项）。

3.9.3.4 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数

在某些处理器平台上，禁用“所有中断等级”可理解为仅禁用 MQX_HARDWARE_INTERRUPT_LEVEL_MAX（MQX_INITIALIZATION_STRUCT 中的字段）及低于此等级的中断等级。这实际允许了对高于最高等级的关键中断请求进行与 MQX 内核执行的异步服务，从而使延时尽可能最短。从 MQX RTOS 角度来看，将这样的中断认为是非屏蔽中断，与 NMI 应用具有相同的限制。

下表汇总了当任务切换到以 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 的值作为定义优先级时，应写入 SR/BASEPRI 寄存器的值。

以 ColdFire 平台为例，当 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 设为 7 切换到优先级为 4 的任务时，会导致 SR 寄存器装载数值 2。这说明优先级低于 3 的中断不能中断该任务。

表 3-46. 适用于 ColdFire 平台的不同任务优先级和不同 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 值的 SR 寄存器值

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	任务优先级							
0	1	2	3	4	5	6	7	
0	不允许。有效的更改方式为 MQX_HARDWARE_INTERRUPT_LEVEL_MAX=1							
1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	2	1	0	0	0	0	0	0
4	3	2	1	0	0	0	0	0
5	4	3	2	1	0	0	0	0
6	5	4	3	2	1	0	0	0
7	6	5	4	3	2	1	0	0
8	不允许。有效的更改方式为 MQX_HARDWARE_INTERRUPT_LEVEL_MAX=70							

在基于 Cortex-M4 和 Cortex-A5 内核的平台上，MQX 中断处理设计为这种方式。Kinetis K MCU 支持 16 级硬件中断优先级。MQX RTOS 映射偶数等级（0, 2, 4, .., 14）用于 MQX 应用程序，而奇数等级（1, 3, .., 15）供内部使用。MQX 应用程序中断等级为 0-7，从 MQX 应用程序等级 0-7 到硬件优先级等级（0、2-14）的映射是在_bsp_int_init()函数中设置的。

使用如下代码在 Kinetis K 上装载 MQX 应用程序定义的 ISR:

```
_int_install_isr(vector, isr_ptr, isr_data);
_bsp_int_init(vector, priority, subpriority, enable);
```

向量—非内核向量编号（例如，LLWU 为 37，在 MCU 头文件的 IRQInterruptIndex 中定义）。

优先级—中断源的优先级。允许值:
MQX_HARDWARE_INTERRUPT_LEVEL_MAX 与 7 之间的任意整数（这两个值包括在内），数值越小，优先级越高。

子优先级—Kinetis K 不具备子优先级。

启用—TRUE 表示在 NVIC 中启用中断向量源。

使用以下代码在 Kinetis K 上装载内核 ISR（旁路 MQX RTOS）:

```
_int_install_kernel_isr(Vector, isr_ptr); /* works only for vector table located in the RAM
*/
_bsp_int_init(vector, priority, subpriority, enable);
```

向量—非内核向量编号（例如，FTM1 为 79，在 MCU 头文件的 IRQInterruptIndex 中定义）。

优先级—中断源的优先级。允许值：0（最高优先级中断）至 7。

子优先级—Kinetis K 不具备子优先级。

启用—TRUE 表示在 NVIC 中启用中断向量源。

请注意，由于 ARM 硬件中断的堆栈特性，内核 ISR 可以是任何具有 void my_kernel_isr(void)说明的 C 语言函数。

不同任务优先级和不同 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 值的 ARM Cortex-M4 BASEPRI 寄存器值如下表所示。注意，最高有效半字节用于设置优先级。关于 BASEPRI 寄存器的说明，请参见《ARM 参考手册》。

示例：BASEPRI=0x20，高位半字节为 0x2，意味着只有硬件优先级为 1 或 0 的中断才能打断该任务。

表 3-47. 适用于基于 ARM® Cortex®-M4 内核平台的不同任务优先级和不同 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 值的 SR 寄存器值

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	任务优先级							
0	1	2	3	4	5	6	7	
0	不允许。有效的更改方式为 MQX_HARDWARE_INTERRUPT_LEVEL_MAX=1							
1	0x20	0x40	0x60	0x80	0xA0	0xC0	0xE0	0
2	0x40	0x60	0x80	0xA0	0xC0	0xE0	0	0
3	0x60	0x80	0xA0	0xC0	0xE0	0	0	0
4	0x80	0xA0	0xC0	0xE0	0	0	0	0
5	0xA0	0xC0	0xE0	0	0	0	0	0
6	0xC0	0xE0	0	0	0	0	0	0
7	0xE0	0	0	0	0	0	0	0
8	不允许。有效的更改方式为 MQX_HARDWARE_INTERRUPT_LEVEL_MAX=70							

不同任务优先级和不同 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 值的 ARM Cortex-A5 中断优先级屏蔽寄存器（GICC_PMR – GIC 寄存器）值如下表所示。注意，最高有效半字节用于设置优先级。关于 GICC_PMR 寄存器的说明，请参见《ARM 通用中断控制器（INTC）架构规格》。

表 3-48. 适用于基于 ARM® Cortex®-A5 内核平台的不同任务优先级和不同 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 值的 SR 寄存器值

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	任务优先级							
0	1	2	3	4	5	6	7	

下一页继续介绍此表...

表 3-48. 适用于基于 ARM® Cortex®-A5 内核平台的不同任务优先级和不同 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 值的 SR 寄存器值 (继续)

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	任务优先级							
0	不允许。有效的更改方式为 MQX_HARDWARE_INTERRUPT_LEVEL_MAX=1							
1	0x20	0x40	0x80	0xA0	0xC0	0xE0	0xFF	0xFF
2	0x40	0x80	0xA0	0xC0	0xE0	0xFF	0xFF	0xFF
3	0x80	0xA0	0xC0	0xE0	0xFF	0xFF	0xFF	0xFF
4	0xA0	0xC0	0xE0	0xFF	0xFF	0xFF	0xFF	0xFF
5	0xC0	0xE0	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
6	0xE0	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
7	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
8	不允许。有效的更改方式为 MQX_HARDWARE_INTERRUPT_LEVEL_MAX=7							

对于 Freescale PowerPC®器件和 ARM® Cortex®-M0+器件，不支持基于运行中任务优先级的自动优先级切换，并且所有外设中断总是由*_int_disable* 禁用，无论 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 如何设定。

3.9.4 更改默认 ISR

当 MQX RTOS 处理中断时，会调用**_int_kernel_isr()**，当以下条件之一成立时，该函数使用中断编号调用默认 ISR:

- 应用程序未对该中断编号安装应用程序定义的 ISR。
- 中断编号在 ISR 表范围外。

应用程序可以使用**_int_get_default_isr()**获取默认 ISR 的指针。

应用程序可以更改默认 ISR，如下表所述。

表 3-49. 默认 ISR

默认 ISR	描述	修改或安装方式
_int_default_isr	当 MQX RTOS 启动时，将其安装为默认 ISR。它会阻塞任务。	修改：**_int_install_default_ isr()**
_int_exception_isr	设置 MQX RTOS 异常处理。	安装：**_int_install_exception_ isr()**
_int_unexpected_ isr	与**_int_default_isr()**相似，但还输出消息到默认控制台，识别未处理的中断。	安装：**_int_install_unexpected_ isr()**

3.9.5 处理异常

为了实现 MQX RTOS 异常处理，应用程序应调用**_int_install_exception_isr()**函数，将_int_exception_isr()装载为默认 ISR。从而，当发生了异常或未处理中断，会调用**_int_exception_isr()**函数。当异常发生时，函数**_int_exception_isr()**执行以下操作:

- 如果当任务正在运行并且存在任务异常 ISR 时发生了异常，MQX RTOS 会运行 ISR；如果不存在任务异常 ISR，MQX RTOS 会通过调用**_task_abort()**函数终止任务。
- 如果当 ISR 正在运行并且存在 ISR 异常 ISR 时发生了异常，MQX RTOS 会终止正在运行的 ISR，并运行 ISR 异常 ISR。
- 函数通过中断堆栈查找关于在异常发生之前运行的 ISR 或任务信息。

附注	如果 MQX RTOS 异常 ISR 确定中断堆栈包含的信息不正确，则它会调用错误代码为 **MQX_CORRUPT_INTERRUPT_STACK** 的**_mqx_fatal_error()**函数。

3.9.6 处理 ISR 异常

应用程序可以为每个 ISR 装载一个 ISR 异常句柄。如果在 ISR 运行时发生异常，MQX RTOS 会调用句柄并终止 ISR。如果应用程序未装载异常句柄，MQX RTOS 将直接终止 ISR。

当 MQX RTOS 调用异常句柄时，它将传递:

- 当前 ISR 编号
- ISR 数据指针
- 异常编号
- 异常帧的堆栈地址

表 3-50. 汇总：处理 **ISR** 异常

_int_get_exception_handler	获取用于 ISR 的当前异常句柄指针。
_int_set_exception_handler	设置用于中断的当前 ISR 异常句柄地址。

3.9.7 处理任务异常

如果任务导致了一个不支持的异常，它可以装载一个供 MQX RTOS 调用的任务异常句柄。

表 3-51. 汇总：处理任务异常

_task_get_exception_handler	获取任务异常句柄。
_task_set_exception_handler	设置任务异常句柄。

3.9.8 示例：安装 ISR

安装 ISR 以截取内核定时器中断。将 ISR 与前一 ISR 链接起来，后者是 BSP 提供的周期定时器 ISR。

```
/* isr.c */
#include <mqx.h>
#include <bsp.h>
#define MAIN_TASK       10
extern void main_task(uint32_t);
extern void new_tick_isr(void *);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice
*/
  { MAIN_TASK,  main_task,  2000,  8,        "Main", MQX_AUTO_START_TASK, 0,     0},
  { 0 }
};
typedef struct
{
   void *      OLD_ISR_DATA;
   INT_ISR_FPTR OLD_ISR;
   _mqx_uint    TICK_COUNT;
} MY_ISR_STRUCT, * MY_ISR_STRUCT_PTR;
/*ISR*-------------------------------------------------------------
*
* ISR Name : new_tick_isr
* Comments :
*   This ISR replaces the existing timer ISR, then calls the
*   old timer ISR.
*END*-------------------------------------------------------------*/
void new_tick_isr
   (
      void * user_isr_ptr
   )
{
   MY_ISR_STRUCT_PTR  isr_ptr;
   isr_ptr = (MY_ISR_STRUCT_PTR)user_isr_ptr;
   isr_ptr->TICK_COUNT++;
   /* Chain to previous notifier */
   (*isr_ptr->OLD_ISR)(isr_ptr->OLD_ISR_DATA);
}
/*TASK*-------------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*   This task installs a new ISR to replace the timer ISR.
*   It then waits for some time, finally printing out the
*   number of times the ISR ran.
*END*-------------------------------------------------------------*/
void main_task
   (
      uint32_t initial_data
   )
{
   MY_ISR_STRUCT_PTR  isr_ptr;
```

```
   isr_ptr = _mem_alloc_zero(sizeof(MY_ISR_STRUCT));
   isr_ptr->TICK_COUNT   = 0;
   isr_ptr->OLD_ISR_DATA =
      int_get_isr_data(BSP_TIMER_INTERRUPT_VECTOR);
   isr_ptr->OLD_ISR      =
      int_get_isr(BSP_TIMER_INTERRUPT_VECTOR);
   _int_install_isr(BSP_TIMER_INTERRUPT_VECTOR, new_tick_isr,
      isr_ptr);
   _time_delay_ticks(200);
   printf("\nTick count = %d\n", isr_ptr->TICK_COUNT);
   _mqx_exit(0);
}
```

3.9.8.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\isr
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序

主任务显示应用程序 ISR 被调用的次数。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 已成为 MQX 开发和构建的首选环境。请参阅《*Freescale MQX™ RTOS* 入门》以获取有关支持的工具链的更多详细信息。

3.10 检测工具

检测工具包括以下组件:

- 日志
- 轻量级日志
- 内核日志
- 堆栈使用情况实用程序

3.10.1 日志

许多实时应用程序需要记录关于重要情况的信息，如事件、状态转换或函数的进入和退出信息。如果应用程序记录了发生的信息，那么您可以通过分析这些信息序列来确定应用程序是否处理正确。如果每条信息都具有时间戳 (以绝对时间表示)，那么您可以确定应用程序在何时使用处理时间，从而了解应优化哪段代码。

附注	为了在某些目标平台上优化代码和数据存储器要求，日志组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

通过日志组件,您可以将数据存入最多 16 个日志中并对其进行检索。每个日志均有一个预确定的条目数。每个条目包含一个时间戳 (以绝对时间表示)、一个序列号和应用程序定义的数据。

表 3-52. 汇总：使用日志

日志使用某些在 *log.h* 中定义的结构和常数。	日志使用某些在 *log.h* 中定义的结构和常数。
_log_create	创建日志。
_log_create_component	创建日志组件。
_log_destroy	销毁日志。
_log_disable	禁止日志记录。
_log_enable	启用日志记录。
_log_read	读取日志。
_log_reset	复位日志中的内容。
_log_test	测试日志组件。
_log_write	写入日志。

3.10.1.1 创建日志组件

您可以使用**_log_create_component()**显式地创建日志组件。如果不显式地创建，MQX RTOS 会在应用程序首次创建日志或内核日志时创建。

3.10.1.2 创建日志

要创建日志，任务要调用**_log_create()**并指定:

- 日志编号，范围为 0 到 15。
- 要在日志中存储的最大**_mqx_uint** 数 (包括标头)。
- 日志满后的行为。默认行为是不再写入数据。另一种行为是新条目覆盖最早的条目。

3.10.1.3 日志条目的格式

每个日志条目包含一个日志头 (**LOG_ENTRY_STRUCT**)，后面紧跟应用程序定义的数据。

```
typedef struct
{
  _mqx_uint  SIZE;
  _mqx_uint  SEQUENCE_NUMBER;
  uint32_t    SECONDS;
  uint16_t    MILLISECONDS;
  uint16_t    MICROSECONDS;
} LOG_ENTRY_STRUCT, * LOG_ENTRY_STRUCT_PTR;
```

《MQX 参考手册》中介绍了这些字段。

3.10.1.4 写入日志

任务通过**_log_write()**函数写入日志。

3.10.1.5 读取日志

任务通过调用**_log_read()**函数读取日志，并指定如何读取。可用的日志读取方式如下:

- 读取最新的条目。
- 读取最旧的条目。
- 读取下一个条目 (与读取最旧的条目配合使用)。
- 读取并删除最旧的条目。

3.10.1.6 禁用和启用日志写入

任何任务都可以使用**_log_disable()**禁止使用特定日志。而后，任何任务都可以使用**_log_enable()**启用日志写入。

3.10.1.7 复位日志

任务可以通过**_log_reset()**函数将日志内容复位为最初的无数据状态。

3.10.1.8 示例: 使用日志

```
/* log.c */
#include <mqx.h>
#include <bsp.h>
#include <log.h>
#define MAIN_TASK  10
#define MY_LOG     1
extern void main_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
```

```
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice
*/
  { MAIN_TASK,  main_task,  2000,  8,        "Main", MQX_AUTO_START_TASK, 0,     0},
  { 0 }
};
typedef struct entry_struct
{
   LOG_ENTRY_STRUCT   HEADER;
   _mqx_uint          C;
   _mqx_uint          I;
} ENTRY_STRUCT, * ENTRY_STRUCT_PTR;
/*TASK*-------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*   This task logs 10 keystroke entries then prints out the log.
*END*-------------------------------------------------------------*/
void main_task
   (
      uint32_t initial_data
   )
{
   ENTRY_STRUCT entry;
   _mqx_uint    result;
   _mqx_uint    i;
   uchar        c;
   /* Create the log component. */
   result = _log_create_component();
   if (result != MQX_OK) {
      printf("Main task - _log_create_component failed!");
      _mqx_exit(0);
   }
   /* Create a log */
   result = _log_create(MY_LOG,
      10 * (sizeof(ENTRY_STRUCT)/sizeof(_mqx_uint)), 0);
   if (result != MQX_OK) {
      printf("Main task - _log_create failed!");
      _mqx_exit(0);
   }
   /* Write data into the log */
   printf("Please type in 10 characters:\n");
   for (i = 0; i < 10; i++) {
      c = getchar();
      result = _log_write(MY_LOG, 2, (_mqx_uint)c, i);
      if (result != MQX_OK) {
         printf("Main task - _log_write failed!");
      }
   }
   /* Read data from the log */
   printf("\nLog contains:\n");
   while (_log_read(MY_LOG, LOG_READ_OLDEST_AND_DELETE, 2,
     (LOG_ENTRY_STRUCT_PTR)&entry) == MQX_OK)
   {
      printf("Time: %ld.%03d%03d, c=%c, i=%d\n",
         entry.HEADER.SECONDS,
         (_mqx_uint)entry.HEADER.MILLISECONDS,
         (_mqx_uint)entry.HEADER.MICROSECONDS,
         (uchar)entry.C & 0xff,
         entry.I);
   }
   /* Delete the log */
   _log_destroy(MY_LOG);
   _mqx_exit(0);
}
```

3.10.1.8.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

```
mqx\examples\log
```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序
4. 在输入控制台键入 10 个字符。

该程序将记录这些字符并在控制台上显示该日志条目。

附注	借助 Freescale MQX RTOS，CodeWarrior Development Studio 已成为 MQX 开发和构建的首选环境。 请参阅《Freescale MQX™ RTOS 入门》文档以获取有关支持的工具链的更多详细信息。

3.10.2 轻量级日志

轻量级日志与日志相似（见日志），但有以下区别:

- 所有轻量级日志中的所有条目大小均相同。
- 可以在指定的存储器位置创建轻量级日志。
- 轻量级日志的时间戳可以是滴答时间或秒/毫秒时间形式，这取决于编译时的 MQX RTOS 配置（欲了解更多信息，请参见在编译时配置 MQX RTOS）。

附注	为了在某些目标平台上优化代码和数据存储器要求，轻量级日志组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

表 3-53. 汇总：使用轻量级日志

轻量级日志使用某些在 *lwlog.h* 中定义的结构和常数。	轻量级日志使用某些在 *lwlog.h* 中定义的结构和常数。
_lwlog_calculate_size	计算一个指定最大条目数的轻量级日志所需的大小。
_lwlog_create	创建轻量级日志。
_lwlog_create_at	在某个位置创建轻量级日志。
_lwlog_create_component	创建轻量级日志组件。
_lwlog_destroy	撤销轻量级日志。
_lwlog_disable	禁止记录轻量级日志。
_lwlog_enable	启用记录轻量级日志。
_lwlog_read	读取轻量级日志。
_lwlog_reset	复位轻量级日志中的内容。
_lwlog_test	测试轻量级日志组件。
_lwlog_write	写入轻量级日志。

3.10.2.1 创建轻量级日志组件

您可以使用**_lwlog_create_component()**显式地创建轻量级日志组件。如果不显式地创建，MQX RTOS 会在应用程序首次创建轻量级日志或内核日志时创建。

3.10.2.2 创建轻量级日志

任务可在特定位置(**_lwlog_create_at()**)创建轻量级日志，也可以让 MQX RTOS 选择位置(**_lwlog_create()**)。

无论使用哪一函数，任务都要指定:

- 1-15 范围内的日志编号（0 保留用于内核日志)。
- 日志中的最大条目数。
- 日志满后的行为。默认行为是不再写入数据。另一种行为是新条目覆盖最早的条目。

如果使用**_lwlog_create_at()**，任务还要指定日志的地址。

3.10.2.3 轻量级日志条目的格式

每个轻量级日志条目均为以下结构。

```
typedef struct lwlog_entry_struct
  _mqx_uint        SEQUENCE_NUMBER;
#if MQX_LWLOG_TIME_STAMP_IN_TICKS == 0
  /* Time at which the entry was written: */
  uint32_t          SECONDS;
  uint32_t          MILLISECONDS;
  uint32_t          MICROSECONDS;
#else
  /* Time (in ticks) at which the entry was written: */
  MQX_TICK_STRUCT  TIMESTAMP;
#endif
  _mqx_max_type    DATA[LWLOG_MAXIMUM_DATA_ENETRIES];
  struct lwlog_entry_struct * NEXT_PTR;
} LWLOG_ENTRY_STRUCT, * LWLOG_ENTRY_STRUCT_PTR;
```

《MQX 参考手册》中介绍了这些字段。

3.10.2.4 写入轻量级日志

任务通过**_lwlog_write()**函数写入轻量级日志。

3.10.2.5 读取轻量级日志

任务通过调用**_lwlog_read()**函数读取轻量级日志，并指定如何读取日志。可用的日志读取方式如下:

- 读取最新的条目。
- 读取最旧的条目。
- 读取下一个条目 (与读取最旧的条目配合使用)。
- 读取并删除最旧的条目。

3.10.2.6 禁用和启用轻量级日志写入

任何任务都可以使用**_lwlog_disable()**禁止使用特定轻量级日志。而后，任何任务都可以使用**_lwlog_enable()**启用日志轻量级写入。

3.10.2.7 复位轻量级日志

任务可以通过**_lwlog_reset()**函数将轻量级日志内容复位为最初的无数据状态。

3.10.2.8 示例: 使用轻量级日志

```
/* lwlog.c */
#include <mqx.h>
#include <bsp.h>
#include <lwlog.h>
#define MAIN_TASK  10
#define MY_LOG     1
extern void main_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice
*/
  { MAIN_TASK,  main_task,  2000,  8,        "Main", MQX_AUTO_START_TASK, 0,     0},
  { 0 }
};
/*TASK*-------------------------------------------------------
*
* Task Name : main_task
* Comments  :
*   This task logs 10 keystroke entries in a lightweight log,
*   then prints out the log.
*END*-------------------------------------------------------------*/
void main_task
   (
      uint32_t initial_data
   )
{
   LWLOG_ENTRY_STRUCT entry;
   _mqx_uint          result;
   _mqx_uint          i;
   uchar              c;
   /* Create the lightweight log component */
   result = _lwlog_create_component();
```

```
   if (result != MQX_OK) {
      printf("Main task: _lwlog_create_component failed.");
      _mqx_exit(0);
   }
   /* Create a log */
   result = _lwlog_create(MY_LOG, 10, 0);
   if (result != MQX_OK) {
      printf("Main task: _lwlog_create failed.");
      _mqx_exit(0);
   }
   /* Write data to the log */
   printf("Enter 10 characters:\n");
   for (i = 0; i < 10; i++) {
      c = getchar();
      result = _lwlog_write(MY_LOG, (_mqx_max_type)c,
         (_mqx_max_type)i, 0, 0, 0, 0, 0);
      if (result != MQX_OK) {
         printf("Main task: _lwlog_write failed.");
      }
   }
   /* Read data from the log */
   printf("\nLog contains:\n");
   while (_lwlog_read(MY_LOG, LOG_READ_OLDEST_AND_DELETE,
      &entry) == MQX_OK)
   {
      printf("Time: ");
#if MQX_LWLOG_TIME_STAMP_IN_TICKS
      _psp_print_ticks((PSP_TICK_STRUCT_PTR)&entry.TIMESTAMP);
#else
      printf("%ld.%03ld%03ld", entry.SECONDS, entry.MILLISECONDS,
         entry.MICROSECONDS);
#endif
      printf(, c=%c, I=%d\n", (uchar)entry.DATA[0] & 0xff,
         (_mqx_uint)entry.DATA[1]);
   }

   /* Destroy the log */
   _log_destroy(MY_LOG);
   _mqx_exit(0);
}
```

3.10.2.8.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\lwlog
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序
4. 在输入控制台键入 10 个字符。

该程序将记录这些字符并在控制台上显示该日志条目。

3.10.3 内核日志

内核日志允许应用程序日志包含以下内容:

- 所有 MQX 函数调用的函数进入和退出信息。
- 指定特定函数调用的函数进入和退出信息。

- 上下文切换。
- 中断。

附注	为了在某些目标平台上优化代码和数据存储器要求，KLog 组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

性能工具使用内核日志数据来分析应用程序的工作和资源使用情况。欲了解更多信息，请参见《MQX 主机工具用户指南》。

表 3-54. 汇总：使用内核日志

内核日志使用某些在 *log.h*、*lwlog.h* 和 *klog.h* 中定义的结构和常数。	内核日志使用某些在 *log.h*、*lwlog.h* 和 *klog.h* 中定义的结构和常数。
_klog_control	控制内核日志记录。
_klog_create	创建内核日志。
_klog_create_at	在指定位置创建内核日志。
_klog_disable_logging_task	对于指定任务禁止内核日志记录。
_klog_enable_logging_task	对于指定任务启用内核日志记录。
_klog_display	显示内核日志中的一个条目。

3.10.3.1 使用内核日志

为了使用内核日志，应用程序遵循以下通用步骤。

1. 如创建轻量级日志组件 中所述创建轻量级日志组件（可选）。
2. 通过**_klog_create()**函数创建内核日志。这与创建轻量级日志相似，见创建轻量级日志组件 中所述。还可以通过**_klog_create_at()**函数在指定位置创建内核日志。
3. 调用**_klog_control()**函数设置对日志记录的控制，指定任意组合的位标志，如下表所述。

表 3-55. 日志记录函数介绍

选择标志用于：		
• MQX 组件	选择用于：	记入日志的函数：
	错误	例如：**_mqx_exit()**、**_task_set_error()**和**_mqx_fatal_error()**函数。
	事件	来自**_event** 系列的大部分函数。
	中断	来自**_int** 系列的某些函数。
	LWSems	**_lwsem** 系列函数。
	存储器	来自**_mem** 系列的某些函数。
	消息	来自**_msg**、**_msgpool** 和**_msgq** 系列的某些函数。

下一页继续介绍此表...

表 3-55. 日志记录函数介绍 (继续)

	互斥	来自**_mutatr** 和**_mutex** 系列的某些函数。
	命名	**_name** 系列函数。
	分区	来自**_partition** 系列的某些函数。
	信号量	来自**_sem** 系列的大部分函数。
	任务	**_sched**、**_task**、**_taskq** 和**_time** 系列函数。
	定时	**_timer** 系列函数；来自**_time** 系列的某些函数。
	看门狗	**_watchdog** 系列函数。
• 仅特定任务（符合资格的任务）	对于要记入日志的每个任务，调用以下函数： **_klog_disable_logging_task()** **_klog_enable_logging_task()**	对于要记入日志的每个任务，调用以下函数： **_klog_disable_logging_task()** **_klog_enable_logging_task()**
• 中断 • 周期性定时器中断（系统时钟） • 上下文切换	• 中断 • 周期性定时器中断（系统时钟） • 上下文切换	• 中断 • 周期性定时器中断（系统时钟） • 上下文切换

3.10.3.2 禁用内核日志记录

内核日志记录会使您的应用程序使用更多资源，运行更缓慢。测试和验证您的应用程序后，您可能要创建不包含记录到内核日志的版本。要为 MQX RTOS 的任意部分删除内核日志记录，您必须将 **MQX_KERNEL_LOGGING** 选项设为 0，重新编译 MQX RTOS。欲了解更多信息，请参见 MQX RTOS 编译时配置选项 。重新编译 Freescale MQX RTOS 中详细介绍了重新编译 MQX RTOS 的完整步骤。

3.10.3.3 示例：使用内核日志

记录所有定时器组件调用和所有周期性定时器中断。

```
/* klog.c */
#include <mqx.h>
#include <bsp.h>
#include <log.h>
#include <klog.h>
extern void main_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function,   Stack, Priority, Name,   Attributes,          Param, Time Slice
*/
  { 10       ,  main_task,  1500,  8,        "Main", MQX_AUTO_START_TASK, 0,     0},
  { 0 }
};
/*TASK*-------------------------------------------------------
*
* Task Name : main_task
```

```
* Comments  :
*   This task logs timer interrupts to the kernel log,
*   then prints out the log.
*END*-----------------------------------------------------------*/
void main_task
   (
      uint32_t initial_data
   )
{
   _mqx_uint result;
   _mqx_uint i;

   /* Create kernel log */
   result = _klog_create(4096, 0);
   if (result != MQX_OK) {
      printf("Main task - _klog_create failed!");
      _mqx_exit(0);
   }
   /* Enable kernel log */
   _klog_control(KLOG_ENABLED | KLOG_CONTEXT_ENABLED |
      KLOG_INTERRUPTS_ENABLED| KLOG_SYSTEM_CLOCK_INT_ENABLED |
      KLOG_FUNCTIONS_ENABLED | KLOG_TIME_FUNCTIONS |
      KLOG_INTERRUPT_FUNCTIONS, TRUE);
   /* Write data into kernel log */
   for (i = 0; i < 10; i++) {
      _time_delay_ticks(5 * i);
   }
   /* Disable kernel log */
   _klog_control(0xFFFFFFFF, FALSE);
   /* Read data from kernel log */
   printf("\nKernel log contains:\n");
   while (_klog_display()){
   }
   _mqx_exit(0);
}
```

3.10.3.3.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\klog
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序

大约 3 秒钟后，**Main_task()**会显示内核日志的内容。

3.10.4 堆栈使用情况实用程序

MQX RTOS 提供内核实用程序使您可以检查和重定义中断堆栈的大小和每个任务堆栈的大小。

表 3-56. 汇总：堆栈使用情况实用程序

为了使用这些实用程序，您必须使用 MQX_MONITOR_STACK 配置 MQX RTOS。欲了解更多信息，请参见 MQX RTOS 编译时配置选项 。重新编译 Freescale MQX RTOS 中详细介绍了重新编译 MQX RTOS 的完整步骤。	为了使用这些实用程序，您必须使用 MQX_MONITOR_STACK 配置 MQX RTOS。欲了解更多信息，请参见 MQX RTOS 编译时配置选项 。重新编译 Freescale MQX RTOS 中详细介绍了重新编译 MQX RTOS 的完整步骤。
_klog_get_interrupt_stack_ usage	获取中断堆栈边界和使用的堆栈总数。
_klog_get_task_stack_usage	获取堆栈大小和指定任务使用的堆栈总数。
_klog_show_stack_usage	计算和显示每个任务使用的堆栈数量和中断堆栈。

3.11 实用程序

实用程序包括：

- 队列
- 名称组件
- 运行时测试
- 其他实用程序

3.11.1 队列

队列组件可以管理元素的双链表。

附注	为了在某些目标平台上优化代码和内存要求，队列组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其它内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

表 3-57. 汇总：使用队列

_queue_dequeue	删除位于队列开始处的元素。
_queue_enqueue	向队列末尾添加元素。
_queue_get_size	获取队列中的元素数量。
_queue_head	获取（但不删除）位于队列开始处的元素。
_queue_init	初始化队列。
_queue_insert	在队列中插入元素。
_queue_is_empty	确定队列是否为空。
_queue_next	获取（但不删除）队列中的下一个元素。
_queue_test	测试队列。
_queue_unlink	从队列中删除指定元素。

3.11.1.1 队列数据结构

队列组件需要两种数据结构，在 *mqx.h* 中进行定义：

- **QUEUE_STRUCT**—保持跟踪队列大小和指向队列首尾的指针。MQX RTOS 在任务创建队列时初始化该结构。
- **QUEUE_ELEMENT_STRUCT**—定义队列元素的结构。该结构是应用程序定义的对象的头结构，该对象是任务要加入队列的元素。

3.11.1.2 创建队列

任务通过**_queue_init()**函数来创建和初始化队列，该函数的参数分别是指向队列对象的指针和最大队列大小。

3.11.1.3 添加元素至队列

任务通过调用**_queue_enqueue()**添加元素至队列末尾，其中的指针指向队列和队列元素对象（任务要排队的对象的头结构）。

3.11.1.4 从队列中删除元素

让任务通过调用**_queue_dequeue()**函数从队列的开始处获取并删除一个元素，该函数的参数是指向队列的指针。

3.11.2 名称组件

通过名称组件，任务可以将 32 位数字与字符串或符号名称相关联。MQX RTOS 在名称数据库中存放关联，处理器上的所有任务均可使用该关联信息。数据库避免使用全局变量。

附注	为了在某些目标平台上优化代码和数据存储器要求，名称组件默认不会编译到 MQX 内核中。为了测试该功能，您需要先在 MQX 用户配置文件中启用该选项，然后重新编译 MQX PSP、BSP 和其他内核组件。欲了解更多详细信息，请参见重新编译 Freescale MQX RTOS 。

表 3-58. 汇总：使用名称组件

名称组件使用某些在 *name.h* 中定义的结构和常数。	名称组件使用某些在 *name.h* 中定义的结构和常数。
_name_add	向名称数据库中添加一个名称（名称是以 NULL 结尾的字符串，最长 32 个字符，包括 NULL）。
_name_create_component	创建名称组件。
_name_delete	从名称数据库中删除一个名称。
_name_find	在名称数据库中查找一个名称并获取其编号。
_name_find_by_number	在名称数据库中查找一个编号并获取其名称。
_name_test	测试名称组件。

3.11.2.1 创建名称组件

您可以使用**_name_create_component()**显式地创建名称组件。如果您不显式地创建该组件，MQX RTOS 会使用应用程序首次使用名称数据库时的默认值创建。

参数及默认值与事件组件的相同，如创建事件组件 页所述。

3.11.3 运行时测试

MQX RTOS 提供内核运行时测试，用于测试大部分 MQX 组件的完整性。

测试决定与组件相关的数据是否有效并没有被破坏。如果数据结构的 **VALID** 字段是已知值，则 MQX RTOS 将认定该数据结构是有效的。如果数据结构的 **CHECKSUM** 字段不正确或指针不正确，则 MQX RTOS 将认定该数据结构被破坏。

应用程序可以在正常工作时使用运行时测试。

表 3-59. 汇总：运行时测试

_event_test	事件
_log_test	日志
_lwevent_test	轻量级事件
_lwlog_test	轻量级日志
_lwmem_test	块大小可变的轻量级存储器
_lwsem_test	轻量级信号量
_lwtimer_test	轻量级定时器
_mem_test	块大小可变的存储器
_msgpool_test	消息池
_msgq_test	消息队列
_mutex_test	互斥

下一页继续介绍此表...

表 3-59. 汇总：运行时测试 (继续)

_name_test	名称组件
_partition_test	块大小固定的存储器（分区）
_queue_test	应用程序队列
_sem_test	信号量
_taskq_test	任务队列
_timer_test	定时器
_watchdog_test	看门狗

3.11.3.1 示例：执行运行时测试

应用程序使用所有 MQX 组件。低优先级任务测试所有组件。如果发现错误，则停止应用程序。

```
/* test.c */
#include <mqx.h>
#include <fio.h>
#include <event.h>
#include <log.h>
#include <lwevent.h>
#include <lwlog.h>
#include <lwmem.h>
#include <lwtimer.h>
#include <message.h>
#include <mutex.h>
#include <name.h>
#include <part.h>
#include <sem.h>
#include <timer.h>
#include <watchdog.h>
extern void background_test_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index,Function,           Stack,Prio,Name,  Attributes,          Param,Time Slice */
 { 10       , background_test_task,2000, 8,   "Main",MQX_AUTO_START_TASK,0,    0},
 { 0 }
};
/*TASK*-------------------------------------------------------------
*
* Task Name : background_test_task
* Comments  :
*   This task is meant to run in the background testing for
* integrity of MQX component data structures.
*END*-------------------------------------------------------------*/
void background_test_task
   (
      uint32_t  parameter
   )
{
   _partition_id  partition;
   _lwmem_pool_id lwmem_pool_id;
   void *        error_ptr;
   void *        error2_ptr;
   _mqx_uint      error;
   _mqx_uint      result;
   while (TRUE) {
      result = _event_test(&error_ptr);
```

```
if (result != MQX_OK){
   printf("\nFailed _event_test: 0x%X.", result);
   _mqx_exit(1);
}
result = _log_test(&error);
if (result != MQX_OK){
   printf("\nFailed _log_test: 0x%X.", result);
   _mqx_exit(2);
}
result = _lwevent_test(&error_ptr, &error2_ptr);
if (result != MQX_OK){
   printf("\nFailed _lwevent_test: 0x%X.", result);
   _mqx_exit(3);
}
result = _lwlog_test(&error);
if (result != MQX_OK){
   printf("\nFailed _lwlog_test: 0x%X.", result);
   _mqx_exit(4);
}
result = _lwsem_test(&error_ptr, &error2_ptr);
if (result != MQX_OK){
   printf("\nFailed _lwsem_test: 0x%X.", result);
   _mqx_exit(5);
}
result = _lwmem_test(&lwmem_pool_id, &error_ptr);
if (result != MQX_OK){
   printf("\nFailed _lwmem_test: 0x%X.", result);
   _mqx_exit(6);
}
result = _lwtimer_test(&error_ptr, &error2_ptr);
if (result != MQX_OK){
   printf("\nFailed _lwtimer_test: 0x%X.", result);
   _mqx_exit(7);
}
result = _mem_test_all(&error_ptr);
if (result != MQX_OK){
   printf("\nFailed _mem_test_all,");
   printf("\nError = 0x%X, pool = 0x%X.", result,
      (_mqx_uint)error_ptr);
   _mqx_exit(8);
}
/*
** Create the message component.
** Verify the integrity of message pools and message queues.
*/
if (_msg_create_component() != MQX_OK){
   printf("\nError creating the message component.");
   _mqx_exit(9);
}
if (_msgpool_test(&error_ptr, &error2_ptr) != MQX_OK){
   printf("\nFailed _msgpool_test.");
   _mqx_exit(10);
}
if (_msgq_test(&error_ptr, &error2_ptr) != MQX_OK){
   printf("\nFailed _msgq_test.");
   _mqx_exit(11);
}
if (_mutex_test(&error_ptr) != MQX_OK){
   printf("\nFailed _mutex_test.");
   _mqx_exit(12);
}
if (_name_test(&error_ptr, &error2_ptr) != MQX_OK){
   printf("\nFailed _name_test.");
   _mqx_exit(13);
}
if (_partition_test(&partition, &error_ptr, &error2_ptr)
   != MQX_OK)
{
   printf("\nFailed _partition_test.");
   _mqx_exit(14);
```

```
      }
      if (_sem_test(&error_ptr) != MQX_OK){
         printf("\nFailed _sem_test.");
         _mqx_exit(15);
      }
      if (_taskq_test(&error_ptr, &error2_ptr) != MQX_OK){
         printf("\nFailed _takq_test.");
         _mqx_exit(16);
      }
      if (_timer_test(&error_ptr) != MQX_OK){
         printf("\nFailed _timer_test.");
         _mqx_exit(17);
      }
      if (_watchdog_test(&error_ptr, &error2_ptr) != MQX_OK){
         printf("\nFailed _watchlog_test.");
         _mqx_exit(18);
      }
      printf("All tests passed.");
      _mqx_exit(0);
   }
}
```

3.11.3.1.1 编译应用程序并将其与 MQX RTOS 链接

1. 转至此目录:

   ```
   mqx\examples\test
   ```

2. 请参阅《*MQX™ RTOS* 版本注释》文档 获取有关如何构建和运行该应用程序的说明。
3. 按照《*MQX™ RTOS* 版本注释》文档 中的说明运行该应用程序

3.11.4 其他实用程序

表 **3-60.** 汇总：其他实用程序

_mqx_bsp_revision	BSP 修订。
_mqx_copyright	MQX 版权字符串的指针。
_mqx_date	指示 MQX RTOS 构建时间的字符串的指针。
_mqx_fatal_error	指示检测到导致 MQX RTOS 或应用程序无法再正常运行的严重错误。
_mqx_generic_revision	通用 MQX 代码的修订。
_mqx_get_counter	获取处理器的唯一 32 位编号。
_mqx_get_cpu_type	获取处理器类型。
_mqx_get_exit_handler	获取 MQX 退出处理程序的指针，MQX RTOS 在退出时调用。
_mqx_get_kernel_data	获取内核数据的指针。
_mqx_get_system_task_id	获取系统任务描述符的任务 ID。
_mqx_get_tad_data	获取任务描述符的 **TAD_RESERVED** 字段。
_mqx_idle_task	空闲任务。
_mqx_io_revision	BSP 的 I/O 修订。
_mqx_monitor_type	监示器类型。
_mqx_psp_revision	PSP 修订。
_mqx_set_cpu_type	设置处理器类型。

下一页继续介绍此表...

表 3-60. 汇总：其他实用程序 (继续)

_mqx_set_exit_handler	设置 MQX 退出处理程序的地址，MQX RTOS 在退出时调用。
_mqx_set_tad_data	设置任务描述符的 **TAD_RESERVED** 字段。
_mqx_version	指示 MQX RTOS 版本的字符串的指针。
_mqx_zero_tick_struct	使用常量 0 初始化的滴答结构， 应用程序可用于将其滴答结构之一初始化为 0。
_str_mqx_uint_to_hex_string	将**_mqx_uint** 值转换为十六进制字符串。
_strnlen	计算有限长度字符串的长度。

3.12 用户模式任务和存储器保护

从 MQX RTOS 3.8 开始支持存储器保护单元(MPU), 该模块集成在选定的 Freescale Kinetis 微处理器器件中。MPU 能够阻止访问和保护最多 16 个存储器区域，阻止代码运行在所谓的“用户模式”下。当软件运行于所谓的“特权”或“管理”模式时，会处理存储器保护和所有其他特殊内核操作（包括中断服务）的设置。

在早期的 MQX RTOS 版本（MQX RTOS 3.7 及以前）中，所有代码总是运行在特权模式下，并且可以访问存储器的任何部分，没有限制。这对于具有高级存储器管理单元（MMU）的器件也适用（现在仍然适用)。MMU 与 MPU 不同，MMU 不仅实现了存储器保护，还提供虚拟内存翻译和为存储器的不同部分设置不同的缓存等功能。在这样的器件上，MQX RTOS 仅支持 MMU 的缓存控制。

MPU 和用户模式任务首先在用于 Kinetis K60 器件的 MQX RTOS 3.8 版本上引入，后来扩展到 MQX API。在 MQX 配置头文件中启用了用户模式支持后, BSP 启动代码将启用 MPU，并为主要 RAM 区域设置为只读模式。保护涵盖了内核拥有的变量、默认存储器分配池和 MQX 调度程序正常工作所需的所有其他数据结构。

用户可以在任务模板列表中将任务声明为“用户任务”。该用户任务运行在受限 CPU 模式下，并且所有 MPU 保护均有效。任务不会破坏内核存储器或影响运行在特权模式下的任务。不过它可以影响其他用户模式任务。如果用户任务试图破坏保护，将产生一个异常并根据系统的配置进行处理。

用户任务可能使用的 MQX API 将受限。通常而言，用户模式任务仅支持轻量级同步对象、轻量级存储器管理和有限的任务创建。

后续章节对用户模式支持的内容进行了详细说明。《MQX 参考手册》提供了所有 API 函数的参考信息。

3.12.1 配置用户模式支持

用户模式支持通过在 user_config.h 文件中将 **MQX_ENABLE_USER_MODE** 定义为 1 启用。默认情况下，此宏定义为 0，用户模式支持处于禁用状态。

当用户模式启用时，可以定义另一配置选项:

- 可以将 **MQX_DEFAULT_USER_ACCESS_RW** 设置为零或非零，以禁用或启用对未显式定义访问模式的全局变量的用户模式访问。请参阅下面有关变量访问的更多详细信息。
- **MQX_ENABLE_USER_STDAPI** 可设置为非零, 以便在用户模式任务中模拟标准 API。禁用时，用户任务可以显式调用_usr_前缀的 API（如 _usr_lwsem_post)。启用此选项时，用户模式任务可以调用标准 API（例如 _lwsem_post)，系统则负责将该调用转发至适当的_usr_ API 函数。

3.12.2 MQX 初始化结构

启用用户模式支持时，**MQX_INITIALIZATION_STRUCT** 扩展为包含额外的运行时配置参数，用于设定 MPU 和用户模式行为。在典型情况下，大部分数值由链接器文件提供, 定义了所有特权和用户任务所需的所有存储器段和 RAM 区域的定义。

MQX 初始化结构中将添加如下数据成员:

- START_OF_KERNEL_AREA 和 END_OF_KERNEL_AREA: 用于用户模式任务的受限访问区域。其覆盖了 KERNEL_DATA 结构、默认存储器堆和其他特权 MQX 结构及数据，包括内核所有的全局变量。
- START_OF_USER_DEFAULT_MEMORY 和 END_OF_USER_DEFAULT_MEMORY: 默认数据部分（.data 用于初始化的全局变量，.bss 用于未初始化的归零全局变量)。
- START_OF_USER_HEAP 和 END_OF_USER_HEAP: 用户堆—在用户模式中用于动态存储器分配的区域。
- START_OF_USER_RW_MEMORY 和 END_OF_USER_RW_MEMORY: 全局变量在用户模式中显式声明读写访问权限的区域。
- START_OF_USER_RO_MEMORY 和 END_OF_USER_RO_MEMORY: 全局变量在用户模式中显式声明只读访问权限的区域。
- START_OF_USER_NO_MEMORY 和 END_OF_USER_NO_MEMORY: 全局变量在用户模式中显式声明无任何访问权限的区域。
- MAX_USER_TASK_PRIORITY: 用于用户任务优先级的有限值—用户任务只能在相同或更低优先级下运行（从数字上来说，这是用户任务可用作优先级的最小数字)。
- MAX_USER_TASK_COUNT: 可在系统中创建的最大用户任务数。

3.12.2.1 初始化默认值

表 3-61. MQX 初始化默认值

MQX 初始化结构成员	BSP 默认宏常量	LINKER 文件符号（IAR EWARM 工具的示例）
START_OF_KERNEL_AREA	**BSP_DEFAULT_START_OF_KERNEL_AREA**	__KERNEL_DATA_START
END_OF_KERNEL_AREA	**BSP_DEFAULT_END_OF_KERNEL_AREA**	__KERNEL_DATA_END
START_OF_USER_DEFAULT_MEMORY	BSP_DEFAULT_START_OF_USER_DEFAULT_MEMORY	__sfb("USER_DEFAULT_MEMORY")
END_OF_USER_DEFAULT_MEMORY	BSP_DEFAULT_END_OF_USER_DEFAULT_MEMORY	__sfe("USER_DEFAULT_MEMORY")
START_OF_USER_HEAP	BSP_DEFAULT_START_OF_USER_HEAP	__sfb("USER_HEAP")
END_OF_USER_HEAP	BSP_DEFAULT_END_OF_USER_HEAP	__USER_AREA_END
START_OF_USER_RW_MEMORY	BSP_DEFAULT_START_OF_USER_RW_MEMORY	__sfb("USER_RW_MEMORY")
END_OF_USER_RW_MEMORY	BSP_DEFAULT_END_OF_USER_RW_MEMORY	__sfe("USER_RW_MEMORY")
START_OF_USER_RO_MEMORY	BSP_DEFAULT_START_OF_USER_RO_MEMORY	__sfb("USER_RO_MEMORY")
END_OF_USER_RO_MEMORY	BSP_DEFAULT_END_OF_USER_RO_MEMORY	__sfe("USER_RO_MEMORY")
START_OF_USER_NO_MEMORY	BSP_DEFAULT_START_OF_USER_NO_MEMORY	__sfb("USER_NO_MEMORY")
END_OF_USER_NO_MEMORY	BSP_DEFAULT_END_OF_USER_NO_MEMORY	__sfe("USER_NO_MEMORY")
MAX_USER_TASK_PRIORITY	BSP_DEFAULT_MAX_USER_TASK_PRIORITY	不适用
MAX_USER_TASK_COUNT	BSP_DEFAULT_MAX_USER_TASK_COUNT	不适用

3.12.3 声明和创建用户模式任务

用户模式任务通过 MQX 任务模板列表中的 **MQX_USER_TASK** 标志定义。根据内核配置，您可以将此标志与 MQX_AUTO_START_TASK、MQX_FLOATING_POINT_TASK、MQX_TIME_SLICE_TASK 及其它标准任务标志混合使用。

应用程序通过使用来自特权任务的_task_create API 以标准方式或从另一用户任务调用_usr_task_create 来创建用户任务。只能通过特权任务创建特权任务。

如上所述，影响用户模式任务创建的 MQX 初始化结构有两个成员：

- **MAX_USER_TASK_PRIORITY**：用户任务优先级的限制值。
- **MAX_USER_TASK_COUNT**：可在系统中创建的最大用户任务数。

3.12.4 全局变量的访问权限

全局变量的用户模式访问可以使用如下声明的修饰符显式定义:

- **USER_RW_ACCESS** - 变量通常从用户模式任务访问。
- **USER_RO_ACCESS** - 变量对用户模式任务为只读。
- **USER_NO_ACCESS** - 变量不可通过用户模式任务访问。

例如:

```
USER_RO_ACCESS int counter;  /* read-only for user-mode task */
USER_NO_ACCESS char state;   /* not accessible for user-mode task */
```

对未显式声明的变量（默认的.data 和.bss 段）的访问权限由 **MQX_DEFAULT_USER_ACCESS_RW** 配置选项决定。如果未定义或定义为 0，则全局变量声明为用户模式任务只读。当配置选项设置为非 0 时，全局变量将具有读写权限。

3.12.5 API

本节概述同样适用于用户模式任务的 API 子集。API 可通过_usr_前缀轻松识别。请注意，仅有在 MQX 配置中启用了用户模式的情况下，才会声明 API 函数原型。

表 3-62. 用户模式 API 概述

USERMODE 函数	PRIVILEGE 原函数
_usr_lwsem_poll	_lwsem_poll
_usr_lwsem_post	_lwsem_post
_usr_lwsem_wait	_lwsem_wait
_usr_lwsem_create	_lwsem_create
_usr_lwsem_wait_for	_lwsem_wait_for
_usr_lwsem_wait_ticks	_lwsem_wait_ticks
_usr_lwsem_wait_until	_lwsem_wait_until
_usr_lwsem_destroy	_lwsem_destroy
_usr_lwevent_clear	_lwevent_clear
_usr_lwevent_set	_lwevent_set
_usr_lwevent_set_auto_clear	_lwevent_set_auto_clear
_usr_lwevent_wait_for	_lwevent_wait_for
_usr_lwevent_wait_ticks	_lwevent_wait_ticks
_usr_lwevent_wait_until	_lwevent_wait_until
_usr_lwevent_get_signalled	_lwevent_get_signalled
_usr_lwevent_create	_lwevent_create

下一页继续介绍此表...

表 3-62. 用户模式 API 概述 (继续)

_usr_lwevent_destroy	_lwevent_destroy
_usr_task_create	_task_create
_usr_task_destroy	_task_destroy
_usr_task_abort	_task_abort
_usr_task_ready	_task_ready
_usr_task_set_error	_task_set_error
_usr_task_get_td	**_task_get_td**
_usr_lwmem_alloc	**_lwmem_alloc**
_usr_lwmem_alloc_from	**_lwmem_alloc_from**
_usr_lwmem_free	**_lwmem_free**
_usr_lwmem_create_pool	**_lwmem_create_pool**
_mem_set_pool_access	不适用
_usr_time_delay_ticks	**_time_delay_ticks**
_usr_time_get_elapsed_ticks	**_time_get_elapsed_ticks**
_usr_lwmsgq_init	**_lwmsgq_init**
_usr_lwmsgq_receive	**_lwmsgq_receive**
_usr_lwmsgq_send	**_lwmsgq_send**

3.12.6 在用户模式中处理中断

MQX RTOS 不支持在用户模式中处理中断, 但是这可以通过轻量级信号量或事件功能轻易实现。中断服务例程 (运行在特权模式下) 可能确定或只是禁止该中断源, 并将信号量或事件传递给应用程序任务。等待事件的任务 (用户模式任务或任务) 激活后, 可以完成中断处理并重新使能中断源。

请注意, Freescale Kinetis 平台允许用户模式访问系统配置桥中选定的外设寄存器。可以使用该桥将用户模式保护扩展到外设模块。

3.13 嵌入式调试

有几种方式可调试基于 MQX RTOS 的应用程序:

- 使用普通调试程序环境, 不考虑 MQX 操作系统。在整个应用程序代码中使用断点和单步执行时, 这种简单的方法可能很有用。
- 在调试程序中使用操作系统感知 (所谓的任务感知调试程序或 TAD)。这种方法有助于在单个任务的上下文中查看经过调试的代码。也有助于以用户友好的方式检查内部 MQX 数据结构。

3.14 在编译时配置 MQX RTOS

您可以通过更改编译时配置选项的值包含或排除 MQX RTOS 附带的某些功能。如果您更改任何配置值，则必须重新编译 MQX RTOS 并将它与目标应用程序重新链接。

由于板级支持包 (BSP) 库也可能依赖于某些 MQX RTOS 配置选项，通常也必须重新编译。

与 BSP 一样，还有使用 MQX OS 服务（如 RTCS、MFS、USB）的其他代码组件。这些组件需要在 MQX RTOS 和 BSP 之后重新编译。

附注	与原 ARC 版本相比，Freescale MQX RTOS 引入了一种不同的 MQX OS 和其他组件的编译时配置方法。 原方法使用编译器命令行-D 选项或 *source\psp\platform\psp_cnfg.asm* 文件。 Freescale MQX RTOS 包含一个中央用户配置文件 *user_config.h*（位于 *config\<board>*目录），可用于覆盖默认配置选项。其他系统组件（如 RTCS、MFS 或 USB）也使用这一配置文件。

3.14.1 MQX RTOS 编译时配置选项

本章节提供了 MQX RTOS 配置选项列表。所有这些选项的默认值均可在 *config\<board>\user_config.h* 文件中重写。

默认值在 *mqx\source\include\mqx_cnfg.h* 文件中进行定义。

附注	不要直接修改 *mqx_cnfg.h* 文件。应总是使用 *config* 目录中的特定电路板或特定项目的 *user_config.h* 文件进行修改。

MQX_CHECK_ERRORS

默认为 1。

1：MQX RTOS 组件对所有参数执行错误检查。

0：MQX RTOS 组件不执行参数检查。不会返回针对特定函数列出的所有错误代码。

MQX_CHECK_MEMORY_ALLOCATION_ERRORS

默认为 1。

1：MQX RTOS 组件检查所有错误的内存分配，并检查分配是否成功。

MQX_CHECK_VALIDITY

默认为 1。

1：MQX RTOS 在访问所有结构时检查 **VALID** 字段。

MQX_COMPONENT_DESTRUCTION

默认为 1。

1：MQX RTOS 包含允许撤销 MQX RTOS 组件（如信号量组件或事件组件）的函数。MQX RTOS 收回分配给组件的所有资源。

MQX_DEFAULT_TIME_SLICE_IN_TICKS

默认为 1。

1：任务模板结构中的默认时间片以滴答为单位。

0：任务模板结构中的默认时间片以毫秒为单位。

该值还会影响任务模板中的时间片字段，因为该值用于设置任务的默认时间片。

MQX_EXIT_ENABLED

默认为 1。

1：MQX RTOS 包含允许应用程序调用**_mqx()**函数时返回的代码。

MQX_HAS_TIME_SLICE

默认为 1。

1：MQX RTOS 包含允许时间片调度处于相同优先级任务的代码。

MQX_HAS_DYNAMIC_PRIORITIES

默认为 1。

1：MQX RTOS 包含动态修改任务优先级的代码，通过调用**_task_set_priority()**函数、优先级继承或优先级升级。

MQX_HAS_EXCEPTION_HANDLER

默认为 1。

1：MQX RTOS 包含处理异常的代码（见 psp/<psp>/int_xcpt.c）和调用 **_task_set_exception_handler** 和**_task_get_exception_handler** 函数设置/获取任务异常句柄例程的代码。

MQX_HAS_EXIT_HANDLER

默认为 1。

1：MQX RTOS 包含在任务退出前执行任务退出句柄的代码。还包括 **_task_set_exit_handler** 和**_task_get_exit_handler** 函数调用。

MQX_HAS_HW_TICKS

默认为 1。

1：MQX RTOS 支持硬件滴答及相关调用：**_time_get_hwticks**、**_time_get_hwticks_per_tick** 和**_psp_usecs_to_ticks**。请注意，BSP 也需要支持硬件滴答。

MQX_HAS_TASK_ENVIRONMENT

默认为 1。

1：MQX RTOS 包含设置和获取任务环境数据指针的代码：**_task_set_environment** 和**_task_get_environment**。

MQX_HAS_TICK

默认为 1。建议保留该选项为使能。

1：MQX RTOS 包含支持滴答时间和延时任务及等待具有超时的同步对象等的所有相关功能。

MQX_KD_HAS_COUNTER

默认为 1。

1：MQX RTOS 内核维护计数器值，通过调用**_mqx_get_counter** 函数在询问时自动递增数值。

MQX_TD_HAS_PARENT

默认为 1。

1：MQX RTOS 任务描述符维护任务的创建者 ID，这通过调用**_task_get_creator** 函数获取。

MQX_TD_HAS_TEMPLATE_INDEX

默认为 1。

1: MQX RTOS 任务描述符维护来自 TASK_TEMPLATE_ STRUCT 数组的初始索引值。该值仅用于维护向后兼容性，MQX RTOS 内核不会使用。

MQX_TD_HAS_TASK_TEMPLATE_PTR

默认为 1。

1：MQX RTOS 任务描述符维护用于任务创建的初始 TASK_TEMPLATE_STRUCT 结构的指针。该指针用于任务重启动调用**_task_restart()**函数和诸如**_task_get_id_from_name()**的多个查找函数。

MQX_TD_HAS_ERROR_CODE

默认为 1。

1：MQX RTOS 任务描述符维护通过调用**_task_set_error** 和**_task_get_error** 函数可访问的错误代码。

MQX_TD_HAS_STACK_LIMIT

默认为 1。

1：MQX RTOS 任务描述符维护任务限值，用于诸如**_task_check_stack** 函数的各种堆栈溢出检查调用。

MQX_INCLUDE_FLOATING_POINT_IO

默认为 0。

1：**_io_printf()**和**_io_scanf()**函数包含浮点 I/O 代码。

MQX_IS_MULTI_PROCESSOR

默认为 1。

1：MQX RTOS 包含支持多处理器 MQX RTOS 应用程序的代码。

MQX_KERNEL_LOGGING

默认为 1。

1: 在进入和退出各组件中的某些函数时，这些函数会写入内核日志。只有在为组件启用日志记录的情况下，该设置才会降低性能。可以通过**_klog_control()**函数来控制为哪个组件使用日志记录。

MQX_LWLOG_TIME_STAMP_IN_TICKS

默认为 1。

1：轻量级日志中的时间戳以滴答为单位。

0：时间戳以秒、毫秒和微秒为单位。

MQX_MEMORY_FREE_LIST_SORTED

默认为 1。

1：MQX RTOS 按地址排序内存区的释放列表。这可以减少存储器碎片，但是会延长 MQX RTOS 释放存储器的时间。

MQX_MONITOR_STACK

默认为 1。

1：MQX RTOS 将所有任务和中断堆栈初始化为已知值，以便 MQX RTOS 组件和调试程序计算出已占用的堆栈大小。只有当 MQX RTOS 创建任务时，该设定才会降低性能。

必须将选项设为 1 才能使用。

- **_klog_get_interrupt_stack_usage()**
- **_klog_get_task_stack_usage()**
- **_klog_show_stack_usage()**

MQX_MUTEX_HAS_POLLING

默认为 1。

1：MQX RTOS 包含支持互斥选项 **MUTEX_SPIN_ONLY** 和 **MUTEX_LIMITED_SPIN** 的代码。

MQX_PROFILING_ENABLE

默认为 0。

1：支持外部分析工具的代码编译到 MQX RTOS 中。分析会增加编译映像的大小，而且会导致 MQX RTOS 运行缓慢。仅在您使用的工具集支持分析时才能使用。

MQX_RUN_TIME_ERR_CHECK_ENABLE

默认为 0。

1：支持外部运行时错误检查工具的代码编译到 MQX RTOS 中。这会增加编译映像的大小，而且会导致 MQX RTOS 运行缓慢。仅在您使用的工具集支持运行时错误检查时才能使用。

MQX_ROM_VECTORS

默认为 0。

1：中断向量表不会复制到 RAM 中。正确设置基于 ROM 的表，以便由默认 MQX RTOS 中断分配器处理所有中断。应用程序仍然可以通过**_int_install_isr** 调用来装载中断服务程序。然而，**_int_install_kernel_isr** 调用不能用于将低级中断服务程序直接装载到向量表中。

MQX_SPARSE_ISR_TABLE

默认为 0。

1：将 MQX RTOS 中断服务程序表分配为“链表数组”，而非线性数组。该选项独立于 MQX_ROM_VECTORS，它通过 MQX RTOS 中由中断分配器管理的“逻辑”表处理。通过该稀疏 ISR 表，只有由**_int_install_isr** 调用装载的 ISR 才会占用 RAM 存储器。这会增加中断延时，因为 MQX RTOS 需要遍历列表以找到被调用的用户 ISR。

MQX_SPARSE_ISR_SHIFT

默认为 3。

当 MQX_SPARSE_ISR_TABLE 定义为 1 时，MQX_SPARSE_ISR_SHIFT 选项确定向量编号移动的位数，以获取 ISR 链表的索引。例如，潜在中断源为 256 个且位移值为 3，则产生 256>>3=32 个列表，每个列表的最大深度为 8 个 ISR 条目。如果位移值为 8，则会产生一个包含所有 ISR 条目的大型链表。

MQX_TASK_CREATION_BLOCKS

默认为 1。该选项仅适用于多处理器应用程序。

1：当任务调用**_task_create()**函数在另一处理器上创建任务时，该任务会阻塞。执行创建的任务将被阻塞，直至新任务完成创建并返回错误代码为止。

MQX_TASK_DESTRUCTION

默认为 1。

1：MQX RTOS 允许终止任务。从而，MQX RTOS 包含释放所有 MQX RTOS 管理的资源以终止任务本身的代码。

MQX_TIMER_USES_TICKS_ONLY

默认为 0。

1：定时器任务处理周期性定时器和一次性定时器要求，使用滴答时间而非秒/毫秒时间作为超时格式。

MQX_USE_32BIT_MESSAGE_QIDS

默认为 0。

0：消息组件数据类型（**_queue_number** 和**_queue_id**）为 **uint16_t**。

1：消息组件数据类型（**_queue_number** 和**_queue_id**）为 uint32_t。这允许处理器上具有超过 256 个消息队列，以及在多处理器网络中具有超过 256 个处理器。

MQX_USE_IDLE_TASK

默认为 1。

1: 内核将创建空闲任务，在没有其他任务就绪时执行，否则，没有任务运行时处理器会停止。

MQX_USE_INLINE_MACROS

默认为 1。

1: MQX RTOS 调用的某些内部函数在函数调用内嵌代码时会改变。该设置会优化 MQX RTOS 速度。

0: 优化 MQX RTOS 代码大小。

MQX_USE_IO

默认为 1。

1: MQX RTOS 设置 I/O 驱动程序所需的 I/O 子系统调用。没有 I/O 子系统，则无法装载或使用驱动程序，并且任务不能使用 stdin/stdout/stderr 句柄。

MQX_USE_LWMEM_ALLOCATOR

默认为 0。

1: 调用**_lwmem** 系列中的相应函数来取代对**_mem** 系列函数的调用。

MQXCFG_ENABLE_FP

默认值取决于 **MQXCFG_MEM_COPY_NEON**。如果 **MQXCFG_MEM_COPY_NEON** 已置位，则默认值为 1。否则，默认值为 0。

如果已置位，则在 MQX RTOS 中启用 FPU 支持。调度程序存储和恢复 FPU 上下文，并提供用于在任务和中断中支持浮点的 API。

MQX_SAVE_FP_ALWAYS

默认值取决于 **MQXCFG_MEM_COPY_NEON**。如果 **MQXCFG_MEM_COPY_NEON** 已置位，则默认值为 1。否则，默认值为 0。

在每个任务中使 **MQX_FLOATING_POINT_TASK** 置位。MQX RTOS 在调度程序中存储和恢复 FPU 上下文。FPU 上下文存储在中断序言中，并在中断结语处恢复。用户不能在运行时禁用 FPU 上下文存储。

MQX_INCLUDE_FLOATING_POINT_IO

默认值为 0。

在 MQX RTOS I/O 函数如 `printf` 和 `scanf` 中启用浮点类型，并且启用浮点转换 API。

MQXCFG_MEM_COPY

默认值为 0。

如果置位，则允许 MQX RTOS 拥有唯一的存储器副本。否则，会使用编译器库中的 memcpy。

MQXCFG_MEM_COPY_NEON

默认值为 0。

如果置位，则 MQX RTOS 使用具有 NEON 指令的特殊存储器副本设置。该功能需要 MQX RTOS 支持 FPU。选项 **MQXCFG_ENABLE_FP** 和 **MQX_SAVE_FP_ALWAYS** 置为 1。

3.14.2 推荐设置

您选择用于编译时配置选项的设置应由应用的需求决定。

附注	MQX 编译过程及其编译时配置特定于给定目标板（在 *config/<board>/user_config.h* 目录中设置）。 您可能希望根据定制电路板甚至是具体应用创建自己的配置。关于此过程的更多详情，请参见为什么要创建新配置？ 。

下表列出了在开发应用时可以使用的一些常用设置。

表 3-63. 编译时配置设置

选项	默认值	调试	速度	大小
MQX_ALLOW_TYPED_MEMORY	1	1	0	0，1
MQX_CHECK_ERRORS	1	1	0	0
MQX_CHECK_MEMORY_ALLOCATION_ ERRORS	1	1	0	0
MQX_CHECK_VALIDITY	1	1	0	0
MQX_COMPONENT_DESTRUCTION	1	0*，1	0*	0*
MQX_DEFAULT_TIME_SLICE_IN_TICKS	0	0，1	1	1
MQX_EXIT_ENABLED	1	0，1	0	0
MQX_HAS_DYNAMIC_PRIORITIES	1	0, 1	0	0
MQX_HAS_EXIT_HANDLER	1	0，1	0	0
MQX_HAS_TASK_ENVIRONMENT	1	0，1	0	0
MQX_HAS_TIME_SLICE	1	0, 1	0	0
MQX_INCLUDE_FLOATING_POINT_IO	0	0，1	0	0
MQX_IS_MULTI_PROCESSOR	1	0，1	0	0
MQX_KD_HAS_COUNTER	1	0，1	0，1	0
MQX_KERNEL_LOGGING	1	1	0	0
MQX_LWLOG_TIME_STAMP_IN_TICKS	1	0	1	1
MQX_MEMORY_FREE_LIST_SORTED	1	1	0	0
MQX_MONITOR_STACK	1	1	0	0
MQX_MUTEX_HAS_POLLING	1	0，1	0	0

下一页继续介绍此表...

表 **3-63.** 编译时配置设置 **(**继续**)**

MQX_PROFILING_ENABLE	0	1	0	0
MQX_ROM_VECTORS	0	0，1	0，1	1
MQX_RUN_TIME_ERR_CHECK_ENABLE	0	1	0	0
MQX_SPARSE_ISR_TABLE	0	0，1	0	1
MQX_SPARSE_ISR_SHIFT（范围 1-8）	3	任意值	更低	更高
MQX_TASK_CREATION_BLOCKS（用于多处理器应用）	1	1	0	0，1
MQX_TASK_DESTRUCTION	1	0，1	0	0
MQX_TD_HAS_ERROR_CODE	1	0，1	0	0
MQX_TD_HAS_PARENT	1	0，1	0	0
MQX_TD_HAS_STACK_LIMIT	1	0，1	0	0
MQX_TD_HAS_TASK_TEMPLATE_PTR	1	0，1	0	0
MQX_TD_HAS_TEMPLATE_INDEX	1	0，1	0	0
MQX_TIMER_USES_TICKS_ONLY	0	0，1	1	1
MQX_USE_32BIT_MESSAGE_QIDS	0	0，1	1	1
MQX_USE_IDLE_TASK	1	0，1	0，1	0
MQX_USE_INLINE_MACROS	1	0，1	1	0
MQX_USE_LWMEM_ALLOCATOR	0	0，1	1	1
MQX_VERIFY_KERNEL_DATA	1	1	0	0

第 4 章
重新编译 MQX RTOS

4.1 为什么要重新编译 MQX RTOS?

从版本 4.0 开始，MQX RTOS 发布版不再提供出厂预编译库。要开始使用 MQX RTOS，您需要先编译所有所需的 MQX 库。阅读本章节，以了解如何实现以及所需的步骤。

通常，当您进行以下操作时，需要编译或重新编译 MQX 库:

- 在新安装不带出厂预编译库的 MQX RTOS 程序包后。
- 如果改变了编译器选项（例如：优化等级）。
- 如果在 *config/<board>/user_config.h* 文件中改变了 MQX 编译时间配置选项。
- 如果开发了新的 BSP（例如：添加了新的 I/O 驱动程序）。
- 如果修改了 MQX 源代码。

4.2 开始之前

在编译 MQX RTOS 之前:

- 阅读 Freescale MQX RTOS 随附的《*Freescale MQX™ RTOS* 版本注释》，获取特定于您的目标环境的信息。
- 确保具备目标环境所需的工具:
- 编译器
- 汇编器
- 链接器
- 库管理器
- 熟悉 MQX 目录结构和重新编译指令，《*Freescale MQX™ RTOS* for Kinetis SDK 入门》文档中提供了相应说明，本章后续部分也介绍了这些指令。

附注	Freescale MQX RTOS 也可方便地通过支持的开发环境进行编译。

4.3 Freescale MQX RTOS 目录结构

下图显示了整个 Freescale MQX RTOS 的目录结构。

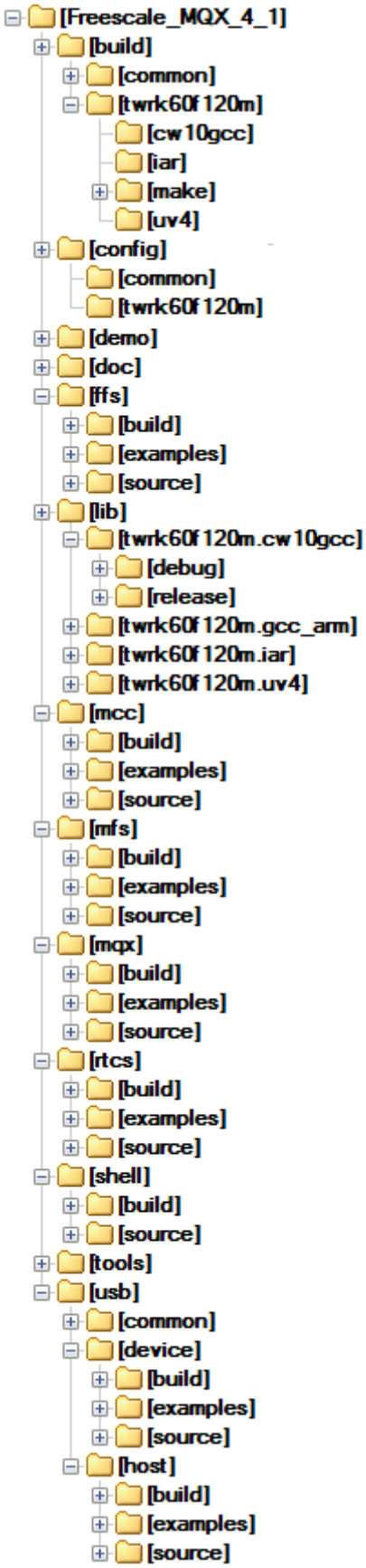

图 4-1. Freescale MQX RTOS 目录结构

4.3.1 MQX RTOS 目录结构

下图详细显示了位于 *mqx* 目录顶层中 MQX RTOS 组件的目录结构。

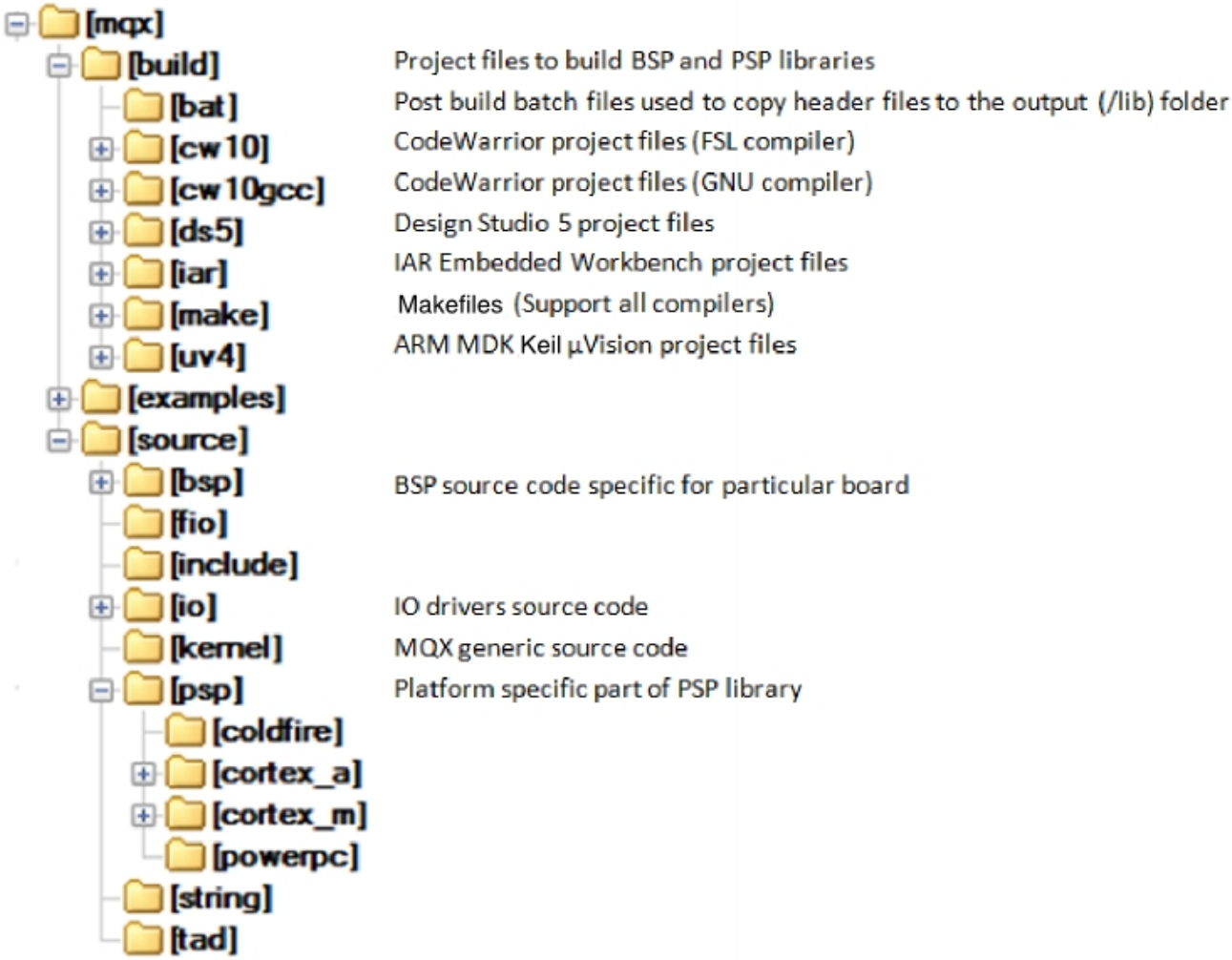

图 4-2. MQX RTOS 目录结构

4.3.2 PSP 子目录

mqx\source\psp\目录包含 PSP 库的独立于平台的代码。例如，ColdFire 子目录中包含专用于 Freescale ColdFire 架构的 MQX 内核部分（内核初始化、用于中断处理的寄存器保存/恢复代码和缓存控制函数等）。该目录还包含每个支持的处理器派生的处理器定义文件。

4.3.3 BSP 子目录

mqx\source\bsp 中的子目录通常根据电路板命名，其中包含低级启动代码、处理器和电路板初始化代码。BSP 还包含数据结构，用于以适合给定电路板的方式初始化各种 I/O 驱动程序。

该代码将会（与 I/O 驱动程序代码一起）编译为 BSP 库。

4.3.4 I/O 子目录

mqx\source\io 中的子目录包含用于 MQX I/O 驱动程序的源代码。通常，每个 I/O 驱动程序目录中的源文件都会进一步拆分为特定器件和独立于器件两部分。适用于给定开发板的 I/O 驱动程序是 BSP 编译项目的一部分，将会编译为 BSP 库。

4.3.5 其他源子目录

在所有其他的源目录中包含 MQX RTOS 的通用部分。通用源会与独立于平台的 PSP 代码一起编译为 PSP 库。

4.4 Freescale MQX RTOS 编译项目

所有必要的编译项目都位于 *mqx\build\<compiler>* 目录下。每种开发板都可以使用另种编译项目：PSP 和 BSP。BSP 项目包含特定于开发板的代码，而 PSP 仅特定于平台（例如 ColdFire）。PSP 项目不包含任何特定于开发板的代码。尽管如此，两种项目的文件名中均引用开发板名称，两者均生成二进制输出文件，并且两个文件将存储到同一个特定于开发板的目录 *lib\<board>.<compiler>*中。

独立于开发板的 PSP 库也会编译到特定于开发板的输出目录中，因为编译时配置文件来自指定开发板的目录 *config\<board>*。换言之，即便 PSP 源代码本身不依赖于开发板特性，但用户也可能希望为不同的开发板创建不同的 PSP。

4.4.1 PSP 编译项目

PSP 项目用于编译 PSP 库，其中包含 *mqx\source\psp* 中依赖于平台的部分，还包含通用 MQX RTOS 代码。

4.4.2 BSP 编译项目

BSP 项目用于编译 BSP 库，其中包含来自 *mqx\source\bsp\<board>*的指定开发板的代码，还有来自 *mqx\source\io* 目录中的选定 I/O 驱动程序。

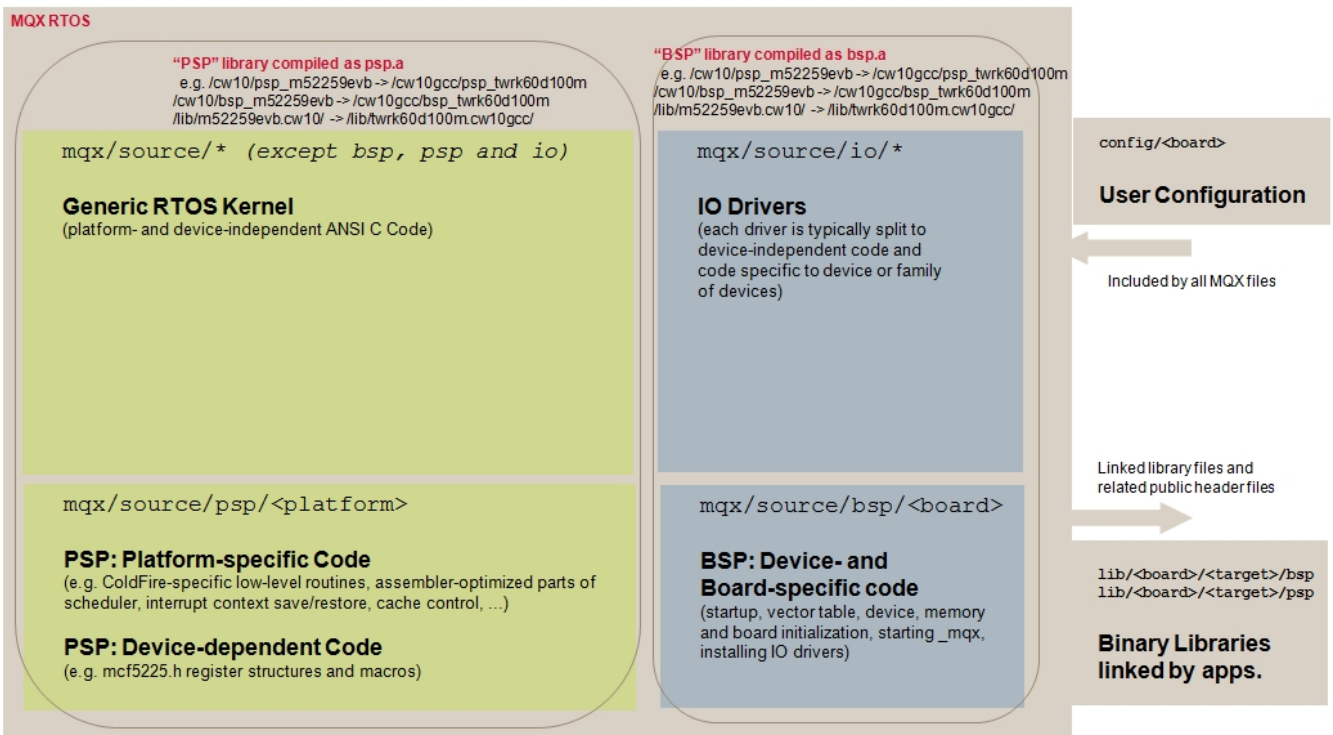

图 4-3. BSP 编译项目

4.4.3 编译后处理

所有编译项目配置为生成位于顶层 *lib\<board>.<compiler>*目录中的二进制库文件。

BSP 和 PSP 编译项目还可设定用于执行编译后的批处理文件，将所有公共头文件复制到目标 *lib* 目录中。这使得输出*/lib* 文件夹成为 MQX 应用程序代码可访问的唯一位置。MQX 应用程序编译项目完全不需要引用 MQX RTOS 源码树。

4.4.4 编译目标

所有受支持的开发环境均提供多种编译配置，称为编译目标。

- 调试目标—编译器优化设为低，以简化调试。将使用该目标编译的库复制到 *lib\<board>.<compiler>\debug* 目录下的相应文件夹中。
- 发布目标—编译器优化设为最高，以实现最小的代码大小和最快的执行速度。这样得到的代码很难进行调试。将使用该目标编译的库复制到 *lib\<board>.<compiler>\release* 目录下的相应文件夹中。

4.5 重新编译 Freescale MQX RTOS

重新编译 MQX RTOS 库非常简单，包括在开发环境中打开合适的 PSP 和 BSP 编译项目，并对其进行编译。不要忘记选择合适的编译目标进行编译或编译所有目标。

关于重新编译 MQX 的具体信息和相关示例，请参见 MQX 安装目录中的版本注释。

4.6 为什么要创建新配置?

当您需要创建一组新编译项目的典型场景包括:

- 您希望在一块开发板上同时在不同应用程序上使用两个或更多不同的内核配置。这只需要简单地“克隆”现有配置目录，并修改现有编译项目（修改名称和输出文件夹）。
- 您需要为指定开发板创建一个新的 BSP。这比较复杂，可能需要开发一些新的 I/O 驱动程序或修改高级配置。不过，可以从现有的最相似的 BSP 入手，对其克隆并改名，然后进行进一步修改。

4.7 克隆现有配置

如之前章节所述，PSP 和 BSP 编译项目（以及其他 MQX 内核组件的项目，如 RTCS、MFS 或 USB）与目标板名字联系在一起。以 TWR-K60D100M 板为例，以下项目由该名字决定:

- 用户配置来自 *config\<board>*目录（例如：*config\twrk60d100m*）。
- 编译项目包含的搜索路径设置为指向用户配置目录。
- 编译项目产生的结果二进制库文件位于 *lib\<board>.<compiler>\<target name>* 输出目录（例如：*lib\twrk60d100m.cw10gcc\debug*）。
- 编译项目命名以反映开发板名称 *mqx\build\<compiler>\bsp_<board>.<prj>*（例如：*mqx\build\cw10gcc\bsp_twrk60d100m\.project*）。
- 编译项目中的链接后批处理文件也根据开发板命名（例如：*mqx\build\bat\bsp_twrk60d100m.bat*）。

以 TWR-K60D100M 为例演示了使用 CodeWarrior 编译工具克隆（复制）现有配置并保存成其他名字的步骤:

- 复制现有 *config\twrk60d100m* 目录，分配新的特定板或特定配置的名称 (例如: *config\twrk60d100m_test*)。
- 在 *lib* 文件夹中创建新的输出目录（例如：*lib\twrk60d100m_test.cw10gcc*)。
- 创建 BSP 和 PSP 编译项目文件夹的副本（*mqx\build\cw10gcc\bsp_twrk60d100m* 文件夹和 *mqx\build\cw10gcc\bsp_twrk60d100m* 文件夹)。
- 打开项目设定，参考旧的配置目录修改包含搜索路径（即编辑 *config\twrk60d100m* 搜索路径为 *config\twrk60d100m_test*)。
- 在项目设定中，将输出目录修改为在 *lib* 中新创建的目录（从 *lib\twrk60d100m.cw10gcc* 改为 *lib\twrk60d100m_test.cw10gcc*)。
- 如果您还想克隆链接后批处理文件，则修改项目设定中的相应内容。(如果您的新 BSP 与驱动程序设定一致，则无需此步骤)。
- 确保您对所有可用编译目标（调试和发布）均完成了项目设定修改。
- 如果需要，对其他 MQX 库（如 RTCS、MFS 或 USB）重复上述步骤。

在新配置和编译项目就绪后，您可以开始修改编译时间配置，而不影响初始的 BSP 库。如果您希望创建一个完整的全新 BSP，需要创建新的 BSP 源文件并修改“克隆”BSP 项目的内容。#developing_a_new_bsp 中说明了新 BSP 的开发流程。

还可以通过 *BSP Cloning Wizard* 工具来克隆现有 BSP，该工具在 MQX 安装软件包中提供。BSP Cloning Wizard 提供创建 BSP 文件和项目副本（克隆）的简便途径。这对于自行准备基于支持 MQX RTOS 处理器开发板的用户而言非常有用。

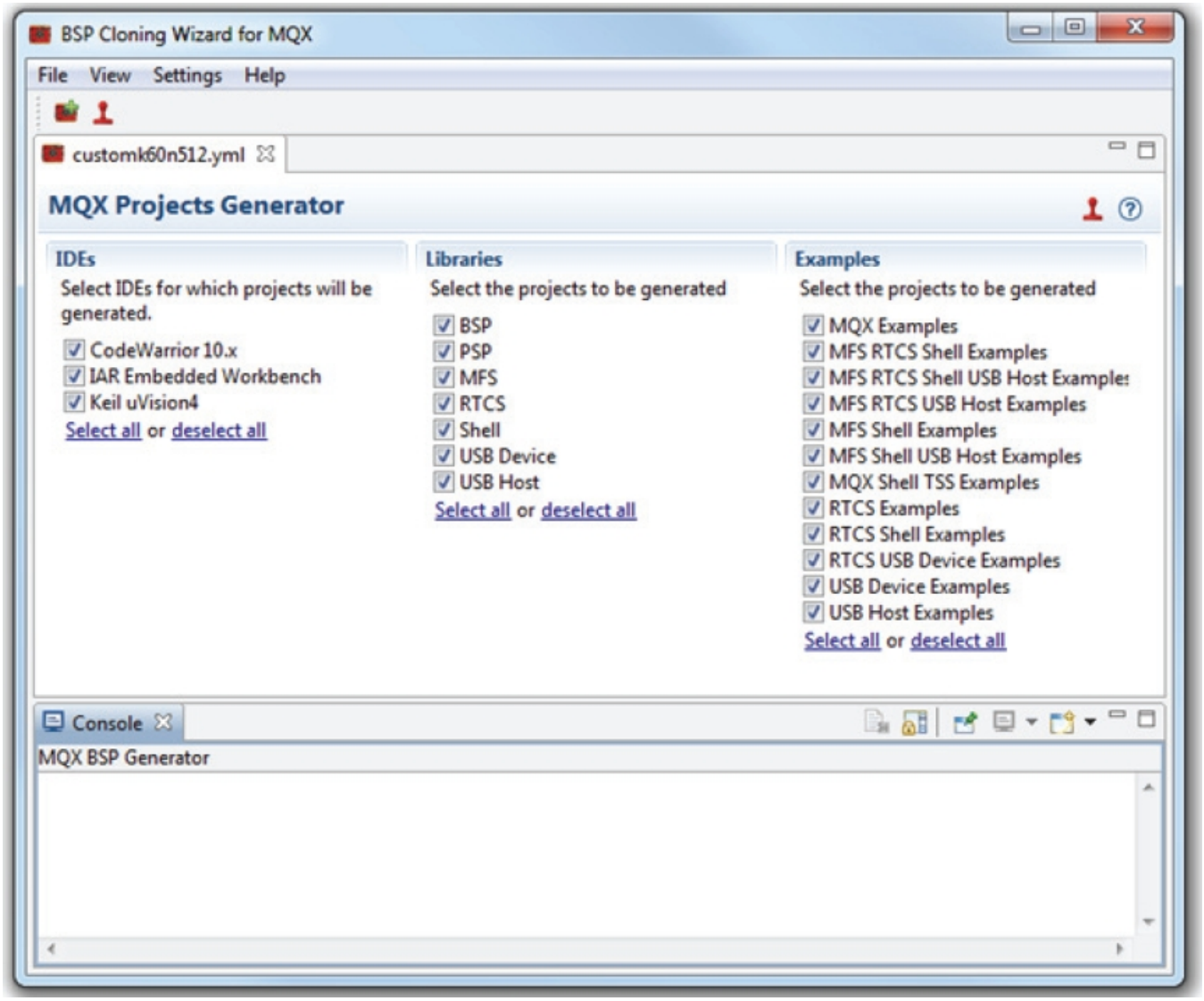

图 4-4. 用于 MQX RTOS 的 BSP Cloning Wizard

第 5 章
开发一个新的 BSP

5.1 什么是 BSP?

板级支持包（BSP）是关于硬件的文件集合，依赖于单板计算机的具体特性。您可能会想开发尚未提供的 BSP。此外，如果您的目标硬件支持定制，则建议您开发一个新的 BSP。

在之前的章节中，您已了解如何克隆现有的 BSP，以及为新的硬件配置编译项目。本章节将进一步说明在开发新 BSP 代码时的注意事项。

5.2 概述

要开发新的 BSP:

1. 选择用于修改的基准 BSP。
2. 克隆选定 BSP（和 PSP）的项目、配置和源代码。
3. 准备 BSP 专用的调试器配置。
4. 修改 BSP 专用的包含文件。
5. 修改启动代码。
6. 修改源代码。
7. 创建 I/O 器件驱动程序的默认初始化信息

5.3 选择基准 BSP

通常最简单的做法就是选择一个现有的基准 BSP，然后对其进行修改以适应您的硬件。在大多数情况下, 选择使用相同或相似处理器基准 BSP。创建现有 BSP 源和项目的最直接的方法就是使用 BSP Cloning Wizard 工具。可以在 Freescale MQX 的启动菜单中找到该应用程序的快捷方式。

也可以手动创建克隆。可遵循以下指示进行操作:

1. 创建一个新的 BSP 源目录，例如:

```
source\bsp\myk60d100mboard
```

2. 进入基准目录，例如:

```
source\bsp\twrk60d100m
```

3. 将基准目录中的内容复制到新目录中。
4. 在新目录中，将旧板级专用名称<*board*>.* 重命名为新 BSP 名称。
5. 创建关联到新 BSP 的其他文件和目录。
   - 新 BSP 配置目录

     ```
     config\bsp\myk60d100mboard
     ```
   - 新建输出目录

     ```
     lib\myk60d100mboard
     ```
6. 如克隆现有配置 中所述克隆 BSP 和 PSP 编译项目。不要忘记修改每个编译目标中的项目设定。
   - 从项目（<*board*>.*）中删除旧板级专用的源代码文件，然后添加新创建的文件。
   - 将包含搜索路径重定向到新的配置目录
   - 将输出库路径重定向到新的输出目录
   - 修改要编译的输出库文件名称（可选）
   - 克隆位于 build\bat 目录中的批处理文件，然后在项目设定中选中它们作为新的链接后操作。
7. 在所有文件中，将所有出现的旧 BSP 或处理器名称（大写和小写）修改为新 BSP/处理器名称。

5.4 编辑调试器配置文件

板级专用配置文件存放在 BSP 源内（位于*/dbg* 子目录），并通过链接后批处理文件复制到输出*/lib* 文件夹。BSP 项目本身不使用调试器配置文件。由特定 BSP 编译的应用程序项目需要在其项目中引用调试器文件。

您可能需要修改调试器初始化文件，诸如*.*cfg* 或*.*mem* 以支持新电路板。调试器初始化文件所需的典型修改包括外部存储器设置（外部总线信号、定时、存储器区域位置等）

附注	以使用调试器工具，在基于相同处理器器件的评估板上使用调试器配置文件为例。

5.5 修改 BSP 专用的包含文件

BSP 专用的包含文件位于:

```
mqx\source\bsp\<board>\
```

其中，<*board*>是 BSP 名称。

下表显示了为新电路板修改 BSP 源文件时所需的工作量。

表 5-1. 修改 BSP 源文件的工作量

文件	导出到相同微处理器的工作量	导出到具有相同子系列的相似处理器的工作量	导出到不同处理器（相同代码和 PSP）的工作量
bsp.h	中	中	高
init_hw.c (bsp_init.c)	中	中	高
bsp_prv.h	中	中	高
bsp_rev.h	低	低	低
enet_ini.c	低	中	高
get_usec.c	低	低	低
gpio_init.c	中	高	高
init_bsp.c	中	高	高
init_<driver_name>.c	低	低	低
<board_name>.h	低	中	高
mqx_init.c	低	低	低
vectors.c	低	中	中
特定于编译器的代码 **cw/*.c**	低	低	低
链接器配置 **cw/*.lcf**	低	中	高
调试器配置 **cw/dbg/*.mem, *.cfg**	低	中	高
PSP 处理器文件	低	高	高

5.5.1 bsp_prv.h

此文件包含:

- BSP 使用的专用函数的原型。
- BSP 中器件的器件初始化结构的原型（位于 *source\io*\中）。

5.5.2 bsp.h

文件包括**#include** 语句，应用程序可使用这些文件访问板级资源和器件驱动程序 API。它还声明了导出到应用程序或 IO 驱动程序的公共 BSP 函数原型 (即，特定于电路板的引脚初始化函数)。

- 处理器专用头文件<*board*>.*h*
- 处理器专用源代码文件
- 用于器件驱动程序 API 的.*h* 文件

5.5.3 <board>.h

<*board*>.*h* 文件（其中<*board*>是目标板名称）申明了以下特定于电路板的定义:

- 电路板类型
- 电路板的存储器映射符，如基址和不同存储器区域的大小 (闪存、RAM 和外部存储器等)。
- 周期性定时器中断的分辨率和频率。
- 总线时钟和系统时钟值。
- 应用程序可装载 ISR 的中断范围。
- 器件驱动程序的中断向量编号和中断优先级，包括周期性定时器。
- MQX 初始化结构的默认值。
- 所有其他电路板独有的硬件定义，如特定于电路板的寄存器、按钮符号名称、LED 和模拟通道等。
- I/O 驱动程序的默认配置选项。

5.6 修改启动代码

BSP 提供默认的启动函数，用于设置运行时环境，然后调用**_mqx()**以启动 MQX RTOS。对于某些电路板，启动代码位于 BSP 的编译器专用子目录中。对于大多数新平台，启动代码是板级独立的，并且位于 PSP 的编译器专用子目录中。根据具体设置，启动代码可以重复利用编译器专用运行时库中标准启动流程的部分代码。

5.6.1 boot.*和<compiler>.c

引导文件（代码为 C 语言或汇编代码）和<*compiler*>.*c* 文件（其中<*compiler*>是编译器工具的名称缩写）配置独立于编译器的代码，用于启动处理器和运行时板级设置。这些文件通常与其他依靠编译器的源和配置文件位于同一个子目录中。

*boot.**文件中的代码处理复位条件:

- 禁止中断。
- 为其他引导启动流程设置初始堆栈。
- 初始化硬件寄存器，如向量表基址、外设寄存器基址和内部存储器基址等。
- 设置主要处理器资源，如时钟源、PLL 和外部总线等。
- 将控制交给标准编译器专用启动函数，进行 C 变量初始化和调用 main()函数。

main()函数在 BSP 源代码中实现 (位于 *mqx_main.c* 文件中)。通过调用*_mqx()*函数，*main()*函数体将控制权转给 MQX 内核。

```
int main
   (
      void
   )
{ /* Body */
   extern const MQX_INITIALIZATION_STRUCT MQX_init_struct;
   /* Start MQX */
   _mqx( (MQX_INITIALIZATION_STRUCT_PTR) &MQX_init_struct );
   return 0;
} /* Endbody */
```

5.7 修改源代码

本节描述了当要支持其他电路板或处理器时，需要修改的主要 BSP 文件。

5.7.1 init_bsp.c

此文件包含:

- 用于 OS 重要函数初始化的预初始化函数，如定时器 (系统滴答)、中断控制器和存储器管理等。(**_bsp_pre_init()**)。
- 用于特定开发板的 IO 初始化函数 (**_bsp_init()**)。
- 周期性定时器 ISR (**_bsp_timer_isr()**)。
- MQX 退出句柄 (**_bsp_exit_handler()**)。
- 支持硬件滴答时间 (如果适用) (**_bsp_get_hwticks()**)。
- 初始化硬件看门狗 (如果适用) (**_bsp_setup_watchdog()**)。

5.7.1.1 _bsp_pre_init()

在初始化期间，MQX RTOS 调用该函数执行以下操作:

- 初始化处理器支持的设备。PSP 可为设备提供 CPU 资源管理，如基于 CPU 的储存器或波特率发生器
- 初始化中断支持。函数**_psp_int_init()**创建并装载 MQX 中断表。

- 初始化缓存和 MMU，并可选择启用它们。PSP 为具有缓存和 MMU 的 CPU 提供支持函数。
- 装载和初始化周期性定时器 ISR。

5.7.1.2 _bsp_init

此函数将初始化 I/O 子系统并安装 I/O 器件驱动器。此代码仅使用条件编译来安装选定的 I/O 驱动器。请参阅 <board>.h，了解默认情况下启用的驱动器。这些设置可在 user_config.h 中或直接在 <board>.h 文件中更改。

5.7.1.3 _bsp_timer_isr()

该函数是周期性定时器中断的中断服务例程。它会清除中断，并根据需要重启定时器。它会调用**_time_notify_kernel()**，以告知 MQX RTOS 发生了中断。

_bsp_timer_isr 句柄还可用于硬件看门狗定时器（如果可用）。

5.7.1.4 _bsp_exit_handler()

当应用程序调用**_mqx_exit()**函数时，该函数将被调用。它会关闭不再使用的器件。

5.7.2 get_usec.c _time_get_microseconds()

该函数返回从上一周期性定时器中断到现在的毫秒数。如果无法确定从上一周期性定时器中断到现在的时间，函数应返回零。

只有当您使用其他定时器时才需要修改该函数；此时，调用其**_timer_get_usec** 函数。

5.7.3 get_nsec.c _time_get_nanoseconds()

该函数返回从上一周期性定时器中断到现在的纳秒数。如果无法确定从上一周期性定时器中断到现在的时间，函数返回零。

只有当您使用其他定时器时才需要修改该函数。此时，调用其**_timer_get_nsec** 函数。

5.7.4 mqx_init.c

该文件包含电路板的默认 MQX 初始化结构，因此简单应用程序或使用默认值（在 *target.h* 中定义）的应用程序无需定义初始化结构。应用程序可以创建新的 MQX 初始化结构，使用部分默认值，并改写其他值。

附注	为保证 MQX 主机工具正常工作，MQX 初始化结构变量必须命名为 **MQX_init_struct**。

5.8 创建 I/O 驱动程序的默认初始化信息

当通过**_bsp_init()**装载 I/O 驱动程序时，可能需要一些初始化文件提供默认信息。

5.8.1 init_<dev>.c

i*nit_*<*dev*>*.c* 文件，其中<*dev*>是器件驱动程序名称，提供装载指定 I/O 驱动程序所需的默认初始化结构和其他信息。

第 6 章
FAQ

6.1 概述

我的应用程序停止了。如何分辨 **MQX RTOS** 是否仍在运行？

如果时间更新了，则说明 MQX RTOS 正在处理周期性定时器中断。如果空闲任务在运行，则 MQX RTOS 也在运行。

6.2 事件

两个任务使用同一个事件组。连接对一个任务起作用，对另一个不起作用。这是为什么？

这两个任务可能共享同一个全局连接，而不是各自具有本地的独立连接。每个任务应调用**_event_open()**或**_event_open_fast()**以获取其自己的连接。

6.3 全局构造函数

我需要在调用**“main”**之前（也就是启动 **MQX RTOS** 之前），使用**“new”**操作符初始化一些全局构造函数。**“new”**操作符调用 **malloc()**，我会将其重新定义用于调用 **MQX** 函数**_mem_alloc()**。我该怎么做？

从**_bsp_pre_init()**初始化构造函数（位于 *init_bsp.c* 中），MQX RTOS 在初始化后会调用存储器管理组件。

6.4 空闲任务

如果空闲任务由于异常而阻塞，会发生什么？

如果空闲任务阻塞，则系统任务 (只有系统任务描述符而没有代码的真正系统任务) 变为活动任务。系统任务描述符设定中断堆栈，然后重新启用中断。从而，应用程序可以继续运行。

6.5 中断

我遇到了周期性间隔的中断，我的应用程序必须快速响应，要超过 **MQX RTOS** 允许的速度。我应该怎么做?

调用**_int_install_kernel_isr()**取代内核 ISR（**_int_kernel_isr()**）。替代的 ISR 必须:

- 在进入时保存所有寄存器内容，并在退出时恢复。
- 一定不能调用任何 MQX 函数。
- 通过应用程序实现的机制（通常是具有头尾指针和总大小字段的环缓冲区）将信息传递给另一个任务（如果需要)。

我的应用程序由多个任务组成，并且应在中断发出某个信号时才运行。我的 **ISR** 应该如何处理中断来合适地控制任务?

如果目标硬件运行，在禁止中断时，将中断优先级设为高于 MQX RTOS 所使用的优先级（见 **MQX_INITIALIZATION_STRUCT** 中的 **MQX_HARDWARE_INTERRUPT_LEVEL_MAX** 字段)。如果这么做，中断将能够中断 MQX RTOS 关键部分。例如，在 ARCtangent 处理器上，MQX RTOS 可以配置为从不禁止等级为 2 的中断，并且只在关键部分使用等级为 1 的中断进行启用/禁止。

如果目标硬件不允许如上所述设置中断优先级，则使用事件组件从 ISR 向多个任务发出信号。任务打开到事件组的连接，其中一个任务为 ISR 提供连接。每个任务调用**_event_wait_any()**或**_event_wait_all()**并阻塞。ISR 调用**_event_set()**来解除任务阻塞。

当我保存，然后从指定中断恢复 **ISR** 时，该如何获得与原始 **ISR** 相关联的数据指针的值?

在装载临时 ISR 之前，调用**_int_get_isr_data()**。该函数会返回指向您传递的指定向量的数据指针。

6.6 存储器

一个任务如何转移不属于它的内存区?

虽然通常是由拥有该内存区的任务进行转移，但是也可以通过**_mem_transfer()**实现利用其他任务转移。

我的任务分配了一个 **10** 字节的存储器块，但总是大于这个空间。这是为什么？

当 MQX RTOS 分配内存区时，该区块需要与合适的存储器边界对齐，并且会关联一个内部头到该区块。这强制占据一个最小空间。

一个任务可以为另一个任务分配内存区吗？

不行。任务只能分配其自己的存储器。不过，任务可以在分配后将存储器发送给另一个任务。

If _partition_test()检测到一个问题，它会尝试修复该问题吗？

不会。这说明存储器已被破坏。需要调试应用程序以确定原因。

当我扩展默认存储器池时，附加的存储器必须与现有存储器池的末端相邻吗？

不用。附加存储器可位于任何位置。

如果我用不相邻的存储器扩展默认存储器池，**_mem_get_highwater()**会返回什么？

高水位标记是 MQX RTOS 分配的内存区中最高的存储器位置。

我在多个处理器上均有任务，因而需要共享存储器。如何在存储器上提供互斥操作？

这取决于您的硬件，可以使用自旋互斥来保护共享存储器。自旋互斥调用 **_mem_test_and_set()**函数，这在硬件支持锁定共享存储器时，对于多处理器是安全的。

6.7 消息传递

如何保证目标消息队列 **ID** 与正确的任务相关联？

创建一个任务，使用名称数据库将每个消息队列编号与一个名称关联起来。然后，每个任务通过指定名称来获取队列编号。

可以在 **PC** 机和目标硬件之间发送消息吗？

可以。创建一个运行在 PC 机上的程序，用于在应用程序之间发送和接收数据包，可以是串行、通过 PCI 或以太网。只要数据包格式正确，MQX RTOS 就能传送任何收到的数据。

我的任务成功地多次调用**_msgq_send()**函数，并且每次都被分配到一个新消息。但是最终在遇到**_msgq_send()**函数时失败。

可能是达到了消息数量限制。每次分配要发送的新消息时，都要检查是否返回 NULL。如果是，则可能是接收任务未释放消息，或未获得机会运行。

6.8 互斥

当拥有互斥数据结构的任务被破坏时，会发生什么？等待锁定互斥的任务会永远等待吗？

不会。所有组件都有清除函数。当任务结束时，清除函数会决定任务使用的资源并将其释放。如果任务锁定了互斥，MQX RTOS 会在任务结束时解锁互斥。任务不应拥有互斥结构存储器；而应创建全局变量形式的结构，或是从系统内存块进行分配。

6.9 信号量

如果“强行撤销”严谨的信号量会发生什么？

当撤销严谨信号量时，强行撤销标志会置位，MQX RTOS 不会立即撤销信号量，知道所有等待任务获取并传递该信号量后才会撤销。(如果信号量为非严谨，MQX RTOS 会立即使所有正在等待信号量的任务就绪。)

两个任务使用一个信号量。连接对一个任务起作用，对另一个不起作用。这是为什么？

这两个任务可能共享同一个全局连接，而不是各自具有本地的独立连接。每个任务应调用**_sem_open()**或**_sem_open_fast()**以获取其自己的连接。

6.10 任务退出句柄与任务异常句柄

两者的区别是什么？

当任务调用**_task_abort()**或当任务从其任务本身返回时，MQX RTOS 调用任务退出句柄。如果装载了 MQX 异常处理，当任务导致了一个不支持的异常时，MQX RTOS 会调用任务异常句柄。

6.11 任务队列

我的应用程序将多个具有相同优先级的任务放在同一个优先级任务队列中。它们是如何排序的？

具有相同优先级的任务按 FIFO 排序。

6.12 任务

是否总是需要至少一个自动启动任务?

是的。在应用程序中，至少需要一个自动启动应用程序任务用于启动应用程序。在多处理器应用中 (应用程序可远程创建任务), 不是每个映像都需要一个自动启动应用程序任务; 然而，每个映像必须包含在任务模板列表中作为自动启动任务的 IPC 任务。 如果处理器上没有创建应用程序任务，则运行空闲任务。

一个自动启动任务创建了所有其他任务并初始化全局存储器。能否在不影响其他子任务的情况下结束该自动启动任务?

可以。当 MQX RTOS 终止创建者时，它会释放创建者的资源（存储器、分区和队列等）和堆栈空间。子任务的资源与创建者相独立，因而不受影响。

创建者任务是否拥有其子任务?

不拥有。两者之间的唯一联系是子任务可以获取其创建者的任务 ID。子任务具有其自己的堆栈空间和自动变量。

什么是任务，如何创建?

如果任务执行相同的根函数，则它们共享相同的代码空间。任务总是从根函数的入口指针开始执行，即使该函数是其创建者的根函数。这与 UNIX 中的 **fork()**函数不同。

我能把一个已创建的任务转移到另一个处理器上吗?

不能。

6.13 时间片

MQX RTOS 如何测量时间片？时间片是绝对的还是相对的？也就是说，如果任务具有一个 **10** 毫秒的时间片，并从 **0** 毫秒时开始执行，那么任务会在处理器到达 **10** 毫秒时结束，还是在处理器执行 **10** 毫秒后结束?

对于 10 毫秒的时间片，在任务激活后，MQX RTOS 会计数发生的周期性定时器中断次数。如果经过了 10 个或更多毫秒数，MQX RTOS 会对任务运行**_sched_yield()** 函数。因而，任务不会获得 10 毫秒的线性时间，因为会发生更高优先级的任务抢占。此外，如果任务调用了调度函数（例如**_task_block()**或**_sched_yield()**），MQX RTOS 会将任务的时间片计数器归零。

超时的话，MQX RTOS 分配的时间是**±BSP_ALARM_FREQUENCY** 滴答/秒。

6.14 定时器

我的应用程序基于多处理器。我有一个主处理器用于发送同步消息到其他处理器，从而使它们复位时间。如何确保复位消息不受应用程序使用的定时器的干扰？

由于改变到绝对时间（**_time_set()**）不会影响定时器，因此启动定时器应为相对时间（**TIMER_ELAPSED_TIME**），而不是绝对时间（**TIMER_KERNEL_TIME_MODE**）。

如果给**_timer_start_oneshot_at()**函数一个过去的到期时间，会发生什么？

MQX RTOS 会将元件置入定时器队列。当下个周期性定时器中断发生时，MQX RTOS 确定当前时间是大于或等于过期时间，从而触发定时器并且 MQX RTOS 调用通知函数。

附录 A
修订历史记录

下表包含了本指南的修改历史记录。

公司网站上提供的文档修订版为最新版本，供您获取最新信息。您的印刷副本可能是较早的版本。要验证您是否拥有最新信息，请访问 freescale.com/mqx。

主题交叉引用	修改说明
MQX 用户指南版本 0	MQX 3.0 附带的初始版本
MQX 用户指南版本 0B	根据 MQX 3.1 版本修改了内容和格式。
在编译时配置 MQX RTOS #rebuilding_mqx 在编译时配置 MQX RTOS	在在编译时配置 MQX RTOS 章节中说明了新 MQX 编译时配置选项。在#rebuilding_mqx 和在编译时配置 MQX RTOS 中更新了 BSP 迁移指令。
信号量 示例：使用内核日志	更新了信号量 章节。添加了示例：使用内核日志 。
分配任务优先级 示例：使用内核日志 在编译时配置 MQX RTOS	在分配任务优先级 中说明了中断等级任务优先级。在示例：使用内核日志 中编辑了 NMI 处理内容。更新了在编译时配置 MQX RTOS 。
MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数	在第 3.4.6 中，从释放资源列表中删除了“轻量级信号量”。添加了关于 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数 的说明。
处理器之间的通信 终止任务 用户模式任务和存储器保护 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数	更新了处理器之间的通信 和终止任务 。新增章节: 用户模式任务和存储器保护 ，删除了“使用 Freescale CodeWarrior Development Studio”章节（与《Freescale MQX™ RTOS 入门》中的内容一致）。根据 Kinetis 平台的相关数据更新了 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数 。
终止任务 互斥量	更新了终止任务 和互斥量 。
MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数 #before_you_begin #rebuilding_mqx #developing_a_new_bsp	更新了 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数 。更新了#before_you_begin 、#rebuilding_mqx 和 #developing_a_new_bsp 。

下一页继续介绍此表...

主题交叉引用	修改说明
MQX RTOS 编译时配置选项 管理任务错误 管理任务 控制高速缓存 超时 管理块大小可变的轻量级存储器 #developing_a_new_bsp	在 MQX RTOS 编译时配置选项 中更新了对于 MQX_CHECK_ERRORS 的描述。更新了管理任务错误 、管理任务 、 控制高速缓存 、 超时 和管理块大小可变的轻量级存储器 。添加了轻量级消息队列组件的说明。更新了任务模板结构定义的示例代码。更新了#developing_a_new_bsp 和 MQX_HARDWARE_INTERRUPT_LEVEL_MAX 配置参数 。
整个文档。	对语言进行了少量编辑，对格式进行了更新。
MQX RTOS 编译时配置选项	在 MQX 编译时配置选项中，添加了 MQXCFG_ENABLE_FP、MQX_SAVE_FP_ALWAYS、MQX_INCLUDE_FLOATING_POINT_IO、MQXCFG_MEM_COPY 和 MQXCFG_MEM_COPY_NEON。
整个文档	更新了整个文档，以反映从 MQX 类型向 C99 类型的转变。
整个文档	更新了电路板参考信息。

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number MQXUG
Revision 12, 02/2014