Freescale MQX
实时操作系统用户手册

MQXUGZHS
第 0 版
2010 年 7 月

飞思卡尔半导体

freescale™
semiconductor

目 录

第一章 前言

1.1 关于MQX

MQX 实时操作系统设计用于单一处理器、多处理器和分布式处理器等形式的嵌入式实时系统。

Freescale 半导体公司成功地搭载 MQX 操作系统软件平台用于 ColdFire 和 PowerPC 系列微处理器。相比于最初发布的 MQX 软件，Freescale MQX 软件更易于配置和使用。现在单一发布版本就包含了 MQX 操作系统外加其它所有软件组件来支持特定的微处理器。有关 Freescale MQX 的发布版本的详细说明如下。

Freescale MQX	本文档将以“Freescale MQX”作为本软件的标识。

MQX 是一个运行时函数库，程序用它来实现实时多任务应用。其主要特征为：大小可裁剪、面向组件的架构和便于使用。

MQX 支持多处理器应用，并且可用于灵活配置嵌入式输入/输出产品，如网络、数据通讯和文档管理等。

本手册通篇都使用 MQX 作为 MQX 操作系统的缩写。

1.2 关于本手册

使用本手册时需要参照：

- MQX参考手册——包含MQX简单和复杂的数据类型，按字母顺序排列的MQX函数原型。

Freescale MQX	Freescale MQX发布版本还包含其它基于MQX操作系统的软件产品。参见RTCS TCP/IP栈、USB主机开发套件、USB设备开发套件、MFS文件系统等用户指南和参考手册。

1.3 版本3.0和2.50的新特点

Freescale MQX	为了延续最初 MQX 发布版本的编号方式， Freescale MQX 发布版本第一版编号为 3.0。尽管主版本编号改变，MQX 并没有主要特性改变，与 2.50 版本兼容。 关于 Freescale MQX RTOS 新特性的最新信息，请参见相关版本的发布文档。

Freescale MQX RTOS 版本 3.0 相比 MQX 版本 2.50 具有如下改进：

- Freescale MQX RTOS 版本包含 MFS、RTCS、USB 等 MQX 关键组件。
- Freescale MQX RTOS 与其它 MQX 组件的默认开发环境是 CodeWarrior Development Studio。新版本还将支持其它开发环境。
- 现在所有关键 Freescale MQX RTOS 组件（PSP、BSP、RTCS、MFS、USB、...）的实时编译配置都由编辑 user_config.h 文件完成，存放于顶层配置文件夹（config）下的板级相关路径。而在以前的版本中，用户配置的宏通过命令行下的 makefile 或者 CodeWarrior 下的预处理文件传入整个编译过程。
- PSP 组件是可裁剪的，基于特定的硬件平台。PSP 组件针对特定处理器设备。PSP 代码仍然保持相对的独立性，而且能够对特定的硬件平台进行更好的内核裁剪。
- MQX 现在支持类型存储（typed memory），即允许一些附加信息能在任务调试插件（task-aware debugger plugin）中显示出来。

MQX2.50 向下兼容 2.40 版本，并且具有如下改进：

- 减少对 32 位类型的依赖——类型由处理器本身数据位数决定。例如一个 16 位的处理器，其一般数据类型大小是 16 位。对于小规模架构的处理器降低了相应代码和数据的大小。
- 时间由一个时钟滴答（tick）来度量——为允许更高分辨率的延迟、定时和时间测量，MQX 用时钟滴答取代秒和毫秒来衡量时间。延迟一段特定的时间或者直到一个特定的时间为止成为可能。这一改进适用于所有使用超时功能的组件。这一内在的改变对于用户来讲是透明的。另外，MQX 加入了一个扩展的日期结构来表示超过 24 世纪的年历日期，精度可以达到皮秒。

- 支持存储管理单元（MMU）——MMU 支持任务间的存储保护。每个任务能建立它自己的被保护数据区域。
- 多个内存池——应用程序可以从一个应用自身定义的全局存储区分配内存块，正如之前应用从默认的内存池分配内存块一样。内存池和 MQX 版本 2.40 的区块（partition）类似，但允许用户分配可变大小的存储块。
- 轻量级的存储模块——与存储组件类似，作为轻量级的组件，MQX 在分配存储块时做较少的检查，且代码和数据量较小。在分配和释放存储空间时，轻量级存储组件更快，但当一个任务结束时回收存储区域速度较慢。
- 能创建处于阻塞状态的任务——应用程序能创建处于阻塞态的任务。MQX 像 MQX2.40 版本一样创建任务，但是不将其加入就绪队列。
- 轻量级事件组——与事件组类似，除了它们更为简单。
- 自动清除事件位——如果一个应用程序用自动清除属性创建了一个事件组，MQX 在事件位被设置后自动地清除该事件位。这一特性使得等待事件位的任务变为就绪态，但是该任务不需要清除事件位。
- 轻量级信号量支持超时等待和轮询方式。
- 任务可以重新启动——应用程序可以从任务的开始函数重新启动任务，且保持同样的任务描述符，任务 ID 号和任务栈。
- 增强的区块组件——区块组件是一个完全注册的 MQX 组件。应用程序可以以处理其它 MQX 组件相同的方式创建和注销这些区块。
- 增强的 EDS 服务器(用于嵌入式调试)—— IPC 支持 EDS 服务器，因此可以跨越一个多处理器 MQX 网络来进行调试。
- 多处理器事件——应用程序能够通过多处理器 MQX 网路中的另一个处理器设置事件位。
- 32 位队列 ID——应用程序除了用 16 位表示队列 ID 外还可以用 32 位表示它们。因此，应用程序能使用大于 255 的数值来表示队列号和处理器号。你可以在编译 PSP 时使能这个选项。
- 更快和更紧凑的 MQX——应用程序可以使用额外的编译时间配置选项来建立一个更小的 MQX 且使用新类型的组件（轻量级组件）来处理存储应用和高速应用。轻量级组件通常使用内存映射的数据结构（没有多处理器功能）。它们比常规的组件速度更快；但是，它们只有较少的特性和保护机制。轻量级组件包括：
  - 轻量级事件（新）——使用内存映射的数据结构定义事件组。
  - 轻量级日志（新）——由固定大小的条目组成。内核日志目前为一个轻量级日志。
  - 轻量级存储组件（新）——使用较少的开销来维持大小可变的存储块堆。MQX 能配置成将轻量级存储组件作为其默认存储组件。
  - 轻量级信号量（lightweight semaphores）——使用内存映射的数据结构来定义信号量。
  - 轻量级定时器（新）——使用内存映射的数据结构来提供可重复的时间驱动服务。

1.4 约定

1.4.1 提示

提示指出有用的信息。

提示：从中断服务程序（ISR）中分配消息的最有效方法就是使用_msg_alloc()

1.4.2 注释

注释标明重要信息。

注释：非严谨的信号量不支持优先级继承。

1.4.3 注意

注意告诉一些命令或程序可能具有不可预期的副作用，或者有可能损坏文件或硬件。

注意：如果你更改了 MQX 的数据类型，一些来自 MQX Embedded 的 MQX™的 PC 端工具可能不能正常工作。

第二章 MQX概述

2.1 MQX的组织结构

MQX 由核心组件（必选）和可选性组件构成。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。为了满足要求，应用程序可通过加入可选组件来扩展和配置核心组件。

图 2-1 显示了以核心组件为中心，外围环绕可选性组件的示意图。

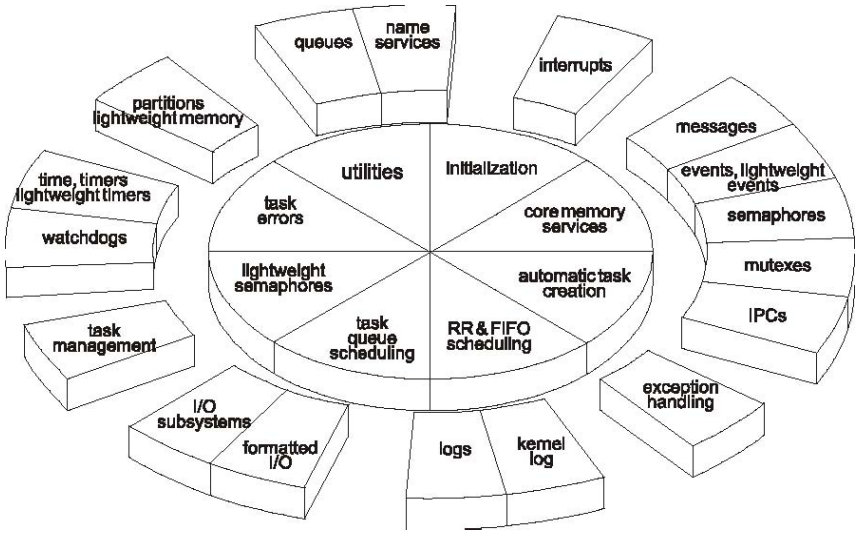


图 2-1

表 2-1 总结了核心组件和可性选组件，它们都将在后面章节中给予描述。

表2-1 核心组件和可选性组件

组件	内容	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	轮转和先进先出	核心
	显式使用任务队列	可选
任务同步和通信	轻量级信号量	核心
	信号量	可选
	轻量级事件	可选
	事件	可选
	互斥	可选
	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
存储管理	可变大小的存储块	核心
	固定大小的存储块（区块）	可选
	存储器管理单元（MMU），高速缓存和虚拟存储	可选
	轻量级存储	可选
中断处理		可选（BSP）
输入/输出驱动	输入/输出子系统	可选（BSP）
	格式化输入/输出	可选（BSP）
检测工具	栈的运用	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，运行测试	核心
队列操纵		核心
命名组件		可选
嵌入式调试	EDS 服务器	可选

2.2 初始化

初始化组件是核心组件。**_mqx()**运行的同时，应用程序开始启动，初始化硬件并启动 MQX。MQX 启动时，创建称为自启动任务的应用程序。

2.3 任务管理

任务管理是核心组件。

与 MQX 启动时自动创建任务一样，应用程序运行时也能创建、管理和终止任务。它能为同一个任务创建多个实例，并且在一个应用程序中没有任务总个数的限制。应用程序可动态改变任一任务的属性。当一个任务终止时，MQX 释放任务资源。

此外，对于每个任务均可指定：

- 退出函数：当任务终止时由 MQX 调用。
- 异常处理：当处于活动状态的任务发生异常时由 MQX 调用。

2.4 调度

调度遵从 POSIX.4 标准（实时扩展）并且支持如下策略：

- FIFO（基于优先级的抢占机制）调度是核心组件——活动任务（active task）是等待时间最长且优先级最高的任务。
- 轮转（又称时间片）调度也是核心组件——活动任务（active task）是优先级最高，等待时间最长且未消耗自身时间片的任务。
- 显式调度（使用任务队列）是一种可选性组件——可以使用任务队列明确地调度任务或创建相对复杂的同步机制。因为任务队列提供尽可能少的功能，因此效率很高。应用程序在创建任务队列时可指定使用 FIFO 或轮转调度策略。

2.5 存储管理

2.5.1 可变大小存储块管理

为了分配和释放可变大小的存储片（称为内存块），MQX 提供了类似于大部分 C 运行函数库提供的 **malloc()**和 **free()**函数功能的核心服务。可以从默认内存池的内部区或外部区分配存储块给任务或系统。分配给任务的存储块是该任务的资源，当任务终止时，MQX 将释放存储块，收回所分配的资源。

2.5.2 固定大小存储块管理（区块）

区块组件是可选性组件。可以分配和管理固定大小的存储片（称作区块）。区块组件支持快速、固定大小的存储分配，减少了存储碎片，节约了存储资源。区块可位于默认内存池的内部（动态区块）和外部（静态区块）。可将区块分配给任务或系统。分配给任务的存储块是该任务的资源，当任务终止时，MQX 将释放存储块，收回所分配的资源。

2.5.3 高速缓存控制

MQX 功能函数能够控制某些 CPU 具有的指令缓存和数据缓存。

2.5.4 存储器管理单元（MMU）控制

对于一些 CPU 而言，在使能高速缓存之前，必须先初始化存储管理单元（MMU）。MQX 功能函数能够初始化、使能和禁止 MMU，以及为 MMU 添加一个存储区域。你可以通过 MMU 页表来控制 MMU。

2.5.5 轻量级存储管理

当一个应用程序受到代码和数据大小限制时，可使用轻量级存储管理。轻量级存储管理只有

少量的接口函数，代码和数据量也较小。因此，某些部分健壮性降低（移除了头部效验），速度变慢（任务销毁次数增多）。

如果你更改了实时编译配置选项，那么在分配存储空间时，MQX 将使用轻量级存储组件。更多内容，参见 3.14 节“实时编译配置 MQX”。

2.6 任务同步

2.6.1 轻量级事件

轻量级事件（LWEvent）是可选性组件。对于使用位状态改变来实现同步的任务而言，这是一个低开销方式。轻量级事件需要的存储空间很少，且速度快。

2.6.2 事件

事件组件是可选性组件，支持格式化为位字段对象的动态管理。任务和中断服务例程可使用事件来实现同步，以位状态改变的形式传送简单信息。有些是命名的快速事件组。事件组具有自动清除事件位，因此，MQX 在事件位被设置后立即清除该事件位。应用程序能够设置位于远程处理器上的事件组内的事件位。

2.6.3 轻量级信号量

轻量级信号量（LWSems）组件是核心组件。对于实现共享资源同步存取的任务而言，这是一种低开销方式。轻量级信号量组件需要的存储空间很少，运行速度快，轻量级信号量计数 FIFO 信号量且没有优先级继承。

2.6.4 信号量

信号量组件是可选性组件。信号量组件用以计数为形式的信号。你可以使用信号量同步任务，控制对共享资源的访问权限，或实现生产/消费信号机制。信号量组件提供 FIFO 队列、优先级队列和优先级继承，可以是严谨的，也可以是非严谨的。有些是命名的快速信号量。

2.6.5 互斥

互斥组件是可选性组件。遵从 POSIX.4a 标准（线程扩展）。在访问共享资源时，互斥组件实现任务间的相互排斥。互斥提供轮询、FIFO 队列、优先级队列、自旋和限制自旋队列、优先级继承以及优先级保护。互斥是严谨的，也就是说，一个任务不能解锁互斥，除非它先锁定了互斥。

2.6.6 消息

消息组件是可选性组件。任务间通过向其它任务打开的消息队列发送消息进行通信。每个任务打开自己的输入消息队列。消息队列由队列 ID 唯一识别，它是在队列创建时由 MQX 分配。只

有打开了消息队列的任务才可以从队列接收消息。任何任务可以发送消息到先前打开的消息队列中，只要它知道打开的队列的 ID。

任务从消息池分配消息。消息池有两种：系统消息池和私有消息池。任何任务可以从系统消息池中分配消息（系统消息）。具有消息池 ID 的任务可以从私有消息池中分配消息（私有消息）。

2.6.7 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。你可以将任务队列中的任务挂起或移除。

2.7 处理器间通信

处理器间通信（IPC）组件是可选性组件。

一个应用程序能够在多个处理器上同时运行，因为每个处理器有一个 MQX 可执行映像。可执行映像通过消息进行通信和协作，这些信息通过内存或处理器间的通信链传输。每个映像中的应用程序任务不必相同，事实上，它们通常是不一样的。

2.8 定时

2.8.1 时间组件

时间组件是可选性组件，你可以在 BSP 级允许或禁止它。时间有两种：消逝时间和绝对时间。你可以改变绝对时间。时间解析度依赖 MQX 启动时为目标硬件设置的应用程序解析度所决定。

2.8.2 轻量级定时器

轻量级定时器组件是可选性组件，为周期性调用应用函数提供低开销机制。轻量级定时器通过建立周期队列安装，在开始阶段加入一个定时器，在某偏移量处到期。

当向队列增加一个轻量级定时器时，可以指定一个告知函数，它将在定时器到期时由 MQX 滴答中断服务程序调用。因为定时器从中断服务程序运行，所以并不是所有的 MQX 函数能够调用。

2.8.3 定时器

定时器组件是可选性组件。它提供周期执行应用程序的功能。MQX 支持一次性定时器（到期一次）和周期性定时器（在给定间隔重复到期）。你可以设置定时器在某指定时间或时间段后启动。

设置了定时器后，可以指定一个告知函数，当定时器到期时由定时器任务调用。告知函数可以用来同步任务，通过发送消息，设置事件，或者使用其它 MQX 同步机制实现。

2.8.4 看门狗

看门狗组件是可选性组件，它帮助用户检测任务级别的崩溃和死锁情况。

2.9 中断和异常处理

中断和异常处理是可选组件（PSP 级别）。MQX 为 BSP 定义范围内的所有硬件中断服务，并为活动任务保存最精简的场景信息。如果 CPU 支持嵌套中断，那么 MQX 也完全支持。一旦进入中断服务程序（ISR）内，应用程序可以重新定义中断级别。为了进一步缩短中断延迟，MQX 推迟任务调度，直到所有中断服务程序都运行之后。除此之外，只有当中断服务程序准备好一个新任务时，MQX 才重新调度。为了减小堆栈规模，MQX 支持独立的中断堆栈。

中断服务程序（ISR）不是任务；它是一个规模小、速度快、能对硬件中断迅速反应的例程。中断服务程序通常用 C 语言编写，它的任务包括重置设备，获取设备数据，以及向相关任务发送信号。通过非阻塞式 MQX 函数，中断服务程序可以用来给一个任务发送信号。

2.10 输入/输出（I/O）驱动

输入/输出驱动是可选性构件（BSP 级别），由格式化输入/输出和输入/输出子系统组成。本手册内容不涉及 I/O 驱动。

2.10.1 格式化输入/输出

MQX 提供格式化输入/输出函数库，它是输入/输出子系统的应用程序接口（API）。

2.10.2 输入/输出子系统

你可以动态安装输入/输出设备驱动程序。此后，任何任务都可以打开它们。

2.11 检测工具

2.11.1 日志

日志组件是可选性组件，让你保存和恢复应用程序专属信息。每个日志条目有一个时间戳和序列号。你可以使用这些信息进行测试、调试、校验和分析运行情况。

2.11.2 轻量级日志

轻量级日志类似于一般日志，但它仅使用固定大小的条目。轻量级日志比普通的应用程序日志速度快，通常用作内核日志。

2.11.3 内核日志

内核日志是可选性组件，让你记录 MQX 活动状态。你可以在指定的位置或者 MQX 选择的位置创建内核日志。你也可以配置内核日志，以便记录所有 MQX 函数调用、场景切换以及中断服务情况。内核日志用在性能评估工具软件中。

2.11.4 栈的运用

MQX 提供的核心功能，可以动态检查中断栈和任务栈，以便判断是否分配了足够的栈空间。

2.12 出错处理

2.12.1 任务出错代码

每个任务有一个任务出错代码，它和任务的场景相关。特定 MQX 功能函数能够读取和更新任务出错代码。

2.12.2 异常处理

你可以为所有未处理的中断指定一个默认的中断服务例程（ISR）。对于中断服务程序产生的异常，你也可以指定一个特定的中断服务例程。

2.12.3 实时测试

MQX 提供了一些核心实时测试函数，在正常运行状态，应用程序能够调用它们。以下组件具有测试函数：

- 事件和轻量级事件
- 内核日志和轻量级日志
- 固定大小的存储块（区块）
- 可变大小的存储块和轻量级存储块
- 消息池和消息队列
- 互斥
- 命名组件
- 队列（应用程序定义的）
- 信号量和轻量级信号量
- 任务队列
- 定时器和轻量级定时器
- 看门狗

2.13 队列操纵

实现队列元素双向链表结构的核心组件。可以初始化队列、加入、移除和读取队列元素。

2.14 命名组件

命名组件是可选性组件。它提供一个名称库，与动态定义的标量（如队列 ID）一一对应。

2.15 嵌入式调试

EDS 服务器是可选性组件。它与运行在 EDS 客户端的主机通信，读写 MQX 信息，从主机获取各种配置信息。

第三章 使用MQX

3.1 前言

本章介绍如何使用 MQX，内容包括可编译和运行的示例。

查看如下内容：	参见：
本章中提到的每个函数的原型	MQX 附录
本章中提到的数据类型	MQX 附录

3.2 初始化并开始运行MQX

MQX 开始于_mqx()函数，该函数以 MQX 初始化结构为参数。基于这一结构中的参数，MQX 完成如下任务：

- 载入并初始化 MQX 经常使用的数据，包括默认内存池，就绪队列，中断堆栈和任务栈；
- 初始化硬件（如芯片选择）；
- 开启定时器；
- 设置默认的时间片值；
- 生成空闲任务，当没有其它任务就绪时将被激活；
- 生成由任务模板列表定义为自启动的任务；
- 开始调度多任务。

3.2.1 MQX初始化结构

MQX 初始化结构中定义应用程序和目标硬件的参数如下：

```
typedef struct mqx_initialization_struct
{
_mqx_uint PROCESSOR_NUMBER;
pointer START_OF_KERNEL_MEMORY;
pointer 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 _PTR_ IO_CHANNEL;
char _PTR_ IO_OPEN_MODE;
```

```
_mqx_uint RESERVED[2];
} MQX_INITIALIZATION_STRUCT, _PTR_ MQX_INITIALIZATION_STRUCT_PTR;
```

如需查看每个域的描述，参见 MQX 附录。

3.2.1.1 默认MQX初始化结构

你可以为 MQX 初始化结构定义自己的初始值，或者使用由每个 BSP 提供的默认值。该默认值被称为 MQX_init_struct 而且保存在相应的 BSP 目录下的 mqx_init.c 文件中。该文件已经被编译并连接到 MQX。

注意：对于任务相关的调试工具来说，MQX 初始化结构必须命名为 MQX_init_struct。

本章中的例子将使用如下 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 */ (pointer)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
};
```

注：初始化 RESERVED 域的元素值为 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，基于该模板可以在处理器上生成任务。

初始化时，MQX 生成每个任务的一个实例，任务模板将其定义为一个自启动任务。同样，当应用程序运行时，它能按任务模板生成其它任务，该模板由任务模板定义或应用程序动态定义。任务模板队列的结尾是一个填入全 0 的任务模板。

```
typedef struct task_template_struct
{
_mqx_uint TASK_TEMPLATE_INDEX;
void _CODE_PTR_ TASK_ADDRESS)(uint_32);
_mem_size TASK_STACKSIZE;
_mqx_uint TASK_PRIORITY;
char _PTR_ TASK_NAME;
```

```
_mqx_uint TASK_ATTRIBUTES;
uint_32 CREATION_PARAMETER;
_mqx_uint DEFAULT_TIME_SLICE;
} TASK_TEMPLATE_STRUCT, _PTR_ TASK_TEMPLATE_STRUCT_PTR;
```

如需查看每个域的描述，请参见 MQX 附录。

3.2.2.1 指定任务优先级

注意：如果指定某个任务优先级为 0，则该任务运行时屏蔽了所有中断。

当你在任务模板列表中指定任务优先级，请注意：

- MQX 为每个优先级生成一个就绪队列直至最低的优先级（最大的数值）
- 当一个应用程序运行时，它不能生成一个优先级低于任务模板列表中的最低优先级（较大的数值）的任务。

3.2.2.2 指定任务属性

你可以为一个任务指定如下属性的任意组合：

- 自启动——当 MQX 开始运行时，它生成任务的一个实例；
- DSP——MQX 保存 DSP 协处理器寄存器作为任务现场的一部分；
- 浮点——MQX 保存浮点寄存器作为任务现场的一部分；
- 时间片——MQX 为任务使用轮循调度（默认是 FIFO 调度）。

3.2.2.3 默认任务模板列表

你可以初始化自己的任务模板列表，也可以使用默认的列表 MQX_template_list。

3.2.2.4 举例：一个任务模板列表

```
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 }
};
```

在这个例子中，world_task 是一个自启动任务，因此在初始化时 MQX 生成一个参数为 0 的任务实例。应用程序定义任务模板索引（MAIN_TASK）。该任务优先级为 5。World_task()函数是任务的入口，栈的大小是 0x2000 个寻址单元。

任务 hello_task 是一个时间片任务，如果使用默认的实时编译配置选项，其时间片数值为 100 毫秒。相关的配置选项参见 3.14 节内容。

Float_task 任务既是一个浮点任务，也是一个自启动任务。

3.2.2.5 实例：创建一个自启动任务

创建一个简单任务，其功能为输出“Hello World”并终止，程序编写如下：

```
/* hello.c */
#include <mqx.h>
#include <fio.h>
/* Task IDs */
#define HELLO_TASK 5
extern void hello_task(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ HELLO_TASK, hello_task, 500, 5, "hello",
MQX_AUTO_START_TASK, 0L, 0},
{ 0, 0, 0, 0, 0,
0, 0L, 0}
};
void hello_task(uint_32 initial_data)
{
printf("\n Hello World \n");
_mqx_exit(0);
}
```

3.2.2.5.1 编译程序并连接到MQX

1．进入如下目录：mqx\examples\hello
2．参考 MQX 发布版本的说明文档，获得创建和运行程序的指令。
3．按照说明文档的指令运行该应用程序。

在输出设备上将会有如下显示：

Hello World

Freescale MQX	针对 Freescale MQX 来说，Codewarrior Development Studio 是 MQX 开发与构建的理想环境。详细的“Hello”程序运行步骤参见下一节。

3.3 使用Freescale CodeWarrior Development Studio

本节将详细介绍使用 Freescale CodeWarrior Development Studio 进行构建、运行、调试 MQX 应用程序的基本步骤。ColdFire M52259 评估板的“Hello”演示程序可作为范例。

关于使用 CodeWarrior 重建 MQX 操作系统和其它核心组件的详细内容，请参见 4.5 节“重建 Freescale MQX RTOS”。

1．从 windows 开始菜单运行 CodeWarrior Development Studio
2．在 CodeWarrrior 集成开发环境中使用 File/Open 菜单，打开为 M52259 评估板开发的“Hello”

应用程序工程。默认情况下，你可以在如下目录找到该工程文件：

C:\Program Files\Freescale\Freescale MQX\mqx\examples\hello\codewarrior\hello_m52259evb.mcp

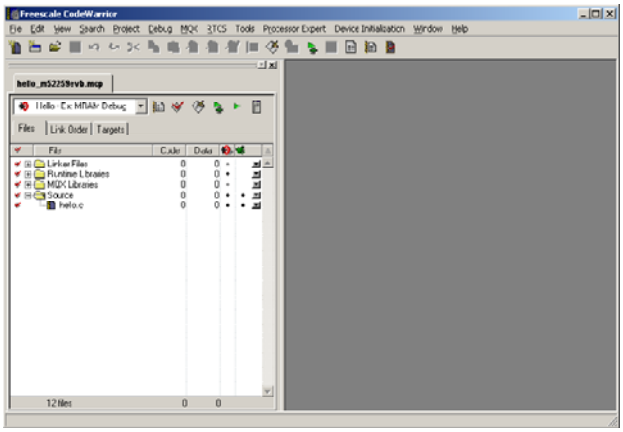


图 3-1

3．选择最适合你需求的构建目标。例如，使用 M52259 评估板，你能够使用 Ext MRAM 调试目标以简化代码下载过程。在该目标中，可执行代码使用外部 MRAM 存储器（而不是内部的 Flash），同时使用片内 SRAM 存储数据。

4．按 F7 键（Project/Make 菜单）生成项目。

注意：该过程要求你在宿主计算机 lib 目录下已有编译好的 MQX PSP 和 BSP 库。Freescale MQX 要求在安装目录中有预编译的默认库，因此这一点必须满足。

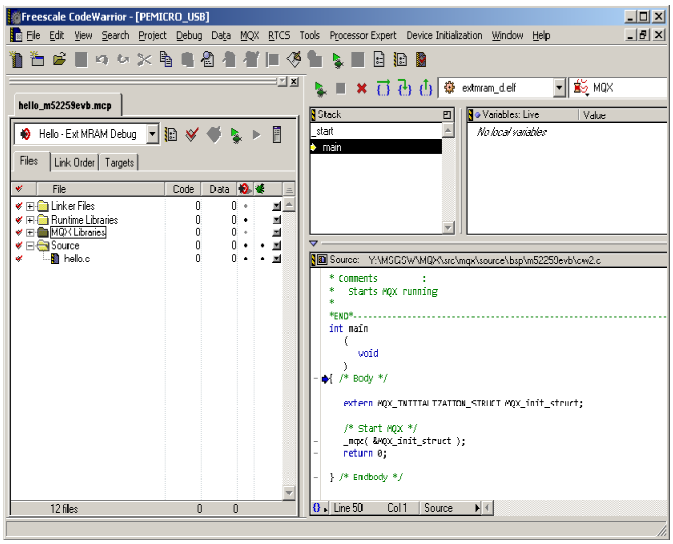
在 Freescale MQX 安装目录查看 *lib/m52259evb.cw*，检查在 *mqx* 子目录中是否有库文件。如果没有，请参照 4.5 节“重建 Freescale MQX RTOS”重新编译 MQX 系统。

5．如果上述步骤均已就绪，构建过程完成时应该没有任何错误或警告。

6．打开超级终端或者宿主计算机上的其它串行终端。

7．以如下配置打开串行终端：115200bps，8 位数据，1 位停止位，无奇偶校验位。

8．在 CodeWarrior 中，按 F5（Project/Debug 菜单）调用 CodeWarrior 调试器，并将可执行应用程序下载到 MRAM 存储器。程序应当在 **main()**函数的默认断点处停止。



9．再按 F5 键继续执行应用程序。

10．在终端窗口中输出 Hello World。

更多关于 Freescale MQX 工程的 CodeWarrior Development Studio 详细信息，参见后续章节、发布版本的说明文档和 Freescale 评估板附送的实验步骤说明书。

3.4 管理任务

从同一任务模板生成的多个任务可以并存，而且每个任务是一个独特的实例。MQX 保存了每个实例的现场，包括程序计数器，寄存器和堆栈。每个任务有一个唯一的 32 位任务编号 ID， MQX 和其它任务通过该编号来区分不同的任务。3.2 节演示了当 MQX 初始化时一个任务如何被自动启动。你也可以在应用程序运行时生成、管理或终止任务。

_task_abort	运行任务退出句柄并释放资源后终止任务
_task_check_stack	判断任务的栈是否超出限制
_task_create	创建并启动一个新的任务
_task_create_blocked	创建一个处于阻塞状态的任务
_task_destroy	释放资源后终止任务
_task_disable_fp	如果该任务属于浮点任务，则屏蔽该任务的浮点开关
_task_enable_fp	打开任务的浮点屏蔽开关
_task_errno	从激活的任务中获取任务出错代码
_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	获取任务的出错代码
_task_get_error_ptr	获取任务的出错代码指针
_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	从任务模板中获取指定名称的首任务 ID
_task_get_index_from_id	获取指定任务 ID 的任务模板索引
_task_get_parameter	获取任务创建参数
_task_get_parameter_for	为一个任务获取创建参数
_task_get_processor	获取任务驻留的处理器编号
_task_get_td	根据任务 ID 获取任务描述的指针
_task_get_template_index	根据任务名称获取任务模板的索引
_task_get_template_ptr	根据任务 ID 获取任务模板的指针
_task_restart	在任务函数开始运行时重启一个任务，保留原来的任务描述、任务 ID 和任务栈
_task_set_environment	为一个任务设置环境数据的指针
_task_set_error	设置任务出错代码
_task_set_exit_handler	设置任务的退出句柄
_task_set_parameter	设置任务的创建参数
_task_set_parameter_for	为一个任务设置任务的创建参数

3.4.1 创建任务

任何任务（创建者）均可以通过调用**_task_create()**或者**_task_create_blocked()**创建其它任务（子任务），并传递处理器编号、任务模板索引和任务创建参数。应用程序定义一个创建参数，通常用于为子任务提供初始化信息。一个任务也可以创建一个在任务模板列表中没有定义过的任务，指定任务模板索引为 0，这时，MQX 把任务创建参数当作一个指向任务模板的指针。

函数初始化子任务的堆栈。**_task_create()**函数将子任务挂入任务优先级的就绪队列。如果子任务比创建者的优先级还高，子任务将变成活动任务，因为它是目前优先级最高的就绪任务。如果创建者有更高或者相等的优先级，则依然保持活动状态。

_task_create_blocked()函数创建一个被阻塞的任务。该任务将保持非就绪状态直到另一个任务调用**_task_ready()**函数。

3.4.2 获取任务IDs

一个任务能够使用**_task_get_id()**函数直接获取其任务 ID。如果使用任务 ID 为参数，你可以用关键字 **MQX_NULL_TASK_ID** 标识当前活动任务的 ID。

一个任务能够使用**_task_get_creator()**函数直接获取其创建者的任务 ID。函数**_task_create()**返回子任务 ID 给创建者。

此外，任务 ID 还可以根据创建该任务的任务模板中的任务名称来确定。该功能可通过**_task_get_id_from_name()**实现，返回的是在任务模板列表中第一次匹配该任务名称的任务 ID。

3.4.3 获取和设置一个任务环境

一个任务可以通过**_task_set_environment** 函数保存一个和应用程序相关的环境指针。其它任

务可以通过**_task_get_environment()**函数访问该环境指针。

3.4.4 管理任务错误

每一个任务都有与任务现场相关的出错代码（任务出错代码）。一些 MQX 函数检测到错误时会及时更新任务出错代码。

如果一个 MQX 函数检测到一个错误而应用程序忽略了该错误，则很可能会有其它错误继续出现。通常第一个错误最能说明问题所在；后续的错误则可能存在误导。在 MQX 将出错代码设置为某一个值而不是 MQX_OK 之后，为了提供一个可靠的机会去诊断错误，MQX 通常不会进一步改变任务错误代码直到任务明确地将其重置为 MQX_OK。

一个任务可以通过如下函数获取任务错误代码：

_task_get_error()

_task_errno()

一个任务通过调用_task_set_error()函数重置其任务出错代码为 MQX_OK。该函数返回先前任务的出错代码并设置任务出错代码为 MQX_OK。

一个任务能通过调用_task_set_error()函数设置其任务错误代码为一个值而不是 MQX_OK。然而，只有当当前任务的错误代码是 MQX_OK 时，MQX 才会更新任务的错误代码。

3.4.5 重启任务

一个应用程序能够通过调用**_task_restart()**函数重启一个任务，该函数从该任务函数的起始位置重启任务，保留了同样的任务描述符、任务 ID 和任务栈。

3.4.6 终止任务

一个任务能够终止自身或者其它任何已知任务 ID 的任务。当一个任务被终止时，他的子任务并不终止。当一个任务终止时，MQX 释放了该任务的所有 MQX 资源。这些资源包括：

动态分配的存储器块和区块

轻量级信号量

消息队列

消息

互斥量

非严谨的信号量

传递后的严谨信号量

取消排队的连接

任务描述符

应用程序能够通过调用**_task_destroy()**或者“礼貌地”调用**_task_abort()**函数来立刻终止一个任务（在 MQX 释放任务所有资源之后）。

当一个任务被“礼貌地”终止，如果该任务被阻塞，MQX 将该任务置入合适的就绪队列。当即将被终止的任务变为活动状态时，将运行一个定义为应用程序的任务退出句柄。该句柄能够清除 MQX 无法处理的资源。

任务退出句柄可以通过**_task_set_exit_handler()**函数设置，也可以通过**_task_get_exit_handler()**函数获取。如果任务从它的任务体返回，则 MQX 也会调用任务退出句柄。

3.4.7 实例：创建任务

该例子基于第 21 页例子增加了第二个任务（world_task）。我们确认例子中的任务模板列表包括了 world_task 的信息，并修改 hello_task 以使它不是一个自启动任务。World_task 任务是一个自启动任务。

当 MQX 启动时，它创建了 world_task。World_task 通过以 word_task 为参数调用**_task_create()**函数去创建 hello_task 任务。MQX 使用 hello_task 模板生成 hello_task 的一个实例。如果**_task_create()**函数调用成功，将返回新的子任务的 ID；否则将返回 **MQX_NULL_TASK_ID**。

新的 hello_task 任务将被置入任务优先级的就绪队列。因为它拥有比 world_task 更高的优先级，它将变为活动状态。活动的任务打印输出 Hello。之后 World_task 任务将变为活动状态并检查 hello_task 任务是否成功创建。如果是，world_task 将输出 World；否则，world_task 将输出一个错误信息。最后 MQX 退出。

如果你更改 world_task 的优先级，使它的优先级和 hello_task 相同，则只输出 World。因为 world_task 和 hello_task 具有相同的优先级并且不会放弃控制权，所以 world_task 在 hello_task 之前运行。既然 hello_task 没有机会再次运行，则 world_task 在输出 World 之后调用**_mqx_exit()**，不会再有任何输出。

3.4.7.1 实例代码

```
/* hello2.c */
#include <mqx.h>
#include <fio.h>
/* Task IDs */
#define HELLO_TASK 5
#define WORLD_TASK 6
extern void hello_task(uint_32);
extern void world_task(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{WORLD_TASK, world_task, 500, 5, "world",
MQX_AUTO_START_TASK, 0L, 0},
{HELLO_TASK, hello_task, 500, 4, "hello",
0, 0L, 0},
{0, 0, 0, 0, 0,
0, 0L, 0}
};
/*TASK*-----------------------------------------------------
*
```

```
* Task Name : world_task
* Comments :
* This task creates hello_task and then prints "World".
*
*END*----------------------------------------------------*/
void world_task(uint_32 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(uint_32 initial_data)
{
printf(" Hello \n");
_task_block();
}
```

3.4.7.2 在MQX下编译和连接应用程序

1. 进入如下目录 mqx\examples\hello2
2. 参照 MQX 发行版本的说明文档，查找关于如何创建和运行应用程序的指令。
3. 按照说明文档中的指令运行应用程序。

在输出设备上将显示如下：

Hello

World

Freescale MQX	针对 Freescale MQX 来说，Codewarrior Development Studio 是 MQX 开发与构建的理想环境。更多细节请参阅 3.3 节“使用 Freescale Codewarrior Development Studio”。

3.5 调度任务

MQX 提供如下任务调度策略：

- FIFO（先来先服务）
- 轮循
- 使用任务队列（详细内容参见 3.7.8 节）

你可以设置处理器和每个任务的调度策略为 FIFO 或者轮循方式，这样，应用程序可能包含使用这些调度策略任意组合的多个任务。

3.5.1 FIFO调度

FIFO 是默认的调度策略。使用先来先服务调度策略，接下来运行的任务（变为活动状态的）总是拥有最高优先级并且等待最久时间的那个任务。活动状态的任务保持运行直至遇到如下状况：

- 活动状态的任务自愿放弃处理器，因为它调用了一个处于阻塞状态的 MQX 函数。
- 产生了中断，因为它拥有比活动任务更高的优先级。
- 更高优先级的任务已经处于就绪状态。

3.5.2 轮循调度

轮循调度与 FIFO 调度比较类似，但不同之处在于每个轮循任务都被分配了最大的时间（时间片），在该时间范围内，该任务才可以被激活。

当且仅当任务的 MQX_TIME_SLICE_TASK 属性在任务模板中被设置时才可以使用轮循调度策略。任务的时间片取决于模板中 DEFAULT_TIME_SLICE 属性值。但如果该值为 0，任务的时间片即为默认的处理器时间片。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 10 倍。由于绝大多数 BSP 的时间间隔为 5 毫秒，则初始的处理器默认时间片为 50 毫秒。你可以通过调用 _sched_set_rr_interval() 或者 _sched_set_rr_interval_ticks() 函数，以处理器的任务 IDMQX_DEFAULT_TASK_ID 为参数，修改默认的处理器时间片。当一个活动的轮循任务的时间片用完时，MQX 将会保存该任务的现场参数，执行切换操作，并检查就绪队列选择激活合适的任务。MQX 将到期的任务移动至任务就绪队列的末尾，并在就绪队列中产生对下一个任务的控制。如果在就绪队列中没有其它的任务，则到期的任务将继续运行。

在轮循调度策略中，相同优先级的任务将以一种时间均等的方式共享处理器时间。

表 3-1　汇总：获取与设置调度信息

_sched_get_max_priority	获取允许设置的最高优先级；返回值为 0
_sched_get_min_priority	获取允许设置的最低优先级
_sched_get_policy	获取调度策略
_sched_get_rr_interval	获取时间片，单位为毫秒
_sched_get_rr_interval_ticks	获取时间片，单位为时钟滴答
_sched_set_policy	设置调度策略
_sched_set_rr_interval	设置时间片，单位为毫秒

_sched_set_rr_interval_ticks	设置时间片，单位为时钟滴答

表 3-2　汇总：任务调度

_sched_yield	移动活动任务至就绪队列的末尾，并将处理器指向就绪队列中具有相同优先级的下一个任务
_task_block	阻塞任务
_task_get_priority	获取任务的优先级
_task_ready	激活一个任务为就绪状态
_task_set_priority	设置任务的优先级
_task_start_preemption	允许任务抢占
_task_stop_preemption	禁止任务抢占

每个任务都处于下列某种逻辑状态：

阻塞状态——任务未准备好而不能被激活，因为该任务在等待某些情况发生；当这些情况发生时，该任务才变为就绪状态；

就绪状态——任务准备好可以被激活，但未进入激活状态，因为其优先级等于或低于当前的激活任务；

激活状态——该任务在运行中。

如果激活状态任务变为阻塞状态或抢占状态，则 MQX 将执行切换操作，此时将从就绪队列中选择合适的任务并激活。MQX 将选择优先级最高的任务进入激活状态。如果多个具有相同优先级的任务进入就绪状态，则就绪队列中的首任务先被激活。也就是说，每个就绪队列执行先来先服务规则。

3.5.2.1 抢占

激活状态的任务可以被抢占。当一个更高优先级的任务变为就绪并因而变成活动任务时，抢占将会发生。先前的活动任务依然处于就绪状态，但不再是活动任务。当一个中断句柄产生一个更高优先级的就绪任务，或者活动任务生成一个更高优先级的就绪任务时，抢占会发生。

3.6 内存管理

3.6.1 使用可变块管理内存

默认情况下，MQX 从自己的默认内存池分配内存块。任务也能在默认内存池之外产生内存池，并从中分配内存块。

两种分配操作类似，均调用了绝大多数 C 语言动态库中的 **malloc( )**和 **free( )函数完成操作**。

注意：不允许将内存块作为消息使用，使用消息时必须从消息池中分配（参见 3.7.7 节）

内存块可以是私有内存块（属于分配它的任务所有）或者系统内存块（不属于任何任务）。当任务结束时，MQX 返回该任务的私有内存块给内存。

当 MQX 分配内存块时，将至少分配所要求大小的内存块（也即分配的内存块可能更大）。

此外，任务可以调用**_mem_transfer()**函数转换内存块的所有权给其它任务。

表 3-3 汇总：采用可变大小的内存块管理内存

_mem_alloc	从默认内存池分配私有内存块
_mem_alloc_from	从指定的内存池分配私有内存块
_mem_alloc_zero	从默认的内存池分配以 0 填充的私有内存块
_mem_alloc_zero_from	从指定的内存池分配以 0 填充的私有内存块
_mem_alloc_system	从默认的内存池分配系统内存块
_mem_alloc_system_from	从指定内存池分配系统内存块
_mem_alloc_system_zero	从默认内存池分配以 0 填充的系统内存块
_mem_alloc_system_zero_from	从指定内存池分配以 0 填充的系统内存块
_mem_copy	拷贝某位置的内存数据到其它位置
_mem_create_pool	在默认内存池之外生成内存池
_mem_extend	追加额外的内存给默认内存池；附加的内存必须在当前默认内存池之外，但不必与其邻接
_mem_extend_pool	追加额外的内存给某非默认内存池；额外的内存必须在该内存池之外，但不必与其邻接
_mem_free	释放默认内存池内、外的内存块
_mem_free_part	释放部分内存块（如果内存块比申请的大，或者比需求的大）
_mem_get_error	获取使用**_mem_test()**函数产生错误时指向内存位置的指针
_mem_get_error_pool	获取使用**_mem_test_pool()**函数产生错误时指向内存位置的指针
_mem_get_highwater	获取默认内存池分配的内存块的高位地址（尽管可能已经被释放）
_mem_get_highwater_pool	获取已分配内存池的高位地址（尽管可能已经被释放）
_mem_get_size	获取内存块的大小，其大小可能大于申请值
_mem_swap_endian	转换为其它端格式
_mem_test	测试默认的内存池；即检查内部校验码以确定内存完整性是否被破坏（通常被破坏是由于应用程序写内存块出界）
_mem_test_and_set	测试并设置内存位置
_mem_test_pool	测试内存池的错误，参见_mem_test()
_mem_transfer	转交内存块所有权给另一任务
_mem_zero	设置全部/部分内存块为全 0

3.6.2 利用可变大小块管理轻量级内存

轻量级内存函数与常规内存函数较为类似，参见 3.6.1 节“利用可变块管理内存”，但它们的

代码和数据量开销较小。

如果你修改了一个 MQX 实时编译配置选项，当分配内存时 MQX 将使用轻量级内存组件。更多信息参见 3.14 节。

表 3-4　汇总：使用可变大小块操纵轻量级内存

轻量级内存使用特定的结构和常量，它们均在 lwmem.h 中定义	
_lwmem_alloc	从默认的轻量级内存池分配私有的轻量级内存块
_lwmem_alloc_from	从指定的轻量级内存池分配私有的轻量级内存块
_lwmem_alloc_zero	从默认的轻量级内存池分配私有的轻量级内存块，并以全 0 填充
_lwmem_alloc_zero_from	从指定的轻量级内存池分配私有的轻量级内存块，并以全 0 填充
_lwmem_alloc_system	从默认的轻量级内存池分配系统轻量级内存块
_lwmem_alloc_system_from	从指定的轻量级内存池分配系统轻量级内存块
_lwmem_alloc_system_zero	从默认的轻量级内存池分配系统的轻量级内存块，并以全 0 填充
_lwmem_alloc_system_zero_from	从指定的轻量级内存池分配系统的轻量级内存块，并以全 0 填充
_lwmem_create_pool	生成轻量级内存池
_lwmem_free	释放轻量级内存块
_lwmem_get_size	获取轻量级内存块的大小，其大小可能大于申请的大小
_lwmem_set_default_pool	设置内存池为默认的轻量级内存池
_lwmem_test	测试所有轻量级内存池
_lwmem_transfer	转交轻量级内存块的所有权为其它任务所有

3.6.3 使用固定大小块管理内存（区块）

使用区块组件，你可以管理固定大小内存块的分区，区块的大小是当任务创建分区的时候指定的。在默认的内存池中有可增长的动态区块，而在默认内存池之外又有不可增长的静态区块。

3.6.3.1 为动态区块生成区块组件

你可以显式地使用**_partition_create_component()**函数生成区块组件。否则，如果不显式地创建，MQX 将在应用程序第一次生成分区时创建它，并且没有其它参数。

3.6.3.2 生成区块

有两种类型的区块可供选择：

区块类型	创建者	调用函数
动态	默认内存池	**_partition_create()**
静态	非默认内存池	**_partition_create_at()**

如果你生成静态区块，你必须保证该内存不会覆盖应用程序所使用的代码或者数据空间。

3.6.3.3 分配和释放区块

应用程序能够分配两种类型的区块：动态区块或静态区块。

区块类型	分配函数	资源归属	调用者
私有 Private	**_partition_alloc()**	申请它的任务	仅被自己
系统 System	**_partition_alloc_system()**	不属于任何任务	任何任务

如果任务被终止时，它的私有区块也被释放。

3.6.3.4 撤销动态区块

如果一个动态区块中的所有分区块都被释放，则任何任务均可调用**_partition_destroy()**函数撤销区块。此外，你不能撤销一个静态区块。

3.6.3.5 举例：两个区块

如下流程图显示一个静态区块和一个动态区块。

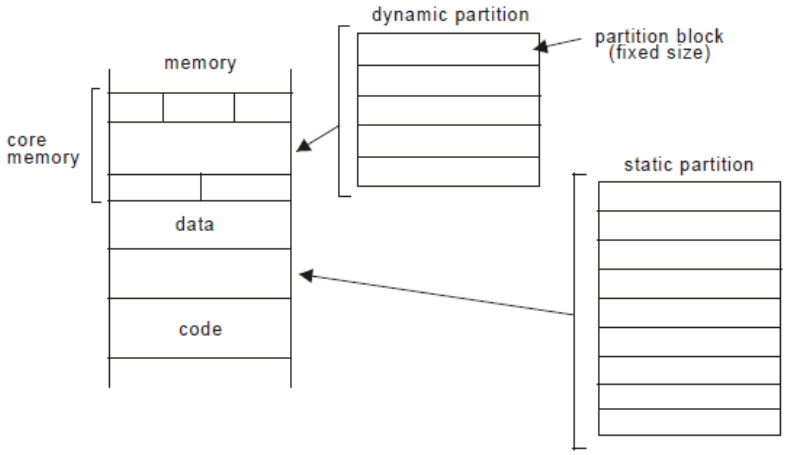


图 3-2

表 3-5　汇总：使用固定大小块管理内存（区块）

_partition_alloc	从分区分配私有区块
_partition_alloc_system	从分区分配系统区块
_partition_alloc_system_zero	从分区分配以全 0 填充的系统区块
_partition_alloc_zero	从分区分配以全 0 填充的私有区块
_partition_calculate_blocks	根据区块的大小和分区的大小（针对静态区块来说），计算区块的数量
_partition_calculate_size	根据分区块的大小和块的数量计算区块的大小

_partition_create	从默认的内存池创建区块（动态区块）
_partition_create_at	在默认的内存池之外，在指定的位置创建分区（静态区块）
_partition_create_component	创建区块组件
_partition_destroy	撤销未分配任何分区块的动态区块
_partition_extend	增加内存到静态分区，增加的内存按照该分区中块的大小被划分成相同大小的区块
_partition_free	返回分区块给该分区
_partition_get_block_size	获取分区中分区块的大小
_partition_get_free_blocks	获取分区中未分配的分区块的数量
_partition_get_max_used_blocks	获取分区中已分配的分区块的数量，也即，高位地址标识已经同时分配的最大数量，但未必是当前分配的数量
_partition_get_total_blocks	获取分区中分区块的数量
_partition_get_total_size	获取分区的大小，包括扩展分区
_partition_test	测试区块组件
_partition_transfer	转交分区块的所有权给其它任务（包括系统等）；只有新的所有者才有权释放分区块

3.6.4 操纵缓冲寄存器

MQX 函数允许我们操纵某些 CPU 的指令寄存器和数据寄存器。

为了我们能够写一个有或者没有缓冲寄存器系统的应用程序，MQX 将这些函数封装成了宏。对于那些没有缓冲寄存器的 CPU 来说，宏不映射到任何函数；对于一些 CPU 来说，使用了统一的缓冲寄存器（该缓冲寄存器既用于数据也用于代码），则**_DCACHE_**和**_ICACHE_**宏块映射到了相同的函数。

3.6.4.1 清除数据缓存

MQX 使用关键词 *flush* 表示清除数据缓存中的所有数据。在缓存中那些未写数据被写入物理存储器中。

3.6.4.2 使数据/指令缓存失效

MQX 使用关键词 *invalidate* 表示使所有缓存实体失效。缓存中遗留的数据或指令如果没有被写入存储器，则将被丢失。后续的访问会重新加载缓存，其数据或者指令来自物理内存。

表 3-6　汇总：控制数据缓存

_DCACHE_DISABLE	禁用数据缓存
_DCACHE_ENABLE	启用数据缓存
_DCACHE_FLUSH	清除数据缓存
_DCACHE_FLUSH_LINE	清除指定地址的数据缓存数据
_DCACHE_FLUSH_MLINES	清除指定区域的数据缓存数据

_DCACHE_INVALIDATE	使数据缓存失效
_DCACHE_INVALIDATE_LINE	使指定地址的数据缓存失效
_DCACHE_INVALIDATE_MLINES	使指定区域的数据缓存失效

表 3-7　汇总：控制指令缓存

_ICACHE_DISABLE	禁用指令缓存
_ICACHE_ENABLE	启用指令缓存
_ICACHE_INVALIDATE	使指令寄存器失效
_ICACHE_INVALIDATE_LINE	使指定地址的指令寄存器失效
_ICACHE_INVALIDATE_MLINES	使指定区域的指令寄存器失效

3.6.5 控制MMU（虚拟存储器）

对于某些 CPU 来说，你在启用缓存之前必须初始化内存管理单元（Memory Management Unit, MMU）。MQX 函数允许初始化、启用、禁用 MMU，或者为其增加一个存储区域。

你能够通过 MMU 页表管理 MMU。

虚拟存储器组件允许应用程序控制 MMU 页表。

下图表示了虚拟地址、MMU 页表、MMU 页、物理页以及物理地址之间的关系。

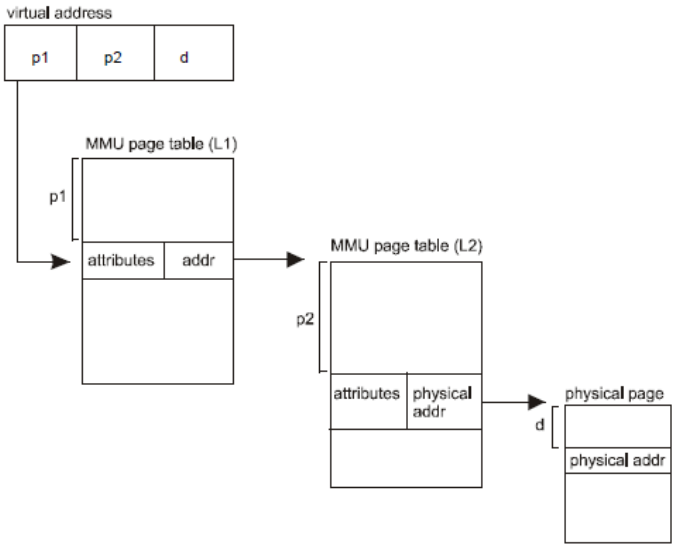


图 3-3

使用虚拟存储组件，应用程序能够管理虚拟内存，使它映射到相应的物理地址。

应用程序能够使用虚拟内存组件为一个任务生成虚拟现场。虚拟现场为该任务提供私有的内存空间，而且仅当该任务活动时才可见。

当 BSP 被初始化时，需要调用如下函数：

表 3-8　汇总：管理虚拟内存

_mmu_add_vcontext	为虚拟现场增加内存区域

_mmu_add_vregion	为 MMU 页表增加内存区域，以便供所有任务和 MQX 使用
_mmu_create_vcontext	为一任务生成虚拟现场
_mmu_create_vtask	使用初始化的虚拟现场生成一任务
_mmu_destroy_vcontext	撤销一任务的虚拟现场
_mmu_get_vmem_attributes	获取一 MMU 页的虚拟内存属性
_mmu_get_vpage_size	获取一 MMU 页的大小
_mmu_set_vmem_attributes	设置一 MMU 页的虚拟内存属性
_mmu_vdisable	禁用虚拟存储器
_mmu_venable	启用虚拟存储器
_mmu_vinit	初始化 MMU，使用 MMU 页表
_mmu_vtop	获取虚拟地址对应的物理地址

3.6.5.1 举例：使用虚拟存储初始化MMU

增加一定数量的内存区用于指令缓存和数据缓存，所有任务均可访问该内存区：

```
_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);
...
}
```

3.6.5.2 举例：创建虚拟现场

在地址 0xA0000000 处设置活动任务可访问 64KB 的私有内存：

```
...
{
pointer virtual_mem_ptr;
uint_32 size;
virtual_mem_ptr = (pointer)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) {
}
...
```

3.6.5.3 举例：使用虚拟现场创建任务

使用虚拟现场创建任务，并拷贝公共数据：

```
...
/* Task template number for the virtual-context task: */
#define VMEM_TTN 10
/* Global variable: */
uchar_ptr data_to_duplicate[0x10000] = { 0x1, 0x2, 0x3 };
...
{
pointer virtual_mem_ptr;
virtual_mem_ptr = (pointer)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) {
}
...
```

3.7 任务同步

你可以通过如下一种或者多种机制实现任务之间的同步，这些机制将会在后续章节中介绍：

- 事件——任务能够等待一组事件位被置位。任务可以设置或者清除这组事件位；
- 轻量级事件——事件的简化实现；
- 信号量——任务等待信号量从非 0 逐步增长，其它任务可以增长信号量。MQX 信号量通过优先级继承防止优先级倒置。关于优先级倒置的更多讨论，参见 3.7.3.2 节。
- 轻量级信号量——简单计数信号量
- 互斥——任务可以使用互斥保证某一时刻仅有一个任务访问共享的数据。为了访问共享数据，任务可对互斥量加锁，如果该互斥量已经被加锁则等待。当任务完成对共享数据的访问，则解锁该互斥量。互斥通过优先级继承和优先级保护来防止优先级倒置。详细内容参见 3.7.3 节。
- 消息传递——MQX 允许任务之间传递数据。当一个任务在消息中填充了数据并发送到一个特别的消息队列。其它任务可以等待该消息队列的消息到达，也即接收消息。
- 任务队列——允许应用程序挂起并恢复任务。

3.7.1 事件

事件能够被用来同步多个任务，或者实现任务与 ISR 之间的同步。

事件组件包括事件组，它是事件位的集合。事件组中的事件位的数量就是**_mqx_uint 中定义的位的数量。**

任务可以等待事件组中的事件位。如果事件位没有被置位，则任务将阻塞。任何其它任务或者 ISR 均可以置位事件位。当事件位被置位，MQX 将从等待的任务中选择满足条件的任务进入就绪队列。如果事件组有自动清除中的事件位，只要该事件位被置位，MQX 将立即清除事件位。并将该任务置入就绪队列。

Freescale MQX	为了在目标平台上优化代码和数据存储器的要求，事件组件将不会默认地在 MQX 内核中编译。为了测试这一特征，你首先需要在 MQX 用户配置文件中启用它，并重新编译 MQX PSP、BSP 和其它核心组件。详细内容参见 4.5 节“重建 Freescale MQX RTOS”。

此外，有两种事件组：命名的事件组，以唯一的字符串标识；快速事件组，以唯一的数字标识。

应用程序能够通过字符串中特定的处理器号码打开远程处理器上的事件组，之后应用程序可

以设置该事件组的任何事件位。但应用程序不能等待远程事件组中任何事件位。

表 3-9　汇总：使用事件组件

事件使用特定的结构和常数，详细定义参见 event.h 文件	
_event_clear	清除事件组中的特定事件位
_event_close	关闭与事件组的连接
_event_create	生成命名的事件组
_event_create_auto_clear	生成命名的事件组，并包含自动清除事件位
_event_create_component	生成事件组件
_event_create_fast	生成快速事件组
_event_create_fase_auto_clear	生成快速事件组，并包含自动事件位
_event_destroy	撤销命名的事件组
_event_destroy_fast	撤销快速事件组
_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	打开与快速事件组的连接
_event_set	设置本地/远程处理器上事件组中特定的事件位
_event_test	测试事件组件
_event_wait_all	为事件组中所有指定事件位等待指定毫秒数
_event_wait_all_for	为事件组中所有指定事件位等待指定时钟滴答周期（包括硬件滴答）
_event_wait_all_ticks	为事件组中所有指定事件位等待指定的时钟滴答数。
_event_wait_all_untill	等待事件组中所有指定事件位，直到指定滴答时间到
_event_wait_any	为事件组中任何指定事件位等待指定毫秒数
_event_wait_any_for	为事件组中任何指定事件位等待指定时钟滴答周期
_event_wait_any_ticks	为事件组中任何指定事件位等待指定的时钟滴答数
_event_wait_any_until	等待事件组中任何指定事件位，直到指定滴答时间到

3.7.1.1 生成事件组件

你可以调用**_event_create_component()**显示地生成事件组件。如果不显式地生成，则 MQX 将在应用程序首次生成事件组时自动使用默认值生成事件组件。

参数	含义	默认值
初始值	能够创建的事件组的初始数量	8
增量	生成所有事件组时事件组的增量，直到达到最大值为止	8
最大值	如果增量非 0，允许创建的事件组的最大数量	0

3.7.1.2 创建事件组

在任务使用事件组件之前，必须创建一个事件组。

创建的事件组类型	调用函数	参数
快速事件组 （包含自动清除事件位）	**_event_create_fast()** **_event_create_fast_auto_clear()**	索引（事件组件被创建时必须符合指定限制条件）
命名事件组 （包含自动清除事件位）	**_event_create()** **_event_create_auto_clear()**	字符串名称

如果事件组生成时带有自动清除事件位，只要这些事件位被置位，MQX 就会清除它们。这一动作使等待这些位的任务进入就绪状态，不需要任务去清除这些位。

3.7.1.3 打开与事件组的连接

在任务能够使用事件组件之前，必须打开与事件组的连接。

打开的事件组类型	调用函数	参数
快速	**_event_open_fast()**	符合指定限制的索引，而且是事件组件被创建时生成的
命名	**_event_open()**	字符串名称

两个函数均给事件组返回唯一的句柄。

3.7.1.4 等待事件位

任务通过函数**_event_wait_all()**或**_event_wait_any()**等待事件组中一定模式的事件位。当事件位被置位时，MQX 将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（也即函数**_event_create_auto_clear()**或**_event_create_fast_auto_clear()**），则 MQX 清除该位，以便等待的任务不必清除它。

3.7.1.5 设置事件位

一任务通过函数**_event_set()**设置事件组中一定模式的事件位。事件组可以在本地或远程处理器上。当一个事件位被置位时，等待它的任务即变为就绪。如果事件组创建时带有自动清除事件位，则只要这些事件位一旦被置位 MQX 就会清除它们。

3.7.1.6 清除事件位

任务通过调用**_event_clear()**函数清除事件组中一定模式的事件位。然而，如果事件组创建时带有自动清除事件位，则只要这些事件位一被置位 MQX 就会清除它们。

3.7.1.7 关闭与事件组的连接

当任务不再使用事件组时，将可以通过**_event_close()**函数关闭与它的连接。

3.7.1.8 撤销事件组

当任务被阻塞，且在等待即将被撤销的事件组中的事件位时，MQX 会将它们移至就绪队列。

3.7.1.9 举例：使用事件

模拟时钟滴答 ISR 在每次运行时设置事件位。服务任务在每次时钟滴答发生时执行某一动作。因此任务需要等待模拟时钟滴答程序设置的事件位。

3.7.1.9.1 样例代码

```
/* 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(uint_32);
extern void service_task(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] = {
{SERVICE_TASK, service_task, 500, 5, "service",
MQX_AUTO_START_TASK, 0L, 0},
{ISR_TASK, simulated_ISR_task, 500, 5, "simulated_ISR",
0, 0L, 0},
{0, 0, 0, 0, 0,
0, 0L, 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(uint_32 initial_data)
{
pointer 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(uint_32 initial_data)
{
pointer 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");
}
}
```

3.7.1.9.2 用MQX编译和连接应用程序

1. 进入以下目录：Mqx\examples\event
2. 参阅 MQX 发布版本的说明文档，查找创建和运行应用程序的相关指令。
3. 按照说明文档中指令运行该程序。

事件任务在每次事件位被置位时都会输出一条消息。

Freescale MQX	对于Freescale MQX来说，Code Warrior Development Studio 是 MQX 开发与重建的理想环境。详细内容参见第 3.3 节“使用 Freescale CodeWarrior Development Studio”。

3.7.2 轻量级事件

轻量级事件是一种简化的、低成本实现的事件。

轻量级事件组件包含轻量级事件组，后者是事件位的集合。轻量级事件组中的事件位的数目是 **mqx_uint** 中定义的位的数目。

任何任务都可以等待轻量级事件组中的事件位。如果事件位没有被置位，任务将阻塞。任何其它任务或者 ISR 均可置位事件位。当事件位被置位时，MQX 将所有满足等待条件的等待任务置入任务就绪队列。如果轻量级事件组有自动清除事件位，只要这些事件位被置位，MQX 将清除它们并使一个任务进入就绪状态。

轻量级事件组由静态数据结构创建而且不是多处理器的。

表 3-10 汇总：使用轻量级事件组件

轻量级事件使用特定的结构与常量，详细内容定义于 lwevent.h 文件中	
_lwevent_clear	清除轻量级事件组中的指定事件位
_lwevent_create	创建轻量级事件组，指出是否包含自动清除事件位
_lwevent_destroy	撤销轻量级事件组
_lwevent_set	设置轻量级事件组的指定事件位
_lwevent_test	测试轻量级事件组件
_lwevent_wait_for	为轻量级事件组中所有或任一指定事件位等待一个指定的时钟滴答周期
_lwevent_wait_ticks	为轻量级事件组中所有或任一指定事件位等待指定的时钟滴答数
_lwevent_wait_until	等待轻量级事件组中所有或任一指定事件位，直到一个指定的时钟滴答时间到

3.7.2.1 生成轻量级事件组

要生成一轻量级事件组，应用程序需要声明一个 **LWEVENT_STRUCT** 类型的变量，调用 **_lwevent_create()**函数进行初始化，用一个指针指向该变量，并用一个标志标识事件组是否有自动

清除事件位。

3.7.2.2 等待事件位

任务可以调用**_lwevent_wait** 系列函数之一去等待轻量级事件组中一定模式的事件位。当等待条件不满足，函数等待指定的时间后终止。

3.7.2.3 设置事件位

任务可以调用**_lwevent_set()**函数设置轻量级事件组中的一定模式事件位。如果有任务等待的事件位得到满足，则 MQX 将该任务置入就绪队列。如果事件组有自动清除事件位，则 MQX 仅将等待的首任务置为就绪状态。

3.7.2.4 清除事件位

任务可以调用**_lwevent_clear()**函数清除轻量级事件组中的一定模式事件位。然而，如果轻量级事件组在创建时带有自动清除事件位，只要这些事件一被置位，MQX 就会清除它们。

3.7.2.5 撤销轻量级事件组

当一个任务不再需要轻量级事件组，则可以调用**_lwevent_destroy()**函数撤销该事件组。

3.7.3 关于信号量类型的对象

MQX 提供了轻量级信号量（LWSems）、信号量和互斥功能。

你可以使用两种信号量实现任务同步与互斥操作。任务等待信号量，如果信号量为 0，则 MQX 阻塞该任务；否则，MQX 降低信号量，并给该任务一信号量，该任务继续运行。如果带有该信号量的任务结束运行时，则它会传递信号量；任务保持就绪状态。如果任务正在等待信号量，MQX 将该任务置入就绪队列；否则，MQX 增加信号量。

你可以使用互斥实现互斥操作。互斥有时也被称为二进制信号量，因为它的计数仅能是 0 或 1。

3.7.3.1 限制

如果信号量类型的对象是严谨的，任务在释放对象之前必须等待并获得该对象；如果对象是非严谨的，则任务在释放对象前不必获得该对象。

3.7.3.2 优先级倒置

任务的优先级倒置是一个经典问题，指的是相关任务的优先级被倒置。当任务使用信号量或互斥操作获取共享资源时，优先级倒置可能会发生。

3.7.3.3 举例：优先级倒置

有三个不同优先级的任务，中优先级的任务阻止了高优先级任务先运行，如下图所示。

序号	任务 1 （高优先级 P1）	任务 2 （中优先级 P2）	任务 3 （低优先级 P3）
1 2			. 运行 . 获取信号量

3 4		. 就绪 . 抢先任务 3 并运行	
5 6 7 8	. 就绪 . 抢先任务 2 并运行 . 获取任务 3 的信号量 . 分配内存块，等待信号量		
9		. 运行并保持	

3.7.3.4 使用优先级继承防止优先级倒置

当你生成 MQX 信号量或互斥操作时，你可以设置一种属性：优先级继承，该属性可用于防止优先级倒置。

如果你设置了优先级继承，在任务锁定信号量或互斥操作时，任务的优先级将不会比等待该信号量或互斥操作的任何任务的优先级低。这样如果一个高优先级的任务等待信号量或互斥，MQX 会临时性地提高任务的优先级至等待任务的优先级水平。

表 3-11　优先级 继承属性

序号	任务 1 （高优先级 P1）	任务 2 （中优先级 P2）	任务 3 （低优先级 P3）
1 2			. 运行 . 获取信号量
3 4		. 就绪 . 抢先任务 3 并运行	
5 6 7 8	. 就绪 . 抢先任务 2 并运行 . 获取任务 3 的信号量 . 提升任务 3 的优先级至 P1 并分配内存块		
9 10 11			. 抢先任务 1 并运行 . 完成工作并传递信号量 . 优先级降至 P3
12 13	. 抢先任务 3 和任务 2 并运行 . 获取信号量		

3.7.3.5 使用优先级保护防止优先级倒置

当你生成 MQX 互斥时，你可以设置优先级保护的互斥属性和互斥优先级。这些属性可以防止优先级倒置。

如果一个要求锁定互斥的任务的优先级并没有互斥优先级的等级高，MQX 则临时性地提高任务的优先级至互斥的优先级，只要任务锁定了该互斥。

表 3-12　互斥属性

序号	任务 1 （高优先级 P1）	任务 2 （中优先级 P2）	任务 3 （低优先级 P3）
1 2			. 运行 . 锁定互斥(在优先级 P1 下);提升优先级至 P1
3 4		. 就绪 . 不抢先任务 3	
5 6	. 就绪 . 不抢先任务 3		
7			. 以互斥方式完成任务并解锁
8			. 优先级降至 P3
9 10	. 抢先任务 3 并运行 . 锁定互斥		

表 3-13　轻量级信号量、信号量与互斥的比较

特性	轻量级信号量	信号量	互斥
超时	是	是	否
队列方式	FIFO	FIFO 优先	FIFO 优先 只是自旋 限制自旋
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	在轻量级信号量和信号量之间
速度	最快	最慢	在轻量级信号量和信号量之间

3.7.4 轻量级信号量

轻量级信号量是一种简化的、低成本的信号量实现。

轻量级信号量由静态数据结构生成，并且不是多处理器的。

表 3-14　汇总：使用轻量级信号量

_lwsem_create	创建轻量级信号量
_lwsem_destroy	撤销轻量级信号量

_lwsem_poll	提升轻量级信号量（非阻塞）
_lwsem_post	传递轻量级信号量
_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	为轻量级信号量等待指定的时钟滴答周期
_lwsem_wait_ticks	为轻量级信号量等待指定的时钟滴答数
_lwsem_wait_until	等待轻量级信号量直到指定的时钟滴答数结束

3.7.4.1 生成轻量级信号量

为了生成轻量级信号量，你可以声明一**LWSEM_STRUCT**类型的变量并调用**_lwsem_create()**函数进行初始化，该函数有一指针指向变量并设置了一个初始的信号量计数。该信号量计数表明了并发地访问信号量资源的请求数，并设置为初始值。

3.7.4.2 等待并传递轻量级信号量

任务可以调用**_lwsem_wait()**函数等待轻量级信号量。如果信号量计数大于 0，则 MQX 将其减量，任务继续运行。如果信号量计数等于 0，MQX 阻塞该任务直至某些其它任务传递了信号量。

为了释放轻量级信号量，任务可以调用**_lwsem_post()**函数传递它。如果没有其它任务在等待该信号量，则 MQX 增加该信号量的计数。

由于轻量级信号量是非严谨的，任务可以不等待而先传递它；因此信号量计数没有限制，也即可以增长至超过初始值。

3.7.4.3 撤销一轻量级信号量

当一个任务不再需要轻量级信号量时，可以调用**_lwsem_destroy()**函数撤销该信号量。

3.7.4.4 举例：生产者与消费者

生产者与消费者任务可通过轻量级信号量来同步。

1. 读任务生成：

——多个写任务并给每个任务分配一个唯一的字符；

——一个写信号量 LWSem

——一个读信号量 LWSem

2. 每个写任务在写入字符到缓冲区之前都必须等待写信号量。当字符被写入，每个写任务传递读信号量，表示字符现在允许读任务读取了。

3. 每个读任务在从缓冲区读取字符之前必须等待读信号量。在字符被读取后，每个读任务必须传递给写信号量，表示缓冲区现在已经可以写入其它字符了。

3.7.4.4.1 数据结构与定义

```
/* 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(uint_32 initial_data);
extern void read_task(uint_32 initial_data);
extern SW_FIFO fifo;
```

3.7.4.4.2 任务模板

```
/* ttl.c */
#include <mqx.h>
#include <bsp.h>
#include "read.h"
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{WRITE_TASK, write_task, 600, 5, "write",
0, 0L, 0},
{READ_TASK, read_task, 500, 5, "read",
MQX_AUTO_START_TASK, 0L, 0},
{0, 0, 0, 0, 0,
0, 0L, 0}
};
```

3.7.4.4.3 写任务的代码

```
/* 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(uint_32 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);
}
}
```

3.7.4.4.4 读任务的代码

```
/* 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(uint_32 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, (uint_32)('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);
}
}
```

3.7.4.4.5 用MQX编译和连接应用程序

1. 进入如下目录：mqx\examples\lwsem
2. 参阅 MQX 发布版本的说明文档，查找构建和运行应用程序的相关指令。
3. 按照说明文档中的指令运行应用程序。

在输出设备上显示如下内容：

A
A
B
C
A
B
…

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior 集成开发环境是 MQX 开发和构建的理想平台。详细内容参考第 3.3 节“使用 Freescale CodeWarrior 集成开发环境”。

3.7.5 信号量

信号量可以用于同步任务和互斥操作。任务关于信号量的主要操作包括等待信号量和传递信号量。

Freescale MQX	为了在目标平台上优化代码和数据存储器需求，信号量组件默认地不在 MQX 内核中编译。为了测试这一特征，你首先需要在 MQX 用户配置文件中启用并重新编译 MQX PSP、BSP 和其它核心组件。详细内容参见 4.5 节“重建 Freescale MQX RTOS”。

表 3-15 汇总：使用信号量

信号量使用特定的结构与参数，详见 sem.h 文件	
_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	撤销命名的信号量
_sem_destroy_fast	撤销快速信号量
_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
_sem_open	打开与命名信号量的连接
_sem_open_fast	打开与快速信号量的连接
_sem_post	传递信号量
_sem_test	测试信号量组件
_sem_wait	等待信号量一定毫秒数
_sem_wait_for	等待信号量一个时钟滴答周期
_sem_wait_ticks	等待信号量一定时钟滴答数
_sem_wait_until	等待信号量直到一定时间到（时钟滴答）

3.7.5.1 信号量使用纵览

为了使用信号量，任务需要执行如下步骤，每一步骤都将在后续章节中详细叙述：

1. 创建信号量组件（可选）
2. 生成信号量
3. 打开与信号量的连接
4. 如果信号量是严谨的，它将等待信号量
5. 当完成使用信号量，它将及时传递信号量
6. 如果不再使用信号量，它将关闭与信号量的连接
7. 如果信号量正在保护一个共享资源，而该共享资源已经不存在或者不再能访问，则任务可以撤

销该信号量

3.7.5.2 生成信号量组件

你可以显式地调用**_sem_create_component()**函数生成信号量组件。如果不显式地生成，MQX 则在应用程序首次创建信号量时使用默认的参数创建组件。参数及其默认值与事件组件的参数相同，详见 3.7.1.1 节。

3.7.5.3 生成信号量

在使用信号量之前，任务需要创建信号量。

创建该类型信号量：	调用：	参数
Fast	**_sem_create_fast()**	索引，它必须在信号量组件被创建时的指定的范围内。
Named	**_sem_create()**	字符串名称

当任务创建信号量时，需要指定如下内容：

- 计数初始值——信号量计数的初始值标识信号量拥有的锁数量（一个任务可以获得多个锁）。
- 优先级队列——如果优先级队列被设置，等待信号量的任务队列必须按照优先级队列，而且 MQX 置信号量为最高优先级的等待任务。
- 如果优先级队列没有被设置，则队列按照先来先服务顺序，而且 MQX 置信号量为最久等待的任务。
- 优先级继承——如果优先级继承被设置，而且有更高优先级的任务在等待信号量，则 MQX 会提高任务的优先级至等待任务的优先级。更多详细内容，请参考 3.7.3.4 节关于优先级继承的讨论。为了使用优先级继承，信号量必须是严谨的。
- 严谨性——如果声明信号量是严谨的，则任务在能够传递信号量之前必须等待信号量。如果信号量是严谨的，初始值是信号量计数的最大值。如果信号量是不严谨的，则计数不受限制。

3.7.5.4 打开与信号量的连接

一个任务在使用信号量之前，必须先打开与信号量的连接。

打开一个与该类型信号量的连接：	调用：	参数
Fast	**_sem_open_fast()**	索引，它必须在信号量组件被创建时的指定的范围内。
Named	**_sem_open()**	字符串名称

以上两个函数都给信号量返回一个唯一句柄。

3.7.5.5 等待信号量与传递信号量

任务调用**_sem_wait_**系列函数之一等待信号量。如果信号量计数为 0，MQX 阻塞该任务直到其它任务传递该信号量（**_sem_post()**）或者特定的任务的定时时间到。如果计数不为 0，则 MQX 将计数减量，任务继续运行。

当任务传递信号量并且有多个任务正在等待信号量时，MQX 将它们置入就绪队列。如果没有任务在等待，则 MQX 增加信号量计数。在这两种情况下，传递信号量的任务保持就绪状态。

3.7.5.6 关闭与信号量的连接

当任务不再需要使用信号量时，它可以调用**_sem_close()**函数关闭与该信号量的连接。

3.7.5.7 撤销信号量

当信号量不再被需要时，任务可以撤销它。

销毁该类型信号量：	调用：	参数：
Fast	**_sem_destroy_fast()**	索引，它必须在信号量组件被创建时的指定的范围内。
Named	**_sem_destroy()**	字符串名称

同样，任务可以确认是否强制销毁。如果强制销毁，MQX 将等待该信号量的任务置为就绪，并在所有任务传递信号量之后撤销该信号量。

如果不是强制销毁方式，则 MQX 在最后一个等待任务获得并传递信号量之后撤销该信号量。（如果信号量是严谨的，则通常采用这一方式）

3.7.5.8 举例：任务同步与互斥操作

该例子基于 3.7.4.1 节轻量级信号量例题，它给出信号量如何实现任务的同步与互斥操作。

该例子实现了多任务可以读、可以写的 FIFO 机制。访问 FIFO 数据结构时需要进行互斥操作。当 FIFO 满时写入数据任务，或者在 FIFO 空时读取数据任务，都要求实现任务同步操作。为此，需要设置如下三个信号量：

- 索引信号量——为了在 FIFO 中实现互斥
- 读信号量——为了同步读任务
- 写信号量——为了同步写任务

该例子涉及到三个任务：主程序、读和写。主程序初始化信号量，创建读和写任务。

3.7.5.8.1 数据结构与定义

```
/* 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];
uint_32 READ_INDEX;
uint_32 WRITE_INDEX;
} SW_FIFO, _PTR_ SW_FIFO_PTR;
/* Function prototypes */
extern void main_task(uint_32 initial_data);
extern void write_task(uint_32 initial_data);
extern void read_task(uint_32 initial_data);
extern SW_FIFO fifo;
```

3.7.5.8.2 任务模板

```
/* ttl.c */
#include <mqx.h>
#include "main.h"
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{MAIN_TASK, main_task, 1000, 5, "main",
MQX_AUTO_START_TASK, 0L, 0},
{WRITE_TASK, write_task, 600, 5, "write",
0, 0L, 0},
{READ_TASK, read_task, 1000, 5, "read",
0, 0L, 0},
{ 0, 0, 0, 0, 0,
0, 0L, 0}
};
```

3.7.5.8.3 主程序任务的代码

主程序任务创建信号量组件、索引、读写信号量以及读写任务。

```
/* 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(uint_32 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);
}
```

3.7.5.8.4 读任务的代码

```
/* 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(uint_32 initial_data)
{
pointer write_sem;
pointer read_sem;
pointer 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);
}
}
```

3.7.5.8.5 写任务的代码

```
/* 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(uint_32 initial_data)
{
pointer write_sem;
pointer read_sem;
pointer 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);
}
}
```

3.7.5.8.6 编译程序并连接MQX

1. 进入如下目录：mqx\examples\sem
2. 参考 MQX 发布版本的说明文档，查找关于如何创建并运行应用程序的相关指令。
3. 按照说明文档的指令运行该应用程序。

读任务输出了写入 FIFO 的数据。

修改程序以去除优先级继承，并再次运行应用程序。

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior 集成开发环境是 MQX 开发和构建的理想平台。详细内容参见第 3.3 节“使用 Freescale CodeWarrior 集成开发环境”。

3.7.6 互斥

互斥通常用来实现互斥操作，也即一次仅允许一个任务访问共享资源，例如数据或者设备等。要访问共享资源，任务需要锁定与资源关联的互斥。一任务拥有互斥，直到它解锁为止。

Freescale MQX	为了在目标平台上优化代码和数据内存需求，互斥组件不会默认地在 MQX 内核中编译。为了测试这个特性，首先你需要在 MQX 用户配置文件中启用它，并重新编译 MQX PSP、BSP 及其它核心组件。更多细节请参阅第 4.5 节“重建 Freescale MQX RTOS”。

互斥和 POSIX.4a（线程扩展）保持兼容，并提供了优先级继承和保护功能以防止优先级倒置。

表 3-16 汇总：使用互斥

互斥使用特定的结构与常量，详细定义参见 mutex.h	
_mutex_create_component	创建互斥组件
_mutex_destroy	撤销互斥
_mutex_get_priority_ceiling	获取互斥优先级
_mutex_get_wait_count	获取等待互斥的任务数
_mutex_init	初始化互斥
_mutex_lock	锁定互斥
_mutex_set_priority_ceiling	设置互斥优先级
_mutex_test	测试互斥组件
_mutex_try_lock	尝试锁定互斥
_mutex_unlock	解锁互斥

3.7.6.1 创建互斥组件

你可以调用**_mutex_create_component()**函数显示地创建互斥组件。如果不显式地创建，则 MQX 将在应用程序首次初始化互斥时创建它，并且不带参数。

3.7.6.2 互斥属性

互斥可以根据任务的等待与调度协议包含若干属性。

3.7.6.3 互斥等待协议

互斥可以包含若干个等待协议之一，这将影响到申请已被锁定资源的任务。

等待协议	如果互斥已经被锁定，请求任务做如下事情：
排队（默认）	分配内存块，直到另一个任务解锁互斥。当互斥被锁定时，要求锁定的第一个任务（不管优先级）锁定互斥。
优先级排队	分配内存块，直到另一个任务解锁互斥。当互斥被锁定时，最高优先级任务请求锁定并锁定互斥。
Spin-only	无限期地自旋（被分配时间片），直到另一个任务解锁互斥。这意味着 MQX 保存请求任务的现场，并在相同的优先就绪队列中分配下一个任务。当就绪队列中的所有任务都运行了，请求任务再次激活。如果互斥仍然被锁定，重复自旋。
Limited spin	如果另一个任务先解锁互斥，自旋小于或等于指定的次数。

Spin-only 协议仅当共享互斥的任务属于如下两种情况之一时才有效：

- 时间片任务
- 具有相同的优先级

如果具有不同优先级的非时间片任务试图共享一个 Spin-only 互斥，资源已被一低优先级的任务锁定，则更高优先级的任务将永远不能获得锁（除非低优先级的任务阻塞）。

Spin-only 协议的互斥易于导致死锁，因此不推荐使用。MQX 为了保持与 POSIX 的兼容性而使用它。

3.7.6.4 调度协议

互斥为了避免优先级倒置，可以有特别的调度协议。这些规则可能在任务已经锁定互斥的时候影响任务的优先级。默认是两个协议都不起作用。

调度协议	含义
优先级继承	如果已经锁定互斥的任务（任务 A）的优先级并不和正在等待锁定互斥的最高优先级任务（任务 B）一样高，当任务 A 具有互斥时，MQX 将任务 A 的优先级提升，使它和任务 B 的优先级相同。
优先级保护	一个互斥具有一个优先级。如果请求锁定互斥的任务（任务 A）的优先级低于互斥的优先级，只要任务 A 具有互斥锁定，MQX 就提升任务 A 的优先级至互斥的优先级。

3.7.6.5 创建与初始化互斥

任务定义互斥时首先声明一个 **MUTEX_STRUCT** 类型的变量。

为了根据默认的等待协议属性和非特定的调度协议去初始化互斥，任务可以调用**_mutex_init()**函数，获得一个指向互斥变量的指针和一个空指针。

但是，如果要创建一个非默认互斥，任务需要执行如下操作：

1. 定义 MUTEX_ATTR_STRUCT 类型的互斥属性结构
2. 调用_mutatr_init()函数初始化属性结构
3. 调用如下各种函数设置相应的属性：

_mutatr_set_prority_ceiling()

_mutatr_set_sched_protocol()

_mutatr_set_spin_limit()

_mutatr_set_wait_protocol()

1. 调用**_mutex_init()**函数初始化互斥，获得指向互斥的指针和属性结构，之后任务即可使用它们。
2. 使用**_mutatr_destroy()**函数撤销互斥属性结构。

表 3-17　汇总：使用互斥属性结构

_mutatr_destroy	撤销互斥属性结构
_mutatr_get_priority_ceiling	获取互斥属性结构的优先级
_mutatr_get_sched_protocol	获取互斥属性结构的调度协议
_mutatr_get_spin_limit	获取互斥属性结构的 limited_spin 数
_mutatr_get_wait_protocol	获取互斥属性结构的等待规则
_mutatr_init	初始化互斥属性结构
_mutatr_set_priority_ceiling	设置互斥属性结构的优先级
_mutatr_set_sched_protocol	设置互斥属性结构的调度协议
_mutatr_set_spin_limit	设置互斥属性结构的 limited_spin 数
_mutatr_set_wait_protocol	设置互斥属性结构的等待协议

3.7.6.6 锁定互斥

为了访问共享资源，任务可以调用**_mutex_lock()**函数锁定与资源关联的互斥。如果互斥未被锁定，任务加锁互斥并继续运行。如果互斥已被锁定，依据 3.7.6.3 节所描述的互斥等待协议，任务将阻塞直到互斥被解锁。

为了确保不被阻塞，任务可以调用**_mutex_trylock()**函数尝试锁定互斥。如果互斥未被锁定，任务加锁互斥并继续运行。如果任务已被锁定，任务不能获得互斥但可以继续运行。

3.7.6.7 解锁互斥

仅有加锁互斥的任务才可以调用**_mutex_unlock()**函数解锁。

3.7.6.8 撤销互斥

如果不再需要互斥，任务可以调用**_mutex_destroy()**函数撤销它。对于等待该互斥的任何任务，MQX 都将其置入就绪队列。

3.7.6.9 举例：使用互斥

互斥被用于实现互斥操作。有两个时间片任务均向同一设备输出。互斥将阻止它们交替地输出内容。

3.7.6.9.1 例题代码

```
/* 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(uint_32 initial_data);
extern void print_task(uint_32 initial_data);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{MAIN_TASK, main_task, 600, 5, "main",
MQX_AUTO_START_TASK, 0L, 0},
{PRINT_TASK, print_task, 600, 6, "print",
MQX_TIME_SLICE_TASK, 0L, 3},
{0, 0, 0, 0, 0,
0, 0L, 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(uint_32 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, (uint_32)string1);
_task_create(0, PRINT_TASK, (uint_32)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(uint_32 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);
}
}
```

3.7.6.9.2 用MQX编译与连接应用程序

1. 进入如下目录：mqx\examples\mutex
2. 参阅 MQX 发布版本的说明文档，查找关于如何创建并运行应用程序的相关指令。
3. 根据说明文档的指令运行应用程序。

3.7.7 消息

任务之间可以通过交换消息实现相互通信。任务从消息池分配消息。任务发送消息到消息队列，并从消息队列接收消息。消息能够指定优先级或标示紧急。任务还可以广播消息。

Freescale MQX	为了在目标平台上优化代码和数据内存需求，互斥组件不会默认地在 MQX 内核中编译。为了测试这个特性，首先你需要在 MQX 用户配置文件中启用它，并重新编译 MQX PSP、BSP 及其它核心组件。更多细节请参阅第 4.5 节“重建 Freescale MQX RTOS”。

表 3-18 汇总：使用消息

消息使用特定的结构定义与常量，详细定义参见 message.h 文件	
_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数
_msg_create_component	创建消息组件
_msg_free	释放消息
_msg_swap_endian_data	转换消息中定义的数据为其它格式
_msg_swap_endian_header	转换消息头部为其它格式
_msgpool_create	创建私有消息池
_msgpool_create_system	创建系统消息池
_msgpool_destroy	撤销私有消息池
_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
_msgq_get_id	转换一队列号码和处理器号码为队列 ID
_msgq_get_notification_function	获取与某消息队列相关的警告函数
_msgq_get_owner	获取拥有消息队列的任务 ID
_msgq_open	打开私有消息队列
_msgq_open_system	打开系统消息队列
_msgq_peek	获取指向消息队列的头一条消息的指针（消息不出列）

_msgq_pool	从消息队列中获取一条消息（非阻塞）
_msgq_receive	从消息队列获取一条消息，并等待指定毫秒数
_msgq_receive_for	从消息队列获取一条消息，并等待指定的时钟滴答周期
_msgq_receive_ticks	从消息队列获取一条消息，并等待指定的时钟滴答数
_msgq_receive_until	从消息队列中获取一条消息，并等待直到指定的时钟滴答
_msgq_send	发送一条消息到消息队列
_msgq_send_broadcast	发送一条消息到多个消息队列
_msgq_send_priority	发送一条优先级消息到消息队列
_msgq_send_queue	直接发送一条消息至消息队列
_msgq_send_urgent	发送一条紧急消息到消息队列
_msgq_set_notification_function	设置消息队列的警告函数
_msgq_test	测试消息队列

3.7.7.1 创建消息组件

你可以调用**_msg_create_component()**函数显式地创建消息组件。否则，当应用程序首次创建消息池或者打开消息队列时 MQX 会创建该组件。

3.7.7.2 使用消息池

任务从消息池分配消息，消息池是任务之前必须创建的。任务能够调用**_msgpool_create()**函数创建私有消息池，也可以调用**_msgpool_create_system()**函数创建系统消息池。

在创建消息池时，任务需要明确如下信息：

- 消息池中消息的大小
- 消息池中消息的初始个数
- 增长因子：MQX 增加到消息池的增长消息数（如任务已分配了所有消息）
- 消息池中的最大消息数（如果增长因子非 0，这里 0 表示消息池能够容纳不限数量的消息）

_msgpool_create_system()函数能够被多次调用以创建多个系统消息池，每次使用不同的属性。

_msgpool_create()函数将返回一个消息池 ID，每个任务都可用它访问私有消息池。

	系统消息池	私有消息池
创建一个消息池	**_msgpool_create_system()**	**_msgpool_create()**
分配一个消息	**_system()** (MQX 搜索所有系统消息池)	**_msgpool_alloc()** (MQX 只搜索指定的私有消息池)
释放一个消息（只由消息拥有者操作）	**_msg_free()**	**_msg_free()**
撤销一个消息池	一个系统消息池不能被撤销	**_msg_destroy()** (当池中所有消息被释放后通过任

		何具有消息池 ID 的任务来调用)

3.7.7.3 分配与释放消息

在任务发送消息之前，它调用**_msg_alloc_system()**或者**_msg_alloc()**函数从系统/私有消息池分配相应大小的消息。

系统消息池不是任何任务的资源，而且任何任务都可以从中分配消息。任何任务可以根据消息池 ID 从私有消息池分配消息。

当任务从两种消息池分配消息时，消息就变成该任务的资源，直到任务调用**_msg_free()**函数释放消息或者调用**_msgq_send** 函数集中的函数将其置入消息队列。当任务调用**_msgq_pool()**函数或者**_msgq_receive** 系列函数之一从消息队列获取消息时，消息变成该任务的资源。只有消息的所有者才能释放消息。

消息以消息头部（**MESSAGE_HEADER_STRUCT**）开始，头部定义了 MQX 需要的路由信息。应用程序定义的数据紧跟头部之后。

```
typedef struct message_header_struct
{
_msg_size SIZE;
#if MQX_USE_32BIT_MESSAGE_QIDS
uint_16 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, _PTR_ MESSAGE_HEADER_STRUCT_PTR;
```

关于每个域的详细定义，参见 MQX 参考手册。

3.7.7.4 发送消息

当任务申请了消息空间并填充消息头部和数据域之后，它可调用**_msgq_send()**函数发送该消息，这一操作将消息发送到消息头部定义的目标消息队列中。发送消息不是一种阻塞动作。

3.7.7.5 消息队列

任务使用消息队列交换消息。消息队列可以是私有的也可以是系统的。当任务根据特定的号码打开消息队列时，MQX 返回一个唯一的应用程序消息队列 ID，之后任务将据此访问消息队列。

任务可调用**_msgq_get_id()**函数将队列号转换为队列 ID。

3.7.7.5.1 16 位队列ID

常用的 16 位队列 ID 包含了处理器编号和队列编号。

位	15 8	7 0
队列号	处理器编号	队列编号

3.7.7.5.2 32 位队列ID

常用的 32 位队列 ID 包含了处理器编号和队列编号。

位	31	16	15
队列号	处理器编号		队列编号

3.7.7.6 使用私有消息队列接收消息

任务可以发送消息到任何私有消息队列，但仅有打开私有消息队列的任务才可以接收消息。一次仅允许有一个任务打开私有消息队列。

任务通过指定队列号码调用**_msgq_open()**函数以打开私有消息队列，该队列号码介于 8 和 MQX 初始化结构定义的最大队列号码之间。如果任务用队列号码 0 去调用**_msgq_open()**函数，MQX 将打开未打开的私有消息队列中的某一个。

打开私有消息队列的任务可以调用**_msgq_close()**函数关闭该队列，这将去除消息队列中的所有消息并释放这些消息。

任务使用**_msgq_receive** 系列函数之一从私有消息队列中接收消息，这将去除特定队列中的首消息并返回消息指针。如果任务指定队列 ID 为 0，它将从任何打开的消息队列中接收一条消息。从私有队列中接收消息是一种阻塞动作，除非任务设置一时限，规定任务等待这条消息的最大时间限制。

3.7.7.7 使用系统消息队列接收消息

系统消息队列不属于任何一个任务所有，而且任务在接收消息时不会阻塞。由于从系统消息队列中接收消息时任务不会阻塞，ISR 能够使用系统消息队列。任务或 ISR 通过**_msgq_open_system()**函数打开系统消息队列。

任务或 ISR 通过**_msgq_pool()**函数从系统消息队列接收消息。如果在系统消息队列中没有消息，该函数将返回 NULL。

3.7.7.8 确定排队消息数

任务通过**_msgq_get_count()**函数确定系统消息队列或私有消息队列中的消息数。

3.7.7.9 警告函数

对于系统消息队列和私有消息队列，任务均可在消息发往队列时使用警告函数。对于系统消息队列，任务在打开队列时使用警告函数。对于私有消息队列，任务在打开队列后使用**_msgq_set_notification_function()**函数设置警告函数。应用程序能够使用警告函数将另一个同步服务（如事件或信号量等）联结到一个消息队列。

3.7.7.10 举例：客户/服务器模式

客户/服务器模式是一个使用消息传递实现通信与任务同步的例子。

服务器任务等待从客户任务的请求消息而被阻塞。当服务器接收请求时，它执行请求并向客户端回送消息。为了在服务器运行时阻塞客户，这里使用双向消息传递模式。

服务器打开输入消息队列，以接收来自客户任务的请求，并创建一个消息池以从中分配请求消息。之后服务器创建大量的客户任务。在实际应用程序中，客户任务大多不是由服务器创建的。

当服务器打开它的消息队列并创建消息池时，它进入接收来自消息队列的消息的循环中，执行后续动作（例如打印数据），并回送消息给客户端。

客户端也打开一个消息队列，从消息池分配一消息，填充消息域，发送消息给服务器，并等待来自服务器的答复。

3.7.7.10.1 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, _PTR_ SERVER_MESSAGE_PTR;
/* Function prototypes */
extern void server_task(uint_32 initial_data);
extern void client_task(uint_32 initial_data);
extern _pool_id message_pool;
```

3.7.7.10.2 任务模板

```
/* ttl.c */
#include <mqx.h>
#include <bsp.h>
#include "server.h"
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{SERVER_TASK, server_task, 600, 5, "server",
MQX_AUTO_START_TASK, 0L, 0},
```

```
{CLIENT_TASK, client_task, 600, 5, "client",
0, 0L, 0},
{0, 0, 0, 0, 0,
0, 0L, 0}
};
```

3.7.7.10.3 服务器任务的代码

```
/* 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(uint_32 param)
{
SERVER_MESSAGE_PTR msg_ptr;
uint_32 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);
}
}
```

3.7.7.10.4 客户任务的代码

```
/* 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(uint_32 index)
{
SERVER_MESSAGE_PTR msg_ptr;
_queue_id client_qid;
client_qid = _msgq_open((_queue_number)(CLIENT_QUEUE_BASE +
index), 0);
while (TRUE) {
/* 分配消息存储空间 */
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_ptr)msg_ptr->DATA) + 1;
msg_ptr->DATA[0] = ('A'+ index);
printf("Client Task %d\n", index);
_msgq_send(msg_ptr);
/* 等待消息返回 */
msg_ptr = _msgq_receive(client_qid, 0);
```

```
/* 释放消息存储空间 */
_msg_free(msg_ptr);
}
}
```

3.7.7.10.5 编译应用程序并将其连接到MQX

1. 进入如下目录：mqx\examples\msg
2. 参阅 MQX 发布版本的说明文档，查找关于如何构建并运行应用程序的相关指令。
3. 运行应用程序。

Freescale MQX	对于 Freescale MQX，CodeWarrior Development Studio 是 MQX 开发和构建的理想环境。详细内容参阅第 3.3 节，“使用 Freescale CodeWarrior Development Studio”

3.7.8 任务队列

任务队列可用来：

- 从 ISR 中安排任务
- 具体的任务调度
- 实现同步机制

表 3-19. 总结：使用任务队列

_taskq_create	按照指定的队列规则（FIFO 或优先级）创建任务队列
_taskq_destroy	撤消任务队列(将等待任务放到合适的就绪队列中)
_taskq_get_value	获取任务队列的长度
_taskq_resume	重启任务队列中挂起的或者所有任务(将它们置入就绪队列中)
_taskq_suspend	挂起任务，将它置入指定的任务队列中（并从任务就绪队列中删除）
_taskq_suspend_task	挂起非阻塞的任务并将其置入指定的任务队列中(并从任务就绪队列中删除)
_taskq_test	测试所有任务队列

3.7.8.1 创建和撤消任务队列

在应用程序执行具体任务调度前，必须通过调用**_taskq_create()**按照任务队列的排队规则初始化任务队列。MQX 创建任务队列并返回一个队列 ID，该队列 ID 被用于访问任务队列。

任务队列不是创建它的任务的资源，而是系统资源，在创建它的任务没有终止时不会被撤消。

任务可通过调用**_taskq_detroy()**撤消一任务队列。如果任务队列中有任务，MQX 将它们移入就绪队列中。

3.7.8.2 挂起任务

任务通过调用**_taskq_suspend()**在指定的任务队列中挂起自己。MQX 按照任务队列规则将任务置入队列中（阻塞任务）。

3.7.8.3 重启任务

任务通过调用**_taskq_resume()**从指定的任务队列中删除一个或所有任务。MQX 将任务置入就绪队列。

3.7.8.4 举例：任务同步

一个任务被一个 ISR 同步，另一个任务模拟中断。Service_task 任务等待周期性的中断，每当中断产生时，打印出消息。任务首先创建任务队列，然后在队列中将自己挂起。Simulated_ISR_task 任务调用**_time_delay()**模拟一个周期性中断。当时间溢出时，调度 service_task 任务。

3.7.8.4.1 代码举例

```
/* taskq.c */
#include <mqx.h>
#include <fio.h>
/* Task IDs */
#define SERVICE_TASK 5
#define ISR_TASK 6
extern void simulated_ISR_task(uint_32);
extern void service_task(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{SERVICE_TASK, service_task, 600, 5, "service",
MQX_AUTO_START_TASK, 0L, 0},
{ISR_TASK, simulated_ISR_task, 600, 5, "simulated_ISR",
0, 0L, 0},
{0, 0, 0, 0, 0,
0, 0L, 0}
};
pointer my_task_queue;
/*任务*-------------------------------------------------------*
* 任务名 : 模拟ISR任务
* 注释 :
* 该任务暂停和恢复任务队列
*结束*-------------------------------------------------------*/
void simulated_ISR_task(uint_32 initial_data)
{
while (TRUE) {
```

```
_time_delay(200);
_taskq_resume(my_task_queue, FALSE);
}
}
/*任务*-------------------------------------------------------*
* 任务名 : service_task
* 注释 :该任务创建任务队列以及simulated_ISR_task任务，然后进入无限循环，打印“Tick”并且挂起任务队列。
*结束*-------------------------------------------------------*/
void service_task(uint_32 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);
}
/* 创建任务*/
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);
}
}
```

3.7.8.4.2 编译应用程序并连接到MQX

1. 进入到目录：mqx\examples\msg
2. 参阅 MQX 发布版本的说明文档，查找关于如何构建并运行应用程序的相关指令。
3. 运行应用程序。

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior Development Studio 是比较理想的开发编译环境。具体细节参考第 3.3 节“使用 Freescale CodeWarrior Development Studio”

3.8 处理器间的通信

通过处理器间通信(IPC)组件，任务可在远程处理器上实现：

- 消息交换
- 创建(阻塞或者非阻塞)任务
- 结束任务
- 打开或关闭指定的事件组
- 设置指定事件组中的事件位

并非所有处理器需要直接连接或者型号相同。IPC 组件通过中间处理器传递消息并转换至合适的端格式。IPC 组件利用设备信息包控制块（PCB）驱动程序通信。

当带有 IPC 组件的 MQX 初始化时，MQX 初始化 IPC 消息驱动，编译消息路由表，该表中定义了消息在处理器中传输的路径。具体硬件的信息，参考对应 MQX 公司发布的资料。

表 3-20. 总结：处理器间通信设置

_ipc_add_ipc_handler	为 MQX 组件添加 IPC 句柄
_ipc_add_io_ipc_handler	为 I/O 组件添加 IPC 句柄
_ipc_msg_route_add	向消息路由表中添加消息
_ipc_msg_route_remove	从消息路由表中删除消息
_ipc_pcb_init	为 PCB 驱动程序初始化 IPC
_ipc_task	初始化 IPC 的任务以及处理器远程服务请求

3.8.1 发送消息到远程处理器

除了消息路由表，每个处理器有一个或多个 IPC，每个 IPC 包含以下内容：

- 输入功能
- 输出功能
- 输出队列

当任务发送消息到消息队列时，MQX 检查目的处理器编号，这个编号包含在队列 ID 中。如果目的处理器不存在，MQX 检查路由表。

如果有路由，路由表显示 IPC 队列到达目的处理器使用的输出队列，然后 MQX 引导消息到达输出队列，最后输出功能运行并传输 IPC 上的消息。

当 IPC 接收到消息时，输入功能运行。输入功能汇总消息然后调用**_msgq_send()函数**。输入功能不需要决定输入的消息是不是给本地处理器的，如果不是，MQX 引导消息到达目的处理器。

3.8.1.1 举例：四核处理器的应用。

图中显示一个简单的四核处理器应用。表中的编号是任意的，只有处理器输出队列编号是唯一的。

每个处理器有两个 IPC，且每个处理器有两个可能的路由。例如，处理器 1 通过一个 IPC 到处理器 2，通过另一个到处理器 4。路由表支持一个路由，因此要选择最适合的路由表。表中为每个处理器路由列出了一个可能的路由表。

3.8.1.1.1 处理器 1 的路由表

源处理器	目的处理器			
	1	2	3	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 的任务发送消息到处理器 3 的消息队列，使用队列 10。处理器 2 到处理器 3 使用队列 20。当处理器 3 的 IPC 接收到消息，MQX 引导消息到目标消息队列。

3.8.2 创建和结束远程处理器上的任务

通过 IPC 组件，任务可以通过向目标处理器发送服务请求创建和结束远程处理器上的任务。IPC 组件处理请求，并向发出请求的处理器做出响应。

3.8.3 访问远程处理器上的事件组

通过 IPC 组件，任务可以打开和关闭指定远程处理器上的事件组以及设置事件位。但任务不可能等待远程处理器上的事件位是否变化。

通过指定处理器编号打开远程处理器上的事件组。下面的例子将打开处理器 4 上的 Fred 事件：

_event_open(“4:fred”,&handle);

3.8.4 创建和初始化IPC

对于需要在处理器间通信的任务，应用程序必须按照以下步骤创建和初始化每个处理器上的 IPC 组件。每个步骤将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. 创建 IPC 路由表
2. 创建 IPC 协议初始化表
3. 预备 IPC 协议功能初始化和数据
4. 创建 IPC 任务(**_ipc_task()**)

3.8.4.1 编译IPC路由表

IPC 路由表定义了内部处理器消息的路由。每个处理器对应一个路由表，称为 **_ipc_routing_table**。在前面的例子中，对于处理器 2，发送给处理器 1 的消息通过队列 20，发送给处理器 3 和 4 的消息通过队列 21。

路由表结构为路由的数组，末尾以 0 填充。

```
typedef struct ipc_routing_struct
{
_processor_number MIN_PROC_NUMBER;
```

```
_processor_number MAX_PROC_NUMBER;
_queue_number QUEUE;
} IPC_ROUTING_STRUCT, _PTR_ IPC_ROUTING_STRUCT_PTR;
```

这些字段在 MQX 参考手册中有详细描述。

3.8.4.1.1 处理器 1 的路由表

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

3.8.4.1.2 处理器 2 的路由表

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

3.8.4.1.3 处理器 3 的路由表

```
IPC_ROUTING_STRUCT _ipc_routing_table[] =
{ {1, 2, 31},
{4, 4, 30},
{0, 0, 0}};
```

3.8.4.1.4 处理器 4 的路由表

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

3.8.4.2 创建IPC协议初始化表

IPC 协议初始化表定义并初始化使用 IPC 的协议。路由表中每个 IPC 输出队列相关的 IPC 必须在协议初始化表中对应一个单元，用来定义实现 IPC 时使用的协议和通信路径。

注意：IPC_PROTOCOL_INIT_STRUCT 结构中 IPC_OUT_QUEUE 域必须与 IPC_ROUTING_STRUCT 中的 QUEUE 域匹配。

协议初始化表结构为协议初始化结构的数组，末尾以 0 填充。

```
typedef struct ipc_protocol_init_struct
{
_mqx_uint (_CODE_PTR_ IPC_PROTOCOL_INIT)(
struct ipc_protocol_init_struct _PTR_ ipc_init_ptr,
pointer ipc_info_ptr);
pointer ipc_info_ptr);
pointer IPC_PROTOCOL_INIT_DATA;
```

```
char _PTR_ IPC_NAME;
_queue_number IPC_OUT_QUEUE;
} IPC_PROTOCOL_INIT_STRUCT, _PTR_ IPC_PROTOCOL_INIT_STRUCT_PTR;
```

这些字段在 MQX 参考手册有详细描述。

当带有 IPC 组件的 MQX 初始化时，调用 **IPC_PROTOCOL_INIT**，将包含了指定 IPC 初始化信息的 **IPC_PROTOCOL_INIT_DATA** 传给 IPC。

3.8.4.3 IPC使用I/O PCB 设备驱动程序

当开发特殊用途的 IPC 时，MQX 提供在 I/O PCB 设备驱动程序上编译的标准 IPC，这样应用程序可以使用任何 I/O PCB 设备驱动程序去接收和发送消息(参见 3.8.4.5，“举例：IPC 初始化信息”)

下面的 **IPC_PROTOCOL_INIT_STRUCT** 在 PCB 设备驱动程序上使用标准 MQX IPC。

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

3.8.4.4 启动IPC任务

IPC 任务检查 IPC 协议初始化表并启动 IPC 服务器来初始化每个 IPC 驱动。IPC 服务器接收从其它处理器发来的消息，完成远程过程调用。应用程序必须在 MQX 初始化结构中为每个处理器定义 IPC 任务为自启动任务。IPC 任务模板为：

```
{ IPC_TTN, _ipc_task, IPC_DEFAULT_STACK_SIZE, 6,
"_ipc_task", 0, 0L, 0}
```

3.8.4.5 举例：IPC初始化信息

在这个例子中，两个处理器在异步串口上创建 IPC 通信，串口使用匹配 MQX 的 PCB 设备驱动程序。每个处理器由中断驱动异步字符设备驱动 “ittyb:”连接。IPC 调用 PCB_MQXA 驱动来发送和接收具有 MQX 定义格式的信息包。Ipc_init_table 使用 PCB I/O 驱动初始化程序 **_ipc_pcb_init()**中的 MQX IPC 以及初始化需要的数据结构。Pcb_init 定义了如下几个部分：

- 打开驱动时所使用的 PCB I/O 驱动名称
- **_io_pcb_mqxa_install()**调用安装程序(如果 PCB I/O 驱动已经安装就不需要再指定)
- PCB I/O 驱动程序初始化细节 pcb_mqxa_init

3.8.4.5.1 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;
uint_32 DATA;
} THE_MESSAGE, _PTR_ THE_MESSAGE_PTR;
```

3.8.4.5.2 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(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ IPC_TTN, _ipc_task, IPC_DEFAULT_STACK_SIZE, 6,
"_ipc_task", MQX_AUTO_START_TASK, 0, 0},
{ MAIN_TTN, main_task, 2000, 8,
"Main", MQX_AUTO_START_TASK, 0, 0},
{ 0, 0, 0, 0,
0, 0, 0, 0 }
};
MQX_INITIALIZATION_STRUCT MQX_init_struct =
{
TEST_ID,
BSP_DEFAULT_START_OF_KERNEL_MEMORY,
BSP_DEFAULT_END_OF_KERNEL_MEMORY,
BSP_DEFAULT_INTERRUPT_STACK_SIZE,
(pointer)MQX_template_list,
BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
BSP_DEFAULT_MAX_MSGPOOLS,
BSP_DEFAULT_MAX_MSGQS,
BSP_DEFAULT_IO_CHANNEL,
BSP_DEFAULT_IO_OPEN_MODE
};
```

```
IPC_ROUTING_STRUCT _ipc_routing_table[] =
{
{ TEST2_ID, TEST2_ID, QUEUE_TO_TEST2 },
{ 0, 0, 0 }
};
IO_PCB_MQXA_INIT_STRUCT pcb_mqxa_init =
{
/* IO_PORT_NAME */ "ittyb:",
/* 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_ittyb:",
/* DEVICE_INSTALL? */ _io_pcb_mqxa_install,
/* DEVICE_INSTALL_PARAMETER*/ (pointer)&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
};
IPC_PROTOCOL_INIT_STRUCT _ipc_init_table[] =
{
{ _ipc_pcb_init, &pcb_init, "Pcb_to_test2", QUEUE_TO_TEST2 },
{ NULL, NULL, NULL, 0}
};
/*任务*-------------------------------------------------------------*
* 任务名 : main_task
* 注释 :任务创建消息池和消息队列，然后发送消息到处理器 2 的队列。如果需要等待返回消息，则在发送新消息之前，保持该消息有效。
*结束*-------------------------------------------------------------*/
void main_task
(
uint_32 dummy
```

```
)
{
_pool_id msgpool;
THE_MESSAGE_PTR msg_ptr;
_queue_id qid;
_queue_id my_qid;
uint_32 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);
}
```

3.8.4.5.3 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(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ IPC_TTN, _ipc_task, IPC_DEFAULT_STACK_SIZE, 6,
"_ipc_task", MQX_AUTO_START_TASK, 0L, 0},
{ RESPONDER_TTN, responder_task, 2000, 9,
"Responder", MQX_AUTO_START_TASK, 0, 0},
{ 0, 0, 0, 0,
0, 0, 0, 0 }
};
MQX_INITIALIZATION_STRUCT MQX_init_struct =
{
TEST2_ID,
BSP_DEFAULT_START_OF_KERNEL_MEMORY,
BSP_DEFAULT_END_OF_KERNEL_MEMORY,
BSP_DEFAULT_INTERRUPT_STACK_SIZE,
(pointer)MQX_template_list,
BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
BSP_DEFAULT_MAX_MSGPOOLS,
BSP_DEFAULT_MAX_MSGQS,
BSP_DEFAULT_IO_CHANNEL,
BSP_DEFAULT_IO_OPEN_MODE
};
IPC_ROUTING_STRUCT _ipc_routing_table[] =
{
{ TEST_ID, TEST_ID, QUEUE_TO_TEST },
{ 0, 0, 0 }
};
IO_PCB_MQXA_INIT_STRUCT pcb_mqxa_init =
{
/* IO_PORT_NAME */ "ittyb:",
/* BAUD_RATE */ 19200,
/* IS POLLED */ FALSE,
/* INPUT MAX LENGTH */ sizeof(THE_MESSAGE),
/* INPUT TASK PRIORITY */ 7,
/* OUTPUT TASK PRIORITY */ 7
```

```
};
IPC_PCB_INIT_STRUCT pcb_init =
{
/* IO_PORT_NAME */ "pcb_mqxa_ittyb:",
/* DEVICE_INSTALL? */ _io_pcb_mqxa_install,
/* DEVICE_INSTALL_PARAMETER*/ &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
};
IPC_PROTOCOL_INIT_STRUCT _ipc_init_table[] =
{
{ _ipc_pcb_init, &pcb_init, "Pcb_to_test", QUEUE_TO_TEST },
{ NULL, NULL, NULL, 0}
};
/*任务*-------------------------------------------------------------
*
*任务名: responder_task
* 注释 :任务创建消息队列，然后等待消息。在接收消息的过程中，消息返回给发送者之前，数据保持增加。
*结束*-------------------------------------------------------------*/
void responder_task(uint_32 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);
}
}
}
```

3.8.4.5.4 编译应用程序并连接MQX

1. 如何创建和运行应用程序，请参阅 MQX 发布版本的说明文档。
2. 进入目录编译处理器 1 的代码：Mqx\examples\ipc\cpu1\
3. 创建工程。
4. 进入目录编译处理器 2 的代码：Mqx\examples\ipc\cpu2\
5. 创建工程。
6. 连接处理器 1 的 ttyb 到处理器 2 的 ttyb。
7. 参照发布文档的指令运行应用程序。

在处理器 1 前启动处理器 2。

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior Development Studio 是比较理想的开发编译环境。具体细节参考第 3.3 节“使用 Freescale CodeWarrior Development Studio”

3.8.5 消息头的端模式转换

当处理器从远程处理器上接收消息时，IPC 输入函数检查消息头中 **CONTROL** 字段，查看来自其它处理器的消息是否使用了其它的端格式。如果不同于本地的格式，则转换成本地的格式，并设置 **CONTROL** 字段来说明端格式。

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

注意：IPC 无法将消息的数据部分转换为其它端格式，因为 IPC 不清楚数据格式。因此需要应用程序专门将接收的消息转换成其它的端格式。使用宏定义 **MSG_MUST_CONVERT_DATA_ENDIAN 来**检查是否必须转换。使用**_msg_swap_endian_data() 函数来**转换消息数据。所有函数均定义在 *message.h* 文件中，更多信息请查看 MQX 参考手册。

3.9 定时

MQX 提供核心计时组件，可以组合选择定时器以及看门狗组件。

3.9.1 MQX定时翻转法

当应用程序开始运行时，MQX 定时器可作为计时中断的 64 位计数器。在 MQX 定时翻转之

前时间比较长，例如假设速率为每纳秒一次，则 MQX 经过 584 年才会翻转一次。

3.9.2 MQX定时精度

MQX 定时器是计时中断的 64 位计数器，但当应用程序获取运行时间时，时间中同时包含一个 32 位的计数器，它记录上次运行中断后经历的运行时间。MQX 从硬件计数器中获取时间，因此应用程序获取的都是比较精确的时间。

3.9.3 定时器组件

定时器是核心组件，提供消逝时间和绝对时间，用秒/毫秒、或时间滴答(tick)、或一个日期（日期时间或扩展的日期时间）表示。

表 3-21. 总结：使用定时器组件

_ticks_to_time	将运行时间转换成秒/毫秒时间
_time_add_day_to_ticks	运行时间加天
_time_add_hour_to_ticks	运行时间加小时
_time_add_min_to_ticks	运行时间加分钟
_time_add_msec_to_ticks	运行时间加毫秒
_time_add_nsec_to_ticks	运行时间加纳秒
_time_add_psec_to_ticks	运行时间加皮秒
_time_add_sec_to_ticks	运行时间加秒
_time_add_usec_to_ticks	运行时间加微秒
_time_delay	将活动的任务挂起指定的毫秒数
_time_delay_for	将活动的任务挂起指定的嘀嗒时间 (包括硬件运行时间)
_time_delay_ticks	将活动的任务挂起指定的滴答时间
_time_delay_until	将活动的任务挂起直到指定的滴答时间
_time_dequeue	从超时队列中移除任务(通过任务 ID 指定)
_time_dequeue_td	从超时队列中移除任务(通过任务描述符指定)
_time_diff	获取两个秒/毫秒时间结构中的时间差
_time_diff_days	获取运行时间中天数的差
_time_diff_hours	获取运行时间中小时数的差
_time_diff_microseconds	获取运行时间中微秒数的差
_time_diff_milliseconds	获取运行时间中毫秒数的差
_time_diff_minutes	获取运行时间中分钟数的差
_time_diff_nanoseconds	获取运行时间中纳秒数的差
_time_diff_picoseconds	获取运行时间中皮秒数的差
_time_diff_seconds	获取运行时间中秒数的差
_time_diff_ticks	获取运行时间差

_time_from_date	从日期时间中获取秒/毫秒时间
_time_get	从秒/毫秒时间获取绝对时间
_time_get_ticks	获取绝对时间的滴答数（包括滴答时间和硬件滴答时间）
_time_get_elapsed	获取处理器应用程序启动后消逝的秒/毫秒数
_time_get_elapsed_ticks	获取处理器应用程序启动后消逝的滴答数
_time_get_hwticks	获取上次滴答后的硬件滴答数
_time_get_hwticks_per_tick	获取每个滴答包含的硬件滴答数
_time_get_microseconds	获取上次周期性定时中断以后的微秒数
_time_get_nanoseconds	获取上次周期性定时中断以后的纳秒数
_time_get_resolution	获取周期性定时中断的计时分辨率
_time_get_ticks_per_sec	获取时钟中断频率
_time_init_ticks	用一些滴答时间参数初始化滴答时间结构
_time_normalize_xdate	标准化扩展日期结构
_time_notify_kernel	当周期性定时中断发生后被 BSP 调用
_time_set	用秒/毫秒格式设置绝对时间
_time_set_hwticks_per_tick	设置每个时钟嘀嗒内的硬件时钟嘀嗒
_time_set_ticks	设置时钟嘀嗒内的绝对时间
_time_set_resolution	设置周期性定时中断的频率
_time_set_timer_vector	设置 MQX 使用的周期性定时中断向量
_time_set_ticks_per_sec	设置时钟中断频率
_time_ticks_to_xdate	转换运行时间为以 1970.1.1 0:00:00.000 为参考的扩展日期格式
_time_to_date	转换秒/毫秒格式为日期格式
_time_to_ticks	转换秒/毫秒格式为滴答时间格式
_time_xdate_to_ticks	转换扩展日期格式为以 1970.1.1 0:00:00.000 为参考的运行时间格式

3.9.3.1 秒/毫秒时间

时间可以表示为秒或者毫秒。但是相对于处理为滴答时间而言，处理为秒/毫秒时间更为复杂一些，CPU 的负载会更大一些。

```
typedef struct time_struct
{
uint_32 SECONDS;
uint_32 MILLISECONDS;
} TIME_STRUCT, _PTR_ TIME_STRUCT_PTR;
```

这些字段在 MQX 参考手册有描述。

3.9.3.2 滴答时间

时间可以表示为滴答时间。相对于表示为秒/毫秒形式，处理简单一些。

```
typedef struct mqx_tick_struct
{
_mqx_uint TICKS[MQX_NUM_TICK_FIELDS];
uint_32 HW_TICKS;
} MQX_TICK_STRUCT, _PTR_ MQX_TICK_STRUCT_PTR;
```

这些字段在 MQX 参考手册有描述。

3.9.3.3 消逝时间

消逝时间为从处理器开始运行所记录的时间。任务可调用**_time_get_elapsed()**获得消逝时间，也可以通过**_time_get_elapsed_ticks()**获取滴答时间。

3.9.3.4 时间分辨率

MQX 启动时通过装载周期定时器 ISR 来设置硬件的时间分辨率。分辨率定义了 MQX 更新时间的频率以及时钟嘀嗒(tick)发生的频率。分辨率一般为每秒 200 个时钟嘀嗒或 5 毫秒。任务通过**_time_get_resolution()**获取毫秒格式的分辨率，通过**_time_get_resolution_ticks()**获取每秒的时钟嘀嗒，通过**_time_get_elapsed()**获取微秒格式的消逝时间以及_time_get_microseconds()获取从上一个时钟嘀嗒中断往后的微秒数；通过**_time_get_elapsed()**获取纳秒格式的消逝时间以及**_time_get_nanoseconds()**获取从上一个时钟嘀嗒中断以后的纳秒数；通过**_time_get_hwticks()**获取从上一个时钟嘀嗒中断以后的硬件时钟嘀嗒数，通过**_time_get_hwticks_per_tick()**获取硬件时钟分辨率。

3.9.3.5 绝对时间

当消息使用了基于共同时间参考的时间戳时，为了使不同处理器的任务可以交换消息，计时器组件提供绝对时间。

例如，初始时绝对时间是以 1970.1.1 0:00:00:000 为参考的时间。应用程序可以调用**_time_set()**秒/毫秒格式中的参考日期（或调用**_time_set_ticks ()**修改运行时间格式）来修改绝对时间。

任务可以通过**_time_set** 或**_time_get_ticks** 获取绝对时间。

除非应用程序修改了绝对时间，否则以下两个函数可获得相同的值。

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

注意：任务使用消逝时间来衡量时间间隔或应用定时器。这样可防止其它使用**_time_set()**和**_time_set_ticks()函数的**任务对测量值的影响，否则会改变绝对时间。

3.9.3.6 日期中的时间格式

为帮助设置或中断以秒/毫秒/时钟嘀嗒表示的绝对时间，定时器组件提供了日期格式或扩展日期格式的时间表示方法。

3.9.3.6.1 DATA_STRUCT

```
typedef struct date_struct
{
uint_16 YEAR;
uint_16 MONTH;
uint_16 DAY;
uint_16 HOUR;
uint_16 MINUTE;
uint_16 SECOND;
uint_16 MILLISEC;
} DATE_STRUCT, _PTR_ DATE_STRUCT_PTR;
```

这些字段在 MQX 参考手册有描述。

3.9.3.6.2 MQX_XDATE_STRUCT

该结构的日期格式比 DATE_STRUCT 更详细，可通过调用**_time_ticks_to_xdate()**和**_time_xdate_to_ticks()**在 **MQX_XDATE_STRUCT** 和 **MQX_TICK_STRUCT** 进行格式转换。

```
typedef struct mqx_xdate_struct
{
uint_16 YEAR;
uint_16 MONTH;
uint_16 MDAY;
uint_16 HOUR;
uint_16 MIN;
uint_16 SEC;
uint_16 MSEC;
uint_16 USEC;
uint_16 NSEC;
uint_16 PSEC;
uint_16 YDAY;
} MQX_XDATE_STRUCT, _PTR_ MQX_XDATE_STRUCT_PTR;
```

这些字段在MQX参考手册有描述。

3.9.3.7 超时

任务可以将时间作为超时参数传给MQX组件。例如，**_msgq_receive**和**_sem_wait**中的函数。任务还可以调用**_time_delay()**函数将自己挂起。当时间超时后，MQX将任务放入准备队列。

3.9.4 定时器

定时器是内核的可选扩展组件。应用程序可用定时器实现：

• 在指定的时间产生通知功能——当 MQX 创建定时器组件时，它启动定时器任务来维护定时

器和应用程序定义通知信息。当定时器溢出时，定时器任务可调用适当的通知函数。

• 在预定的时间周期结束后进行通信。

Freescale MQX	为了优化某些目标平台上的代码和数据存储，MQX 内核中定时器组件默认不被编译。为了检测这个特征，首先需要在 MQX 用户配置文件中启用它，然后重新编译 MQX PSP，BSP 和其它核心组件。更多细节请参见 4.5 节“重新编译 Freescale MQX RTOS”。

任务可以在任何指定时间开启定时器，或者在当前时间基础上过一段时间再开启。定时器可以使用消逝时间或者绝对时间。

定时器有两种类型：

• 一次性定时器，只超时一次

• 周期性定时器，根据指定的时间间隔重复超时，当定时器超后，MQX 将定时器复位。

表 3-22. 总结：定时器

定时器使用特定的结构和常量，在 timer.h 中定义	
_timer_cancel	取消定时器请求
_timer_create_component	创建定时器组件
_timer_start_oneshot_affer	开启定时器，在一段毫秒级延时后超时一次
_timer_start_oneshot_affer_ticks	开启定时器，在一段运行时间延时后超时一次
_timer_start_oneshot_at	开启定时器，在一段指定时间延时后超时一次
_timer_start_oneshot_at_ticks	开启定时器，在指定时间超时一次
_timer_start_periodic_at	在指定时间开启周期性定时器
_timer_start_periodic_at_ticks	在指定时间开启周期性定时器（以滴答时间为单位）
_timer_start_periodic_every	每隔一些毫秒开启周期性定时器
_timer_start_periodic_every_ticks	每隔一段运行时间开启周期性定时器
_timer_test	测试定时器组件

3.9.4.1 创建定时器组件

可以通过调用**_timer_create_component()**创建定时器组件，同时指定优先级和定时器任务堆栈长度，这些都是在定时器组件创建时由 MQX 创建的。定时器任务管理定时器队列，为通知函数提供场景信息。

如果没有创建定时器组件，MQX 则会在应用程序第一次开启定时器时利用默认值创建。

参数	**默认值**
定时器任务的优先级	1
定时器任务的堆栈长度	500

3.9.4.2 开启定时器

任务可调用以下函数之一开启定时器：

- **_timer_start_oneshot_after(),_timer_start_oneshot_after_ticks()**
- **_timer_start_oneshot_at(),_timer_start_oneshot_at_ticks()**
- **_timer_start_periodic_at(),_timer_start_periodic_at_ticks()**

• **_timer_start_periodic_every(),_timer_start_periodic_every_ticks()**

当定时器调用任意一个函数时，MQX 向定时器队列加入定时器请求。当定时器超时，则通知函数运行。

注意：定时器任务的堆栈空间必须包括通知函数所需要的堆栈空间。

3.9.4.3 取消定时器请求

任务通过调用**_timer_cancel()函数**取消定时器请求，同时需要向该函数传入由**_timer_start** 函数返回的定时器句柄。

3.9.4.4 举例：定时器的使用

每秒钟模拟 LED 亮暗。一个定时器启动 LED，另一个关闭它。

3.9.4.4.1 样例代码

```
/* 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(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{MAIN_TASK, main_task, 2000, 8, "Main", MQX_AUTO_START_TASK,
0L, 0},
{0, 0, 0, 0, 0, 0,
0L, 0 }
};
/*函数*------------------------------------------------------------*
*  函数名: LED_on
*  返回值  : none
*  注释  :该定时器函数打印”ON”
*结束*------------------------------------------------------------*/
void LED_on
(
_timer_id id,
pointer data_ptr,
MQX_TICK_STRUCT_PTR tick_ptr
)
{
```

```
printf("ON\n");
}
/*函数*-------------------------------------------------------*
*  函数名: LED_off
*  返回值  : none
*  注释  :该定时器函数打印“OFF”
*结束*-------------------------------------------------------*/
void LED_off
(
_timer_id id,
pointer data_ptr,
MQX_TICK_STRUCT_PTR tick_ptr
)
{
printf("OFF\n");
}
/*任务*-------------------------------------------------------*
*  任务名  : main_task
*  注释  :任务创建两个定时器，每个定时器周期为 2 秒，第二个与第一个定时器定时相差 1 秒
*结束*-------------------------------------------------------*/
void main_task
(
uint_32 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);
}
```

3.9.4.4.2 利用MQX编译和连接应用程序

1. 进入如下路径：mqx\examples\timer
2. 关于如何创建和运行应用程序，请参考 MQX 的发布说明文档。
3. 根据发布说明文档中的指示运行应用程序。

每次定时器通知函数运行时，都会发出一个消息。

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior Development Studio 是优先推荐的集成开发环境，详细内容请参看第 3.3 节“使用 Freescale CodeWarrior Development Studio”

3.9.5 轻量级定时器

轻量级定时器是用来扩展定时器组件的一个扩展组件，可以为应用程序提供定期通知。任务可以创建一个队列并定期添加定时器，定时器溢出的周期和队列移动的时间相等，但常会有偏移。

表 3-23. 总结：轻量级定时器的使用

轻量级定时器使用特定的结构和常量，它们在 lwtimer.h 中定义	
_lwtimer_add_timer_to_queue	为定期队列创建一个轻量级定时器
_lwtimer_cancel_period	从队列中移除所有定时器
_lwtimer_cancel_timer	从队列中移除一个定时器
_lwtimer_create_periodic_queue	创建一个可以添加定时器的队列
_lwtimer_test	测试所有队列和定时器

3.9.5.1 启动轻量级定时器

任务通过一个指向 **LWTIMER_PERIOD_STRUCT 结构**的指针调用**_lwtimer_create_periodic_queue()**启动轻量级定时器，这指定了队列周期。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。

当定时器溢出时，定时器指定的通知函数将运行。

3.9.5.2 取消轻量级定时器请求

任务可使用 **LWTIMER_STRUCT** 地址来调用**_lwtimer_cancel_timer()**取消轻量级定时器的请求。也可以使用 **LWTIMER_PERIOD_STRUCT** 地址来调用**_lwtimer_cancel_period()函数**取消

所有轻量级定时器队列中定时器。

3.9.6 看门狗

大多数嵌入式体系有硬件看门狗定时器，如果应用程序不在特定的时间内复位定时器（可能因为发生死锁或其它错误），则硬件会进行复位操作。因此，硬件看门狗是监控整个处理器，而不是监控各个任务的。

Freescale MQX	为了优化目标代码和数据，存储器需要一个目标平台，MQX 内核在默认情况下并不编译看门狗组件。为了测试此功能，你需要在 MQX 用户配置文件中启用它并重新编译 MQX BSP，BSP 和其它核心组件。详细信息请参看第 4.5 节“重建 Freescale MQX RTOS”

MQX 看门狗组件为每个任务提供软件看门狗，如果一个任务崩溃或者超时，那么看门狗将提供一种检测问题的方法。首先，任务用特定时间值启动看门狗，如果在看门狗溢出之前没有中止或复位看门狗，则 MQX 将调用能够恢复错误的函数。

表 3-24. 总结：看门狗的使用

看门狗使用特定的结构和常量，它们在 watchdog.h 中定义	
_watchdog_create_component	创建看门狗组件
_watchdog_start	启动或复位看门狗（毫秒级）
_watchdog_start_ticks	启动或复位看门狗（ticks 级）
_watchdog_stop	中止看门狗
_watchdog_test	测试看门狗组件

3.9.6.1 创建看门狗组件

在任务使用看门狗之前，应用程序必须调用_watchdog_create_component()函数创建看门狗组件。

3.9.6.2 启动或复位看门狗

可以通过下面函数之一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()**。

如果在看门狗溢出之前任务无法复位或者中止看门狗，MQX 就会调用溢出函数。

3.9.6.3 终止看门狗

任务可以调用**_watchdog_stop()函数终止**看门狗。

3.9.6.4 举例：看门狗的使用

任务可以对周期性定时器中断向量创建看门狗，并指定溢出函数（**handle_watchdog_expiry()**）。然后开启一个将在两秒钟溢出的看门狗。为了防止看门狗溢出，任务必须在两秒钟之内停止或者复位看门狗。

```
/*watchdog.c */
```

```
#include <mqx.h>
#include <bsp.h>
#include <watchdog.h>
#define MAIN_TASK 10
extern void main_task(uint_32);
extern void handle_watchdog_expiry(pointer);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ MAIN_TASK, main_task, 2000, 8, "Main", MQX_AUTO_START_TASK,
0L, 0 },
{ 0, 0, 0, 0, 0, 0,
0L, 0 }
};
/*函数*-------------------------------------------------------
*
*  函数名  : handle_watchdog_expiry
*  返回值  : none
*  注释  :当看门狗中断发生时调用该函数
*结束*-------------------------------------------------------------*/
void handle_watchdog_expiry(pointer 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
(
uint_32 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);
}
}
```

3.9.6.4.1 使用MQX编译和连接应用程序

1. 进入到下面的路径：mqx\examples\watchdog
2. 关于如何创建和运行应用程序，请参考 MQX 的发布说明文档。
3. 根据发布说明文档中的指示运行应用程序。

当看门狗到期时，主任务将打印一信息给输出设备。

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior Development Studio 是优先推荐的集成开发环境，详细信息请参看第 3.3 节“使用 Freescale CodeWarrior Development Studio”。

3.10 中断和异常处理

MQX 使用中断服务例程(ISR)来处理硬件中断和异常。ISR 并不是一个任务，而是一个能快速响应硬件中断和异常事件的高速的短例程。ISR 通常是用 C 语言编写的。ISR 的任务包括：

- 维护设备
- 清除错误环境
- 指示一个任务

当 MQX 调用一个 ISR 时，它将传递一个由应用程序定义的参数，然后应用程序安装 ISR。例如，这个参数可能是指向一个具体设备配置结构的指针。

注意：该参数不能是指向任务堆栈的数据，因为这部分内存对于 ISR 可能是不可访问的。

由于中断服务优先级的不同，ISR 可能会遇到禁用的中断。因此，ISR 执行一个最小数量的函数是十分重要的。ISR 通常会使任务处于就绪状态，任务的优先级决定了对来自中断设备信息的处理速度。ISR 有多种方法使得任务处于就绪状态：事件、轻量级信号量、信号量、消息或者任务队列。

MQX 提供了一个用汇编语言编写的 ISR 内核，这个内核会在其它任何 ISR 运行之前运行，并完成如下任务：

- 保护活动任务的现场
- 切换到中断堆栈
- 调用合适的 ISR
- 当 ISR 返回后，恢复具有最高优先级的就绪任务的现场。

当 MQX 启动后，会为所有可能的中断装载默认的 ISR 内核(**_int_kernel_isr()**)。当 ISR 返回到 ISR 内核后，如果此时 ISR 有一个处于就绪状态的、优先级更高的中断任务，ISR 内核就会进行任务调度。这将意味着，前一个活动任务的现场将被存储起来，而更高优先级的任务则成为当前的活动任务。以下图表为 MQX 处理中断过程：

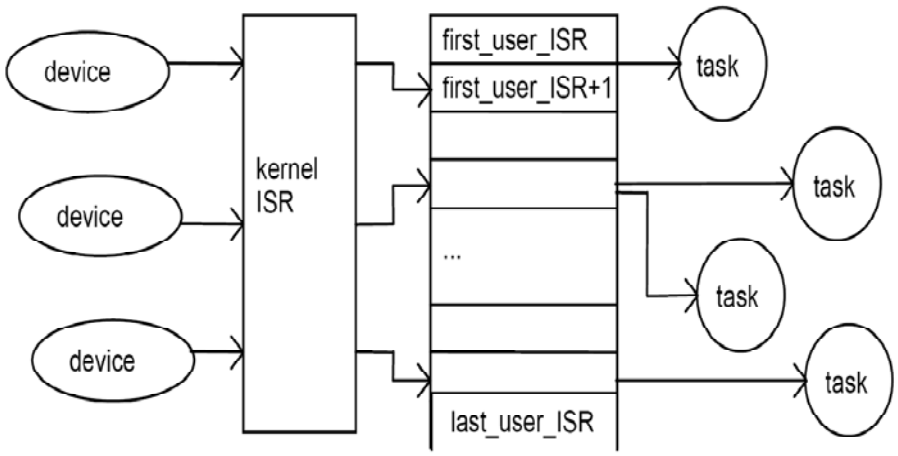


表 3-25 总结：中断和异常处理

_int_disable	禁止硬件中断
_int_enable	使能硬件中断
_int_get_isr	为向量获取ISR
_int_get_isr_data	获取与中断有关的数据指针
_int_get_isr_depth	获取当前ISR的嵌套深度
_int_get_kernel_isr	为中断获取ISR核
_int_get_previous_vector_table	当MQX启动时，获取已经存储的中断向量表的指针
_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ISR核
_int_install_unexpected_is	装载_int_unexpected_isr()做为默认ISR核
_int_kernel_isr	默认ISR核
_int_set_isr_data	设置与特定的中断相关的数据
_int_set_vector_table	改变向量表位置

3.10.1 中断处理初始化

当 MQX 启动后，首先将初始化其 ISR 表，此表中包含了每个中断的入口，入口组成如下：

- 指向 ISR 的指针
- 数据，用于 ISR 参数传递
- 用于 ISR 异常处理的指针

最初，每个入口的 ISR 都是默认的 **ISR _int_default_isr()**，调用此函数可以阻塞当前的活动任务。

3.10.2 装载应用程序定义的ISR

当中断产生时，MQX 将调用**_int_install_isr()函数**，使用应用程序定义的、面向特定中断的

ISR 代替默认 ISR。应用程序必须在初始化设备之前完成这个替换。

_int_install_isr()的参数包括为：

- 中断号
- 指向 ISR 函数的指针
- ISR 数据
- 应用程序定义的 ISR 通常会指向一个任务，指定方法有：

——设置一个事件位(**_event_set()**)。

——传递一个轻量级信号量 (**_lwsem_post()**)。

——传递一个非严谨的信号量(**_sem_post()**)。

——向消息队列发送一个消息。ISR 也可以从系统消息队列中收到消息(**_msgq_send** family)。

提示：从 ISR 分配消息的最有效的方法是使用_msg_alloc()函数。

- 从任务队列中撤销一个任务，放入就绪任务队列中。(**_taskq_resume()**).

3.10.3 针对ISR的限制

下表包含了针对 ISR 限制信息：

3.10.3.1 ISR不能调用的函数

如果 ISR 调用下面任何一个函数，MQX 都将返回错误。

组件	函数
Events	**_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 family** **_event_wait_any family**
Lightweight events	**_lwevent_destroy()** **_lwevent_test()** **_lwevent_wait family**
Lightweight logs	**_lwlog_create_component()**
Lightweight semaphores	**_lwsem_test()** **_lwsem_wait()**
Logs	**_log_create_component()**
Messages	**_msg_create_component()** **_msgq_receive family**
Mutexes	**_mutex_create_component()**

	_mutex_lock()
Names	**_name_add()** **_name_create_component()** **_name_delete()**
Partitions	**_partition_create_component()**
Semaphores	**_sem_close()** **_sem_create()** **_sem_create_component()** **_sem_create_fast()** **_sem_destroy()** **_sem_destroy_fast()** **_sem_post() (for strict semaphores only)** **_sem_wait family**
Task queues	**_taskq_create()** **_taskq_destroy()** **_taskq_suspend()** **_taskq_suspend_task()** **_taskq_test()**
Timers	**_timer_create_component()**
Watchdogs	**_watchdog_create_component()**

3.10.3.2 ISR不应该调用的函数

ISR 不应该调用如下函数，否则 MQX 可能阻塞或长时间运行：

- **most functions from the _io_ family**
- **_event_wait family**
- **_int_default_isr()**
- **_int_unexpected_isr()**
- **_klog_display()**
- **_klog_show_stack_usage()**
- **_lwevent_wait family**
- **_lwsem_wait family**
- **_msgq_receive family**
- **_mutatr_set_wait_protocol()**
- **_mutex_lock()**
- **_partition_create_component()**
- **_task_block()**
- **_task_create() and _task_create_blocked()**
- **_task_destroy()**
- **_time_delay family**

3.10.4 修改默认ISR

当 MQX 处理中断时，它将调用**_int_kernel_isr()**，如果以下条件都成立，则该函数也称为默认 ISR：

- 应用程序没有为中断装载一个应用程序定义的 ISR。
- 该中断在 ISR 表范围之外。

利用**_int_get_default_isr()函数**，应用程序可以获得指向默认 ISR 的指针。

应用程序可以调用下表的函数来修改 ISR：

默认 ISR	描述	移除或装载
_int_default_isr	MQX 启动后，装载默认 ISR	移除 **_int_install_default_isr()**
_int_exception_isr	实现 MQX 异常处理	装载 **_int_install_exception_isr()**
_int_unexpected_isr	与**_int_default_isr()**类似，但是向控制台发送消息，确认未处理中断	装载 **_int_install_unexpected_isr()**

3.10.5 异常处理

为了进行异常处理，应用程序必须调用**_int_install_exception_isr()函数**，该函数将会装载**_int_exception_isr()**作为默认的 ISR。因此，当有异常或者未处理的中断产生时，MQX 就会调用**_int_exception_isr()**。当异常发生时，函数**_int_exception_isr()**执行情况如下：

- 当一个任务正在运行而且任务异常 ISR 存在时，如果异常发生，MQX 将运行 ISR；如果任务异常 ISR 不存在，MQX 则将通过调用**_task_abort()**终止该任务。

•当一个 ISR 正在运行而且 ISR 异常 ISR 存在时，MQX 将终止这个正在运行的 ISR 并启动异常处理 ISR。

- 这个函数将沿着中断堆栈寻找在异常发生前运行的 ISR 或者任务。

注意：如果 MQX 异常 ISR 的确定中断堆栈可能包含不正确的信息，它将用错误代码 **MQX_CORRUPT_INTERRUPT_STACK** 调用**_mqx_fatal_error()**。

3.10.6 ISR异常处理

应用程序可以为每一个 ISR 装载 ISR 异常处理程序。当正在运行的 ISR 发生异常时，MQX 将调用异常处理程序并中止ISR。如果应用程序没有装载异常处理程序，则MQX将简单终止ISR。

当 MQX 调用异常处理器时，首先验证：

- 当前 ISR 号
- ISR 的数据指针
- 异常号
- 异常框架堆栈的地址

表 3-26. 总结: ISR 异常处理

_int_get_exception_handler	为 ISR 获取当前异常处理程序指针
_int_set_exception_handler	为中断设置当前异常处理程序地址

3.10.7 任务异常处理

当任务出现异常时，MQX 将装载任务异常处理程序。

表 3-27. 总结: 任务异常处理

_task_get_exception_handler	获取任务异常处理程序
_task_set_exception_handler	设置任务异常处理程序

3.10.8 举例：装载ISR

为了拦截内核定时器中断，需要装载一个 ISR，并连接 ISR 到上一个 ISR，也即由 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(uint_32);
extern void new_tick_isr(pointer);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ MAIN_TASK, main_task, 2000, 8, "Main",
MQX_AUTO_START_TASK, 0L, 0 },
{ 0, 0, 0, 0, 0,
0, 0L, 0 }
};
typedef struct
{
pointer OLD_ISR_DATA;
void (_CODE_PTR_ OLD_ISR)(pointer);
_mqx_uint TICK_COUNT;
} MY_ISR_STRUCT, _PTR_ 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
(
pointer 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
(
uint_32 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);
}
```

3.10.8.1 利用MQX编译和连接应用程序

1. 进入下面的路径：mqx\examples\isr
2. 关于如何创建和运行应用程序，请参考 MQX 的发布说明文档。
3. 根据发布说明文档的指示运行应用程序。
   主任务将会显示调用应用程序 ISR 的次数。

Freescale MQX	对于 Freescale MQX 来说，CodeWarrior Development Studio 是优先推荐的集成开发环境，详细信息请参看第 3.3 节“使用 Freescale CodeWarrior Development Studio”

3.11 工具

可用的工具包括如下：

- 日志
- 轻量级日志
- 内核级日志
- 堆栈使用工具

3.11.1 日志

许多实时应用程序都需要记录重要信息，例如事件、状态转换、或功能的进入和退出。如果应用程序记录了所有发生的信息，你可通过分析这些信息的序列来确定程序处理是否正确。如果每条信息都有一个时间戳，你便可以进一步确定，程序在哪里花费了处理时间，哪些代码应需优化。

Freescale MQX	为了在一些目标平台下优化目标代码和数据内存需求，MQX 内核在默认情况下并不编译日志组件。为了测试此功能，你需要在 MQX 用户配置文件中启用它并重新编译 MQX BSP，BSP 和其它核心组件。详细信息请参看 4.5 节。

利用该日志组件，你可以将数据存储到16个长度的日志中，并进行进一步检索。每个日志都有预定的条目数限制，每一条目都有一个时间戳（绝对时间）、一个序列号和程序定义的数据。

表 3-28. 总结：使用日志

日志使用特定的结构和常量，它们在 log.h 中定义	
_log_create	创建日志
_log_create_component	创建日志组件
_log_destroy	销毁日志
_log_disable	禁用记录日志
_log_enable	启用记录日志
_log_read	读日志
_log_reset	重置日志内容
_log_test	测试日志组件
_log_write	写日志

3.11.1.1 创建日志组件

你可以利用**_log_create_component()来**明确地创建日志组件，如果你没有创建它，则 MQX 将在应用程序中创建日志或内核日志时默认地创建它。

3.11.1.2 创建日志

任务可以调用**_log_create()**函数创建一个日志，并要求：

- 日志号，范围必须介于 0～15。
- **_mqx_uint** 的最大数值将存储在日志中（包括标题）。
- 当日志满的时候如何处理。默认做法是额外数据将不能被写入；另一个可选做法是新的条目覆盖旧的。

3.11.1.3 格式化日志条目

每个日志条目包含一个日志标题**(LOG_ENTRY_STRUCT)**，其后紧跟应用程序定义的数据。

```
typedef struct
{
_mqx_uint SIZE;
_mqx_uint SEQUENCE_NUMBER;
uint_32 SECONDS;
uint_16 MILLISECONDS;
uint_16 MICROSECONDS;
} LOG_ENTRY_STRUCT, _PTR_ LOG_ENTRY_STRUCT_PTR;
```

MQX 参考文档中详细描述了该字段。

3.11.1.4 写日志

调用**_log_write()**函数可以写日志。

3.11.1.5 读日志

通过调用**_log_read()**来读取日志，并规定如何读取日志，可选的方法有：

- 读最新的条目
- 读最旧的条目
- 读下一个条目（常用于“读最旧的条目”）
- 读最旧的条目并删除它

3.11.1.6 禁用和启用记录日志

任何一个任务都可以通过**_log_disable()**来禁用一个具体的日志，并通过调用**_log_enable()**来启用一个具体的日志。

3.11.1.7 重置日志

通过调用**_log_reset()函数**重置一个日志。

3.11.1.8 举例：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(uint_32 initial_data);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ MAIN_TASK, main_task, 2000, 8, "Main",
MQX_AUTO_START_TASK, 0, 0},
{ 0, 0, 0, 0, 0,
0, 0, 0 }
};
typedef struct entry_struct
{
LOG_ENTRY_STRUCT HEADER;
_mqx_uint C;
_mqx_uint I;
} ENTRY_STRUCT, _PTR_ ENTRY_STRUCT_PTR;
/*TASK*-------------------------------------------------------------
*
* Task Name : main_task
* Comments :
* This task logs 10 keystroke entries then prints out the log.
*END*-------------------------------------------------------------*/
void main_task
(
uint_32 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);
}
```

3.11.1.8.1 利用MQX编译和连接应用程序

1. 进入下面的路径：mqx\examples\log
2. 关于如何创建和运行应用程序，请参考 MQX 的发布说明文档。
3. 根据发布说明文档的指示运行应用程序。

4. 向输入控制台键入 10 个字符。

该程序记录这些字符，并在控制台显示日志。

3.11.2 轻量级日志

轻量级日志与日志类似，但有如下不同点：

- 所有轻量级日志的条目都等长。
- 可以在存储器某特定位置创建轻量级日志。
- 轻量级日志可以时钟嘀嗒/秒或毫秒为单位加时间戳，这取决于 MQX 在编译时如何配置(更多信息请参照 3.14 节“MQX 实时编译配置”)。

表 3-29. 总结：使用轻量级日志

轻量级日志使用特定的结构和常量，它们在 lwlog.h 中定义	
_lwlog_calculate_size	计算一个轻量级日志所需的最大条目的数量
_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	销毁轻量级日志
_lwlog_disable	禁止记录轻量级日志
_lwlog_enable	使能记录轻量级日志
_lwlog_read	读轻量级日志
_lwlog_reset	重置轻量级日志
_lwlog_test	测试轻量级日志
_lwlog_write	写轻量级日志

3.11.2.1 创建轻量级日志组件

可以调用**_lwlog_create_component()**函数明确地创建日志组件，如果你没有创建它，则 MQX 将在应用程序创建日志或内核日志时默认地创建它。

3.11.2.2 创建轻量级日志

任务可以利用**_lwlog_create_at()**在特定位置创建轻量级日志，或者让 MQX 来选择创建位置(**_lwlog_create()**)。

上述两种功能均规定：

- 日志号的范围为 1～15（0 保留给内核日志）。
- 日志中条目的最大编号。
- 当日志满的时候怎样做。默认做法是没有额外数据被写入；另一种可选的做法是添加新的条目并覆盖旧条目。

调用**_lwlog_create_at()**函数，任务还可以进一步指定日志地址。

3.11.2.3 格式化轻量级日志条目

每个轻量级日志都有如下结构：

```
typedef struct lwlog_entry_struct
_mqx_uint SEQUENCE_NUMBER;
#if MQX_LWLOG_TIME_STAMP_IN_TICKS == 0
/* Time at which the entry was written: */
uint_32 SECONDS;
uint_32 MILLISECONDS;
uint_32 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 _PTR_ NEXT_PTR;
} LWLOG_ENTRY_STRUCT, _PTR_ LWLOG_ENTRY_STRUCT_PTR;
```

MQX 的参考文件中描述了这个字段。

3.11.2.4 写轻量级日志

利用**_lwlog_write()函数**来写日志。

3.11.2.5 读轻量级日志

通过**_lwlog_read()函数**来读取日志，并规定如何读取日志，可选方法有：

- 读最新的条目
- 读最旧的条目
- 读下一个条目（应用于“读最旧的条目”）
- 读最旧的条目并删除它

3.11.2.6 禁用和启用轻量级日志

任何一个任务都可以通过**_lwlog_disable()**来禁用一个具体的日志，或通过**_lwlog_enable()**来启用一个具体的日志。

3.11.2.7 重置轻量级日志

通过**_lwlog_reset()函数**重置一个具体的日志。

3.11.2.8 举例：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(uint_32 initial_data);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
```

```
{
{ MAIN_TASK, main_task, 2000, 8, "Main",
MQX_AUTO_START_TASK, 0, 0},
{ 0, 0, 0, 0, 0,
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
(
uint_32 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.");
}}
```

3.11.2.8.1 使用MQX编译并连接应用程序

1. 进入目录 mqx\examples\lwlog
2. 关于如何编译和运行的指令，请 参考 MQX 的发布说明文档
3. 根据说明文档的指令运行应用程序
4. 通过输入控制台键入十个字符

程序会将输入的字符记录到轻量级日志，并通过控制台显示。

3.11.3 内核日志

内核日志允许应用程序日志包括以下部分：

- 用于实现 MQX 功能所调用函数的进入和退出信息
- 用于实现特殊功能所调用函数的进入和退出信息
- 现场切换
- 中断

Freescale MQX	在一些目标平台上，为了实现优化代码和数据存储的需要，默认情况下内核日志在 MXQ 内核是不进行编译的。为了测试这个特性，需要在用户配置的第一个文件中启用它，并且还需要编译 MQX PSP，BSP，和其它核心组件。更多内容参考 4.5 节“重建 Freescale MQX RTOS”。

性能工具通常使用内核日志来分析一个应用程序如何操作以及如何使用资源的问题，更多详细信息请参考 MQX 主机工具用户指南。

内核日志需要用到以下数据结构和内容，这些都定义在 log.h lwlog.h 和 klog.h 文件中	
_klog_control	控制内核日志
_klog_create	创建内核日志
_klog_create_at	在一个确定的位置创建内核日志
_klog_disable_logging_task	对特殊任务关闭内核日志
_klog_enable_logging_task	对特殊任务启用内核日志
_klog_display	在内核日志中显示一个实体

3.11.3.1 使用内核日志

为了使用内核日志，应用程序需要经过下列步骤

1. 可以选择创建如 3.11.2 节所描述的轻量级日志组件

2. 通过**_klog_create()**语句创建内核日志，这和在 3.11.2 节创建轻量级日志相类似。也可以通过**_klog_create_at()**在一个确定的位置创建内核日志。

3. 调用**_klog_control()**建立对日志的控制，并确定具体的位标志组合，描述如下：

可选标志位：

MQX组件 可选：	被记录的功能：
Errors	例如：**_mqx_exit(),_task_set_error(),_mqx_fatal_error()**
Events	大部分来自**_event** 家族
Interrupts	一部分来自于**_int** 家族
LWSems	**_lwsem** 家族
Memory	一部分来自于**_mem** 家族
Messages	来自于**_msg,_msgpool** 和**_msgq** 家族
Mutexes	来自于**_mutatr** 和**_mutex** 家族
Name	**_name** 家族
Partitions	**_partition** 家族
Semaphores	大部分来自**_sem** 家族
Tasking	**_sched,_task,_taskq,_time** 家族
Timing	**_timer** 家族，一部分来自**_time** 家族
Watchdogs	**_watchdogs** 家族
具体任务（该任务有资格记录内核日志），对于每一个要求记录日志的任务，可以调用以下函数：**_klog_disable_logging_task(),_klog_enable_logging_task()**	
中断	
周期计时器中断（系统时钟）	
现场切换	

3.11.3.2 禁用内核日志

内核日志可以使你利用更多的资源，但是同时使你的系统运行很慢。当测试和验证应用程序之后，你最好使用一个不记录内核日志功能的版本。如果想要去掉 **MQX** 任何部分的内核日志功能，必须重新编译 MQX，并且将 **MQX_KERNEL_LOGGING** 设置为 0。更多的参考信息见 3.14.1 节（MQX 实时编译配置选项）。重编译 MQX 的完整程序见 4.5 节“重建 Freescale MQX RTOS”。

3.11.3.3举例：内核日志的应用

该例子将记录所有的定时器组件和所有定期定时中断。

```
/* klog.c */
#include <mqx.h>
#include <bsp.h>
#include <log.h>
#include <klog.h>
extern void main_task(uint_32 initial_data);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ 10, main_task, 2000, 8, "Main",
MQX_AUTO_START_TASK, 0, 0},
{ 0, 0, 0, 0, 0,
```

```
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
(
uint_32 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);
}
```

3.11.3.3.1 编译应用程序并将其与MQX链接

1. 进入目录：mqx\examples\klog
2. 关于如何编译和运行应用程序，请参考 MQX 的发布说明文档。
3. 根据说明文档中的指令运行应用程序。
   三秒之后，**Main_task()**将显示内核日志的内容。

3.11.4堆栈使用工具

MQX 提供核心工具以检查和重定义中断堆栈和每个任务堆栈的大小。

表 3-32 总结：堆栈使用工具

为了使用这些工具，你必须已经配置过 MQX 的 MQX_MONITOR_STACK。更多内容参见 3.14.1 节“MQX 实时编译配置选项”，重编译 MQX 的完整程序见 4.5 节“重建 Freescale MQX RTOS”。	
_klog_get_interrupt_stack_usage	获得中断堆栈的边界和已经使用的堆栈大小
_klog_get_task_stack_usage	获得一个特定任务的堆栈大小和已经使用的堆栈大小
_klog_show_stack_usage	计算并显示每个任务和中断堆栈已经使用的空间大小

3.12 工具

工具包括：

- 队列
- 命名组件
- 实时测试
- 其它工具

3.12.1队列

队列组件可以用于管理双向元素链表。

Freescale MQX	在一些目标平台上，为了优化代码和数据存储的需要，默认情况下队列组件在 MQX 内核中是不进行编译的。为了测试这个特性，需要在用户配置的第一个文件中启用它，并且还需要编译 MQX PSP，BSP，和其它核心元件。更多内容参考 4.5 节“重建 Freescale MQX RTOS”。

表 3-33.总结：使用队列

_queue_dequeue	出队列，删除队列开始处的元素
_queue_enqueue	在队列的末尾添加元素
_queue_get_size	获得队列中元素的个数
_queue_head	获取（但不删除）队列开始处的元素

_quedu_init	初始化队列
_queue_insert	添加元素到队列
_queue_is_empty	判断队列是否为空
_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	从队列中移除指定元素

3.12.1.1 队列的数据结构

队列组件需要两个数据结构，它们都包含在文件 *mqx.h* 中：

- **QUEUE_STRUCT——**记录队列中元素的个数、指示队列的队列头和队列尾的位置。当一个任务创建一个队列时，**MQX** 初始化该结构。
- **QUEUE_ELEMENT_STRUCT——**定义队列中元素的数据结构。该结构是一个应用程序的头结构所定义的对象，这个对象正是该任务要加入到队列中的元素。

3.12.1.2 创建队列

任务通过调用**_queue_init()**来创建和初始化队列，创建一个指向该队列的指针并且初始化该队列的最大长度。

3.12.1.3 向一个队列增加元素

一个任务通过调用**_queue_enqueue()**向队列中添加一个元素，创建一个指向该队列的指针和该队列元素对象，该对象是任务要加入到队列中的头结构。

3.12.1.4 从队列中移除元素

一个任务通过调用**_queue_dequeue()**从队列开始处得到和删除一个元素，并且删除该对象中的指针。

3.12.2 命名组件

通过命名组件，任何一个任务可以关联一个 32-bit 字符串或数字的名字。MQX 将这种名字关联存储在数据库中，以便于处理器上的所有任务都可以使用。而数据库中尽量避免使用全局变量。

Freescale MQX	在一些目标平台上，为了优化代码和数据存储的需要，默认情况下队列组件在 MQX 内核中是不进行编译的。为了测试这个特性，需要在用户配置的第一个文件中启用它，并且还需要编译 MQX PSP，BSP，和其它核心元件。更多内容参考 4.5 节 “重建 Freescale MQX RTOS”。

表 3-34.总结：使用命名组件

命名组件所用到的数据结构和内容都定义在 name.h 文件中	
_name_add	向名字数据库中添加名字（名称是 NULL 结尾的字符串，最多 32 个字符，包括 NULL）。
_name_create_component	创建命名组件
_name_delete	从数据库中删除一个名字

_name_find	在数据库中查找一个名字并且得到它的序号
_name_find_by_number	再数据库中查找一个序号并且得到它的名字
_name_test	测试命名组件

3.12.2.1创建命名组件

应用程序可以通过调用**_name_create_component()**来创建命名组件。如果没有创建，MQX 将在应用程序第一次使用命名数据库时默认地创建它。其中使用的参数和默认值与在 3.7.1.1 节讲述的事件组件默认创建类似。

3.12.3 实时测试

MQX 提供了核心的实时测试功能，用来测试 MQX 大部分组件的完整性。

测试能够确定与该组件相关联的数据有效或者没有受到破坏。

如果一个数据结构的 **VALID 字段**是已知值，那么 MQX 将认定以该数据结构创建的数据是有效的。

如果一个数据结构的 **CHECKSUM** 字段或者其指针是不正确的，那么 MQX 将认定以该数据结构创建的数据是不正确的。

应用程序可以在正常运行期间使用实时测试。

表 3-35.总结：实时测试

_event_test	事件
_log_test	日志
_lwevent_test	轻量级事件
_lwlog_test	轻量级日志
_lwmem_test	可变块大小的轻量级存储
_lwsem_test	轻量级信号量
_lwtimer_test	轻量级定时器
_mem_test	可变块大小的存储
_msgpool_test	内存池
_msgq_test	内存队列
_mutex_test	互斥
_name_test	命名组件
_partition_test	固定大小块的存储
_queue_test	应用程序实现的队列
_sem_test	信号量
_taskq_test	任务队列
_timer_test	定时器
_watchdog_test	看门狗

3.12.3.1 举例：实时测试

该应用程序将进行所有的实时测试，一个低优先级的任务进行所有的组件测试。如果发现错

误，应用程序将停止。

```
/* 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(uint_32);
TASK_TEMPLATE_STRUCT MQX_template_list[] =
{
{ 10, background_test_task, 800, 8, "Main",
MQX_AUTO_START_TASK, 0L, 0},
{ 0, 0, 0, 0, 0,
0, 0L, 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
(
uint_32 parameter
)
{
_partition_id partition;
_lwmem_pool_id lwmem_pool_id;
pointer error_ptr;
```

```
pointer 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);
}
}
```

3.12.3.1.1 使用MQX编译与连接应用程序

1. 进入目录 mqx\examples\test
2. 关于如何编译和运行应用程序，请参考 MQX 的发布说明文档。
3. 根据说明文档给出的指令运行程序。

3.12.4 其它工具

表 3-36.总结：其它工具

_mqx_bsp_revision	BSP 版本
_mqx_copyright	返回 MQX 版权字符串指针
_mqx_date	返回 MQX 创建日期字符串指针
_mqx_fatal_error	检测到错误，该错误是非常严重的以至于 MQX 或者应用程序不能正常运行
_mqx_generic_revision	修订 MQX 代码的版本
_mqx_get_counter	获得处理器 32 位唯一序列号
_mqx_get_cpu_type	获得处理器类型
_mqx_get_exit_handler	获得 MQX 结束句柄指针，当 MQX 退出时调用
_mqx_get_kernel_data	获得执行内核数据的指针
_mqx_get_system_task_id	获得任务描述符的描述 ID
_mqx_get_tad_data	获得任务描述符的 **TAD_RESERVED** 字段
_mqx_idle_task	空闲任务
_mqx_io_revision	BSP IO 版本
_mqx_monitor_type	监控器类型

_mqx_psp_revision	PSP 版本
_mqx_set_cpu_type	设置处理器类型
_mqx_set_exit_handler	设置处理器退出句柄，当 MQX 结束时调用
_mqx_set_tad_data	设置任务描述符 TAD_RESERVED 字段
_mqx_version	返回描述 MQX 版本字符串的指针
_mqx_zero_tick_struct	恒定零初始化时钟结构，应用程序可以使用它来初始化它的时钟结构归零
_str_mqx_uint_to_hex_string	将一个_mqx_uint 数据转换为 16 进制字符串
_strnlen	计算定长字符串的长度

3.13 嵌入式调试

有如下几种方法调试基于 MQX 的应用程序：

- 使用简单调试环境，该方法不关心所使用的 MQX 操作系统。当在应用程序代码使用断点和单步调试时，该方法有很好的效果。
- 使用操作系统相关的调试器（称之为任务相关调试器，TAD）。这种方法有助于查看个别任务的调试代码，还有助于以用户友好的方式查看 MQX 内部数据结构。
- 在目标代码中使用 EDS 服务器和 EDS 客户端（Freescale MQX 远程调试工具）。该工具通过串口、TCP/IP 或者其它的通信接口链接到目标系统，提供和 TAD 类似的信息。

Freescale MQX	更多关于 Freescale MQX 远程调试工具的信息，请参考 Freescale MQX 主机用户手册。 Freescale MQX 不支持的 IPC 组件（原 EDS 公司通讯所必需的），而是使用 EDSerial 或 TCP / IP 服务器。

3.14 实时编译配置MQX

MQX 可以同某些功能一起编译，你可以通过改变实时编译配置选项值来包含或者移除这些功能。如果改变了某些配置的值，必须重编译 MQX 并将它与目标应用程序连接。

由于 BSP 库文件也可能依赖于一些 MQX 配置选项，因此它也必须被重新编译。

和 BSP 相类似，也有其它的代码组件使用 MQX 操作系统服务，例如，RTCS、MFS、USB，因此在 MQX 和 BSP 之后也要对这些组件重新编译。

Freescale MQX	和以前的 ARC 版本相比较，Freescale 提供了一种不同的方法来编译 MQX OS 和其它组件的编译时配置功能。 原来的方法使用编译器的命令行 –D 选项或 *source\psp\platform\psp_cnfg.asm* 文件作为前缀包含文件。不过这种方法现在已经过时了。 在 Freescale MQX 中，还有一部分用户使用 *config\board* 目录下的配置文件，这个文件可用于覆盖默认配置选项。 同样的配置文件可以用于其它的系统组件，如 RTCS、MFS

	或 USB 中。

3.14.1 MQX编译时配置选项

这节将提供一个配置时的选项清单。所有这些选项的默认值都会被 *config\board\user_config.h* 文件内相应的值所覆盖。

这些默认值定义在 *mqx\source\include\mqx_cnfg.h* 文件中。

注意：不要试图直接更改 *mqx_cnfg.h* 文件。请使用在配置目录中特定项目或者特定平台的 *user_config.h* 文件。

MQX_CHECK_ERRORS

默认值：1

1：MQX 组件执行所有的参数错误检查

MQX_CHECK_MEMORY_ALLOCATION_ERRORS

默认值：1

1：MQX 组件检查所有的内存分配错误，并验证所有的分配都是正确的。

MQX_CHECK_VALIDITY

默认值：1

1：当需要访问时，MQX 检查所有结构的 VALID 字段。

MQX_COMPONENT_DESTRUCTION

默认值：1

1：使 MQX 具有这样的功能：将 MQX 组件如信号量组件或事件组件销毁。MQX 收回分配给这些组件的所有资源。

MQX_DEFAULT_TIME_SLICE_IN_TICKS

默认值：1

1：在任务模板结构中默认的时间分片单位是 tick

0：在任务模板结构中默认的时间分片单位是毫秒

这个值也影响任务模板中的 time-slice 字段，因为这个值用来设定任务的默认时间分片

MQX_EXIT_ENABLED

默认值：1

1：MQX 包含代码允许应用程序从_mqx()中返回

QX_HAS_TIME_SLICE

默认值：1

1：MQX 包括代码允许时间片调度

MQX_INCLUDE_FLOATING_POINT_IO

默认值：0

0：_io_printf（）和_io_scanf（）包括浮点 I / O 代码。

MQX_IS_MULTI_PROCESSOR

默认值：1

1：MQX 允许代码支持多处理器 MQX 应用

MQX_KERNEL_LOGGING

默认值：1

1：每个组件中的某些功能被写入到内核日志。当进入或者退出这些功能时，该设置会降低性能，当且仅当你启用组件的日志记录时。可以通过_klog_control()来控制哪一个组件被记录到日志中。

MQX_LWLOG_TIME_STAMP_IN_TICKS

默认值：1

1：轻量级记录时间戳是 ticks

0：时间戳是秒，毫秒，微妙

MQX_MEMORY_FREE_LIST_SORTED

默认值：1

1：MQX 通过地址来排序内存中的空闲空间列表。这可以减少内存碎片，但是增加了 MQX 释放内存的时间

MQX_MONITOR_STACK

默认值：1

1：MQX 通过一个已知的值来初始化任务和中断堆栈，因此 MQX 组件和调试器能够计算出所有堆栈空间的大小。仅当 MQX 创建一个任务时，这个设置将降低性能。必须使用以下函数之一来设置该选项：

— **_klog_get_interrupt_stack_usage()**

— **_klog_get_task_stack_usage()**

— **_klog_show_stack_usage()**

MQX_MUTEX_HAS_POLLING

默认值：1

1：MQX 可以支持互斥选项 **MUTEX_SPIN_ONLY** 和 **MUTEX_LIMITED_SPIN**。

MQX_PROFILING_ENABLE

默认值：1

1：支持将外部工具编译到 MQX，分析增加了编译映像的大小，也使得 MQX 运行速度较慢。当且仅当所使用的工具集支持分析时，可以分析这个工具。字段值必须和在 *source\psp\psp\psp_cnfg.comp* 文件中的名字一样。

MQX_RUN_TIME_ERR_CHECK_ENABLE

默认值：0

0：支持将外部实时错误检查工具编译到 MQX。这增加了编译映像的大小，也减慢了 MQX 的运行速度。当你所使用的工具集支持该功能时，可以使用实时错误检查。该定义的取值必须和文件 *source\psp\psp\psp_cnfg.comp* 同名的取值一致。

MQX_TASK_CREATION_BLOCKS

默认值：1。该选项仅用于多处理器应用。

1：当调用**_task_create()**时，任务块在另一个处理器上创建一个任务。直到新任务创建完并且返回错误代码时，才创建任务块。

MQX_TASK_DESTRUCTION

默认值：1。

1：MQX 可以将任务终止。因此，MQX 将允许代码释放所有 MQX 管理的资源，这些资源是被终止任务所拥有的。

MQX_TIMER_USES_TICKS_ONLY

默认值：0。

0：定时器任务处理周期性定时器和一次性定时器以 tick 而不是以秒或者毫秒来请求超时报告。

MQX_USE_32BIT_MESSAGE_QIDS

默认值：0。

0：消息组件数据类型（**_queue_number** 和**_queue_id**）是 uint_16

1：消息组件数据类型（**_queue_number** 和**_queue_id**）是 uint_32。这允许一个处理器上有 256 个消息队列，同时允许在一个多处理器的系统中允许有多于 256 个处理器。

MQX_USE_32BIT_TYPES

默认值：0。

0：MQX 被迫处于 32 位模式，而不管它本身是多少位的处理器。当你用 2.5 版本的 MQX 编译链接 2.4 版本的应用程序时，这将减少编译时产生的警告。

MQX_USE_IDLE_TASK

默认值：1。

1：当 MQX 初始化时创建空闲任务

0：当 MQX 初始化时不创建空闲任务

MQX_USE_INLINE_MACROS

默认值：1。

1：一些为 MQX 提供的功能将函数转换为行内代码，这将增加 MQX 的速度。

0：MQX 将优化代码大小

MQX_USE_LWMEM_ALLOCATOR

默认值：0。

0：对_mem 函数集的调用将替换为_lwmem 相应函数的调用。

3.14.2 推荐设置

这些实时编译配置选项的设置取决于你自己应用程序的需求。

Freescale MQX	在 MQX 构建和编译过程时配置具体目标板（在目录 congfig/<board>/user_config.h 下）。你可能想创建自己的配置，特定于定制版或者具体的应用。关于这一过程的更多信息，请参考 4.6 节。

下表列出了一些常见的设置，可能在你开发应用程序时有更多地使用：

选项	default	debug	speed	size
MQX_CHECK_ERRORS	1	1	0	0
MQX_CHECK_MEMORY	1	1	0	0
_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_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_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
MQX_PROFILING_ENABLE	0	1	0	0
MQX_RUN_TIME_ERR	0	1	0	0
_CHECKING_ENABLE				
MQX_TASK_CREATION	1	1	0	0，1
_BLOCKS (for multiprocessor applications)				
MQX_TASK_DESTRUCTION	1	0，1	0	0
MQX_TIMER_USES_TICKS_ONLY	0	0，1	1	1
MQX_USE_32BIT_MESSAGE_QIDS	1	1	1	1
MQX_USE_32BIT_TYPES	0	0	0	0
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

第四章 重建MQX

4.1 为什么要重建MQX？

如果你进行以下几种操作，则需要重建 MQX：

- 改变编译选项（例如优化级别）。
- 改变 MQX 实时编译配置选项。
- 开发一个新的 BSP（例如增加一个新的输入/输出驱动程序）。
- 合并 MQX 源代码的改变。

注意：我们不建议你修改 MQX 的数据结构。否则，一些 MQX 的主要工具可能会因此而无法正常工作。只有当你对 MQX 非常有经验时才能修改 MQX 的数据结构。而且如果你修改了 MQX 的数据结构，必须重新编译和重建整个 MQX。

重建 MQX 也提供了对 MQX 内核方便地源代码级调试。当使用 MQX 分配的出厂预编译库时，调试器可能会混淆源文件的路径。

4.2 开始之前

在你编译或建立 MQX 之前，必须：

阅读 Freescale MQX 附带的发布说明文档，以获取针对你的目标环境的信息。

在你的目标环境里包括以下工具：

——编译器

——汇编器

——链接器

——库

必须熟悉 MQX 的目录结构和重建指令，它们都在说明文档中进行了说明，并且本章的后续部分也提供了对指令的说明。

Freescale MQX	使用支持 MQX 的开发环境能方便地建立 Freescale MQX，例如：Freescale 公司的 CodeWarrior 集成开发环境是被优先推荐的选择。

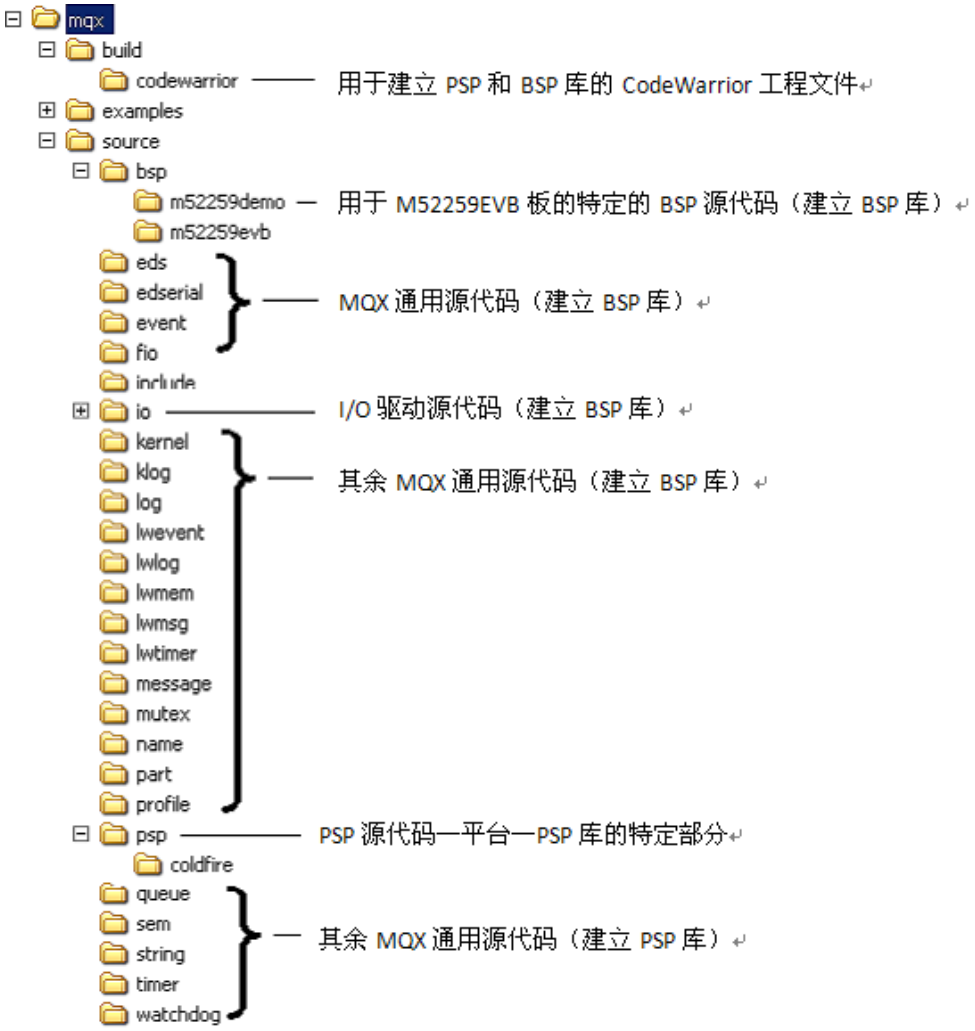
4.3 FreescaleMQX的目录结构

Freescale MQX	除了少数例外，Freescale MQX 的目录结构和早期的 MQX 版本是相同的。主要不同点在于多个组件（MQX、MFS、RTCS、USB…）被安装在同一个顶级目录里，并且它们共享 *config* 和 *lib* 目录。

下图给出了整个 Freescale MQX RTOS 的目录结构：

4.3.1 MQX RTOS目录结构

下图详细地给出了位于 MQX 顶级目录中的 MQX RTOS 部分的目录结构：



4.3.2 PSP子目录

目录 *mqx\source\psp* 中包含了平台支持的 PSP 库代码。例如 ColdFire 子目录包含了 MQX 内核部分，包括 Freescale ColdFire 架构（例如内核初始化，寄存器保存/恢复中断处理，堆栈处理，缓存控制等功能）。此外，此目录中还包含对每个所支持的处理器的定义文件。

4.3.3 BSP子目录

在 *mqx\source\bsp* 中子目录通常按照电路控制板名称来定义，包含了低级启动代码、处理器和电路控制板初始化代码。BSP 还包含数据结构，初始化指定电路控制板各种输入/输出驱动程序。

这些代码（与输入/输出驱动代码一起）编译后形成 BSP 库。

4.3.4 输入/输出（I/O）子目录

在 *mqx\source\io* 中子目录包含 MQX I/O 驱动程序的源代码。通常，每个 I/O 驱动目录中的源文件进一步分为设备相关文件和设备无关文件。适用于给定主板的 I/O 驱动程序是 BSP 工程的组成部分，并且被编译成 BSP 库。

4.3.5 其它源子目录

在所有其它的源目录中包含 MQX RTOS 的通用部分。这些通用源目录连同平台相关的 PSP 代码被编译进 PSP 库。

4.4 FreescaleMQX构建工程

所有必要的工程文件位于 *mqx\build\<compiler>*目录中。每一种电路控制板有两种工程：PSP 工程和 BSP 工程。BSP 工程包含与电路控制板相关的代码。PSP 工程只包含平台相关（如 ColdFire）的代码，它不包含任何与电路控制板相关的代码。尽管如此，这两种工程都在它们的文件名中包含了电路控制板的名称，并且都生成了二进制输出文件存放到同一个与电路控制板相关目录 *lib\<board>.<compiler>*中。

为什么独立于电路控制板的 PSP 库也被编译到与单板计算机相关的输出目录中呢？这只是因为实时编译配置文件来源于该 config\<board>目录。换句话说，即使 PSP 源代码本身并不取决于电路控制板的特性，用户可能需要为不同的电路控制板建立不同的 PSP 库。

4.4.1 PSP构建工程

PSP 工程用于生成 PSP 库，它包含来源于 *mqx\source\psp* 中的平台相关代码，也包含通用的 MQX RTOS 代码。

4.4.2 BSP构建工程

BSP 工程用于生成 BSP 库，它包含来源于 *mqx\source\bsp\<board>*中的与单板计算机相关的代码，也包含来源于 *mqx\source\io* 目录中的选定的输入/输出驱动代码。

4.4.3 构建后处理

所有的构建工程被配置为在顶级目录 *lib\<board>.<compiler>*中创建二进制库文件。例如 M52259 评估板的 CodeWarrior 库文件被生成在目录 *lib\m52259evb.cw* 中。

BSP 工程和 PSP 工程都将执行构建后的批处理文件，从而把所有的公共头文件复制到目标目录 *lib* 中。这使得输出文件夹/ lib 成为 MQX 应用程序代码访问的唯一位置。MQX 应用程序构建工程不需要访问 MQX RTOS 源代码树的任何地方。

4.4.4 构建目标

CodeWarrior 开发环境具有多种构建配置，故称为构建目标。在 Freescale MQX RTOS 中的所有工程包含至少两种构建目标：

调试目标—编译器优化被设置成低以便轻松实现调试。使用这种目标生成的库文件命名时以“_d”为后缀（如 *lib\m52259evb.cw\mqx\mqx_d.a*）。

发布目标—编译器优化被设置成最大值以实现最小的代码量和快速的执行。发布目标代码很难进行调试。生成的库文件名没有任何后缀（如 *lib\m52259evb.cw\mqx\mqx.a*）。

4.5 重建Freescale MQX RTOS

重建 Freescale MQX RTOS 的库文件是一件简单的过程，包括在开发环境中为 PSP 和 BSP 打开正确的工程，并且进一步构建。别忘记选择正确的被建立的构建目标或建立所有的目标。

关于重建 MQX 的具体资料以及相关例子，请参阅 MQX 的发布说明文档。

4.6创建客户MQX配置并构建工程

4.6.1 为什么创建一个新的配置？

当你需要创建一组新的构建工程时，典型的情况包括：

- 想在同一台电路控制板上运行不同的应用程序时，需要两个或多个不同的内核配置。这是一个相当简单的任务。该任务就是“克隆”现有的配置目录，并且修改现有的构建工程（修改名称和输出文件夹）。
- 需要为客户的电路控制板创建一个新的 BSP。这是一个更复杂的任务，并且可能包括一些新的输入/输出驱动程序的开发，或者更改高级的配置。然而，第一步要从现有最相似的 BSP 开始，克隆后指定一个新名称，并作进一步修改。

4.6.2 克隆现有配置

正如前面的章节所描述的，PSP 和 BSP 构建工程（以及其它 MQX 核心组件如 RTCS、MFS 或 USB）都被绑定到目标电路控制板的名称上。以 M52259 评估板为例，以下各项与该单板计算机的名称相关：

- 用户配置取自 *config\<board>*目录（如：*config\m52259evb*）。
- 构建工程的 include-search 路径被设置为指向用户配置目录。
- 构建工程生成的二进制库文件输出目录为 *lib\<board>.<compiler>*。（如：*lib\m52259evb.cw*）。
- 构建工程参照电路控制板的名称来命名 *mqx\build\<compiler>\bsb_<board>.<prj>*（如：*mqx\build\codewarrior\bsp_m52259evb.mcp*）。
- 构建工程中建立的后链接批处理文件名也和电路控制板名相关。（如：

mqx\build\codewarrior\bsp_m52259evb.bat）。

以 M52259 评估板为例，使用 CodeWarrior 构建工具克隆一个现有的配置并且保存为不同名称的步骤如下：

- 复制现存的 *config\m52259evb* 目录，并且指定一个新的特定单板计算机的名称或者特定配置的名称给它。（如：*config\m52259evb_test*）。
- 在 *lib* 文件夹中创建一个新的输出目录（如：*lib\m52259evb_test.cw*）。
- 复制BSP和PSP构建工程（*mqx\build\codewarrior\bsp_m52259evb.mcp* and *psp_m52259evb.mcp*），并给它们都指定一个新名称。
- 打开工程设置窗口，并更改 include-search 路径为引用老的用户配置目录
- 在工程设置窗口中，更改输出文件夹为一个在 *lib* 目录中新创建的文件夹。（*lib\m52259evb_test.cw*）。
- 如果想复制后链接批处理文件，并更改相应的工程设置，这个步骤不是必须的。
- 确认已经完成了对所有可获得的构建目标的工程设置更改（调试目标和发布目标）。
- 必要情况下对其它 MQX 库（如 RTCS、MFS 或 USB）重复以上所有步骤操作。

在将一个新的配置和构建工程准备好之后，你可以开始在不影响原有 BSP 库的情况下修改构建配置。如果你想要创建一个全新的 BSP，你需要创建新的 BSP 源文件，并更改“克隆”BSP 工程的内容。

下一节将帮助你更好地理解新 BSP 开发时面临的挑战。

第五章 开发一个新的BSP

5.1 什么是BSP?

单板计算机支持包是一个与硬件相关的文件集合，与该单板计算机的具体特点相关。你可能想要开发尚未提供的 BSP,此外如果你的目标硬件支持定制,那么同样建议你去开发一个新的 BSP。

在上一节，你已经学习了如何克隆一个现存的 BSP，并为新的硬件配置构建工程。这一节将进一步描述在开发一个新的 BSP 代码时需要记住什么。

5.2 开发一个新的BSP

开发一个新的 BSP 时，你必须按照以下步骤进行：

1. 选择一个基准 BSP 进行修改。
2. 和目标板进行通信。
3. 修改 BSP 特定的包含文件。
4. 修改启动代码。
5. 修改源代码。
6. 创建对 I/O 设备驱动的默认初始化设置。

5.3 选择一个基准BSP然后开始工作

选择一个基准 BSP，并更改以适用于你的硬件，这通常是最简单的方法。在大多数情况下，选择一个使用相同处理器的基准 BSP。

1．创建一个新的目录，如：*source\bsp\mybsp860*

2．进入基准目录，如：*source\bsp\mcfds860*

3．复制基准目录中的所有内容到新目录。

4．在新目录中，更改 *target.**文件名称，将基准 BSP 名称改为新的 BSP 名称。

5．在复制过来的文件中，将所有和基准 BSP 匹配（大写和小写）的名称更改为新的 BSP 名称。

5.4 和目标板通信

为了保证能通过调试器和目标板进行通信,你可能需要修改调试器的初始化文件,例如*.*cfg* 。

5.5 修改BSP特定的包含文件

BSP特有的包含文件保存在*mqx\source\bsp\target*目录中，其中*target*是BSP的名称。

这些文件是：

- *bsp_prv.h*
- *bsp.h*
- *target.h*

5.5.1 bsp_prv.h

该文件包括：

- BSP 使用的私有函数的原型声明。
- BSP 设备的初始化结构原型（在 *source\io*中）。

5.5.2 bsp.h

该文件包含了对以下文件的**#include** 语句，以使应用程序能使用和访问主板的资源和设备：

- 处理器特有的文件
- 设备的.h 文件
- *target.h*

5.5.3 target.h

文件 *target.h* 包括下列单板计算机所特有的定义：

- 单板计算机的型号
- 单板计算机的内存映射，例如内存（如 EPROM 和 RAM）基地址和 I/O 设备地址。
- 周期性定时器中断频率和精度
- 中断的范围，以使应用程序可以安装中断服务程序。
- 包括定时器等设备的中断向量号。
- MQX 初始化结构的默认值。
- 任何其它单板计算机特有的硬件定义，如板卡特有的寄存器。

5.6 修改启动代码

BSP 提供了建立运行环境的默认启动函数，并且通过调用**_mqx()**启动 MQX。你可以用 C 语言(*comp.c*)、汇编语言、或者两者的结合来编写启动函数。

5.6.1 comp.c

文件 comp.c 中的 comp 代表编译器，文件的内容取决于编译器运行时设置的特定要求。

文件包含调用**_mqx()**的代码，它传递一个指针指向 **MQX_init_struct**。与处理器-编译器组合一起，**_mqx()将**直接从 *boot.comp* 中被调用。

5.6.2 boot.comp

对于文件 *boot.comp*，comp 代表单板计算机所使用的编译器，并且 *boot.comp* 文件使用编译器的特定指令。

在某些情况下，BSP 依赖于编译器自身的启动代码。如果是这样的话，*boot.comp* 可能无法退出。

该文件完成以下操作：

- 禁用中断。
- 初始化硬件寄存器。
- 建立一最小的堆栈以被 MQX 用于初始化。
- 在 *comp.c* 中调用一个特定编译器来启动函数。

你需要添加启动代码以替换调试器初始化文件。

5.7 修改源代码

你需要修改以下这些文件，这在后面章节会有描述：

- *init_bsp.c*
- *get_usec.c*
- *get_nsec.c*
- *mqx_init.c*

5.7.1 init_bsp.c

该文件包括：

- 特定板的初始化函数**_bsp_enable_card()**
- 周期性定时器中断函数**_bsp_timer_isr()**
- MQX 退出处理程序**_bsp_exit_handler()**

5.7.1.1 **_bsp_enable_card()**

在初始化过程中，MQX 将调用这个函数完成以下操作：

- 初始化处理器所支持的设施。

PSP 能提供管理 CPU 资源的设施，例如基于 CPU 级的内存或波特率发生器。

- 初始化中断支持。

函数**_psp_int_init()**创建和安装了 MQX 中断表。

- 初始化 cache 和 MMU 并且有选择地启用它们。

PSP 为拥有 caches 和 MMUs 的 CPUs 提供了支持函数。

- 安装和初始化定时器 ISR。
- 安装 I/O 设备驱动程序并初始化 I/O 子系统，此后需要修改此代码。

5.7.1.2 **_bsp_timer_isr()**

该函数是定时器中断函数。它的功能是清中断，必要时还可重启定时器。它调用**_time_notify_kernel()**函数以使 MQX 知道中断的发生。

5.7.1.3 **_bsp_exit_handler()**

当应用程序调用**_mqx_exit()**函数时，该函数被调用。它关闭不再使用的设备。如果单板计算机有一个使用中的设备，则应尝试重新启动调试监视器。如果你使用的是目标监视器，则应添加更多的代码。

5.7.2 get_usec.c

5.7.2.1 **_time_get_microseconds()**

该函数返回自上次定时器中断后的微秒数。如果无法计算自上次定时器中断后的时间，则函数会返回 0。

只有当你使用一个不同的计时器时才修改该函数，在这种情况下，调用它的**_timer_get_usec** 函数。

5.7.3 get_nsec.c

5.7.3.1 **_time_get_nanoseconds()**

该函数返回自上次定时器中断后的纳秒数。如果无法计算自上次定时器中断后的时间，则函数会返回 0。

只有当你使用一个不同的计时器时才修改该函数，在这种情况下，调用它的_timer_get_nsec 函数。

5.7.4 mqx_init.c

该函数包含单板计算机默认的 MQX 初始化结构，简单的应用程序或使用默认值（在 *target.h* 中定义）的应用程序就无须定义一个初始化结构。一个应用程序能够创建一个新的 MQX 初始化结构，该结构使用一些默认值和其它的给定值。

5.8 为I/O驱动程序创建默认的初始化文件

当包含**_bsp_enable_card()**函数的 I/O 驱动程序被安装时，一些初始化文件可能需要给定默认信息。

5.8.1 initdev.c

文件名中的 *dev* 表示设备。该文件为特定的 I/O 驱动程序提供了默认的初始化信息。

5.9 支持编译器的文件

一些文件可能需要为编译器、链接器和 MQX 编译环境提供支持信息。下面列出这些文件的一部分：

文件名	使用	内容

Link.comp	链接器	程序代码和数据的默认位置；可修改
**.cfg* **.met* **.txt*	调试器	初始化信息

5.10 构建新的BSP

有关详细说明，请参看 5.2 节。该节还介绍了批处理文件和 make 文件，你可能想以它们作为模型构建新的 BSP。

在你创建或修改了所有新的 BSP 文件之后，你必须做如下几点：

如果你更改其它文件名或增加新的文件，请修改 *bsp.mk*（它编译所有在 *source\target*目录中修改过的文件）。

在 *build* 目录中通过运行 make 来构建 BSP。

第六章 FAQs

6.1 概述

我的应用程序终止了，这时我该如何判断 MQX 仍在运行？

在更新时间时，MQX 正在处理周期性定时器中断。如果空闲任务正在运行，则可以判断出 MQX 仍在运行。

6.2 事件

两个任务使用一个事件组。连接只为一个任务服务，并不为另一个服务，为什么？

两个任务可能共享同一个全局连接，而不是它们自己的局部连接。每个任务必须调用**_event_open()**或**_event_open_fast()**以获取自己的连接。

6.3 全局构造

我需要在调用“main”之前初始化一些全局构造，它们使用了“new”运算符。也就是说，在我启动 MQX 之前，“new”运算符调用 **malloc()**函数，该函数被重定义为调用 MQX 函数**_mem_alloc()**。我该如何实现呢？

用函数**_bsp_enable_card()**（定义在 *init_bsp.c* 中）初始化构造，该函数在初始化了内存管理组件后由 MQX 调用。

6.4 空闲任务

当有空闲任务块由于某一异常而阻塞，这时发生了什么？

空闲任务块阻塞时，系统任务成为活动任务，因为系统任务实际上是一个没有代码的系统任务描述符。系统任务描述符会设置中断堆栈，然后重新启动中断，因此，应用程序能够继续执行。

6.5 中断

一个中断会定期发生并且我的应用程序必须非常快地响应——比 MQX 允许的速度还要快。我该怎么做？

调用**_int_install_kernel_isr()**函数代替内核中断服务程序（**_int_kernel_isr()**）。你所替换的中断服务程序必须：

- 在入口保存所有寄存器，并在出口恢复它们。
- 不可以调用任何 MQX 函数。
- 由一个应用程序运行机制（通常是具有头部和尾部指针以及数据大小字段的环缓冲区）将信息传递给其它任务（如果需要）。

我的应用程序由几个任务组成，它们仅在中断产生的某个信号到来时运行。如何让我的中断服务程序能够与合适的任务进行通信？

如果目标硬件允许的话，当禁用中断时（参阅 **MQX_INITIALIZATION_STRUCT** 中的 **MQX_HARDWARE_INTERRUPT_LEVEL_MAX** 域），设置中断优先级高于 MQX。如果你这么做，中断程序能够中断一个 MQX 临界区。例如，在一个 ARCtangent 处理器上，MQX 能设置成永远禁止 2 级中断并仅使用 1 级中断禁用/启用的临界区。如果目标硬件不允许你设置前面段落中所描述过的中断优先级，使用事件组件从中断服务程序发送一个信号到几个任务。这些任务开放到一个事件组的连接，并且任务之一是给中断服务程序提供连接。每个任务调用 **_event_wait_any()** 或**_event_wait_all()**函数并且停止任务执行。中断服务程序调用**_event_set()**开启任务执行。

当我为一个特定的中断保存并恢复一个中断服务程序时，我如何获得和原始中断服务程序关联的数据指针的值？

在你安装临时中断服务程序之前请调用**_int_get_isr_data()**。该函数返回一个指针，指向你传给它的特定向量。

6.6 内存

一个任务如何传递一个不属于它的内存块？

虽然任务总是传递它自己的内存，但不拥有内存的任务可以通过**_mem_transfer()**函数实现内存的传递。

我的任务分配了一个 10 字节的内存块，但是它总是得到更多，为什么？

当 MQX 分配一个内存块时，它对齐这个块适当的内存边界并且给这个块加一个内部的头，并且还强制分配一个最小尺寸。

一个任务能给另一个任务分配一个内存块吗？

不能。每个任务只能分配它自己的内存。然而，一个任务随后能将内存传送给另一个任务。

如果_partition_test()检测到一个问题，他是否要试图去修复该问题？

不会。这时表明内存已被破坏，请调试应用程序找到原因。

当我超出了默认的内存池，额外内存是否必须连续地连接到现存内存池的末尾？

不是。额外的内存可以置于任何地方。

如果我采用非连续的内存而超出了默认内存池，_mem_get_highwater()会返回什么？

函数名中的 highwater 标志表示内存的最高位置，MQX 已经从中分配了一个内存块。

我有一些基于多处理器的任务，它们需要共享内存。我如何才能提供对内存的互斥使用？

根据你的硬件，您可以使用一个自旋互斥锁来保护共享内存。自旋互斥锁调用 **_mem_test_and_set()**函数，当硬件支持锁定共享内存时，该函数提供了多处理器安全。

6.7 信息传递

我如何保证目标消息队列 IDs 和正确的任务关联起来？

创建一个任务，使用名称数据库把每个消息队列号和一个名称关联起来。每个任务通过指定

名称来获得队列号。

我能在一台 PC 机和我的目标硬件间传送消息吗？

能。创建一个运行在你的 PC 机上的程序，通过串行、PCI 或以太网，该程序能连续地发送数据包到应用程序，也能从应用程序连续地接收数据包。只要正确地格式化数据包，MQX 传送任何接收的数据。

我的任务每次使用一个新分配的消息成功地调用了_msgq_send()几次。最终_msgq_send()调用失败了。

你可能已经用完了消息。每次你分配一个新的消息发送出去，要检查返回值是否为空。如果是，接收任务可能没有释放消息，或者没有机会运行。

6.8 互斥

当一个拥有互斥数据结构的任务被破坏时，会发生什么？你的正在等待锁定互斥的任务会永远等待吗？

不会，所有的组件都具有清除函数。当一个任务被终止时，清除函数确定任务正在使用的资源并释放它们。如果一个任务已有一个互斥锁锁定，MQX 在它终止任务时解锁该互斥锁。一个任务不应该拥有互斥结构的内存；而应该创建结构作为全局变量或者分配一块系统内存块。

6.9 信号量

如果我“强行破坏”一个严谨的信号量会发生什么？

当你破坏一个严格的信号量时，如果强行破坏标志被设置，MQX 不会破坏信号量直到所有等待的任务获得和送出信号量。（反之，如果信号量是不严谨的，MQX 则立即将所有正在等待信号量的任务变为就绪态。）

两个任务使用一个信号量。连接只为一个任务服务，而不服务于另一个，为什么？

两个任务可能共享同样的全局连接，而不是有它们自己的局部连接。每个任务应该调用 **_sem_open()** 或 **_sem_open_fast()**以获取它的连接。

6.10 任务退出处理与任务异常处理

这两者有什么不同？

当一个任务调用**_task_abort()**函数时或者当一个任务从它的任务主体返回时，MQX 调用任务退出处理程序。

如果 MQX 已安装了异常处理程序，此时任务产生了一个不支持的异常，MQX 则调用任务异常处理程序来处理。

6.11 任务队列

我的应用程序提出了几项任务，它们在优先任务队列中具有相同的优先级。它们如何排序？

在一个优先级中的任务按照先进先出的顺序排队。

6.12 任务

我总是至少需要一个自启动的任务吗？

是的。在一个应用程序中，为了启动应用程序至少需要一个自启动的应用程序任务。

在一个多处理器应用程序（该应用程序可以远程创建任务）中，每个程序映像不必具有一个自启动的应用程序任务；然而，每个程序映象必须将作为一个自启动任务的 IPC 任务包含在任务模板中。如果在一个处理器上没有应用程序任务被创建，空闲任务将会运行。

一个自动启动任务创建其它所有任务并且初始化全局内存，我能在不影响子任务的情况下终止它吗？

能。当 MQX 终止创建者，它释放创建者的资源（内存、区块、队列等）和堆栈空间。子任务的资源独立于创建者并且不会被影响。

作为创建者的任务拥有它的子任务吗？

不。两者之间唯一的联系是子任务能获取创建者的任务 ID。子任务有自己的堆栈空间和自变量。

什么是任务，它们如何被创建？

如果任务执行相同的根函数，他们就共享相同的代码空间。一个任务总是在根函数的入口点开始执行，即使该函数是它的创建者的根函数。这是和 UNIX 中的 **fork()**函数有不同的行为。

我能把创建的任务移动到另一个处理器吗？

不能。

6.13 时间片

MQX 如何来度量一个时间片？时间片是绝对的还是相对的？也就是说，如果一个任务有一个 10ms 的时间片并且在时间等于 0ms 时启动，它是在时间等于 10ms 时放弃处理器吗？或者是在执行了 10ms 后放弃处理器吗？

基于 10ms 的时间片，MQX 计算当任务运行时所发生的定时器中断数。如果 10 个或更多毫秒已经过去，MQX 为任务有效地运行**_sched_yield()**函数。因此，一个任务没有获得 10ms 的线性时间，因为更高优先级的任务将抢占执行。并且，如果任务调用一个调度函数（例如**_task_block()**或**_sched_yield()**），MQX 会设置任务的时间片计数器归零。

因为有超时，MQX 分配的时间是±**BSP_ALARM_FREQUENCY** 嘀嗒/秒。

6.14 定时器

我的应用程序是基于多处理器的。我有一个主处理器向其它处理器发送同步消息，使得它们重置时间。

我如何才能确保重置消息不会干扰应用程序所使用的定时器？

为了改变绝对时间**(_time_set())**时不影响定时器，启动定时器的操作要采用消逝时间（**TIMER_ELAPSED_TIME**），而不是采用绝对时间（**TIMER_KERNEL_TIME_MODE**）。

如果给 if_timer_start_oneshot_at()函数一个已经过去的、到期时间，则将会发生什么？

MQX 把这个值放进了定时器队列。当下一个周期性定时器中断发生时，MQX 确定当前时间大于或等于已到期时间，则定时器会触发且 MQX 会调用警告函数。

如何联系我们：

主页：
www.freescale.com

Web 支持：
http://www.freescale.com/support

美国 / 欧洲或未列出的地方：
飞思卡尔半导体公司
技术信息中心， EL516
2100 East Elliot Road
Tempe, Arizona 85284
1-800-521-6274 or +1-480-768-2130
www.freescale.com/support

欧洲、中东和非洲：
Freescale Halbleiter Deutschland GmbH
技术信息中心
Schatzbogen 7
81829 Muenchen, 德国
+44 1296 380 456 (英国)
+46 8 52200080 (English)
+49 89 92103 559 (德国)
+33 1 69 35 48 48 (法国)
www.freescale.com/support

日本：
飞思卡尔半导体 （日本）有限公司
总部
ARCO Tower 15F
1-8-1, Shimo-Meguro, Meguro-ku,
东京 153-0064
日本
0120 191014 or +81 3 5437 9125
support.japan@freescale.com

亚太地区：
飞思卡尔半导体 （中国）有限公司
京汇大厦 23 层
建国路 118 号
朝阳区
北京 100022
中国
+86 10 5879 8000
support.asia@freescale.com

如果仅需要印刷品：
Freescale Semiconductor Literature Distribution Center
1-800-441-2447 或 +1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor@hibbertgroup.com

文件编号：MQXUGZHS
版本 0
07/2010

freescale™
semiconductor