NXP MQX™ RTOS

User's Guide

MQXUG
Rev 5.2,
07/2020

Embedded Access Inc

CHAPTER 1 BEFORE YOU BEGIN 10

1.1 About MQX™ RTOS 10

1.2 About This Book 10

1.3 Conventions 11

1.3.1 Tips 11

1.3.2 Notes 11

1.3.3 Cautions 11

CHAPTER 2 MQX RTOS AT A GLANCE 12

2.1 Organization of MQX RTOS 12

2.1 Initialization 14

2.2 Task Management 14

2.3 Scheduling 14

2.4 Managing Memory with dynamic memory allocators 14

2.5 Managing Memory with Fixed-Size Blocks (Partitions) 15

2.6 Controlling Caches 16

2.7 Controlling an MMU 16

2.8 Lightweight Memory Management 16

2.9 Lightweight Events 16

2.10 Events 16

2.11 Lightweight Semaphores 17

2.12 Semaphores 17

2.13 Mutexes 17

2.14 Lightweight Message Queue 17

2.15 Messages 18

2.16 Task Queues 18

2.17 Inter-Processor Communication 18

2.18 Time Component ... 18

2.19 Lightweight Timers ... 19

2.20 Timers ... 19

2.21 Watchdogs ... 19

2.22 Interrupt and Exception Handling ... 19

2.23 I/O Drivers ... 20

2.24.1 Formatted I/O ... 20
2.24.2 I/O Subsystem ... 20

2.24 Logs ... 20

2.25 Lightweight Logs ... 20

2.26 Kernel Log ... 20

2.27 Stack Usage ... 21

2.28 Task Error Codes ... 21

2.29 Exception Handling ... 21

2.30 Run-Time Testing ... 21

2.31 Queue Manipulation ... 21

2.32 Name Component ... 22

CHAPTER 3 USING MQX RTOS ... 23

3.1 Before You Begin ... 23

3.2 Initializing and Starting MQX RTOS ... 23

3.2.1 MQX RTOS Initialization Structure ... 23
3.2.1.1 Default MQX RTOS Initialization Structure ... 24
3.2.2 Task Template List ... 24
3.2.2.1 Assigning Task Priorities ... 25
3.2.2.2 Assigning Task Attributes ... 25
3.2.2.3 Default Task Template List ... 26
3.2.2.4 Example: A Task Template List ... 26
3.2.2.5 Example: Creating an Autostart Task ... 26

3.3 Managing Tasks ... 28

3.3.1 Creating Tasks ... 29
3.3.2 Getting Task IDs ... 29

3.3.3 Setting a Task Environment 30
3.3.4 Managing Task Errors....30
3.3.5 Restarting Tasks 30
3.3.6 Terminating Tasks 31
3.3.7 Example: Creating Tasks 32
3.3.7.1 Code for the Creating Tasks Example 32
3.3.7.2 Compiling the Application and Linking it with MQX RTOS....34

3.4 Scheduling Tasks 34
3.4.1 FIFO Scheduling....34
3.4.2 Round Robin Scheduling 35
3.4.2.1 Preemption....36

3.5 Managing Memory with Variable-Size Blocks.... 37
3.5.1 Managing Lightweight Memory with Variable-Size Blocks 38
3.5.2 Managing Memory with Fixed-Size Blocks (Partitions) 40
3.5.2.1 Creating the Partition Component for Dynamic Partitions 40
3.5.2.2 Creating Partitions....40
3.5.2.3 Allocating and Freeing Partition Blocks 40
3.5.2.4 Destroying a Dynamic Partition 41
3.5.2.5 Example: Two Partitions 41
3.5.3 Controlling Caches 42
3.5.3.1 Flushing Data Cache 43
3.5.3.2 Invalidating Data or Instruction Cache....43
3.5.4 Controlling the MMU (Virtual Memory) 44
3.5.4.1 Example: Initializing the MMU with Virtual Memory 45
3.5.4.2 Example: Setting Up a Virtual Context....46
3.5.4.3 Example: Creating Tasks with a Virtual Context 46

3.6 Synchronizing Tasks.... 47
3.6.1 Events....48
3.6.1.1 Creating the Event Component....49
3.6.1.2 Creating an Event Group 49
3.6.1.3 Opening a Connection to an Event Group 50
3.6.1.4 Waiting for Event Bits (Events) 50
3.6.1.5 Setting Event Bits 50
3.6.1.6 Clearing Event Bits 51
3.6.1.7 Closing a Connection to an Event Group 51
3.6.1.8 Destroying an Event Group 51
3.6.1.9 Example: Using Events 51
3.6.2 Lightweight Events....53
3.6.2.1 Creating a Lightweight Event Group 54
3.6.2.2 Waiting for Event Bits 54
3.6.2.3 Setting Event Bits 54
3.6.2.4 Clearing Event Bits 55
3.6.2.5 Destroying a Lightweight Event Group 55
3.6.3 About Semaphore-Type Objects....55
3.6.3.1 Strictness 55
3.6.3.2 Priority Inversion....55
3.6.3.3 Example: Priority Inversion 56

3.6.3.4 Avoiding Priority Inversion with Priority Inheritance 56
3.6.3.5 Avoiding Priority Inversion with Priority Protection 57
3.6.4 Lightweight Semaphores 58
3.6.4.1 Creating a Lightweight Semaphore 58
3.6.4.2 Waiting for and Posting a Lightweight Semaphore 59
3.6.4.3 Destroying a Lightweight Semaphore 59
3.6.4.4 Example: Producers and Consumer 59
3.6.5 Semaphores 62
3.6.5.1 Using a Semaphore 63
3.6.5.2 Creating the Semaphore Component 63
3.6.5.3 Creating a Semaphore 63
3.6.5.4 Opening a Connection to a Semaphore 64
3.6.5.5 Waiting for a Semaphore and Posting a Semaphore 64
3.6.5.6 Closing a Connection to a Semaphore 65
3.6.5.7 Destroying a Semaphore 65
3.6.5.8 Example: Task Synchronization and Mutual Exclusion 65
3.6.6 Mutexes 70
3.6.6.1 Creating the Mutex Component 71
3.6.6.2 Mutex Attributes 71
3.6.6.3 Waiting Protocols 71
3.6.6.4 Scheduling Protocols 72
3.6.6.5 Creating and Initializing a Mutex 72
3.6.6.6 Locking a Mutex 73
3.6.6.7 Unlocking a Mutex 73
3.6.6.8 Destroying a Mutex 73
3.6.6.9 Example: Using a Mutex 73
3.6.7 Messages 75
3.6.7.1 Creating the Message Component 76
3.6.7.2 Using Message Pools 76
3.6.7.3 Allocating and Freeing Messages 77
3.6.7.4 Sending Messages 78
3.6.7.5 Message Queues 78
3.6.7.6 Using Private Message Queues to Receive Messages 78
3.6.7.7 Using System Message Queues to Receive Messages 79
3.6.7.8 Determining the Number of Pending Messages 79
3.6.7.9 Notification Functions 80
3.6.7.10 Example: Client/Server Model 80
3.6.8 Lightweight Message Queue 83
3.6.8.1 Initialization of a Lightweight Message Queue 83
3.6.8.2 Sending Messages 84
3.6.8.3 Receiving Messages 84
3.6.8.4 Example: Client/Server Model 84
3.6.9 Task Queues 87
3.6.9.1 Creating and Destroying Task Queues 87
3.6.9.2 Suspending a Task 87
3.6.9.3 Resuming a Task 88
3.6.9.4 Example: Synchronizing Tasks 88

3.7 Communication Between Processors 89
3.7.1 Sending Messages to Remote Processors 90

3.7.1.1 Example: Four-Processor Application 91
3.7.2 Creating and Destroying Tasks on Remote Processors....91
3.7.3 Accessing Event Groups on Remote Processors....91
3.7.4 Creating and Initializing IPC 92
3.7.4.1 Building an IPC Routing Table 92
3.7.4.2 Building an IPC Protocol Initialization Table 93
3.7.4.3 IPC Using I/O PCB Device Drivers....94
3.7.4.4 Starting IPC Task....94
3.7.4.5 Example: IPC Initialization Information....94
5.7.1.1 3.7.5 Endian Conversion of Message Headers....99

3.8 Timing.... 100
3.8.1 Rollover of MQX RTOS Time 100
3.8.2 Accuracy of MQX RTOS Time 100
3.8.3 Time Component 100
3.8.3.1 Second/Millisecond Time....102
3.8.3.2 Time Stamp 102
3.8.3.3 Tick Time 102
3.8.3.4 Elapsed Time 103
3.8.3.5 Time Resolution 103
3.8.3.6 Absolute Time 103
3.8.3.7 Time in Date Formats....104
3.8.3.8 Timeouts....105
3.8.4 Timers....105
3.8.4.1 Creating the Timer Component 106
3.8.4.2 Starting Timers 106
3.8.4.3 Cancelling Outstanding Timer Requests 107
3.8.4.4 Example: Using Timers....107
3.8.5 Lightweight Timers....109
3.8.5.1 Starting Lightweight Timers 109
3.8.5.2 Cancelling Outstanding Lightweight Timer Requests 110
3.8.6 Watchdogs 110
3.8.6.1 Creating the Watchdog Component 111
3.8.6.2 Starting or Restarting a Watchdog....111
3.8.6.3 Stopping a Watchdog....111
3.8.6.4 Example: Using Watchdogs....111

3.9 Handling Interrupts and Exceptions.... 113
3.9.1 Initializing Interrupt Handling 115
3.9.2 Installing Application-Defined ISRs 115
3.9.3 Restrictions on ISRs....116
3.9.3.1 Functions That the ISR Cannot Call 116
3.9.3.2 Functions That ISRs should not call....117
3.9.3.3 Non-Maskable Interrupts....118
3.9.3.4 MQX_HARDWARE_INTERRUPT_LEVEL_MAX Configuration Parameter....118
3.9.4 Changing Default ISRs 121
3.9.5 Handling Exceptions....122
3.9.6 Handling ISR Exceptions....122
3.9.7 Handling Task Exceptions....123
3.9.8 Example: Installing an ISR 123

3.9.8.1 Compiling the Application and Linking it with MQX RTOS 124

3.10 Instrumentation 124
3.10.1 Logs 125
3.10.1.1 Creating the Log Component 125
3.10.1.2 Creating a Log 126
3.10.1.3 Format of a Log Entry 126
3.10.1.4 Writing to a Log 126
3.10.1.5 Reading From a Log 126
3.10.1.6 Disabling and Enabling Writing to a Log 126
3.10.1.7 Resetting a Log 127
3.10.1.8 Example: Using Logs 127
3.10.2 Lightweight Logs 128
3.10.2.1 Creating the Lightweight Log Component 129
3.10.2.2 Creating a Lightweight Log 129
3.10.2.3 Format of a Lightweight Log Entry 130
3.10.2.4 Writing to a Lightweight Log 130
3.10.2.5 Reading From a Lightweight Log 130
3.10.2.6 Disabling and Enabling Writing to a Lightweight Log 130
3.10.2.7 Resetting a Lightweight Log 130
3.10.2.8 Example: Using Lightweight Logs 131
3.10.3 Kernel Log 132
3.10.3.1 Using Kernel Log 133
3.10.3.2 Disabling Kernel Logging 134
3.10.3.3 Example: Using Kernel Log 134
3.10.4 Stack Usage Utilities 136

3.11 Utilities 136
3.11.1 Queues 136
3.11.1.1 Queue Data Structures 137
3.11.1.2 Creating a Queue 137
3.11.1.3 Adding Elements To a Queue 137
3.11.1.4 Removing Elements From a Queue 137
3.11.2 Name Component 137
3.11.2.1 Creating the Name Component 138
3.11.3 Run-Time Testing 138
3.11.3.1 Example: Doing Run-Time Testing 139
3.11.4 Additional Utilities 142

3.12 Embedded Debugging 143

3.13 Configuring MQX RTOS at Compile Time 143
3.13.1 MQX RTOS Compile-Time Configuration Options 143
3.13.1.1 MQX_COMPONENT_DESTRUCTION 144
3.13.1.2 MQX_DEFAULT_TIME_SLICE_IN_TICKS 144
3.13.1.3 MQX_EXIT_ENABLED 144
3.13.1.4 MQX_HAS_TIME_SLICE 144
3.13.1.5 MQX_HAS_DYNAMIC_PRIORITIES 144
3.13.1.6 MQX_HAS_EXCEPTION_HANDLER 144
3.13.1.7 MQX_HAS_EXIT_HANDLER 145

3.13.1.8 MQX_HAS_HW_TICKS 145
3.13.1.9 MQX_HAS_TICK 145
3.13.1.10 MQX_TD_HAS_TEMPLATE_INDEX 145
3.13.1.11 MQX_TD_HAS_STACK_LIMIT 145
3.13.1.12 MQX_INCLUDE_FLOATING_POINT_IO 145
3.13.1.13 MQX_IS_MULTI_PROCESSOR 146
3.13.1.14 MQX_KERNEL_LOGGING 146
3.13.1.15 MQX_LWLOG_TIME_STAMP_IN_TICKS 146
3.13.1.16 MQX_MEMORY_FREE_LIST_SORTED 146
3.13.1.17 MQX_MONITOR_STACK 146
3.13.1.18 MQX_MUTEX_HAS_POLLING 147
3.13.1.19 MQX_PROFILING_ENABLE 147
3.13.1.20 MQX_RUN_TIME_ERR_CHECK_ENABLE 147
3.13.1.21 MQX_ROM_VECTORS 147
3.13.1.22 MQX_SPARSE_ISR_TABLE 147
3.13.1.23 MQX_SPARSE_ISR_SHIFT 148
3.13.1.24 MQX_TASK_CREATION_BLOCKS 148
3.13.1.25 MQX_TASK_DESTRUCTION 148
3.13.1.26 MQX_TIMER_USES_TICKS_ONLY 148
3.13.1.27 MQX_USE_IDLE_TASK 148
3.13.1.28 MQX_USE_IO 148
3.13.1.29 MQX_USE_LWMEM_ALLOCATOR 149
3.13.1.30 MQXCFG_ENABLE_FP 149
3.13.1.31 MQX_SAVE_FP_ALWAYS 149
3.13.1.32 MQX_INCLUDE_FLOATING_POINT_IO 149
3.13.1.33 MQXCFG_MEM_COPY 149
3.13.1.34 MQXCFG_MEM_COPY_NEON 149
3.13.2 Recommended Settings 150

CHAPTER 4 REBUILDING NXP MQX RTOS 152

4.1 Why Rebuild MQX RTOS? 152

4.2 Before You Begin 152

4.3 NXP MQX RTOS Directory Structure 153
4.3.1 MQX RTOS Directory Structure 153
4.3.2 PSP Subdirectories 154
4.3.3 BSP Subdirectories 154
4.3.4 I/O Subdirectories 155
4.3.5 Other Source Subdirectories 155

4.4 NXP MQX RTOS Build Projects 155
4.4.1 PSP Build Project 155
4.4.2 BSP Build Project 155
4.4.3 Post-Build Processing 156
4.4.4 Build Targets 156

4.5 Rebuilding NXP MQX RTOS 157

CHAPTER 5 FAQS 158

6.1 General 158

6.2 Events 158

6.3 Global Constructors 158

6.4 Idle Task 158

6.5 Interrupts 159

6.6 Memory 159

6.7 Message Passing 160

6.8 Mutexes 161

6.9 Semaphores 161

6.10 Task Exit Handler Versus Task Exception Handler 161

6.11 Task Queues 162

6.12 Tasks 162

6.13 Time Slices 163

6.14 Timers 163

Chapter 1 Before You Begin

1.1 About MQX™ RTOS

The MQX™ Real-Time Operating System is designed for uniprocessor, multiprocessor, and distributed-processor embedded real-time systems.

To leverage the success of the MQX operating system, NXP Semiconductors adopted this software platform for its microprocessors. Compared to the original MQX RTOS distributions, the NXP MQX RTOS distribution was made simpler to configure and use. One single release now contains the MQX operating system plus all the other software components supported for a given microprocessor part. In this document, the sections specific to NXP MQX RTOS release are marked as below.

Table 1-1. Note formatting

Note	This is how notes specific to NXP MQX RTOS release are marked in this document.

MQX RTOS provides a run-time library of functions that programs use to become real-time multitasking applications. The main features of MQX RTOS are scalable size, component-oriented architecture, and ease of use.

MQX RTOS supports multiprocessor applications and can be used with flexible embedded I/O products for networking, data communications, and file management.

Throughout this book, we use MQX RTOS as the abbreviation for Message Queue Executive Real Time Operating System.

Table 1-2. Relative paths

<MQX_DIR>	Directory where MQX RTOS is located within KSDK. Specifically, <KSDK_DIR>\rtos\mqx.
<board>	Replaces board name (for example, TWR-K64F120M).
<mcu>	Replaces processor name (for example, MK64F120M).
<tool>	Replaces toolchain name (for example, IAR).
<target>	Replaces project target name (for example, Debug).
<library>	Replaces library name (for example, PSP).

1.2 About This Book

Use this book in conjunction with:

- MQX RTOS Reference - contains MQX RTOS simple and complex data types and alphabetically-ordered listings of MQX RTOS function prototypes.

Table 1-3. Release Contents

Note	NXP MQX RTOS release includes also other software products, based on MQX operating system. See also user guides and reference manuals for RTCS TCP/IP stack, MFS File System and the Release Notes for your specific release of MQX.

1.3 Conventions

The following tips, notes, and cautions represent the conventions used in MQX RTOS documentation.

1.3.1 Tips

Tips point out useful information.

Table 1-4. Generic Tip Format

Tip	The most efficient way to allocate a message from an ISR is to use _msg_alloc().

1.3.2 Notes

Notes point out important information.

Table 1-5. Generic Notes Format

Note	Non-strict semaphores do not have priority inheritance.

1.3.3 Cautions

Cautions tell you about commands or procedures that could have unexpected or undesirable side effects or could be dangerous to your files or your hardware.

Table 1-6. Generic Cautions Format

Caution	If you modify MQX RTOS data types, some MQX RTOS Host Tools might not operate

Chapter 2 MQX RTOS at a Glance

2.1 Organization of MQX RTOS

MQX RTOS consists of core (non-optional) and optional components. Functions that MQX RTOS or an application calls are the only functions included in the application image for core components. To match application requirements, an application can be extended by adding optional components.

The following diagram shows core components in the center with optional components around the outside.

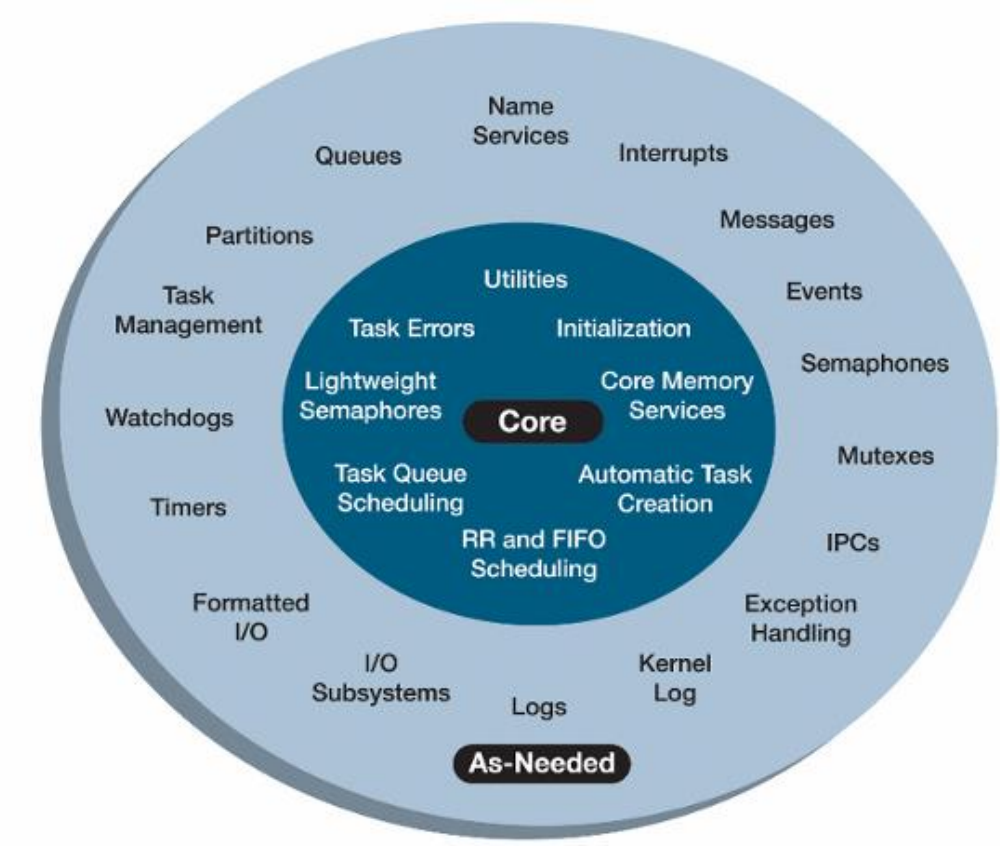

Figure 2-1. Core and Optional Components

The following table summarizes core and optional components, each of which is briefly described in subsequent sections of the chapter.

Table 2-1. Core and Optional Components

Component	Includes	Type
Initialization	Initialization and automatic task creation	Core
Task management	Dynamic task management	Core
Scheduling	Round robin and FIFO	Core
	Explicit using task queues	Optional
Task synchronization and communication	Lightweight semaphores	Core
	Semaphores	Optional
	Lightweight events	Optional
	Events	Optional
	Mutexes	Optional
	Lightweight message queue	Optional
	Messages	Optional
	Task queues	Optional
Interprocessor communication		Optional
Timing	Time component	Optional (BSP)
	Lightweight timers	Optional
	Timers	Optional
	Watchdogs	Optional
Memory management	Memory with variable-size blocks	Core
	Memory with fixed-size blocks (partitions)	Optional
	MMU, cache, and virtual memory	Optional
	Lightweight memory	Optional
Interrupt handling		Optional (BSP)
I/O drivers	I/O subsystem (NIO)	Optional (BSP)
	Formatted I/O	MQX_STDLIB
Instrumentation	Stack usage	Core
	Kernel log	Optional
	Logs	Optional
	Lightweight logs	Optional
Error handling	Task error codes, exception handling, runtime testing	Core
Queue manipulation		Core
Name component		Optional

2.1 Initialization

Initialization is a core component. The application starts when _mqx() runs. The function initializes the hardware and starts MQX RTOS. When MQX RTOS starts, it creates tasks that the application defines as autostart tasks.

2.2 Task Management

Task management is a core component.

Because it automatically creates tasks when MQX RTOS starts, an application can also create, manage, and terminate tasks as the application runs. It can create multiple instances of the same task, and there is no limit to the total number of tasks in an application. The application can dynamically change the attributes of any task. MQX RTOS frees task resources, when it terminates a task.

Also, for each task you can specify:

An exit function, which MQX RTOS calls when it terminates the task.
- An exception handler, which MQX RTOS calls if an exception occurs while the task is active.

2.3 Scheduling

Scheduling complies with POSIX.4 (real-time extensions) and supports these policies:

- FIFO (also called priority-based preemptive) scheduling is a core component - the active task is the highest-priority task that has been ready the longest.
- Round robin (also called time slice) scheduling is a core component - the active task is the highest-priority task that has been ready the longest without consuming its time slice.
- Explicit scheduling (using task queues) is an optional component - you can use task queues to explicitly schedule tasks or to create more complex synchronization mechanisms. Because task queues provide minimal functionality, they are fast. An application can specify a FIFO or round robin scheduling policy when it creates the task queue.

2.4 Managing Memory with dynamic memory allocators

To allocate and free variable-size pieces (called memory blocks) of memory, MQX

RTOS provides core services that are similar to malloc() and free(), which most C run-time libraries provide. You can choose between two implementations of dynamic allocation - LWMEM or MEM. For a comparison, see the following table:

Table 2-2. RAM footprint comparison [in bytes]

	LWMEM	MEM
Per block overhead	12	28
Per instance/pool overhead	44	72

Note: Values may vary depending on the compiler used and the target CPU.

Table 2-3. Code size footprint comparison [in bytes]

	LWMEM	MEM
Release target build size	1048	1968

Note: Values may vary depending on the compiler used and the target CPU.

Table 2-4. Execution time [in cycles]

	LWMEM	MEM
Average allocation time	360	450
Average deallocation time	300	1000
malloc() WCET* after several seconds of run in webserver application	4200+	4500+
free() WCET* after several seconds of run in webserver application	4300+	4300+
Time complexity	O(n)	O(n)

For systems which have only a few kilobytes of RAM, it is advised to use the LWMEM allocator. It has a lower initial RAM footprint and its linear time complexity is not a problem if the number of free blocks in the free list cannot grow much - the RAM is small and there is a minimum size for an allocated block.

2.5 Managing Memory with Fixed-Size Blocks (Partitions)

Partitions are an optional component. You can allocate and manage fixed-size pieces (called partition blocks) of memory. The partition component supports fast, deterministic memory allocation, which reduces memory fragmentation and conserves memory resources. Partitions can be in the default memory pool (dynamic partitions) and outside it (static partitions). You can allocate partition blocks to a task or to the system. Partition blocks allocated to a task are a resource of the task, and MQX RTOS frees them if the allocating task terminates.

2.6 Controlling Caches

MQX RTOS functions let you control the instruction cache and data cache that some CPUs have.

2.7 Controlling an MMU

For some CPUs, you must initialize the memory management unit (MMU) before you enable caches. MQX RTOS functions let you initialize, enable, and disable an MMU, and add a memory region to it. You can control an MMU by using MMU page tables.

2.8 Lightweight Memory Management

If an application is constrained by data- and code-size requirements, lightweight memory can be used. It has fewer interface functions and smaller code and data sizes. As a result, some areas have less robustness (removal of header checksums) and are slower (task-destruction times).

If you change a compile-time configuration option, MQX RTOS uses the lightweight-memory component when it allocates memory. For more information, see Configuring MQX RTOS at Compile Time.

2.9 Lightweight Events

Lightweight events (LWEvents) are an optional component. They are a low-overhead way for tasks to synchronize using bit state changes. Lightweight events require a minimal amount of memory and run quickly.

2.10 Events

Events are an optional component. They support the dynamic management of objects that are formatted as bit fields. Tasks and interrupt service routines can use events to synchronize and convey simple information in the form of bit-state changes. There are named and fast-event groups. Event groups can have autoclearing event bits, whereby MQX RTOS clears the bits immediately after they are set. An application can set event bits in an event group that is on a remote processor.

2.11 Lightweight Semaphores

Lightweight semaphores (LWSems) are a core component. They are a low-overhead way for tasks to synchronize their access to shared resources. LWSems require a minimal amount of memory and run quickly. LWSems are counting FIFO semaphores without priority inheritance.

2.12 Semaphores

Semaphores are an optional component. They are counting semaphores. You can use semaphores to synchronize tasks. You can use a semaphore to guard access to a shared resource, or to implement a producer/consumer-signalling mechanism. Semaphores provide FIFO queuing, priority queuing, and priority inheritance. Semaphores can be strict or non-strict. There are named and fast semaphores.

2.13 Mutexes

Mutexes are an optional component. A mutex provides mutual exclusion among tasks, when they access a shared resource. Mutexes provide polling, FIFO queuing, priority queuing, spin-only and limited-spin queuing, priority inheritance, and priority protection. Mutexes are strict; that is, a task cannot unlock a mutex, unless it had first locked the mutex.

2.14 Lightweight Message Queue

Lightweight message queue is an optional component. It deals with low-overhead implementation of standard MQX RTOS messages. Tasks send messages to lightweight message queues and receive messages from lightweight message queues. A message in the message pool has a fixed size, a multiple of 32 bits. Blocking reads and blocking writes are provided.

2.15 Messages

Messages are an optional component. Tasks can communicate with each other by sending messages to message queues that are opened by other tasks. Each task opens its own input-message queues. A message queue is uniquely identified by its queue ID, which MQX RTOS assigns when the queue is created. Only the task that opens a message queue can receive messages from the queue. Any task can send to any previously opened message queue, if it knows the queue ID of the opened queue.

Tasks allocate messages from message pools. There are system-message pools and private-message pools. Any task can allocate a message (system message) from system-message pools. Any task with the pool ID can allocate a message (private message) from a private-message pool.

2.16 Task Queues

In addition to providing a scheduling mechanism, task queues provide a simple and efficient way to synchronize tasks. You can suspend tasks in the task queue and remove them from the task queue.

2.17 Inter-Processor Communication

Inter-processor communication (IPC) is an optional component.

An application can run concurrently on multiple processors with one executable image of MQX RTOS on each processor. The images communicate and cooperate using messages that are transferred by memory or over communication links using inter-processor communication. The application tasks in each image need not be the same and, indeed, are usually different.

2.18 Time Component

Time is an optional component that you can enable and disable at the BSP level. There is elapsed time and absolute time. You can change absolute time. The time resolution depends on the application-defined resolution that is set for the target hardware when MQX RTOS starts.

2.19 Lightweight Timers

Lightweight timers are an optional component and provide a low-overhead mechanism for calling application functions at periodic intervals. Lightweight timers are installed by creating a periodic queue, then adding a timer to expire at some offset from the start of the period.

When you add a lightweight timer to the queue, you specify a notification function that is called by the MQX RTOS tick ISR when the timer expires. Since the timer runs from an ISR, not all MQX RTOS functions can be called from the timer.

2.20 Timers

Timers are an optional component. They provide periodic execution of an application function. MQX RTOS supports one-shot timers (they expire once) and periodic timers (they expire repeatedly at a given interval). You can set timers to start at a specified time or after a specified duration.

When you set a timer, you specify the notification function that timer task calls when the timer expires. The notification function can be used to synchronize tasks by sending messages, setting events, or using one of the other MQX RTOS synchronization mechanisms.

2.21 Watchdogs

Watchdogs are option components that let the user detect task starvation and deadlock conditions at the task level.

2.22 Interrupt and Exception Handling

Interrupt and exception handling is optional at the PSP level. MQX RTOS services all hardware interrupts within a range that the BSP defines, and saves a minimum context for the active task. MQX RTOS supports fully nested interrupts, if the CPU supports nested interrupts. Once inside an interrupt service routine (ISR), an application can re-enable any interrupt level. To further reduce interrupt latencies, MQX RTOS defers task rescheduling until after all ISRs have run. In addition, MQX RTOS reschedules only if a new task has been made ready by an ISR. To reduce stack size, MQX RTOS supports a separate interrupt stack.

An ISR is not a task; it is a small, high-speed routine that reacts quickly to hardware interrupts. An ISR is usually written in C language. Its duties include resetting the device, getting its data, and signaling the appropriate task. An ISR can be used to signal a task with any of the non-blocking MQX RTOS functions.

2.23 I/O Drivers

I/O drivers are an optional component at the BSP level. They consist of formatted I/O and the I/O subsystem . I/O drivers are not described in this book.

2.24.1 Formatted I/O

MQX RTOS provides formatted I/O functions that are the API to the I/O subsystem.

2.24.2 I/O Subsystem

You can dynamically install I/O device drivers, after which any task can open them.

2.24 Logs

Logs are an optional component that lets you store and retrieve application-specific information. Each log entry has a timestamp and sequence number. You can use the information to test, debug, verify, and analyze performance.

2.25 Lightweight Logs

Lightweight logs are similar to logs, but use only fixed-sized entries. They are faster than the conventional application logs and are used by kernel log.

2.26 Kernel Log

Kernel log is an optional component that lets you record MQX RTOS activity. You can create kernel log at a specific location or let MQX RTOS choose the location. You can configure kernel log to record all MQX RTOS function calls, context switches, and interrupt servicing. Performance tool uses kernel log.

2.27 Stack Usage

MQX RTOS has core functions that let you dynamically examine the interrupt stack and the stack usage by all tasks, so that you can determine whether you have allocated enough stack space.

2.28 Task Error Codes

Each task has a task error code, which is associated with the task's context. Specific MQX RTOS functions read and update the task error code.

2.29 Exception Handling

You can specify a default ISR that runs for all unhandled interrupts, and an ISR-specific exception handler that runs if the ISR generates an exception.

2.30 Run-Time Testing

MQX RTOS provides core run-time test functions that an application can call during its normal operation. There are test functions for the following components:

- events and lightweight events
- kernel log and lightweight logs
- memory with fixed-size blocks (partitions)
- memory with variable-size memory blocks and lightweight memory
- message pools and message queues
- mutexes
- name component
- queues (application-defined)
- semaphores and lightweight semaphores
- task queues
- timers and lightweight timers
- watchdogs

2.31 Queue Manipulation

There is a core component that implements a double-linked list of queue elements. You can initialize a queue, add elements, remove elements, and peek at elements.

2.32 Name Component

The name component is optional. It provides a names database that maps a string to a dynamically defined scalar, such as a queue ID.

Chapter 3 Using MQX RTOS

3.1 Before You Begin

This chapter describes how to use MQX RTOS. It includes examples that you can compile and run.

Table 3-1. References

Information Type	Reference Doc
Prototype for each function that is mentioned in this chapter.	MQX RTOS Reference Manual
Data types that are mentioned in this chapter.	MQX RTOS Reference Manual

3.2 Initializing and Starting MQX RTOS

MQX RTOS is started with **_mqx()**, which takes the MQX RTOS initialization structure as its argument. Based on the values in the structure, MQX RTOS does the following:

- It sets up and initializes the data that MQX RTOS uses internally, including the default memory pool, ready queues, the interrupt stack, and task stacks.
- It initializes the hardware (for example, chip selects).
- It enables timers.
- It sets the default time slice value.
- It creates the Idle task, which is active if no other task is ready.
- It creates tasks that the task template list defines as autostart tasks.
- It starts scheduling the tasks.

3.2.1 MQX RTOS Initialization Structure

The MQX RTOS initialization structure defines parameters of the application and target hardware.

```
typedef struct mqx_initialization_struct
{
  _mqx_uint               PROCESSOR_NUMBER;
  void *                  START_OF_KERNEL_MEMORY;
  void *                  END_OF_KERNEL_MEMORY;
  _mqx_uint               INTERRUPT_STACK_SIZE;
  TASK_TEMPLATE_STRUCT_PTR   TASK_TEMPLATE_LIST;
  mqx_uint                MQX_HARDWARE_INTERRUPT_LEVEL_MAX;
  mqx_uint                MAX_MSGPOOLS;
  mqx_uint                MAX_MSGQS;
```

```
   char *                    IO_CHANNEL;
   char *                    IO_OPEN_MODE;
   mqx_uint                  RESERVED[2];
} MQX_INITIALIZATION_STRUCT, * MQX_INITIALIZATION_STRUCT_PTR;
```

For a description of each field, see *NXP MQX™ RTOS Reference Manual*.

3.2.1.1 Default MQX RTOS Initialization Structure

You can either define your own initialization values of the MQX RTOS initialization structure or use the default initialization that is provided with each BSP. The default initialization variable is called **MQX_init_struct** and is in mqx_init.c in the appropriate BSP directory. The function has been compiled and linked with MQX RTOS.

Note	For task-aware debugging host tools to work, the MQX RTOS initialization structure variable must be called **MQX_init_struct**.

The examples in this chapter use the following **MQX_init_struct**.

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

Note	Initialize both elements of the **RESERVED** field to zero.

3.2.2 Task Template List

The task template list, which is a list of task templates (**TASK_TEMPLATE_STRUCT**), defines an initial set of templates that are used to create tasks on the processor.

At initialization, MQX RTOS creates one instance of each task, whose template defines it as an autostart task. In addition, while an application is running, it can create other tasks using a task template that either the task template list defines or the application defines dynamically. The end of the task template list is a zero-filled task template.

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

For a description of each field, see the *NXP MQX™ RTOS Reference Manual*.

3.2.2.1 Assigning Task Priorities

Note	If you assign priority zero to a task, the task runs with interrupts disabled. On some target processor platforms (e.g., ColdFire), certain task priority levels are reserved and are mapped to processor interrupt priority levels. Tasks running at such a special priority may prevent lower priority interrupts to be serviced. See more details about interrupt handling in section Handling Interrupts and Exceptions.

When you assign task priorities in the task template list, note that:

- MQX RTOS creates one ready queue for each priority up to the lowest priority (highest number).
- While an application is running, it cannot create a task that has a lower priority (a higher number) than the lowest-priority task in the task template list.

3.2.2.2 Assigning Task Attributes

You can assign any combination of the following attributes to a task:

- Autostart - when MQX RTOS starts, it creates one instance of the task.
- DSP - MQX RTOS saves the DSP co-processor registers as part of the task's context.
- Floating point - MQX RTOS saves floating-point registers as part of the task's context.
- Time slice - MQX RTOS uses round robin scheduling for the task (the default is FIFO scheduling).

3.2.2.3 Default Task Template List

You can initialize your own task template list or use the default, which is called **MQX_template_list.**

3.2.2.4 Example: A Task Template List

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

world_task

The world_task is an autostart task. So, at initialization, MQX RTOS creates one instance of the task with a creation parameter of zero. The application defines the task template index (MAIN_TASK). The task is of priority five. The function **world_task()** is the code-entry point for the task. The stack size is 0x2000 single-addressable units.

hello_task

The hello_task task is a time-slice task with a time slice of 100, in milliseconds, if the default compile-time configuration options are used. For information about these options, see page Configuring MQX RTOS at Compile Time.

Float_task

The Float_task task is both a floating-point task and an autostart task.

3.2.2.5 Example: Creating an Autostart Task

A single task prints Hello World.

```
/* hello.c */

#include <mqx.h>
#include <bsp.h>
#include <fio.h>
#include <log.h>
#include <klog.h>
```

```
/* Task IDs */
#define HELLO_TASK 5

extern void hello_task(uint32_t);


const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
    /* Task Index,   Function,   Stack, Priority, Name, Attributes, Param, Time Slice */
    { HELLO_TASK,   hello_task, 1500,   8,       "hello", MQX_AUTO_START_TASK, 0, 0 },
    { 0 }
};


static uint32_t g_i = 0;

/*TASK*-----------------------------------------------------
*
* Task Name    : hello_task
* Comments     :
*    This task prints " Hello World "
*
*END*-----------------------------------------------------*/
void hello_task
    (
        uint32_t initial_data
    )
{
    (void)initial_data; /* disable 'unused variable' warning */

    _int_install_unexpected_isr();

    while (1)
    {
        printf("Hello World %d\n",g_i++);
        _time_delay(1000);
    }
}
```

Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   ```
   mqx\examples\hello
   ```

2. See the *MQX™ RTOS Release Notes* document for instructions on how to build and run the application.

The following appears on the output device:

```
Hello World 1
Hello World 2
Hello World 3
```

- •
- •
- •

3.3 Managing Tasks

Multiple tasks, created from the same task template can coexist, and each task is a unique instance. MQX RTOS maintains each instance by saving its context; that is, its program counter, registers, and stack. Each task has an application-unique 32-bit task ID, which MQX RTOS and other tasks use to identify the task.

The section on initialization (page Initializing and Starting MQX RTOS) shows how a task can be started automatically when MQX RTOS initializes. You can also create, manage, and terminate tasks, while the application runs.

Table 3-2. Summary: Managing Tasks

_task_abort	Terminates the task after running its task exit handler and releasing its resources.
_task_check_stack	Determines whether the task's stack is out of bounds.
_task_create	Allocates and starts (makes ready) a new task.
_task_create_blocked	Allocates a new task in the blocked state.
_task_create_at	Creates a new task with the stack location specified.
_task_destroy	Terminates the task after freeing its resources.
_task_disable_fp	Disable floating-point context switching for the task, if the task is a floating- point task.
_task_enable_fp	Enables floating-point context switching for the task.
_task_errno	Gets the task error code for the active task.
_task_get_creator	Gets the task ID of the task that created the task.
_task_get_environment	Gets a pointer to the environment data for a task.
_task_get_error	Gets the task error code.
_task_get_error_ptr	Gets a pointer to the task error code.
_task_get_exit_handler	Gets a task's exit handler.
_task_get_id	Gets the task ID.
_task_get_id_from_name	Gets the task ID of the first task with this name in the task template.
_task_get_index_from_id	Gets the task template index for the task ID.
_task_get_parameter	Gets the task-creation parameter.
_task_get_parameter_for	Gets the task-creation parameter for a task.
_task_get_processor	Gets the processor number on which a task resides.
_task_get_td	Converts a task ID to a pointer to a task descriptor.
_task_get_template_index	Gets the task template index of a task name.
_task_get_template_ptr	Gets a pointer to the task template for the task ID.

_task_restart	Restarts a task at the beginning of the task's function; keeps the same task descriptor, task ID, and task stack.
_task_set_environment	Sets a pointer to the environment data for a task.
_task_set_error	Sets the task error code.
_task_set_exit_handler	Sets the task's exit handler.
_task_set_parameter	Sets the task creation parameter.
_task_set_parameter_for	Sets the task creation parameter for a task.

3.3.1 Creating Tasks

Any task (creator) can create another task (child) by calling **_task_create()**, **_task_create_at()**or **_task_create_blocked()**, and passing the processor number, a task template index, and a task-creation parameter. The application defines one creation parameter, which is normally used to provide initialization information to the child. A task can also create a task that is not defined in the task template list, by specifying a template index of zero. In this case, MQX RTOS interprets the task-creation parameter as a pointer to a task template.

The functions initialize the child's stack. The function **_task_create()** puts the child in the ready queue for the task's priority. If the child is of higher priority than the creator, the child becomes the active task, because it is the highest-priority ready task. If the creator is of higher or equal priority, it remains the active task.

The function **_task_create_blocked()** creates a task that is blocked. The task is not ready to run, until another task calls **_task_ready()**.

The function **_task_create_at()** creates a task with the stack location specified, i.e., task stack is not dynamically allocated but has to be allocated before the **_task_create_at()**function is issued.

3.3.2 Getting Task IDs

A task can directly get its task ID with **_task_get_id()**. If a function takes a task ID as a parameter, you can specify **MQX_NULL_TASK_ID** to refer to the task ID of the active task.

A task can directly get the task ID of its creator with **_task_get_creator()**. The function **_task_create()** returns the child's task ID to the creator.

A task ID can also be determined from the task name in the task template, from which the task was created. This is done with **_task_get_id_from_name()**, which returns the task ID of the first task that matches the name in the task template list.

3.3.3 Setting a Task Environment

A task can save an application-specific environment pointer with **_task_set_environment()**. Other tasks can access the environment pointer with **_task_get_environment().**

3.3.4 Managing Task Errors

Each task has an error code (the task error code) associated with the task's context. Some MQX RTOS functions update the task error code when they detect an error.

If an MQX RTOS function detects an error and the application ignores the error, additional errors might still occur. Usually the first error best indicates the problem; subsequent errors might be misleading. To provide a reliable opportunity to diagnose problems after MQX RTOS sets the task error code to a value other than **MQX_OK**, MQX RTOS does not further change the task error code until the task explicitly resets it to **MQX_OK**.

A task can get its task error code from:

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

A task resets its task error code by calling **_task_set_error()** with **MQX_OK**. The function returns the previous task error code and sets the task error code to **MQX_OK**. Using **_task_set_error(),** a task can attempt to set its task error code to a value other than **MQX_OK**. However, only if the current task error code is **MQX_OK**, does MQX RTOS change the task error code to the new value.

If **MQX_CHECK_ERRORS** is set to 0 (see MQX RTOS Compile-Time Configuration Options), then not all error codes listed for a particular function are returned.

3.3.5 Restarting Tasks

An application can restart a task by calling **_task_restart()**, which restarts the task at the beginning of its function with the same task descriptor, task ID, and task stack.

3.3.6 Terminating Tasks

A task can terminate itself or any other task, whose task ID it knows. When a task is terminated, its children are not terminated. When a task is terminated, MQX RTOS frees the task's MQX RTOS-managed resources. These resources include:

- dynamically allocated memory blocks and partition blocks
- message queues
- messages
- mutexes
- non-strict semaphores
- strict semaphores after posting them
- queued connections are dequeued
- task descriptor

Note	The user is responsible for destroying all lightweight objects (lightweight semaphores, lightweight events, lightweight timers, etc.) before terminating a task as this is not done by the MQX RTOS task termination functions!

An application can terminate a task immediately (after MQX RTOS frees the task's resources) with **_task_destroy()** or gracefully with **_task_abort()**. While **_task_destroy()** causes the task destroy to happen from the context of the caller and is performed immediately, **_task_abort()** causes the victim task to be removed from any queues it is blocked on, its PC is effectively set to the task exit handler and then the victim task is added to the ready to run queue. Normal task scheduling and priority rules apply, so the actual task destruction may be deferred indefinitely (or for a long time). The implication is that there is no guarantee that the victim task is destroyed upon return from **_task_abort().**

When the to-be-terminated task becomes active, an application-defined task exit handler runs. The exit handler could clean up resources that MQX RTOS does not manage.

The task exit handler is set with **_task_set_exit_handler()**, and obtained with _task_get_exit_handler()**.**

MQX RTOS also calls the task exit handler if the task returns from its task body.

3.3.7 Example: Creating Tasks

This example adds a second task (world_task) to the example on page Example: Creating an Autostart Task. We modify that example's task template list to include information about world_task, and to change hello_task, so that it is not an autostart task. The world_task task is an autostart task.

When MQX RTOS starts, it creates world_task. The world_task then creates hello_task by calling **_task_create()** with hello_task as a parameter. MQX RTOS uses the hello_task template to create an instance of hello_task.

If **_task_create()** is successful, it returns the task ID of the new child task; otherwise, it returns **MQX_NULL_TASK_ID**.

The new hello_task task is put in the ready queue for the task's priority. Since it has a higher priority than world_task, it becomes active. The active task prints Hello. The world_task task then becomes active and checks to see whether hello_task was created successfully. If it was, world_task prints World; otherwise, world_task prints an error message.

If you change the priority of world_task to be of the same priority as hello_task, the output is World Hello only. The world_task runs before hello_task, because world_task has the same priority and does not relinquish control with a blocking function. When the world_task becomes blocked, the hello_task becomes active.

3.3.7.1 Code for the Creating Tasks Example

```
/* hello2.c */

#include <mqx.h>


#include <fio.h>


/* Task IDs */
#define HELLO_TASK   5
#define WORLD_TASK   6
extern void
hello_task(uint32_t); extern
void world_task(uint32_t);
const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function,  Stack, Priority, Name,    Attributes,    Param, Time Slice */
 { WORLD_TASK, world_task, 1000,    9,      "world", MQX_AUTO_START_TASK, 0,   0 },
```

```
  { HELLO_TASK, hello_task, 1000,    8,     "hello", 0,                    0,   0 },
  { 0 }  };

#include <mqx.h>
#include <bsp.h>
#include <fio.h>


/* Task IDs */
#define HELLO_TASK  5
#define WORLD_TASK  6

extern void hello_task(uint32_t);
extern void world_task(uint32_t);

const TASK_TEMPLATE_STRUCT MQX_template_list[] =
 {
  /* Task Index, Function,  Stack, Priority, Name,    Attributes,    Param, Time Slice */
  { WORLD_TASK, world_task, 1000,    8,     "world", MQX_AUTO_START_TASK, 0,   0 },
  { HELLO_TASK, hello_task, 1000,    8,     "hello", 0,                    0,   0 },
  { 0 }  };


/*TASK*-----------------------------------------------------
*
* Task Name    : hello_task
* Comments     :
*    This task creates world_task and then prints "Hello ".
*
*END*-----------------------------------------------------*/

void hello_task(uint32_t initial_data)
{
   _task_id world_task_id;

   (void) initial_data;

   world_task_id = _task_create(0, WORLD_TASK, 0);
   if (world_task_id == MQX_NULL_TASK_ID)
   {
      printf ("\n Could not create world_task\n");
      _task_block();
   }

   while (1)
   {
      printf("Hello ");
      _sched_yield();
   }
}


/*TASK*-----------------------------------------------------
*
```

```
* Task Name     : world_task
* Comments      :
*    This task prints "World\n".
*
*END*-----------------------------------------------------*/

void world_task(uint32_t initial_data)
{

    while (1)
    {
        printf("World\n");
       _sched_yield();
    }
}
```

3.3.7.2 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   `mqx\examples\hello2`

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.

This message appears on the output device:

```
Hello
World
```

3.4 Scheduling Tasks

MQX RTOS provides these task-scheduling policies:

- FIFO
- Round Robin
- Explicit, using task queues (described in a subsequent section on page Lightweight Message Queue).

You can set the scheduling policy to FIFO or round robin for the processor and separately for each task. As a result, an application might consist of tasks that use any combination of FIFO or round robin scheduling.

3.4.1 FIFO Scheduling

FIFO is the default scheduling policy. With FIFO scheduling, the task that runs (becomes active) next is the highest-priority task that has been waiting the longest time. The active task runs, until any of the following occurs:

- The active task voluntarily relinquishes the processor, because it calls a blocking MQX RTOS function.
- An interrupt occurs that has higher priority than the active task.
- A task that has priority higher than the active task, becomes ready.

You can change the priority of a task with **_task_set_priority()**.

3.4.2 Round Robin Scheduling

Round robin scheduling is similar to FIFO scheduling, but with the additional constraint that each round robin task has a maximum amount of time (the time slice), during which it can be active.

A task uses round robin scheduling only if the **MQX_TIME_SLICE_TASK** attribute is set in its task template. The task's time slice is determined by the value of the template's **DEFAULT_TIME_SLICE**. However, if the value is zero, the task's time slice is the default time slice for the processor. Initially, the default time slice for the processor is ten times the interval of the periodic timer interrupt. Since the interval on most BSPs is five milliseconds, the initial default time slice for the processor is usually 50 milliseconds. You can change the default time slice for the processor with **_sched_set_rr_interval()** or **_sched_set_rr_interval_ticks()**, passing the task ID parameter as **MQX_DEFAULT_TASK_**ID.

When the time slice expires for an active round robin task, MQX RTOS saves the task's context. MQX RTOS then performs a dispatch operation, in which it examines the ready queues to determine, which task should become active. MQX RTOS moves the expired task to the end of the task's ready queue, an action that causes control to pass to the next task in the ready queue. If there are no other tasks in the ready queue, the expired task continues to run.

With round robin scheduling, tasks of the same priority can share the processor in a time-equitable manner.

Table 3-3. Summary: Getting and Setting Scheduling Info

_sched_get_max_priority	Gets the highest priority allowed for any task; always returns zero.
_sched_get_min_priority	Gets the lowest priority for any task.

_sched_get_policy	Gets the scheduling policy.
_sched_get_rr_interval	Gets the time slice in milliseconds.
_sched_get_rr_interval_ticks	Gets the time slice in tick time.
_sched_set_policy	Sets the scheduling policy.
_sched_set_rr_interval	Sets the time slice in milliseconds.
_sched_set_rr_interval_ticks	Sets the time slice in tick time.

Table 3-4. Summary: Scheduling Tasks

_sched_yield	Moves the active task to the end of its ready queue, which yields the processor to the next ready task of equal priority.
_task_block	Blocks the task.
_task_get_priority	Gets a task's priority.
_task_ready	Makes a task ready.
_task_set_priority	Sets a task's priority.
_task_start_preemption	Re-enables preemption for the task.
_task_stop_preemption	Disables preemption for the task.

Each task is in one of the following logical states:

- Blocked - task is not ready to become active, because it is waiting for a condition to occur; when the condition occurs, the task becomes ready.
- Ready - task is ready to become active, but it is not active, because it is of the same priority as, or lower priority than the active task.
- Active - task is running.

If the active task becomes blocked or is preemptied, MQX RTOS performs a dispatch operation, in which it examines the ready queues to determine, which task should become active. MQX RTOS makes the highest-priority ready task the active task. If more than one task of the same priority is ready, the task at the start of that ready queue becomes the active task. That is, each ready queue is in FIFO order.

3.4.2.1 Preemption

The active task can be preemptied. Preemption occurs, when a higher-priority task becomes ready, and thus becomes the active task. The previously active task is still ready, but is no longer the active task. Preemption occurs, when an interrupt handler causes a higher-priority task to become ready, or the active task makes a higher-priority task ready.

3.5 Managing Memory with Variable-Size Blocks

By default, MQX RTOS allocates memory blocks from its default memory pool. Tasks can also create memory pools outside the default memory pool, and allocate memory blocks from them.

Both allocation processes are similar to using **malloc()** and **free()**, which are in most C run-time libraries.

Note	You cannot use a memory block as a message. You must allocate messages from message pools (see Messages).

A memory block can be a private memory block (a resource owned by the task that allocates it) or a system memory block (not owned by any task). When a task is terminated, MQX RTOS returns the task's private memory blocks to memory.

When MQX RTOS allocates a memory block, it allocates a block of at least the requested size (the block might be larger).

A task can transfer ownership of a memory block to another task (**_mem_transfer()**).

Table 3-5. Summary: Managing Memory with Variable-Size Blocks

_mem_alloc	Allocates a private memory block from the default memory pool.
_mem_alloc_from	Allocates a private memory block from the specified memory pool.
_mem_alloc_zero	Allocates a zero-filled private memory block from the default memory pool.
_mem_alloc_zero_from	Allocates a zero-filled private memory block from the specified memory pool.
_mem_alloc_system	Allocates a system memory block from the default memory.
_mem_alloc_system_from	Allocates a system memory block from the specified memory pool.
_mem_alloc_system_zero	Allocates a zero-filled system memory block from the default memory pool.
_mem_alloc_system_zero_from	Allocates a zero-filled system memory block from the specified memory pool.
_mem_alloc_align	Allocates an aligned private memory block from the default memory pool.
_mem_alloc_align_from	Allocates an aligned private memory block from the specified memory pool.
_mem_alloc_system_align	Allocates an aligned system memory block from the default memory pool.

_mem_alloc_system_align_from	Allocates an aligned system memory block from the specified memory pool.
_mem_alloc_at	Allocates a private memory block at the defined start address.
_mem_copy	Copies data from one memory location to another.
_mem_create_pool	Creates a memory pool outside the default memory pool.
_mem_extend	Adds additional memory to the default memory pool; the additional memory must by outside the current default memory pool, but need not be contiguous with it.
_mem_extend_pool	Adds additional memory to a memory pool that is outside the default memory pool; the additional memory must be outside the memory pool, but it needs not to be contiguous with the pool.
_mem_free	Frees a memory block that is inside or outside the default memory pool.
_mem_free_part	Frees part of a memory block (used if the memory block is larger than requested, or if it is larger than needed).
_mem_get_error	Gets a pointer to the memory block that caused **_mem_test**() to indicate an error.
_mem_get_error_pool	Gets a pointer to the last memory block that caused **_mem_test_pool()** to indicate an error.
_mem_get_highwater	Gets the highest memory address that has been allocated in the default memory pool (it might have since been freed).
_mem_get_highwater_pool	Gets the highest memory pool address that has been allocated (it might have since been freed)
_mem_get_size	Gets the size of a memory block; the size might be larger than the requested size.
_mem_swap_endian	Converts to the other endian format.
_mem_test	Tests the default memory pool; this is, checking the internal checksums to determine, whether the integrity of the memory has been violated (usually the cause of failure is that an application writes past the end of a memory block).
_mem_test_and_set	Tests and sets a memory location.
_mem_test_pool	Tests the memory pool for errors, as described for **_mem_test()**.
_mem_transfer	Transfers ownership of a memory block to another task.
_mem_zero	Sets all or part of a memory block to zero.

3.5.1 Managing Lightweight Memory with Variable-Size Blocks

Lightweight memory functions are similar to the functions for regular memory that are described in Managing Memory with Variable-Size Blocks. However, they have less overhead in data and code.

If you change an MQX RTOS compile-time configuration option, MQX RTOS uses the lightweight memory component when it allocates memory. For more information, see page Configuring MQX RTOS at Compile Time.

Table 3-6. Summary: Managing Lightweight Memory with Variable-Size Blocks

Lightweight memory uses certain structures and constants, which are defined in *lwmem.h*.	Lightweight memory uses certain structures and constants, which are defined in *lwmem.h*.
_lwmem_alloc	Allocates a private lightweight-memory block from the default lightweight-memory pool.
_lwmem_alloc_from	Allocates a private lightweight-memory block from the specified lightweight-memory pool.
_lwmem_alloc_zero	Allocates a zero-filled private lightweight-memory block from the default lightweight-memory pool.
_lwmem_alloc_zero_from	Allocates a zero-filled private lightweight-memory block from the specified lightweight-memory pool.
_lwmem_alloc_system	Allocates a system lightweight-memory block from the default lightweight-memory pool.
_lwmem_alloc_system_from	Allocates a system lightweight-memory block from the specified lightweight-memory pool.
_lwmem_alloc_system_zero	Allocates a zero-filled system lightweight-memory block from the default lightweight-memory pool.
_lwmem_alloc_system_zero_from	Allocates a zero-filled system memory block from the specified lightweight-memory pool.
_lwmem_alloc_align	Allocates an aligned private lightweight-memory block from the default lightweight-memory pool.
_lwmem_alloc_align_from	Allocates an aligned private lightweight-memory block from the specified lightweight-memory pool.
_lwmem_alloc_system_align	Allocates an aligned system lightweight-memory block from the default lightweight-memory pool.
_lwmem_alloc_system_align_from	Allocates an aligned system lightweight memory block from the specified lightweight memory pool.
_lwmem_alloc_at	Allocates a private lightweight-memory block at the defined start address.
_lwmem_create_pool	Creates a lightweight-memory pool.
_lwmem_free	Frees a lightweight-memory block.

_lwmem_get_size	Gets the size of a lightweight-memory block; the size might be larger than the requested size.
_lwmem_set_default_pool	Sets the pool to be used for the default lightweight-memory pool.
_lwmem_test	Tests all lightweight memory pools.
_lwmem_transfer	Transfers ownership of a lightweight-memory block to another task.

3.5.2 Managing Memory with Fixed-Size Blocks (Partitions)

With the partition component, you can manage partitions of fixed-size memory blocks, whose size the task specifies when it creates the partition. There are dynamic partitions (in the default memory pool) that can grow, and static partitions (outside the default memory pool) that cannot grow.

3.5.2.1 Creating the Partition Component for Dynamic Partitions

You can explicitly create the partition component with _partition_create_component(). If you do not explicitly create it, MQX RTOS creates it the first time an application creates a partition. There are no parameters.

3.5.2.2 Creating Partitions

There are two types of partitions.

Table 3-7. Static and Dynamic Partitions

Type of partition:	Created from:	By calling:
Dynamic	Default-memory pool	**_partition_create()**
Static	Outside default-memory pool	**_partition_create_at()**

If you create a static partition, you must ensure that the memory does not overlap code or data space that your application uses.

3.5.2.3 Allocating and Freeing Partition Blocks

An application can allocate two types of partition blocks from either a dynamic or static partition.

Table 3-8. Private and System Partition Blocks

Type of partition block:	Allocated by calling:	Is a resource of:	Can be freed by:
Private	**_partition_alloc()**	Task that allocated it	Owner only
System	**_partition_alloc_system()**	No one task	Any task

If the task is terminated, its private partition blocks are freed.

3.5.2.4 Destroying a Dynamic Partition

If all the partition blocks in a dynamic partition are freed, any task can destroy the partition by calling **_partition_destroy()**. You cannot destroy a static partition.

3.5.2.5 Example: Two Partitions

The following diagram shows one static partition and one dynamic partition.

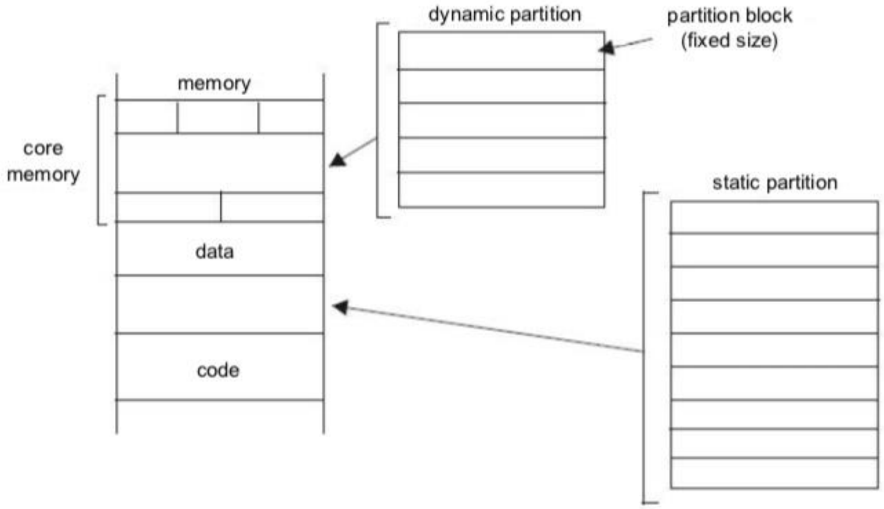

Figure 3-1. Example: Two Partitions

Table 3-9. Summary: Managing Memory with Fixed-Sixe Blocks (Partitions)

_partition_alloc	Allocates a private partition block from a partition.

_partition_alloc_system	Allocates a system partition block from a partition.
_partition_alloc_system_zero	Allocates a zero-filled system partition block from a partition.
_partition_alloc_zero	Allocates a zero-filled private partition block from a partition.
_partition_calculate_blocks	Calculates the number of partition blocks from the partition block size and the partition size (for static partitions).
_partition_calculate_size	Calculates the size of a partition from the partition block size and the number of blocks.
_partition_create	Creates a partition from the default memory pool (dynamic partition).
_partition_create_at	Creates a partition at a specific location outside the default memory pool (static partition).
_partition_create_component	Creates the partition component.
_partition_destroy	Destroys a dynamic partition that has no allocated partition blocks.
_partition_extend	Adds memory to a static partition; the added memory is divided into partition blocks that are the same size as other blocks in the partition.
_partition_free	Returns a partition block to a partition.
_partition_get_block_size	Gets the size of partition blocks in a partition.
_partition_get_free_blocks	Gets the number of free partition blocks in a partition.
_partition_get_max_used_blocks	Gets the number of allocated partition blocks in a partition; this is, a highwater mark that indicates the maximum number that have been allocated simultaneously, not necessarily the number that are currently allocated.
_partition_get_total_blocks	Gets the number of partition blocks in a partition.
_partition_get_total_size	Gets the size of a partition, including extensions.
_partition_test	Tests the partition component.
_partition_transfer	Transfers ownership of a partition block to another task (including the system); only the new owner can free the partition block.

3.5.3 Controlling Caches

MQX RTOS functions let you control the instruction cache and data cache that some CPUs have.

So that you can write an application that applies to both cached and non-cached systems, MQX RTOS wraps the functions in macros. For CPUs that do not have the cache, the macros do not map to a function. Some CPUs implement a unified cache (one cache is used for both data and code), in which case, the **_DCACHE_** and **_ICACHE_** macros map to the same function.

3.5.3.1 Flushing Data Cache

MQX RTOS uses the term flush to mean flushing the entire data cache. Unwritten data that is in the cache is written to physical memory.

3.5.3.2 Invalidating Data or Instruction Cache

MQX RTOS uses the term invalidate to mean invalidating all the cache entries. Data or instructions that are left in the cache, and have not been written to memory, are lost. A subsequent access reloads the cache with data or instructions from physical memory.

Table 3-10. Summary: Controlling Data Caches

_DCACHE_DISABLE	Disables the data cache.
_DCACHE_ENABLE	Enables the data cache.
_DCACHE_FLUSH	Flushes the entire data cache.
_DCACHE_FLUSH_LINE	Flushes the data-cache line containing the specified address.
_DCACHE_FLUSH_ MLINES	Flushes the data-cache lines containing the specified memory region.
_DCACHE_INVALIDATE	Invalidates the data cache.
_DCACHE_INVALIDATE_ LINE	Invalidates the data-cache line containing the specified address.
_DCACHE_INVALIDATE_ MLINES	Invalidates the data-cache lines containing the specified memory region.

Table 3-11. Summary: Controlling Instruction Caches

_ICACHE_DISABLE	Disables the instruction cache.
_ICACHE_ENABLE	Enables the instruction cache.
_ICACHE_INVALIDATE	Invalidates the instruction cache.
_ICACHE_INVALIDATE_ LINE	Invalidates the instruction cache line containing the specified address.
_ICACHE_INVALIDATE_MLINES	Invalidates the instruction cache lines containing the specified memory region.

Note	The flushing and invalidating functions always operate with whole cache lines. In case the data entity is not aligned to the cache line size, these operations affect data that precedes and follows data area currently being flushed/invalidated. The MQX RTOS memory allocators align data entity to the cache line size by default. Once an entity is declared statically the alignment to the cache line size is not guaranteed (unless align pragma is used).

3.5.4 Controlling the MMU (Virtual Memory)

For some CPUs, you must initialize the memory management unit (MMU) before you enable caches. MQX RTOS functions let you initialize, enable, and disable an MMU, and add a memory region to it. MMU functions are not supported on all architectures.

You can control an MMU by using MMU page tables.

The virtual memory component lets an application control the MMU page tables.

The following diagram shows the relationship between virtual address, MMU page tables, MMU pages, physical page, and physical address.

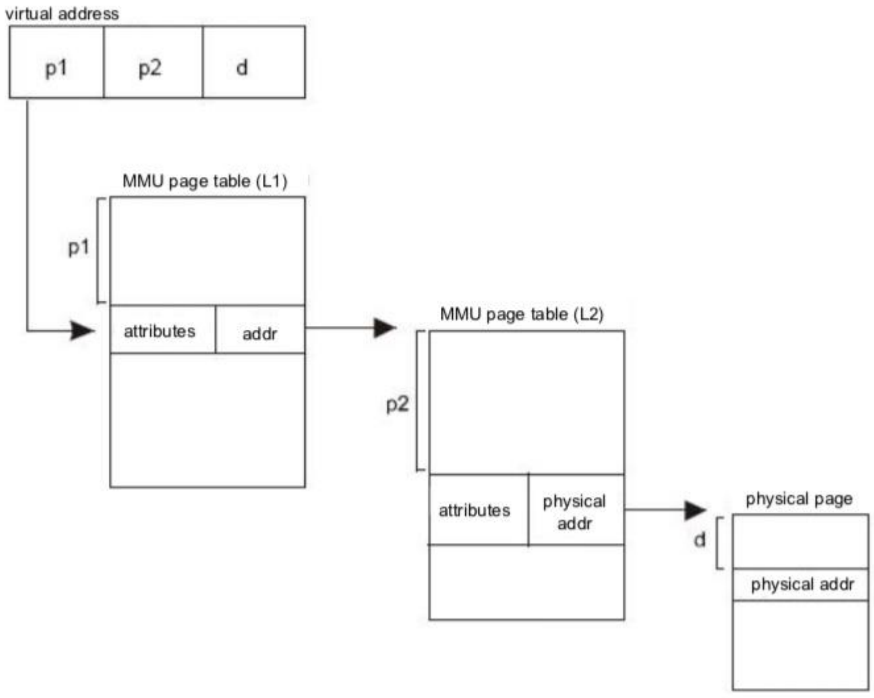

Figure 3-2. Virtual and Physical Addresses

With the virtual memory component, an application can manage virtual memory, which maps to physical addresses.

An application can use the virtual memory component to create a virtual context for a task. Virtual context provides memory that is private to a task, and is visible only while the task is the active task.

The functions are called when the BSP is initialized.

Table 3-12. Summary: Managing Virtual Memory

_mmu_add_vcontext	Adds a memory region to a virtual context.
_mmu_add_vregion	Adds a memory region to the MMU page tables that all tasks and MQX RTOS can use.
_mmu_create_vcontext	Creates a virtual context for a task.
_mmu_create_vtask	Creates a task with an initialized virtual context.
_mmu_destroy_vcontext	Destroys a virtual context for a task.
_mmu_get_vmem_attributes	Gets the virtual memory attributes of an MMU page.
_mmu_get_vpage_size	Gets the size of an MMU page.
_mmu_set_vmem_attributes	Modifies the virtual memory attributes of an MMU page.
_mmu_vdisable	Disables virtual memory.
_mmu_venable	Enables virtual memory.
_mmu_vinit	Initializes the MMU to use MMU page tables.
_mmu_vtop	Gets the physical address that corresponds to a virtual address.

3.5.4.1 Example: Initializing the MMU with Virtual Memory

Add a number of memory regions to support both instruction caching and data caching. All tasks can access the regions.

```
_mqx_uint _bsp_enable_operation(void)
{
   ...
   _mmu_vinit(MPC860_MMU_PAGE_SIZE_4K, NULL);
   /* Set up and initialize the instruction cache: */
   _mmu_add_vregion(BSP_FLASH_BASE, BSP_FLASH_BASE,
      BSP_FLASH_SIZE, PSP_MMU_CODE_CACHE |
      PSP_MMU_CACHED);
   _mmu_add_vregion(BSP_DIMM_BASE, BSP_DIMM_BASE,
      BSP_DIMM_SIZE, PSP_MMU_CODE_CACHE | PSP_MMU_CACHED);
   _mmu_add_vregion(BSP_RAM_BASE, BSP_RAM_BASE,
      BSP_RAM_SIZE, PSP_MMU_CODE_CACHE | PSP_MMU_CACHED);
   /* Set up and initialize the data cache: */
   _mmu_add_vregion(BSP_FLASH_BASE, BSP_FLASH_BASE,
      BSP_FLASH_SIZE, PSP_MMU_DATA_CACHE |
```

```
      PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_PCI_MEMORY_BASE,
      BSP_PCI_MEMORY_BASE, BSP_PCI_MEMORY_SIZE,
      PSP_MMU_DATA_CACHE | PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_PCI_IO_BASE,
      BSP_PCI_IO_BASE, BSP_PCI_IO_SIZE,
      PSP_MMU_DATA_CACHE |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_DIMM_BASE, BSP_DIMM_BASE,
      BSP_DIMM_SIZE, PSP_MMU_DATA_CACHE |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_add_vregion(BSP_RAM_BASE,
      BSP_RAM_BASE,    BSP_COMMON_RAM_SIZE,
      PSP_MMU_DATA_CACHE                  |
      PSP_MMU_CACHE_INHIBITED);
   _mmu_venable();
   _ICACHE_ENABLE(0);
   _DCACHE_ENABLE(0);
   ...
}
```

3.5.4.2 Example: Setting Up a Virtual Context

Set the active task to access 64 KB of private memory at 0xA0000000.

```
...
{
void *
        virtual_mem_pt
r; uint32_t
        size;
virtual_mem_ptr = (void
*)0xA0000000; size = 0x10000L;
...

result =
_mmu_create_vcontext(MQX_NULL_TASK_ID); if
(result != MQX_OK) {
}
result =
   _mmu_add_vcontext(MQX_NULL_TASK_ID,
   virtual_mem_ptr, size, 0);
if (result != MQX_OK) {
}
...
```

3.5.4.3 Example: Creating Tasks with a Virtual Context

Create tasks with a virtual context and a copy of common data.

```
...
/* Task template number for the virtual-context
task: */ #define VMEM_TTN    10
/* Global variable: */
```

```
unsigned char * data_to_duplicate[0x10000] = { 0x1, 0x2, 0x3 };
...
{
void *  virtual_mem_ptr;
virtual_mem_ptr = (void
*)0xA0000000;
...
result = _mmu_create_vtask(VMEM_TTN, 0,
   &data_to_duplicate, virtual_mem_ptr,
   sizeof(data_to_duplicate), 0);
if (result == MQX_NULL_TASK_ID) {
}
result = _mmu_create_vtask(VMEM_TTN, 0,
   &data_to_duplicate, virtual_mem_ptr,
   sizeof(data_to_duplicate), 0);
if (result == MQX_NULL_TASK_ID) {
  }
...
}
```

3.6 Synchronizing Tasks

You can synchronize tasks by using one or more of the following mechanisms, which are described in subsequent sections:

- Events - tasks can wait for a combination of event bits to become set. A task can set or clear a combination of event bits.
- Lightweight events - simpler implementation of events.
- Semaphores - tasks can wait for a semaphore to be incremented from non-zero. A task can post (increment) the semaphore. MQX RTOS semaphores prevent priority inversion by providing priority inheritance. For a discussion of priority inversion, see page Priority Inversion.
- Lightweight semaphores - simple counting semaphores.
- Mutexes - tasks can use a mutex to ensure that only one task at a time accesses shared data. To access shared data, a task locks a mutex, waiting if the mutex is already locked. When the task is finished accessing the shared data, it unlocks the mutex. Mutexes prevent priority inversion by providing priority inheritance and priority protection. For details, see page Mutexes.
- Message passing - lets tasks transfer data between themselves. A task fills a message with data and sends it to a particular message queue. Another task waits for messages to arrive at the message queue (receives messages).
- Lightweight Message Queue - simpler implementation of Messages.
- Task queues - let an application suspend and resume tasks.

3.6.1 Events

Events can be used to synchronize a task with another task or with an ISR.

The event component consists of event groups, which are groupings of event bits. The number of event bits in an event group is the number of bits in **_mqx_uint**.

Any task can wait for event bits in an event group. If the event bits are not set, the task blocks. Any other task or ISR can set the event bits. When the event bits are set, MQX RTOS puts all waiting tasks, whose waiting condition is met, into the task's ready queue. If the event group has autoclearing event bits, MQX RTOS clears the event bits as soon as they are set, and makes one task ready.

Note	To optimize code and data memory requirements on some target platforms, the event component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file, and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

There can be named event groups, which are identified by a unique string name, and fast event groups, which are identified by a unique number.

An application can open an event group on a remote processor by specifying the processor number in the string that it uses to open the event group. After opening the remote-processor event group, an application can set any event bit in the event group. An application cannot wait for event bits in a remote event group.

Table 3-13. Summary: Using the Event Component

Event[1]	Description
_event_clear	Clears the specified event bits in an event group.
_event_close	Closes a connection to an event group.
_event_create	Creates a named event group.
_event_create_auto_clear	Creates a named event group with autoclearing event bits.
_event_create_component	Creates the event component.
_event_create_fast	Creates a fast event group.
_event_create_fast_auto_clear	Creates a fast event group with autoclearing event bits.
_event_destroy	Destroys a named event group.
_event_destroy_fast	Destroys a fast event group.
_event_get_value	Gets the value of an event group.
_event_get_wait_count	Gets the number of tasks waiting for event bits in an event group.

_event_open	Opens a connection to a named event group.
_event_open_fast	Opens a connection to a fast event group.
_event_set	Sets the specified event bits in an event group on the local processor or on a remote processor.
_event_test	Tests the event component.
_event_wait_all	Waits for all the specified event bits in an event group for a specified number of milliseconds.
_event_wait_all_for	Waits for all the specified event bits in an event group for a specified tick-time period (including hardware ticks).
_event_wait_all_ticks	Waits for all the specified event bits in an event group for a specified number of ticks.
_event_wait_all_until	Waits for all the specified event bits in an event group until a specified tick time.
_event_wait_any	Waits for any of the specified event bits in an event group for a specified number of milliseconds.
_event_wait_any_for	Waits for any of the specified event bits in an event group for a specified tick time period.
_event_wait_any_ticks	Waits for any of the specified event bits in an event group for a specified number of ticks.
_event_wait_any_until	Waits for any of the specified event bits in an event group until a specified tick time.

1. Events use certain structures and constants, which are defined in event.h.

3.6.1.1 Creating the Event Component

You can explicitly create the event component with **_event_create_component()**. If you do not explicitly create it, MQX RTOS creates it with default values the first time an application creates an event group.

Table 3-14. Default Event Component Values

Parameter	**Meaning**	**Default**
Initial number	Initial number of event groups that can be created	8
Grow number	Number of additional event groups that can be created if all the event groups are created, until the maximum number is reached	8
Maximum number	If grow number is not 0, maximum number of event groups that can be created	0 (unlimited)

3.6.1.2 Creating an Event Group

Before a task can use the event component, it must create an event group.

Table 3-15. Event Group Creation

To create this type of event group:	Call:	With:
Fast (with autoclearing event bits)	**_event_create_fast()** **_event_create_fast_ auto_clear()**	Index (must be within the limits specified, when the event component was created)
Named (with autoclearing event bits)	**_event_create()** **_event_create_auto_ clear()**	String name

If an event group is created with autoclearing event bits, MQX RTOS clears the bits as soon as they are set. This action makes ready any tasks that are waiting for the bits, without the tasks having to clear the bits.

3.6.1.3 Opening a Connection to an Event Group

Before a task can use the event component, it must open a connection to a created event group.

Table 3-16. Event Group Open

To open a connection to this type of event group:	Call:	With:
Fast	**_event_open_fast()**	Index, which must be within the limits that were specified, when the event component was created.
Named	**_event_open()**	String name

Both functions return a unique handle to the event group.

3.6.1.4 Waiting for Event Bits (Events)

A task waits for a pattern of event bits (a mask) in an event group with **_event_wait_all()** or **_event_wait_any()**. When a bit is set, MQX RTOS makes ready the tasks that are waiting for the bit. If the event group is created with autoclearing event bits (**_event_create_auto_clear()** or **_event_create_fast_auto_clear()**), MQX RTOS clears the bit so that the waiting tasks need not clear it.

3.6.1.5 Setting Event Bits

A task can set a pattern of event bits (a mask) in an event group with **_event_set()**. The event group can be local or on a remote processor. When an event bit is set, tasks waiting for the bit are made ready. If the event group is created with autoclearing event bits, MQX RTOS clears the bits as soon as they are set.

3.6.1.6 Clearing Event Bits

A task can clear a pattern of event bits (a mask) in an event group with **_event_clear()**. However, if the event group is created with autoclearing event bits, MQX RTOS clears the bits as soon as they are set.

3.6.1.7 Closing a Connection to an Event Group

When a task no longer needs to use an event group, it can close its connection to the group with **_event_close()**.

3.6.1.8 Destroying an Event Group

If tasks are blocked, waiting for an event bit in the to-be-destroyed event group, MQX RTOS moves them to their ready queues.

3.6.1.9 Example: Using Events

Simulated_tick ISR sets an event bit each time it runs. Service task performs a certain action each time a tick occurs, and therefore waits for the event bit that Simulated_tick sets.

3.6.1.9.1 Code for the Using Events Example

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


#include <fio.h>


#include


<event.h>
/* Task IDs */
#define SERVICE_TASK 5
#define ISR_TASK   6
/* Function Prototypes */
```

```
extern void simulated_ISR_task(uint32_t);

extern void service_task(uint32_t);
const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function,         Stack, Prio, Name,       Attributes,    Param, TS */
 { SERVICE_TASK, service_task,   2000,  8, "service",    MQX_AUTO_START_TASK, 0, 0},
 { ISR_TASK,   simulated_ISR_task, 2000,                                   8,
 "simulated_ISR", 0,                   0, 0},
 { 0 }
};
/*TASK*-----------------------------------------------------
*
* Task Name   : simulated_ISR_task
* Comments    :
*    This task opens a connection to the event. After
        *    delaying the event bits are set.
        *END* --------------------------------------------*/
        void simulated_ISR_task(uint32_t initial_data)
        {
           void *   event_ptr;
           /* open event connection */
           if (_event_open("global", &event_ptr) !=
              MQX_OK) { printf("\nOpen Event failed");
              _mqx_exit(0);
           }

           while (TRUE) {

               _time_delay(1000);
              if (_event_set(event_ptr, 0x01) !=
                 MQX_OK) { printf("\nSet Event
                 failed");
              _mqx_exit(0);
       }
   }
}

/*TASK*-----------------------------------------------------
*
* Task Name   : service_task
* Comments    :
*    This task creates an event and the simulated_ISR_task
*    task. It opens a connection to the event and waits.
*    After all bits have been set "Tick" is printed and
*    the event is cleared.
*END*---------------------------------------------*/
void service_task(uint32_t initial_data)
{
   void *  event_ptr;
   _task_id second_task_id;
   /* setup event */
   if (_event_create("global") !=
      MQX_OK) { printf("\nMake event
      failed");
      _mqx_exit(0);
   }
   if (_event_open("global", &event_ptr) !=
      MQX_OK) { printf("\nOpen event failed");
```

```
      _mqx_exit(0);

   }
   /* create task */
   second_task_id = _task_create(0,
      ISR_TASK, 0); if (second_task_id ==
      MQX_NULL_TASK_ID) {
         printf("Could not create simulated_ISR_task \n");
           _mqx_exit(0);

      }
   while (TRUE) {
      if (_event_wait_all(event_ptr, 0x01, 0) !=
         MQX_OK) { printf("\nEvent Wait failed");
           _mqx_exit(0);

      }
      if (_event_clear(event_ptr, 0x01) !=
         MQX_OK) { printf("\nEvent Clear
         Failed");
           _mqx_exit(0);

      }
      printf(" Tick \n");
   }
}
```

3.6.1.9.2 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   ```
   mqx\examples\event
   ```

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.

Event task prints a message each time an event bit is set.

3.6.2 Lightweight Events

Lightweight events are a simpler, low-overhead implementation of events.

The lightweight event component consists of lightweight event groups, which are groupings of event bits. The number of event bits in a lightweight event group is the number of bits in **_mqx_uint**.

Any task can wait for event bits in a lightweight event group. If the event bits are not set, the task blocks. Any other task or ISR can set the event bits. When the event bits are set, MQX RTOS puts all waiting tasks, whose waiting condition is met, into the task's ready queue. If the lightweight event group has autoclearing event bits, MQX RTOS clears the event bits as soon as they are set and makes one task ready.

Lightweight event groups are created from static-data structures and are not multi-processor.

Table 3-17. Summary: Using the Lightweight Event Component

Event[1]	Description
_lwevent_clear	Clears the specified event bits in a lightweight event group.
_lwevent_create	Creates a lightweight event group, indicating whether it has autoclearing event bits.
_lwevent_destroy	Destroys a lightweight event group.
_lwevent_set	Sets the specified event bits in a lightweight event group.
_lwevent_test	Tests the lightweight event component.
_lwevent_wait_for	Waits for all or any of the specified event bits in a lightweight event group for a specified tick-time period.
_lwevent_wait_ticks	Waits for all or any of the specified event bits in a lightweight event group for a specified number of ticks.
_lwevent_wait_until	Waits for all or any of the specified event bits in a lightweight event group until a specified tick time.

1. Lightweight events use certain structures and constants, which are defined in lwevent.h.

3.6.2.1 Creating a Lightweight Event Group

To create a lightweight event group, an application declares a variable of type **LWEVENT_STRUCT**, and initializes it by calling **_lwevent_create()** with a pointer to the variable and a flag indicating, whether the event group has autoclearing event bits.

3.6.2.2 Waiting for Event Bits

A task waits a pattern of event bits (a mask) in a lightweight event group with one of the **_lwevent_wait** functions. If the waiting condition is not met, the function waits for a specified time to expire.

3.6.2.3 Setting Event Bits

A task sets a pattern of event bits (a mask) in a lightweight event group with **_lwevent_set()**. If tasks are waiting for the appropriate bits, MQX RTOS makes them ready. If the event group has autoclearing event bits, MQX RTOS makes ready only the first task that is waiting.

3.6.2.4 Clearing Event Bits

A task can clear a pattern of event bits (a mask) in a lightweight event group with **_lwevent_clear()**. However, if the lightweight event group is created with autoclearing event bits, MQX RTOS clears the bits as soon as they are set.

3.6.2.5 Destroying a Lightweight Event Group

When a task no longer needs a lightweight event group, it can destroy the event group with **_lwevent_destroy()**.

3.6.3 About Semaphore-Type Objects

MQX RTOS provides lightweight semaphores (LWSems), semaphores, and mutexes.

You can use both types of semaphores for task synchronization and mutual exclusion. A task waits for a semaphore. If the semaphore count is zero, MQX RTOS blocks the task; otherwise, MQX RTOS decrements the semaphore count, gives the task the semaphore, and the task continues to run. When the task is finished with the semaphore, it posts the semaphore; the task remains ready. If a task is waiting for the semaphore, MQX RTOS puts the task in the task ready queue; otherwise, MQX RTOS increments the semaphore count.

You can use mutexes for mutual exclusion. A mutex is sometimes called a binary semaphore because its counter can be only zero or one.

3.6.3.1 Strictness

If a semaphore-type object is strict, a task must first wait for and get the object, before it can release the object. If the object is non-strict, a task does not need to get the object before it releases the object.

3.6.3.2 Priority Inversion

Task priority inversion is a classic condition, where the relative priorities of tasks appear to be reversed. Priority inversion might occur, when tasks use semaphores or mutexes to gain access to a shared resource.

3.6.3.3 Example: Priority Inversion

There are three tasks of three different priorities. The mid-priority task prevents the highest-priority task from running.

Table 3-18. Priority Inversion Example

Sequence	Task_1 (highest priority P1)	Task_2 (mid priority P2)	Task_3 (lowest priority P3)
1 2			• Runs • Gets semaphore
3		• Is made ready	
4		• Preempties Task_3 and runs	
5	• Is made ready		
6	• Preempties Task_2 and runs		
7	• Tries to get semaphore that Task_3 has		
8	• Blocks, waiting for the semaphore		
9		• Runs and keeps running	

3.6.3.4 Avoiding Priority Inversion with Priority Inheritance

When you create an MQX RTOS semaphore or mutex, one of the properties that you can specify is priority inheritance, which prevents priority inversion.

If you specify priority inheritance, during the time that a task has locked a semaphore or mutex, the task's priority is never lower than the priority of any task that waits for the semaphore or mutex. If a higher-priority task waits for the semaphore or mutex, MQX RTOS temporarily raises the priority of the task that has the semaphore or mutex to the priority of the waiting task.

Table 3-19. Priority Inheritance Properties

Sequence	**Task_1 (highest priority P1)**	**Task_2 (mid priority P2)**	**Task_3 (lowest priority P3)**
1			• Runs
2			• Gets semaphore
3		• Is made ready	
4		• Preempties Task_3	

		and runs	
5	• Is made ready		
6	• Preempties Task_2 and runs		
7	• Tries to get semaphore that Task_3 has		
8	• Raises priority of Task_3 to P1 and blocks		
9			• Preempts Task_1 and runs
10			• Finishes work and posts semaphore
11			• Priority is lowered to P3
12	• Preempts Task_3 and Task_2 and runs		
13	• Gets semaphore		

3.6.3.5 Avoiding Priority Inversion with Priority Protection

When you create an MQX RTOS mutex, you can specify the mutex attributes of priority protection and a mutex priority. These attributes prevent priority inversion.

If the priority of a task that requests to lock the mutex is not at least as high as the mutex priority, MQX RTOS temporarily raises the task's priority to the mutex priority for as long, as the task has the mutex locked.

Table 3-20. Mutex Attributes

Sequence	Task_1 (highest priority P1)	Task_2 (mid priority P2)	Task_3 (lowest priority P3)
1			• Runs
2			• Locks mutex (with priority P1); priority is boosted to P1
3		• Is made ready	
4		• Does not preempt Task_3	
5	• Is made ready		
6	• Does not preempt Task_3		
7			• Finishes with mutex and unlocks it
8			• Priority is lowered to P3
9	• Preempts Task_3 and runs		

10	• Locks mutex		

Table 3-21. Comparison of Lightweight Semaphores, Semaphores, and Mutexes

Feature	LWSem	Semaphore	Mutex
Timeout	Yes	Yes	No
Queuing	FIFO	FIFO Priority	FIFO Priority Spin only Limited spin
Strict	No	No or yes	Yes
Priority inheritance	No	Yes	Yes
Priority protection	No	No	Yes
Size	Smallest	Largest	Between lightweight semaphores and semaphores
Speed	Fastest	Slowest	Between lightweight semaphores and semaphores

3.6.4 Lightweight Semaphores

Lightweight semaphores are a simpler, low-overhead implementation of semaphores.

Lightweight semaphores are created from static-data structures, and are not multi-processor.

Table 3-22. Summary: Using Lightweight Semaphores

_lwsem_create	Creates a lightweight semaphore.
_lwsem_destroy	Destroys a lightweight semaphore.
_lwsem_poll	Polls for a lightweight semaphore (non-blocking).
_lwsem_post	Posts a lightweight semaphore.
_lwsem_test	Tests the lightweight semaphore component.
_lwsem_wait	Waits for a lightweight semaphore.
_lwsem_wait_for	Waits for a lightweight semaphore for a specified tick-time period.
_lwsem_wait_ticks	Waits for a lightweight semaphore for a specified number of ticks.
_lwsem_wait_until	Waits for a lightweight semaphore, until a specified number of ticks have elapsed.

3.6.4.1 Creating a Lightweight Semaphore

To create a lightweight semaphore, you declare a variable of type **LWSEM_STRUCT**, and initialize it by calling _lwsem_create() with a pointer to the variable and an initial semaphore count. The semaphore count, which indicates the number of requests that can be concurrently granted the lightweight semaphore, is set to the initial count.

3.6.4.2 Waiting for and Posting a Lightweight Semaphore

A task waits for a lightweight semaphore with **_lwsem_wait()**. If the semaphore count is greater than zero, MQX RTOS decrements it, and the task continues to run. If the count is zero, MQX RTOS blocks the task, until some other task posts the lightweight semaphore.

To release a lightweight semaphore, a task posts it with **_lwsem_post()**. If no tasks are waiting for the lightweight semaphore, MQX RTOS increments the semaphore count.

Since lightweight semaphores are non-strict, tasks can post without waiting first; therefore, the semaphore count is not bounded and can increase beyond the initial count.

3.6.4.3 Destroying a Lightweight Semaphore

When a task no longer needs a lightweight semaphore, it can destroy it with **_lwsem_destroy()**.

3.6.4.4 Example: Producers and Consumer

Producer and consumer tasks synchronize each other with lightweight semaphores.

1. Read task creates:
   - Multiple Write tasks and assigns a unique character to each.
   - One write LWSem.
   - One read LWSem.
2. Each Write task waits for the Write LWSem, before it writes a character into the buffer. When the character is written, each Write task posts the Read LWSem, signaling that a character is available to the Read task.
3. Read waits for the Read LWSem, before it consumes the character. After it consumes the character, it posts the Write LWSem, signaling that the buffer is ready for another character.

3.6.4.4.1 Definitions and Structures for the Example

```
/* read.h */
/* Number of Writer Tasks
*/ #define NUM_WRITERS 3
/* Task IDs */
#define WRITE_TASK 5
#define READ_TASK  6
/* Global data structure accessible by read and write tasks.
** Contains two lightweight semaphores that govern access to the
** data variable.
*/
typedef struct sw_fifo
```

```
{
   LWSEM_STRUCT  READ_SEM;

   LWSEM_STRUCT  WRITE_SEM;
   uchar         DATA;
} SW_FIFO, _PTR_ SW_FIFO_PTR;

/* Function prototypes */
extern void write_task(uint32_t initial_data);
extern void read_task(uint32_t initial_data);
extern SW_FIFO fifo;
```

3.6.4.4.2 Task Templates for the Producers and Consumers Example

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

3.6.4.4.3 Code for a Write Task

```
/* write.c */

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

      }
      fifo.DATA = (uchar)initial_data;
      _lwsem_post(&fifo.READ_SEM);
   }
}
```


3.6.4.4.4 Code for Read Task

```
/* read.c */

#include <mqx.h>
#include <bsp.h>
#include "read.h"
SW_FIFO    fifo;
/*TASK*-----------------------------------------------------
*
* Task Name : read_task
* Comments : This task creates two semaphores and
*            NUM_WRITER write_tasks. Then it waits
*            on the read_sem and finally outputs the
*            "data" variable.
*END*----------------------------------------------- */
void read_task(uint32_t initial_data)
{
   _task_id   task_id;
   _mqx_uint  result;
   _mqx_uint  i;
   /* Create the lightweight semaphores */
   result = _lwsem_create(&fifo.READ_SEM, 0);
   if (result != MQX_OK) {
      printf("\nCreating read_sem failed: 0x%X", result);
      _mqx_exit(0);

   }
   result = _lwsem_create(&fifo.WRITE_SEM, 1);
   if (result != MQX_OK) {
      printf("\nCreating write_sem failed: 0x%X", result);
      _mqx_exit(0);

   }
   /* Create write tasks */
   for (i = 0; i < NUM_WRITERS; i++) {
      task_id = _task_create(0, WRITE_TASK, (uint32_t)('A' + i));
      printf("\nwrite_task created, id 0x%lX", task_id);
   }
   while (TRUE) {
      result = _lwsem_wait(&fifo.READ_SEM);
      if (result != MQX_OK) {
         printf("\n_lwsem_wait failed: 0x%X", result);
         _mqx_exit(0);

      }
      putchar('\n');
      putchar(fifo.DATA
      );
      _lwsem_post(&fifo.WRITE_SEM);
   }
}
```

3.6.4.4.5 Compiling the Application and Linking It with MQX RTOS

1. Go to this directory:

```
mqx\examples\lwsem
```


2. See the *MQX™ RTOS Release Notes document* (document MQXRN) for instructions on how to build and run the application.

The following appears on the output device:

```
A
A
B
C
A
B
...
```

3.6.5 Semaphores

Semaphores can be used for task synchronization and mutual exclusion. The main operations that a task performs on a semaphore, are to wait for the semaphore and to post the semaphore.

Note	To optimize code and data memory requirements on some target platforms, the Semaphore component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Table 3-23. Summary: Using Semaphores

Semaphore[1]	Description
_sem_close	Closes a connection to a semaphore.
_sem_create	Creates a semaphore.
_sem_create_component	Creates the semaphore component.
_sem_create_fast	Creates a fast semaphore.
_sem_destroy	Destroys a named semaphore.
_sem_destroy_fast	Destroys a fast semaphore.
_sem_get_value	Gets the current semaphore count.
_sem_get_wait_count	Gets the number of tasks waiting for a semaphore.
_sem_open	Opens a connection to a named semaphore.
_sem_open_fast	Opens a connection to a fast semaphore.
_sem_post	Posts (frees) a semaphore.

_sem_test	Tests the semaphore component.
_sem_wait	Waits for a semaphore for a number of milliseconds.
_sem_wait_for	Waits for a semaphore for a tick-time period.
_sem_wait_ticks	Waits for a semaphore for a number of ticks.
_sem_wait_until	Waits for a semaphore until a time (in tick time).

1. Semaphores use certain structures and constants, which are defined in sem.h.

3.6.5.1 Using a Semaphore

To use a semaphore, a task executes the following steps, each of which is described in subsequent sections.

1. Optionally, creates the semaphore component.
2. Creates the semaphore.
3. Opens a connection to the semaphore.
4. If the semaphore is strict, it waits for the semaphore.
5. When finished using the semaphore for the time being, it posts the semaphore.
6. If it no longer needs the semaphore, it closes its connection to the semaphore.
7. If the semaphore is protecting a shared resource that ceases to exist or is no longer accessible, the task can destroy the semaphore.

3.6.5.2 Creating the Semaphore Component

You can explicitly create the semaphore component with **_sem_create_component()**. If you do not explicitly create it, MQX RTOS creates it with default values the first time an application creates a semaphore.

The parameters and their default values are the same as for the event component, which is described on page Creating the Event Component.

3.6.5.3 Creating a Semaphore

Before a task can use a semaphore, it must create the semaphore.

Table 3-24. Semaphore Creation

Semaphore Type	Call	With
Fast	**_sem_create_fast()**	Index, which must be within the limits that were specified when the semaphore component was created.

Named	**_sem_create()**	String name

When the task creates the semaphore, it also specifies:

- Initial count - the initial value for the semaphore count represents the number of locks that the semaphore has. (A task can get multiple locks).
- Priority queuing - if priority queuing is specified, the queue of tasks waiting for the semaphore is in priority order, and MQX RTOS puts the semaphore to the highest-priority waiting task.
- If priority queuing is not specified, the queue is in FIFO order, and MQX RTOS puts the semaphore to the longest-waiting task.
- Priority inheritance - if priority inheritance is specified and a higher-priority task is waiting for the semaphore, MQX RTOS raises the priority of the tasks that have the semaphore to the priority of the waiting task. For more information, see the discussion on priority inheritance on page Avoiding Priority Inversion with Priority Inheritance. To use priority inheritance, the semaphore must be strict.
- Strictness - if strictness is specified, a task must wait for the semaphore, before it can post the semaphore. If a semaphore is strict, the initial count is the maximum value of the semaphore count. If the semaphore is non-strict, the count is unbounded.

3.6.5.4 Opening a Connection to a Semaphore

Before a task can use a semaphore, it must open a connection to the semaphore.

Table 3-25. Opening a Connection to a Semaphore

Semaphore Type	Call	With
Fast	**_sem_open_fast()**	Index, which must be within the limits that were specified when the semaphore component was created.
Named	**_sem_open()**	String name

Both functions return a unique handle to the semaphore.

3.6.5.5 Waiting for a Semaphore and Posting a Semaphore

A task waits for a semaphore using one of the functions from the **_sem_wait**_family of functions. If the semaphore count is zero, MQX RTOS blocks the task, until another task posts (**_sem_post()**) the semaphore or the task-specified timeout expires. If the count is not zero, MQX RTOS decrements the count, and the task continues to run.

When a task posts a semaphore, and there are tasks waiting for the semaphore, MQX RTOS puts them in their ready queues. If there are no tasks waiting, MQX RTOS increments the semaphore count. In either case, the posting task remains ready.

3.6.5.6 Closing a Connection to a Semaphore

When a task no longer needs to use a semaphore, it can close its connection with the semaphore with _sem_close().

3.6.5.7 Destroying a Semaphore

When the semaphore is no longer needed, a task can destroy it.

Table 3-26. Semaphore Destroying

Semaphore Type	Call	With
Fast	**_sem_destroy_fast()**	Index, which must be within the limits that were specified when the semaphore component was created.
Named	**_sem_destroy()**	String name

As well, the task can specify, whether to force destruction. If destruction is forced, MQX RTOS readies tasks that are waiting for the semaphore, and destroys the semaphore after all the tasks that have the semaphore post the semaphore.

If destruction is not forced, MQX RTOS destroys the semaphore after the last waiting task gets and posts the semaphore. (This is always the action if the semaphore is strict).

3.6.5.8 Example: Task Synchronization and Mutual Exclusion

This example builds on the lightweight semaphore example on page Example: Producers and Consumer. It shows, how semaphores can be used for task synchronization and mutual exclusion.

The example manages a FIFO that multiple tasks can write to and read from. Mutual exclusion is required for access to the FIFO data structure. Task synchronization is required to block the writing tasks when the FIFO is full, and to block the reading tasks when the FIFO is empty. Three semaphores are used:

- Index semaphore for mutual exclusion on the FIFO.
- Read semaphore to synchronize the readers.
- Write semaphore to synchronize the writers.

The example consists of three tasks: Main, Read, and Write. Main initializes the semaphores, and creates Read and Write.

3.6.5.8.1 Definitions and Structures for the Example

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

/* Function prototypes */
extern void main_task(uint32_t initial_data);
extern void write_task(uint32_t initial_data);
extern void read_task(uint32_t initial_data);
extern      SW_FIFO fifo;
```

3.6.5.8.2 Task Templates for the Task Synchronization and Mutual Exclusion Example

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


3.6.5.8.3 Code for Main Task

The Main task creates:

- The semaphore component
- The Index, Read, and Write semaphores
- Read and Write tasks

```
/* main.c */

#include <mqx.h>
#include <bsp.h>
#include <sem.h>
#include "main.h"
SW_FIFO  fifo;
/*TASK*-----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*     This task initializes three semaphores, creates NUM_WRITERS
*     write_tasks, and creates one read_task.
*END*--------------------------------------------------------- */
void main_task(uint32_t initial_data)
{
   _task_id  task_id;
   _mqx_uint i;
   fifo.READ_INDEX = 0;
   fifo.WRITE_INDEX = 0;
   /* Create semaphores: */
   if (_sem_create_component(3, 1, 6) != MQX_OK) {
      printf("\nCreating semaphore component failed");
      _mqx_exit(0);

   }
   if (_sem_create("write", ARRAY_SIZE, 0) != MQX_OK) {
      printf("\nCreating write semaphore failed");
      _mqx_exit(0);

   }
   if (_sem_create("read", 0, 0) != MQX_OK) {
      printf("\nCreating read semaphore failed");
      _mqx_exit(0);

   }
   if (_sem_create("index", 1, 0) != MQX_OK) {
      printf("\nCreating index semaphore failed");
      _mqx_exit(0);

   }
   /* Create tasks: */
   for (i = 0; i < NUM_WRITERS; i++) {
      task_id = _task_create(0, WRITE_TASK, i);
      printf("\nwrite_task created, id 0x%lx", task_id);
   }
   task_id = _task_create(0, READ_TASK, 0);
   printf("\nread_task created, id 0x%lx", task_id);
}
```

3.6.5.8.4 Code for the Read Task

```
/* read.c */

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

   }
   if (_sem_open("index", &index_sem) != MQX_OK) {
      printf("\nOpening index semaphore failed");
      _mqx_exit(0);

   }
   if (_sem_open("read", &read_sem) != MQX_OK) {
      printf("\nOpening read semaphore failed");
      _mqx_exit(0);

   }
   while (TRUE) {
      /* Wait for the semaphores: */
      if (_sem_wait(read_sem, 0) != MQX_OK) {
         printf("\nWaiting for read semaphore failed");
         _mqx_exit(0);

      }
      if (_sem_wait(index_sem, 0) != MQX_OK) {
        printf("\nWaiting for index semaphore failed");
        _mqx_exit(0);

      }
      printf("\n 0x%lx", fifo.DATA[fifo.READ_INDEX++]);
      if (fifo.READ_INDEX >=ARRAY_SIZE) {
         fifo.READ_INDEX = 0;
      }
      /* Post the semaphores: */
```

```
      _sem_post(index_sem);
      _sem_post(write_sem);
   }
}
```

3.6.5.8.5 Code for the Write Task

```
/* write.c */

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

   }
   if (_sem_open("index", &index_sem) != MQX_OK) {
      printf("\nOpening index semaphore failed");
      _mqx_exit(0);

   }
   if (_sem_open("read", &read_sem) != MQX_OK) {
      printf("\nOpening read semaphore failed");
      _mqx_exit(0);

   }
   while (TRUE) {
      /* Wait for the semaphores: */
      if (_sem_wait(write_sem, 0) != MQX_OK) {
         printf("\nWaiting for write semaphore failed");
         _mqx_exit(0);

      }
      if (_sem_wait(index_sem, 0) != MQX_OK) {
         printf("\nWaiting for index semaphore failed");
         _mqx_exit(0);

      }
      fifo.DATA[fifo.WRITE_INDEX++] = _task_get_id();

      if (fifo.WRITE_INDEX >=ARRAY_SIZE) {
         fifo.WRITE_INDEX = 0;
      }
```

```
      /* Post the semaphores: */
      _sem_post(index_sem);
      _sem_post(read_sem);
   }
}
```

3.6.5.8.6 Compiling the application and linking it with MQX RTOS

1. Go to this directory:

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

2. See the *MQX RTOS Release Notes* for instructions how to build and run the application.

Read task prints the data that is written to the FIFO.

Modify the program to remove priority inheritance, and run the application again.

3.6.6 Mutexes

Mutexes are used for mutual exclusion, so that only one task at a time uses a shared resource such as data or a device. To access the shared resource, a task locks the mutex associated with the resource. The task owns the mutex, until it unlocks the mutex.

Note	To optimize code and data memory requirements on some target platforms, the Mutex component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file, and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Mutexes provide priority inheritance and priority protection to prevent priority inversion.

Table 3-27. Summary: Using Mutexes

Mutex[1]	Description
_mutex_create_component	Creates the mutex component.
_mutex_destroy	Destroys a mutex.
_mutex_get_priority_ceiling	Gets the priority of a mutex.
_mutex_get_wait_count	Gets the number of tasks that are waiting for a mutex.
_mutex_init	Initializes a mutex.
_mutex_lock	Locks a mutex.
_mutex_set_priority_ceiling	Sets the priority of a mutex.
_mutex_test	Tests the mutex component.

_mutex_try_lock	Tries to lock a mutex.
_mutex_unlock	Unlocks a mutex.

1. Mutexes use certain structures and constants, which are defined in mutex.h.

3.6.6.1 Creating the Mutex Component

You can explicitly create the mutex component with **_mutex_create_component()**. If you do not explicitly create it, MQX RTOS creates it the first time an application initializes a mutex. There are no parameters.

3.6.6.2 Mutex Attributes

A mutex can have attributes with respect to its waiting and scheduling protocols.

3.6.6.3 Waiting Protocols

A mutex can have one of several waiting protocols, which affect tasks that request to lock an already locked mutex.

Table 3-28. Mutex Waiting Protocols

Waiting protocol[1]	Description
Queuing (default)	Blocks, until another task unlocks the mutex. When the mutex is unlocked, the first task (regardless of priority) that requested the lock, locks the mutex.
Priority queuing	Blocks, until another task unlocks the mutex. When the mutex is unlocked, the highest-priority task that requested the lock, locks the mutex.
Spin only	Spins (is timesliced) indefinitely, until another task unlocks the mutex. This means that MQX RTOS saves the requesting task's context, and dispatches the next task in the same-priority ready queue. When all the tasks in this ready queue have run, the requesting task becomes active again. If the mutex is still locked, the spin repeats.
Limited spin	Spins for a specified number of times, or fewer, if another task unlocks the mutex first.

1. If the mutex is already locked, the requesting task does this.

Spin-only protocol functions properly, only if the tasks that share the mutex are either:

- time-slice tasks
- the same priority

If non-time-slice tasks of different priority try to share a spin-only mutex, a higher-priority task that wants to lock the mutex that is locked by a lower-priority task never gets the lock (unless the lower-priority task blocks).

Spin-only protocol mutexes are prone to deadlock and are not recommended.

3.6.6.4 Scheduling Protocols

A mutex can have special scheduling protocols that avoid priority inversion. The policies might affect the priority of a task during the time that the task has the mutex locked. The default is for neither protocol to be in effect.

Table 3-29. Mutex Scheduling Protocols

Scheduling protocol	Meaning
Priority inheritance	If the priority of the task that has locked the mutex (task_A) is not as high as the highest-priority task that is waiting to lock the mutex (task_B), MQX RTOS raises the priority of task_A to be the same as the priority of task_B, while task_A has the mutex.
Priority protection	A mutex can have a priority. If the priority of a task that requests to lock the mutex (task_A) is not at least as high as the mutex priority, MQX RTOS raises the priority of task_A to the mutex priority for as long as task_A has the mutex locked.

3.6.6.5 Creating and Initializing a Mutex

A task creates a mutex by first defining a variable of type **MUTEX_STRUCT**.

To initialize the mutex with the default attributes of a queuing waiting protocol and no special scheduling protocols, the task calls **_mutex_init()** with a pointer to the mutex variable and a NULL pointer.

However, to initialize the mutex with attributes other than the default, the task does the following:

1. It defines a mutex attributes structure of type **MUTEX_ATTR_STRUCT**.
2. It initializes the attributes structure with **_mutatr_init()**.
3. It calls various functions to set the appropriate attributes, choosing from:
4.
    - **_mutatr_set_priority_ceiling()**
    - **_mutatr_set_sched_protocol()**
    - **_mutatr_set_spin_limit()**
    - **_mutatr_set_wait_protocol()**
5. It initializes the mutex by calling **_mutex_init()** with pointers to the mutex and to the attributes structure. When the mutex is initialized, any task can use it.
6. It destroys the mutex attributes structure with **_mutatr_destroy()**.

Table 3-30. Summary: Using a Mutex Attributes Structure

_mutatr_destroy	Destroys a mutex attributes structure.
_mutatr_get_priority_ceiling	Gets the priority of a mutex attributes structure.
_mutatr_get_sched_protocol	Gets the scheduling protocol of a mutex attributes structure.
_mutatr_get_spin_limit	Gets the limited-spin count of a mutex attributes structure.
_mutatr_get_wait_protocol	Gets the waiting policy of a mutex attributes structure.
_mutatr_init	Initializes a mutex attributes structure.
_mutatr_set_priority_ceiling	Sets the priority value in a mutex attributes structure.
_mutatr_set_sched_protocol	Sets the scheduling protocol of a mutex attributes structure.
_mutatr_set_spin_limit	Sets limited-spin count of a mutex attributes structure.
_mutatr_set_wait_protocol	Sets the waiting protocol of a mutex attributes structure.

3.6.6.6 Locking a Mutex

To access a shared resource, a task can lock the mutex that is associated with the resource by calling **_mutex_lock()**. If the mutex is not already locked, the task locks it and continues to run. If the mutex is already locked, depending on the mutex waiting protocols that are described on page Waiting Protocols, the task might block until the mutex is unlocked.

To be sure that it does not block, a task can try to lock a mutex with **_mutex_trylock()**. If the mutex is not already locked, the task locks it and continues to run. If the task is already locked, the task does not get the mutex, but continues to run.

3.6.6.7 Unlocking a Mutex

Only the task that has locked a mutex can unlock it (**_mutex_unlock()**).

3.6.6.8 Destroying a Mutex

If a mutex is no longer needed, a task can destroy it with **_mutex_destroy()**. If any tasks are waiting for the mutex, MQX RTOS puts them in their ready queues.

3.6.6.9 Example: Using a Mutex

A mutex is used for mutual exclusion. There are two time-slice tasks, both of which print to the same device. A mutex prevents the output from being interleaved.

3.6.6.9.1 Code for Using a Mutex Example

```
/* main.c */

#include <mqx.h>
#include <bsp.h>
#include <mutex.h>
/* Task IDs */
#define MAIN_TASK      5
#define PRINT_TASK     6
extern void main_task(uint32_t initial_data);
extern void print_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function, Stack, Priority, Name, Attributes,      Param, Time Slice */
  { MAIN_TASK,  main_task, 1000,    8,     "main",MQX_AUTO_START_TASK,0,     0 },
  { PRINT_TASK, print_task,1000,    9,     "print",   0,              0,     3 },
  { 0 }
};
MUTEX_STRUCT  print_mutex;

/*TASK*--------------------------------------------------------
*
* Task Name : main_task
* Comments : This task creates a mutex, and then two
*           instances of the print task.
*END*--------------------------------------------------- */
void main_task(uint32_t initial_data)
{
    MUTEX_ATTR_STRUCT mutexattr;
    char* string1 = "Hello from Print task 1\n";
    char* string2 = "Print task 2 is alive\n";

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

   }

   /* Initialize the mutex: */
   if (_mutex_init(&print_mutex, &mutexattr) != MQX_OK) {
      printf("Initializing print mutex failed.\n");
      _mqx_exit(0);

   }
   /* Create the print tasks */
   _task_create(0, PRINT_TASK, (uint32_t)string1);
   _task_create(0, PRINT_TASK, (uint32_t)string2);
}

/*TASK*--------------------------------------------------------
*
* Task Name : print_task
* Comments : This task prints a message. It uses a mutex to
*           ensure I/O is not interleaved.
*END*--------------------------------------------------- */
void print_task(uint32_t initial_data)
{
   while(TRUE) {
      if (_mutex_lock(&print_mutex) != MQX_OK) {
```



```
            printf("Mutex lock failed.\n");
           _mqx_exit(0);

        }
         _io_puts((char *) initial_data);
         _mutex_unlock(&print_mutex);
      }
}
```

3.6.6.9.2 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   ```
   mqx\examples\mutex
   ```

2. See the *MQX™ RTOS Release Notes document* for instructions on how to build and run the application.

3.6.7 Messages

Tasks can communicate with each other by exchanging messages. Tasks allocate messages from message pools. Tasks send messages to message queues, and receive messages from message queues. Messages can be assigned a priority or marked urgent. Tasks can send broadcast messages.

Note	To optimize code and data memory requirements on some target platforms, the Message component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file, and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Table 3-31. Summary: Using Messages

Messages use certain structure definitions and constants, which are defined in *message.h*.	Messages use certain structure definitions and constants, which are defined in *message.h*.
_msg_alloc	Allocates a message from a private-message pool.
_msg_alloc_system	Allocates a message from a system-message pool.
_msg_available	Gets the number of free messages in a message pool.
_msg_create_component	Creates the message component.
_msg_free	Frees a message.
_msg_swap_endian_data	Converts the application-defined data in a message to the other endian format.
_msg_swap_endian_header	Converts the message header to the other endian format.
_msgpool_create	Creates a private-message pool.

_msgpool_create_system	Creates a system-message pool.
_msgpool_destroy	Destroys a private-message pool.
_msgpool_test	Tests all message pools.
_msgq_close	Closes a message queue.
_msgq_get_count	Gets the number of messages in a message queue.
_msgq_get_id	Converts a queue number and processor number to a queue ID.
_msgq_get_notification_function	Gets the notification function that is associated with a message queue.
_msgq_get_owner	Gets the task ID of the task that owns a message queue.
_msgq_open	Opens a private-message queue.
_msgq_open_system	Opens a system-message queue.
_msgq_peek	Gets a pointer to the message that is at the head of a message queue (does not dequeue the message).
_msgq_poll	Poll (non-blocking) for a message in a message queue.
_msgq_receive	Receives a message from a message queue, and waits for a specified number of milliseconds.
_msgq_receive_for	Receives a message from a message queue, and waits for a specified tick-time period.
_msgq_receive_ticks	Receives a message from a message queue, and waits for a specified number of ticks.
_msgq_receive_until	Receives a message from a message queue, and waits for a specified tick time.
_msgq_send	Sends a message to a message queue.
_msgq_send_broadcast	Sends a message to multiple message queues.
_msgq_send_priority	Sends a priority message to a message queue.
_msgq_send_queue	Sends a message directly to a message queue (circumvents inter-processor routing).
_msgq_send_urgent	Sends an urgent message to a message queue.
_msgq_set_notification_function	Sets the notification function for a message queue.
_msgq_test	Tests message queues.

3.6.7.1 Creating the Message Component

You can explicitly create the message component with **_msg_create_component()**. If you do not explicitly create it, MQX RTOS creates it the first time that an application creates a message pool or opens a message queue.

3.6.7.2 Using Message Pools

Tasks allocate messages from message pools, which a task must first create. A task can create a private-message pool (**_msgpool_create()**) or a system-message pool (**_msgpool_create_system()**).

A task specifies the following info, when it creates a message pool:

- Size of the messages in the pool.
- Initial number of messages in the pool.
- Grow factor: the number of additional messages that MQX RTOS adds to the pool, if tasks have allocated all the messages.
- Maximum number of messages in the pool (if the grow factor is not zero, zero means here that the pool can contain an unlimited number of messages).

The function **_msgpool_create_system()** can be called multiple times to create multiple system-message pools, each with different characteristics.

The function **_msgpool_create()** returns a pool ID, which any task can use to access the private-message pool.

Table 3-32. Using Message Pools

	System-message pool	**Private-message pool**
Create a message pool	**_msgpool_create_system()**	**_msgpool_create()**
Allocate a message	**_msg_alloc_system()** (MQX RTOS searches all system-message pools.)	**_msg_alloc()** (MQX RTOS searches only the specified private-message pool.)
Free a message (message owner only)	**_msg_free()**	**_msg_free()**
Destroy a message pool	A system-message pool cannot be destroyed.	**_msgpool_destroy()** (By any task with the pool ID after all messages in the pool are freed.)

3.6.7.3 Allocating and Freeing Messages

Before a task sends a message, it allocates a message (**_msg_alloc_system()** or **_msg_alloc()**) of the appropriate size from a system- or private-message pool. System-message pools are not the resource of any task, and any task can allocate a message from them. Any task with the pool ID can allocate a message from a private-message pool.

When a task allocates a message from either type of pool, the message becomes the resource of the task, until the task frees the message (**_msg_free()**) or puts it in a message queue (**_msgq_send** family of functions). When a task gets a message from a message queue (**_msgq_poll()** or **_msgq_receive** family), the message becomes the resource of the task. Only the task that has the message as its resource can free the message.

Messages begin with a message header (**MESSAGE_HEADER_STRUCT**) that defines the information that MQX RTOS needs to route the message. Application-defined data follows the message header.

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

For a description of each field, see MQX RTOS Reference Manual.

3.6.7.4 Sending Messages

After a task allocates a message and fills in the message header fields and any data fields, it sends the message with **_msgq_send()**, which sends the message to the target message queue that is specified in the message header. Sending a message is not a blocking action.

3.6.7.5 Message Queues

Tasks use message queues to exchange messages. There can be private message queues and system message queues. When a task opens a message queue (specified by a message queue number), MQX RTOS returns an application-unique queue ID, which tasks subsequently use to access the message queue.

A task can convert a queue number to a queue ID with **_msgq_get_id()**.

The most-significant byte of a 16-bit queue ID contains the processor number, and the least-significant byte contains the queue number.

Table 3-33. Queue ID

bit position	15 - 8	7 - 0
queue ID	processor number	queue number

3.6.7.6 Using Private Message Queues to Receive Messages

A task can send a message to any private message queue, but only the task that opened a private message queue can receive messages from it. Only one task at a time can have the private message queue open.

A task opens a private message queue (**_msgq_open()**) by specifying its queue number, which is a value between eight and the maximum queue number that is specified in the MQX RTOS initialization structure. (Queue numbers of one through seven are reserved.) If a task calls **_msgq_open()** with queue number zero, MQX RTOS opens any of the task's unopened private message queues.

The task that opened a private message queue can close it with **_msgq_close()**, which removes all messages from the message queue and frees the messages.
A task receives a message from one of its private message queues with a function from the **_msgq_receive** family, which removes the first message in the specified queue and returns a pointer to the message. If the task specifies queue ID zero, it receives a message from any of its open message queues. Receiving a message from a private message queue is a blocking action, unless the task specifies a timeout, which is the maximum time the task waits for a message.

3.6.7.7 Using System Message Queues to Receive Messages

System message queues are not owned by a task, and a task does not block waiting to receive a message from one. Since it is not possible to block waiting for a message in a system message queue, ISRs can use system message queues. A task or ISR opens a system message queue with **_msgq_open_system()**.

A task or ISR receives messages from a system message queue with _msgq_poll(). If there are no messages in the system message queue, the function returns NULL.

3.6.7.8 Determining the Number of Pending Messages

A task can determine how many messages are in a system message queue or in one of its private message queues with **_msgq_get_count()**.

3.6.7.9 Notification Functions

With both system and private message queues, a task can specify a notification function that runs, when a message is sent to the queue. For system message queues, the task specifies the notification function when it opens the queue. For private message queues, the task sets the notification function with **_msgq_set_notification_function()**, after it opens the queue. Applications can use notification functions to couple another synchronization service (such as an event or semaphore) to a message queue.

3.6.7.10 Example: Client/Server Model

This client/server model shows communication and task synchronization using message passing.

Server task blocks waiting for a request message from Client task. When Server receives the request, it executes the request and returns the message to Client. Two-way message exchange is used, in order to block Client while Server runs.

Server opens an input message queue that it uses to receive requests from Client tasks and creates a message pool, from which it allocates request messages. Server then creates a number of Client tasks. In a real application, the Client tasks most likely would not be created by Server.

When Server has opened its message queue and created its message pool, it enters a loop, receiving messages from the message queue, acting on them (in this case, printing the data), and returning the message to Client.

Client also opens a message queue. It allocates a message from the message pool, fills in the message field, sends the message to Server, and waits for a response from Server.

3.6.7.10.1 Message Definition

```
/* server.h */
#include <mqx.h>
#include <message.h>
/* Number of clients */
#define NUM_CLIENTS 3
/* Task IDs */
#define SERVER_TASK    5
#define CLIENT_TASK    6
/* Queue IDs */
#define SERVER_QUEUE     8
#define CLIENT_QUEUE_BASE  9
/* This struct contains a data field and a message struct. */
```



```
typedef struct {
   MESSAGE_HEADER_STRUCT HEADER;
   uchar                 DATA[5];
} SERVER_MESSAGE, * SERVER_MESSAGE_PTR;
/* Function prototypes */
extern void server_task(uint32_t initial_data);
extern void client_task(uint32_t initial_data);
extern _pool_id message_pool;
```

3.6.7.10.2 Task Templates for the Client/Server Model Example

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

3.6.7.10.3 Code for Server Task

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

```
      /* Return the message: */
      msg_ptr->HEADER.TARGET_QID = msg_ptr->HEADER.SOURCE_QID;
      msg_ptr->HEADER.SOURCE_QID = server_qid;
      _msgq_send(msg_ptr);
   }
}
```

3.6.7.10.4 Code for Client Task

```
/* client.c */
#include <string.h>
#include <mqx.h>
#include <bsp.h>
#include "server.h"

/*TASK*--------------------------------------------------------
*
* Task Name   : client_task
* Comments     This task creates a message queue and
   allocates a message in the message pool. It sends the
   message to the server_task and waits for a reply. It
   then frees the message.
*END*------------------------------------------------------ */
void client_task(uint32_t index)
{

   SERVER_MESSAGE_PTR msg_ptr;
   _queue_id          client_qid;

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

   while (TRUE) {
   /* Allocate a message: */
      msg_ptr = (SERVER_MESSAGE_PTR)
      _msg_alloc(message_pool); if(msg_ptr == NULL){
         printf("\nCould not allocate a message\n");
         _mqx_exit(0);

      }/* if */
      msg_ptr->HEADER.SOURCE_QID = client_qid;
      msg_ptr->HEADER.TARGET_QID = _msgq_get_id(0, SERVER_QUEUE);
      msg_ptr->HEADER.SIZE = sizeof(MESSAGE_HEADER_STRUCT) +
         strlen((char *)msg_ptr->DATA) + 1;
      msg_ptr->DATA[0] = ('A'+ index);

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

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


3.6.7.10.5 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

```
mqx\examples\lwmsgq
```

2. See the MQX™ RTOS Release Notes document for instructions on how to build and run the application.
3. Run the application.

3.6.8 Lightweight Message Queue

Lightweight message queues are a simpler, low-overhead implementation of standard MQX RTOS messages. Tasks send messages to lightweight message queues and receive messages from lightweight message queues. A message in the message pool has a fixed size, a multiple of 32 bits. Blocking reads and blocking writes are provided.

Note	To optimize code and data memory requirements on some target platforms, the Lightweight message queue component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file, and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Table 3-35. Summary: Using the Lightweight Message Queue Component

Lightweight message queue component uses certain structure definitions and constants, which are defined in *lwmsgq.h.*	Lightweight message queue component uses certain structure definitions and constants, which are defined in *lwmsgq.h.*
_lwmsgq_init	Create a lightweight message queue.
_lwmsgq_receive	Get a message from a lightweight message queue.
_lwmsgq_send	Puts a message on a lightweight message queue.

3.6.8.1 Initialization of a Lightweight Message Queue

Lightweight message queue is initialized by calling the **_lwmsgq_init()**function.

Message pool has to be allocated statically before the initialization of this component. When a task initializes the lightweight message queue the number of messages to be created and the size of one message has to be specified.

3.6.8.2 Sending Messages

A task sends a message to the Lightweight message queue using the **_lwmsgq_send()**function. Special structure of the message is not required, however the message size must match the message size specified in the _lwmsgq_init() function.

If the queue is full, the task either blocks and waits or the error code is returned. There is also the possibility to block the task after the message is sent.

3.6.8.3 Receiving Messages

A task gets a message from the Lightweight message queue using the **_lwmsgq_receive()**function. This function removes the first message from the queue and copies the message to the user buffer. The message becomes a resource of the task.

If the queue is empty, the reading task performs timeout. There is also the possibility to block the reading task if the lightweight message queue is empty.

3.6.8.4 Example: Client/Server Model

This example is the modified version of the client/server example described in Example: Client/Server Model. The Message component is replaced by the Lightweight message queue component.

Server task initializes the message queues, creates three client tasks, and then waits for a message. After receiving a message, the task returns the message to the sender. Client task sends a message to the server task and then waits for a reply.

3.6.8.4.1 Message Definition

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

3.6.8.4.2 Task Templates for the Client/Server Model

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

3.6.8.4.3 Code for Server Task

```
/* server.c */
#include <mqx.h>
#include <bsp.h>
#include <lwmsgq.h>
#include "server.h"


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

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

   while (TRUE) {
```



```
      _lwmsgq_receive((void *)server_queue, msg, LWMSGQ_RECEIVE_BLOCK_ON_EMPTY, 0, 0);
      printf(" %c \n", msg[0]);
      __lwmsgq_send((void *)client_queue, msg, LWMSGQ_SEND_BLOCK_ON_FULL);
   }
}
```

3.6.8.4.4 Code for Client Task

```
/* client.c */
#include <string.h>
#include <mqx.h>
#include <bsp.h>
#include <lwmsgq.h>
#include "server.h"


/*TASK*-----------------------------------------------------
*
* Task Name : client_task
* Comments : This task sends a message to the server_task and
*   then waits for a reply.
*END*----------------------------------------------------- */
void
   client_task
   (
      uint32_t index
   )
{
   _mqx_uint        msg[MSG_SIZE];

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

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

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

3.6.8.4.5 Compiling the application and linking it with MQX RTOS

1. Go to this directory:

   ```
   /mqx/examples/msg
   ```

2. See the *MQX RTOS Release Notes* for instructions how to build and run the application.
3. Run the application.

3.6.9 Task Queues

You can use a task queue to:

- Schedule a task from an ISR.
- Do explicit task scheduling.
- Implement custom synchronization mechanisms.

Table 3-36. Summary: Using Task Queues

_taskq_create	Creates a task queue with the specified queuing policy (FIFO or priority).
_taskq_destroy	Destroys a task queue (and puts any waiting tasks in the appropriate ready queues).
_taskq_get_value	Gets the size of a task queue.
_taskq_resume	Restarts a task that is suspended in a task queue, or restarts all tasks that are in a task queue (and puts them in their ready queues).
_taskq_suspend	Suspends a task and puts it in the specified task queue (and removes it from the task's ready queue).
_taskq_suspend_task	Suspends the non-blocked task and puts it in the specified task queue (and removes it from the task's ready queue).
_taskq_test	Tests all task queues.

3.6.9.1 Creating and Destroying Task Queues

Before an application can perform explicit task scheduling, it must first initialize a task queue by calling **_taskq_create()** with the queuing policy for the task queue. MQX RTOS creates the task queue and returns a queue ID, which the task subsequently uses to access the task queue.

A task queue is not a resource of the task that created it. It is a system resource and is not destroyed when its creating task is terminated.

A task can explicitly destroy a task queue with **_taskq_destroy()**. If there are tasks in the task queue, MQX RTOS moves them to their ready queues.

3.6.9.2 Suspending a Task

A task can suspend itself in a specific task queue with **_taskq_suspend()**. MQX RTOS puts the task in the queue (blocks the task) according to the queuing policy of the task queue.

3.6.9.3 Resuming a Task

A task calls **_taskq_resume()** to remove either one or all tasks from a specific task queue. MQX RTOS puts them in their ready queues.

3.6.9.4 Example: Synchronizing Tasks

A task is synchronized with an ISR. A second task simulates the interrupt.

The service_task task waits for a periodic interrupt, and prints a message every time the interrupt occurs. The task first creates a task queue, then suspends itself in the queue. The simulated_ISR_task task simulates a periodic interrupt with **_time_delay()**, and when the timeout expires, it schedules service_task.

3.6.9.4.1 Code as an Example

```
/* taskq.c */
#include <mqx.h>
#include <fio.h>

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

void *  my_task_queue;
/*TASK*-----------------------------------------------------
*
* Task Name   : simulated_ISR_task
* Comments    :
*    This task pauses and then resumes the task queue.
*END*----------------------------------------------------*/

void simulated_ISR_task(uint32_t initial_data)
{
   while (TRUE) {
      __time_delay(200);
      __taskq_resume(my_task_queue, FALSE);
   }
}

/*TASK*-----------------------------------------------------
*
* Task Name   : service_task
* Comments    :
*    This task creates a task queue and the simulated_ISR_task
```

```
*    task. Then it enters an infinite loop, printing "Tick" and
*    suspending the task queue.
*END*------------------------------------------------------*/
void service_task(uint32_t initial_data)
{
    _task_id second_task_id;
   /* Create a task queue: */
   my_task_queue = _taskq_create(MQX_TASK_QUEUE_FIFO);
   if (my_task_queue == NULL) {
      _mqx_exit(0);

   }
   /* Create the task: */
   second_task_id = _task_create(0, ISR_TASK, 0);
      if (second_task_id == MQX_NULL_TASK_ID) {
         printf("\n Could not create simulated_ISR_task\n");
 _       _mqx_exit(0);

      }

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

3.6.9.4.2 Compiling the Application and Linking it with MQX RTOS

1. Go to the Example application directory:

   ```
   mqx/examples/taskq
   ```

2. See the MQX™ RTOS Release Notes document for instructions on how to build and run the application.
3. Run the application.

3.7 Communication Between Processors

With the inter-processor communication (IPC) component, tasks can do the following on remote processors:

- exchange messages
- create tasks (blocked or not blocked)
- destroy tasks
- open and close named event groups
- set event bits in named event groups

All the processors need not be directly connected or be of the same type. The IPC component routes messages through intermediate processors and converts them to the appropriate endian format. The IPC component communicates over packet control block (PCB) device drivers.

When MQX RTOS with the IPC component initializes, it initializes IPC message drivers and builds message routing tables, which define the paths that messages take to go from one processor to another. For information that might be specific to your hardware, refer to the release notes that accompany your MQX RTOS release.

Table 3-37. Summary: Setting Up Inter-Processor Communication

_ipc_add_ipc_handler	Adds an IPC handler for an MQX RTOS component.
_ipc_add_io_ipc_handler	Adds an IPC handler for an I/O component.
_ipc_msg_route_add	Adds a route to the message routing table.
_ipc_msg_route_remove	Removes a route from the message routing table.
_ipc_pcb_init	Initializes an IPC for a PCB driver.
_ipc_task	Task that initializes IPCs, and processes remote service requests.

3.7.1 Sending Messages to Remote Processors

As well as having a message routing table, each processor has one or more IPCs, each of which consists of:

- input function
- output function
- output queue

When a task sends a message to a message queue, MQX RTOS examines the destination processor number, which is part of the queue ID. If the destination processor is not local, MQX RTOS checks the routing table.

If there is a route, the table indicates the output queue of the IPC to use, in order to reach the destination processor. MQX RTOS then directs the message to that output queue. The output function runs and transmits the message on the IPC.

When an IPC receives a message, the input function runs. The input function assembles the message and calls **_msgq_send()**. The input function needs not to determine, whether the input message is for the local processor. If the message is not for the local processor, MQX RTOS routes the message to the destination processor.

3.7.1.1 Example: Four-Processor Application

The diagram shows a simple, four-processor application. The numbers in the table are arbitrary, but processor-unique, output queue numbers.

Each processor has two IPCs. There are two possible routes between each processor; for example, processor one has one IPC to processor two, and one to processor four. The routing table supports one route, so the best route should be selected. The table illustrates one possibility for each of the processor's routing tables.

3.7.1.1.1 Routing Table for Processor 1

Table 3-38. Routing Table

Source processor	Destination processor1	Destination processor2	Destination processor3	Destination processor4
1	-	10	10	11
2	21	-	20	20
3	31	31	-	30
4	40	41	41	-

As in the table, when a task on processor one sends a message to a message queue on processor three, MQX RTOS sends the message from processor one to processor two using queue ten, and then from processor two to processor three using queue 20. When the IPC on processor three receives the message, MQX RTOS directs the message to the destination (target) message queue.

3.7.2 Creating and Destroying Tasks on Remote Processors

With IPC component, a task can create and destroy tasks on a remote processor by sending service requests to IPC task on that processor. IPC task runs the request, and responds to the requesting processor.

3.7.3 Accessing Event Groups on Remote Processors

With the IPC component, a task can open and close a named event group on a remote processor and set event bits in the event group. However, a task cannot wait for event bits on a remote processor.

Event groups are opened on remote processors by specifying the processor number (followed by a colon) in the name of the event. The following example would open the event Fred on processor number four:

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

3.7.4 Creating and Initializing IPC

For tasks to communicate across processors, the application must create and initialize the IPC component on each processor, as summarized in the following steps. Each step is described in subsequent sections using information from the routing table previous example.

1. Build the IPC routing table.
2. Build the IPC protocol initialization table.
3. Provide IPC protocol initialization functions and data.
4. Create IPC task (**_ipc_task()**).

3.7.4.1 Building an IPC Routing Table

The IPC routing table defines the routes for inter-processor messages. There is one routing table per processor and it is called _ipc_routing_table. In the previous example, on processor two, messages for processor one are directed to queue number 20; messages for processors three and four are directed to queue number 21.

The routing table is an array of routing structures and ends with a zero-filled entry.

```
typedef struct ipc_routing_struct
{
  _processor_numberMIN_PROC_NUMBER;
  _processor_numberMAX_PROC_NUMBER;

  _queue_number    QUEUE;
} IPC_ROUTING_STRUCT, * IPC_ROUTING_STRUCT_PTR;
```

The fields are described in the MQX RTOS Reference Manual.

3.7.4.1.1 Routing Table for Processor One

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


3.7.4.1.2 Routing Table for Processor Two

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

3.7.4.1.3 Routing Table for Processor Three

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

3.7.4.1.4 Routing Table for Processor Four

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

3.7.4.2 Building an IPC Protocol Initialization Table

The IPC protocol initialization table defines and initializes the protocols that implement the IPC. Each IPC output queue in the routing table refers to an IPC that must have a corresponding entry in the protocol initialization table, defining the protocol and communication path that implement the IPC.

Note	The IPC_OUT_QUEUE field in IPC_PROTOCOL_INIT_STRUCT must match the QUEUE field in IPC_ROUTING_STRUCT.

The protocol initialization table is an array of protocol initialization structures and ends with a zero-filled entry.

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

The fields are described in the MQX RTOS Reference Manual.

When MQX RTOS with the IPC component initializes, it calls the **IPC_PROTOCOL_INIT** function for each IPC in the table. It passes to the IPC the **IPC_PROTOCOL_INIT_DATA**, which contains IPC-specific initialization information.

3.7.4.3 IPC Using I/O PCB Device Drivers

While you can develop special-purpose IPCs, MQX RTOS provides a standard IPC that is built on I/O packet control block (PCB) device drivers.

Using this IPC, an application can use any I/O PCB device driver to receive and send messages (See IPC Initialization Information).

Here is an **IPC_PROTOCL_INIT_STRUCT** that is set up to use the standard MQX RTOS IPC over PCB device drivers:

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

3.7.4.4 Starting IPC Task

IPC task examines the IPC protocol initialization table and starts the IPC server, which initializes each IPC driver. The IPC server accepts messages from other processors to perform remote procedure calls.

The application must define IPC task as an autostart task in the MQX RTOS initialization structure for each processor. The pointer to the IPC initialization structure of the IPC_INIT_STRUCT type has to be passed to the IPC task as a creation parameter. This structure contains IPC routing table and IPC initialization table pointers. If not provided the default IPC_INIT_STRUCT is used. The task template for IPC task is:

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

3.7.4.5 Example: IPC Initialization Information

In this example, two processors set up IPC communication over an asynchronous serial port using the PCB device drivers that accompany MQX RTOS. Each processor is connected by interrupt-driven asynchronous character device drivers "ittyb:". The IPC uses the PCB_MQXA driver, which sends and receives packets that have an MQX RTOS-defined format.

The ipc_init_table uses the MQX RTOS IPC over PCB I/O driver initialization function **_ipc_pcb_init()** and the data structure needed for its initialization, pcb_init, which defines:

- The PCB I/O driver name to use when opening the driver.
- The installation function to call, in this case **_io_pcb_mqxa_install()** (needs not to be specified, if the PCB I/O driver was previously installed).
- The PCB I/O driver-specific initialization pcb_mqxa_init.

3.7.4.5.1 IPC Initialization Information

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

3.7.4.5.2 Code for Processor One

```
/* ipc1.c */

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

IPC_PCB_INIT_STRUCT pcb_init =
{
   /* IO_PORT_NAME */             "pcb_mqxa_ittyx:",
   /* DEVICE_INSTALL? */          _io_pcb_mqxa_install,
   /* DEVICE_INSTALL_PARAMETER*/  (void *)&pcb_mqxa_init,
   /* IN_MESSAGES_MAX_SIZE */     sizeof(THE_MESSAGE),
   /* IN_MESSAGES_TO_ALLOCATE */  8,
   /* IN MESSAGES_TO_GROW */      8,
   /* IN_MESSAGES_MAX_ALLOCATE */ 16,
   /* OUT_PCBS_INITIAL */         8,
```

```
   /* OUT_PCBS_TO_GROW */         8,
   /* OUT_PCBS_MAX */             16
};

const IPC_ROUTING_STRUCT ipc_routing_table[] =
{
   { TEST2_ID, TEST2_ID, QUEUE_TO_TEST2 },
   { 0, 0, 0 }
};

const IPC_PROTOCOL_INIT_STRUCT ipc_init_table[] =
{
   { _ipc_pcb_init, &pcb_init, "Pcb_to_test2", QUEUE_TO_TEST2 },
   { NULL, NULL, NULL, 0}
};

static const IPC_INIT_STRUCT ipc_init = {
    ipc_routing_table,
    ipc_init_table
};

const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
   /* Task Index, Function,Stack, Priority, Name, Attributes, Param,  Time Slice */
    { IPC_TTN, ipc_task, IPC_DEFAULT_STACK_SIZE, 8, "_ipc_task", MQX_AUTO_START_TASK,
        (uint32_t)&ipc_init,  0 },
    { MAIN_TTN, main_task, 2000,               9,  "Main",     MQX_AUTO_START_TASK,
        0, 0 },
    { 0 }
};

MQX_INITIALIZATION_STRUCT   MQX_init_struct =
{
   /* PROCESSOR_NUMBER */              TEST_ID,
   /* START_OF_KERNEL_MEMORY */        BSP_DEFAULT_START_OF_KERNEL_MEMORY,
   /* END_OF_KERNEL_MEMORY */          BSP_DEFAULT_END_OF_KERNEL_MEMORY,
   /* INTERRUPT_STACK_SIZE */          BSP_DEFAULT_INTERRUPT_STACK_SIZE,
   /* TASK_TEMPLATE_LIST */            (void *)MQX_template_list,
   /* MQX_HARDWARE_INTERRUPT_LEVEL_MAX */ BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
   /* MAX_MSGPOOLS */                  8,
   /* MAX_MSGQS */                     16,
   /* IO_CHANNEL */                    BSP_DEFAULT_IO_CHANNEL,
   /* IO_OPEN_MODE */                  BSP_DEFAULT_IO_OPEN_MODE
};

/*TASK*-----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*     This task creates a message pool and a message queue then
*     sends a message to a queue on the second CPU.
*     It waits for a return message, validating the message before
*     sending a new message.
*END*----------------------------------------------------- */
void
   main_task
   (
      uint32_t dummy
   )
{
   _pool_id      msgpool;
   THE_MESSAGE_PTR msg_ptr;
```



```
   _queue_id     qid;
   _queue_id     my_qid;

   uint32_t       test_number = 0;
   my_qid = _msgq_open(MAIN_QUEUE,0);
   qid    = _msgq_get_id(TEST2_ID,RESPONDER_QUEUE);
   msgpool = _msgpool_create(sizeof(THE_MESSAGE), 8, 8, 16);
   while (test_number < 64) {
      msg_ptr =
      (THE_MESSAGE_PTR)_msg_alloc(msgpool);
      msg_ptr->HEADER.TARGET_QID = qid;
      msg_ptr->HEADER.SOURCE_QID = my_qid;
      msg_ptr->DATA = test_number++;
      putchar('-');
      _msgq_send(msg_ptr);
      msg_ptr = _msgq_receive(MSGQ_ANY_QUEUE, 10000);
      if (msg_ptr == NULL) {
         puts("Receive failed\n");
         _mqx_exit(1);
      } else if (msg_ptr->HEADER.SIZE != sizeof(THE_MESSAGE)) {
         puts("Message wrong size\n");
         _mqx_exit(1);
      } else if (msg_ptr->DATA != test_number) {
         puts("Message data incorrect\n");
         _mqx_exit(1);
      }
      __msg_free(msg_ptr);
   }
   puts("All complete\n");
   _mqx_exit(0);
}
```

3.7.4.5.3 Code for Processor Two

```
/* ipc2.c */

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

IPC_PCB_INIT_STRUCT pcb_init =
{
   /* IO_PORT_NAME */              "pcb_mqxa_ittyx:",
```

```
   /* DEVICE_INSTALL? */            _io_pcb_mqxa_install,
   /* DEVICE_INSTALL_PARAMETER*/   (void *)&pcb_mqxa_init,
   /* IN_MESSAGES_MAX_SIZE */       sizeof(THE_MESSAGE),
   /* IN_MESSAGES_TO_ALLOCATE */    8,
   /* IN_MESSAGES_TO_GROW */        8,
   /* IN_MESSAGES_MAX_ALLOCATE */ 16,
   /* OUT_PCBS_INITIAL */           8,
   /* OUT_PCBS_TO_GROW */           8,
   /* OUT_PCBS_MAX */               16
};

const IPC_ROUTING_STRUCT ipc_routing_table[] =
{
   { TEST_ID, TEST_ID, QUEUE_TO_TEST },
   { 0, 0, 0 }
};

const IPC_PROTOCOL_INIT_STRUCT ipc_init_table[] =
{
   { _ipc_pcb_init, &pcb_init, "Pcb_to_test", QUEUE_TO_TEST },
   { NULL, NULL, NULL, 0}
};

static const IPC_INIT_STRUCT ipc_init = {
    ipc_routing_table,
    ipc_init_table
};

const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function, Stack,            Priority, Name, Attributes, Param, Time
Slice */
  { IPC_TTN,    ipc_task,IPC_DEFAULT_STACK_SIZE, 8,"_ipc_task", MQX_AUTO_START_TASK,
   (uint32_t)&ipc_init,    0 },
  { RESPONDER_TTN, responder_task, 2000,9, "Responder",MQX_AUTO_START_TASK, 0, 0 },
    { 0 }
};

MQX_INITIALIZATION_STRUCT    MQX_init_struct =
{
   /* PROCESSOR_NUMBER */              TEST2_ID,
   /* START_OF_KERNEL_MEMORY */        BSP_DEFAULT_START_OF_KERNEL_MEMORY,
   /* END_OF_KERNEL_MEMORY */          BSP_DEFAULT_END_OF_KERNEL_MEMORY,
   /* INTERRUPT_STACK_SIZE */          BSP_DEFAULT_INTERRUPT_STACK_SIZE,
   /* TASK_TEMPLATE_LIST */            (void *)MQX_template_list,
   /* MQX_HARDWARE_INTERRUPT_LEVEL_MAX */ BSP_DEFAULT_MQX_HARDWARE_INTERRUPT_LEVEL_MAX,
   /* MAX_MSGPOOLS */                  8,
   /* MAX_MSGQS */                     16,
   /* IO_CHANNEL */                    BSP_DEFAULT_IO_CHANNEL,
   /* IO_OPEN_MODE */                  BSP_DEFAULT_IO_OPEN_MODE
};

/*TASK*-----------------------------------------------------
*
* Task Name : responder_task
* Comments :
*     This task creates a message queue then waits for a message.
*     Upon receiving the message the data is incremented before
*     the message is returned to the sender.
*END*-----------------------------------------------------*/
void responder_task(uint32_t dummy) {
   THE_MESSAGE_PTR  msg_ptr;
```



```
   _queue_id        qid;
   _queue_id        my_qid;
   puts("Receiver running. \n");
   my_qid = _msgq_open(RESPONDER_QUEUE, 0);
   while (TRUE) {
      msg_ptr = _msgq_receive(MSGQ_ANY_QUEUE, 0);
      if (msg_ptr != NULL) {
         qid = msg_ptr->HEADER.SOURCE_QID;
         msg_ptr->HEADER.SOURCE_QID = my_qid;
         msg_ptr->HEADER.TARGET_QID = qid;
         msg_ptr->DATA++;
         putchar('+');
         _msgq_send(msg_ptr);
      } else {
         puts("RESPONDER RECEIVE ERROR\n");
         _mqx_exit(1);
      }
   }
}
```

3.7.4.5.4 Compiling the Application and Linking it with MQX RTOS

1. See the *MQX™ RTOS Release Notes* document for instructions on how to build and run the application.
2. Go to this directory to compile for processor one:

   ```
   mqx\examples\taskq
   ```

3. Build the project.
4. Go to this directory to compile for processor two:

   ```
   mqx\examples\ipc\cpu2\
   ```

5. Build the project.
6. Connect ttyb: of processor one to ttyb: of processor two.
7. Run the application according to the instructions in the *MQX™ RTOS Release Notes document*. Start processor two before processor one.

5.7.1.1 3.7.5 Endian Conversion of Message Headers

When a processor receives a message from a remote processor, the IPC input function examines the **CONTROL** field in the message header to determine, whether the message is from a processor that uses the other endian format. In that case the input function converts the message header to the local processor's own endian format, and sets the

CONTROL field to specify its endian format.

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

Note	The IPC cannot convert the data portion of the message to the other endian format, because it does not know the format of the data. It is the responsibility of the application to convert the data portion of received messages to the other endian format. To check whether conversion is necessary, use the macro **MSG_MUST_CONVERT_DATA_ENDIAN**. To convert the message data, use **_msg_swap_endian_data()**. Both functions are defined in *message.h*. For more information, see MQX RTOS Reference Manual.

3.8 Timing

MQX RTOS provides the core-time component, which can be extended with optional timer and watchdog components.

3.8.1 Rollover of MQX RTOS Time

MQX RTOS keeps the time internally as a 64-bit count of the number of tick interrupts, since the application started to run. This provides a very long time before MQX RTOS time rolls over. For example, if the tick rate was once per nanosecond, the MQX RTOS time rolls over when 584 years have passed.

3.8.2 Accuracy of MQX RTOS Time

MQX RTOS keeps the time internally as a 64-bit count of the number of tick interrupts, but when an application requests the tick time, the time also includes a 32-bit number that represents the number of hardware "ticks" that have occurred since the last tick interrupt. Typically, MQX RTOS reads this value from the hardware counter that is used to program the timer. As a result, the application receives the time as accurately, as it can possibly be determined.

3.8.3 Time Component

Time is a core component that offers time as elapsed time and absolute time, expressed as seconds and milliseconds time stamp and (second/millisecond time), as ticks (tick time), or as a date (date time and tm struct).

Table 3-39. Summary: Using the Time Component

_ticks_to_time	Converts tick time to second/millisecond time.
_time_add_day_to_ticks	Adds days to tick time.
_time_add_hour_to_ticks	Adds hours to tick time.

_time_add_min_to_ticks	Adds minutes to tick time.
_time_add_msec_to_ticks	Adds milliseconds to tick time.
_time_add_nsec_to_ticks	Adds nanoseconds to tick time.
_time_add_psec_to_ticks	Adds picoseconds to tick time.
_time_add_sec_to_ticks	Adds seconds to tick time.
_time_add_usec_to_ticks	Adds microseconds to tick time.
_time_delay	Suspends the active task for the specified number of milliseconds.
_time_delay_for	Suspends the active task for the specified tick-time period (including hardware ticks).
_time_delay_ticks	Suspends the active task for the specified number of ticks.
_time_delay_until	Suspends the active task until the specified tick time.
_time_dequeue	Removes a task (specified by its task ID) from the timeout queue.
_time_dequeue_td	Removes a task (specified by its task descriptor) from the timeout queue.
_time_diff	Gets the second/millisecond time difference between two second/millisecond time structures.
_time_diff_days	Gets the time difference in days between two tick times.
_time_diff_hours	Gets the difference in hours between two tick times.
_time_diff_microseconds	Gets the difference in microseconds between two tick times.
_time_diff_milliseconds	Gets the difference in milliseconds between two tick times.
_time_diff_minutes	Gets the difference in minutes between two tick times.
_time_diff_nanoseconds	Gets the difference in nanoseconds between two tick times.
_time_diff_picoseconds	Gets the difference in picoseconds between two tick times.
_time_diff_seconds	Gets the difference in seconds between two tick times.
_time_diff_ticks	Gets the tick-time difference between two tick times.
_time_from_date	Gets second/millisecond time from date time.
_time_get	Gets the absolute time in second/millisecond time.
_time_get_ticks	Gets the absolute time in tick time (includes ticks and hardware ticks).
_time_get_elapsed	Gets the second/millisecond time that has elapsed, since the application started on this processor.
_time_get_elapsed_ticks	Gets the tick time that has elapsed, since the application started on this processor.
_time_get_hwticks	Gets the number of hardware ticks since the last tick.
_time_get_hwticks_ per_tick	Gets the number of hardware ticks per tick.
_time_get_microseconds	Gets the calculated number of microseconds, since the last periodic timer interrupt.
_time_get_nanoseconds	Gets the calculated number of nanoseconds, since the last periodic timer interrupt.
_time_get_resolution	Gets the resolution of the periodic timer interrupt.
_time_get_ticks_per_sec	Gets the frequency (in ticks per second) of the clock interrupt.
_time_init_ticks	Initializes a tick-time structure with a number of ticks.

_time_notify_kernel	Called by the BSP, when a periodic timer interrupt occurs.
_time_set	Sets the absolute time in second/millisecond time.
_time_set_hwticks_per_tick	Sets the number of hardware ticks per tick.
_time_set_ticks	Sets the absolute time in tick time.
_time_set_resolution	Sets the frequency of the periodic timer interrupt.
_time_set_timer_vector	Sets the periodic timer interrupt vector that MQX RTOS uses.
_time_set_ticks_per_sec	Sets the frequency (in ticks per second) of the clock interrupt.
_time_to_date	Converts second/millisecond time to date time.
_time_to_ticks	Converts second/millisecond time to tick time.
mktime	Converts the broken-down time value, expressed as local time, to calendar time representation.
gmtime_r	Converts the calendar time to broken-down time representation, expressed in Coordinated Universal Time (UTC).
timegm	Converts the broken-down time structure, expressed as UTC time, to a calendar time representation.
localtime_r	Converts the calendar time to a broken-down time representation, expressed in local time.

3.8.3.1 Second/Millisecond Time

Time is available in seconds and milliseconds. To process second/millisecond time is more complex and CPU intensive, than processing tick time.

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

The fields are described in MQX RTOS Reference Manual.

3.8.3.2 Time Stamp

Time stamp is a system to describe instants in time, which are defined as the number of seconds that have elapsed since the Epoch, 00:00:00 UTC, 1-1-1970.

```
typedef uint32_t time_t
```

3.8.3.3 Tick Time

Time is available in tick time. To process tick time is simpler and less CPU intensive, than processing second/millisecond time.

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

The fields are described in MQX RTOS Reference Manual.

3.8.3.4 Elapsed Time

Elapsed time is the amount of time since MQX RTOS started on the processor. A task can get the elapsed time in second/millisecond time with **_time_get_elapsed()**, and in tick time with **_time_get_elapsed_ticks()**.

3.8.3.5 Time Resolution

When MQX RTOS starts, it installs the periodic timer ISR, which sets the time resolution for the hardware. The resolution defines, how often MQX RTOS updates time, or how often a tick occurs. The resolution is usually 200 ticks per second or five milliseconds. A task can get the resolution in milliseconds with **_time_get_resolution()** and in ticks per second with **_time_get_resolution_ticks()**.

A task can get elapsed time in microsecond resolution by calling **_time_get_elapsed()**, followed by **_time_get_microseconds()**, which gets the number of microseconds since the last periodic timer interrupt.

A task can get elapsed time in nanosecond resolution by calling **_time_get_elapsed()** followed by **_time_get_nanoseconds()**, which gets the number of nanoseconds since the last periodic timer interrupt.

A task can also get the number of hardware ticks since the last interrupt by calling **_time_get_hwticks()**. A task can get the resolution of the hardware ticks by calling _**time_get_hwticks_per_tick()**.

3.8.3.6 Absolute Time

So that the tasks on different processors can exchange information that is timestamped from a common reference, the time component offers absolute time.

Initially, absolute time is the time since the reference date of 0:00:00.000 January 1, 1970. An application can change the absolute time by changing the reference date in second/millisecond time with **_time_set()**, or in tick time with **_time_set_ticks()**.
A task gets the absolute time in second/millisecond time with **_time_get()** or in tick time

with **_time_get_ticks()**.

Unless an application changes the absolute time, the following pairs of functions return the same values:

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

Note	A task should use elapsed time to measure an interval or implement a timer. This prevents the measurement from being affected by other tasks that might call **_time_set()** or **_time_set_ticks()**, and thereby change the absolute time.

3.8.3.7 Time in Date Formats

To help you set and interpret absolute time that is expressed in second/millisecond time or tick time, the time component offers time expressed in a date format and a broken-down time structure (tm struct).

3.8.3.7.1 DATE_STRUCT

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

The fields are described in MQX RTOS Reference Manual.

3.8.3.7.2 TM STRUCT

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

The fields are described in MQX RTOS Reference Manual.

3.8.3.8 Timeouts

A task can supply the time as a timeout parameter to several MQX RTOS components, for example, functions in the **_msgq_receive**, **_lwmsgq_receive**, **_sem_wait**, **_lwsem_wait**, **_event_wait** and **_lwevent_wait** families. Note, that the resolution of all time functions is always one tick.

_time_delay(), **_event_wait_all()**, **_event_wait_any()**, **_sem_wait()**, **msgq_receive()** and **_sched_set_rr_interval()** functions wait at least the specified time in milliseconds. This time is usually bigger than the requested time, depending on the tick length, on other scheduled events and their priorities.

_time_delay_ticks() function waits at least the requested number of tick interrupts.
_time_delay_ticks(1) waits at least to the first tick interrupt.
_time_delay(0) and **_time_delay_tick(0)** cause **shed_yield()** function calling. For ticks higher than zero, the actual waiting time is typically shorter than ticks multiplied by tick time in milliseconds.

A task can also explicitly suspend itself by calling a function from the **_time_delay** family. When the time expires, MQX RTOS puts the task in the task's ready queue.

3.8.4 Timers

Timers are an optional component that extends the core-time component. An application can use timers:

- To cause a notification function to run at a specific time - when MQX RTOS creates the timer component, it starts Timer task, which maintains timers and their application-defined notification functions. When a timer expires, Timer Task calls the appropriate notification function.
- To communicate that a time period has expired.

Note	To optimize code and data memory requirements on some target platforms, the Timer component is not compiled in the MQX kernel by default. To test this feature, you need to enable it first in the MQX user configuration file and recompile the MQX PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

A task can start a timer at a specific time or at some specific time after the current time. Timers can use elapsed time or absolute time.

There are two types of timers:

- One-shot timers, which expire once.
- Periodic timers, which expire repeatedly at a specified interval. When a periodic timer expires, MQX RTOS resets the timer.

Table 3-40. Summary: Using Timers

Timers use certain structures and constants, which are defined in *timer.h.*	
_timer_cancel	Cancels an outstanding timer request.
_timer_create_component	Creates the timer component.
_timer_start_oneshot_after	Starts a timer that expires once after a time delay in milliseconds.
_timer_start_oneshot_after_ticks	Starts a timer that expires once after a time delay in ticks.
_timer_start_oneshot_at	Starts a timer that expires once at a specific time (in second/ millisecond time).
_timer_start_oneshot_at_ticks	Starts a timer that expires once at a specific time (in tick time).
_timer_start_periodic_at	Starts a periodic timer at a specific time (in second/millisecond time).
_timer_start_periodic_at_ticks	Starts a periodic timer at a specific time (in tick time).
_timer_start_periodic_every	Starts a periodic timer every number of milliseconds.
_timer_start_periodic_every_ticks	Starts a periodic timer every number of ticks.
_timer_test	Tests the timer component.

3.8.4.1 Creating the Timer Component

You can explicitly create the timer component by calling **_timer_create_component()** with the priority and stack size for Timer task, which MQX RTOS creates, when it creates the timer component. Timer task manages timer queues and provides a context for notification functions.

If you do not explicitly create the timer component, MQX RTOS creates it with default values the first time an application starts a timer.

Table 3-41. Default Timer Task Parameters

Parameter	Default
Priority of Timer task	1
Stack size for Timer task	500

3.8.4.2 Starting Timers

A task starts a timer with one of the following:

- **_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()**

When a task calls one of these functions, MQX RTOS inserts a timer request into the queue of outstanding timers. When the timer expires, the notification function runs.

Note	The stack space for Timer task should include the stack space that the notification function needs.

3.8.4.3 Cancelling Outstanding Timer Requests

A task can cancel an outstanding timer request by calling **_timer_cancel()** with the timer handle that was returned from one of the **_timer_start** family of functions.

3.8.4.4 Example: Using Timers

Simulate a LED being turned on and off every second. One timer turns the LED on, and another turns it off. The timers expire every two seconds, offset by one second.

3.8.4.4.1 Code for Timer Example

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

#include <timer.h>
#define TIMER_TASK_PRIORITY  2
#define TIMER_STACK_SIZE     1000
#define MAIN_TASK            10
extern void main_task(uint32_t);

const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function, Stack, Priority, Name, Attributes,       Param, Time Slice*/
  { MAIN_TASK, main_task, 2000,  8,       "Main", MQX_AUTO_START_TASK, 0,  0},
  { 0 }
};

/*FUNCTION*------------------------------------------------------
*
* Function Name : LED_on
* Returned Value : none
* Comments      :
*    This timer function prints "ON"
```



```
*END*-------------------------------------------------- */
void
   LED_on
   (
      _timer_id id,
      void * data_ptr,
      MQX_TICK_STRUCT_PTR tick_ptr
   )
{
   printf("ON\n");
}

/*FUNCTION*------------------------------------------------------
*
* Function Name : LED_off

* Returned Value : none
* Comments      :
*    This timer function prints "OFF"
*END*-------------------------------------------------- */
void
   LED_off
   (
      _timer_id id,
      void * data_ptr,
      MQX_TICK_STRUCT_PTR tick_ptr
   )
{
   printf("OFF\n");
}

/*TASK*----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*    This task creates two timers, each of a period of 2 seconds,
*    the second timer offset by 1 second from the first.
*END*-------------------------------------------------- */
void
   main_task
   (
      uint32_t initial_data
   )
{
   MQX_TICK_STRUCT
   ticks;
   MQX_TICK_STRUCT
   dticks;
   _timer_id     on_timer;
   _timer_id     off_timer;
   /*
   ** Create the timer component with more stack than the default
   ** in order to handle printf() requirements:
   */
  _timer_create_component(TIMER_DEFAULT_TASK_PRIORITY, 1024);
  _time_init_ticks(&dticks, 0);
  _time_add_sec_to_ticks(&dticks, 2);
  _time_get_ticks(&ticks);
  _time_add_sec_to_ticks(&ticks, 1);
   on_timer = _timer_start_periodic_at_ticks(LED_on, 0,
              TIMER_ELAPSED_TIME_MODE,&ticks, &dticks);
  _time_add_sec_to_ticks(&tick, 1);
```

```
    off_timer = _timer_start_periodic_at_ticks(LED_off, 0,
                  TIMER_ELAPSED_TIME_MODE, &ticks, &dticks);
  _time_delay_ticks(600);
  printf("\nThe task is finished!");
  _timer_cancel(on_timer);
  _timer_cancel(off_timer);
  _mqx_exit(0);

}
```

3.8.4.4.2 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   `mqx\examples\timer`

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.

A message is printed each time the timer notification function runs.

3.8.5 Lightweight Timers

Lightweight timers are an optional component that extends the core time component. Lightweight timers provide periodic notification to the application.

A task can create a periodic queue and add timers to it. The timers expire at the same rate as the queue's period, but offset from the period's expiry time.

Table 3-42. Summary: Using Lightweight Timers

Lightweight timers use certain structures and constants, which are defined in *lwtimer.h.*	Lightweight timers use certain structures and constants, which are defined in *lwtimer.h.*
_lwtimer_add_timer_to_queue	Adds a lightweight timer to a periodic queue.
_lwtimer_cancel_period	Removes all the timers from a periodic queue.
_lwtimer_cancel_timer	Removes a timer from a periodic queue.
_lwtimer_create_periodic_queue	Creates a periodic queue (with a period of a specified number of ticks), to which lightweight timers can be added.
_lwtimer_test	Tests all the periodic queues and their timers.

3.8.5.1 Starting Lightweight Timers

A task starts a lightweight timer by first creating a periodic queue by calling **_lwtimer_create_periodic_queue()** with a pointer to a variable of type **LWTIMER_PERIOD_STRUCT**, which specifies the queue's period (in ticks). It then adds a timer to the queue by calling **_lwtimer_add_timer_to_queue()** with the address of the periodic queue variable and a pointer to a variable of type **LWTIMER_STRUCT**, which specifies the function that is called when the timer expires.

When the timer expires, the notification function specified by the timer runs.

Note	Because the notification function runs in the context of the kernel timer ISR, it is subject to the same restrictions as the ISR (see page Restrictions on ISRs). The MQX RTOS interrupt stack size should include the stack space that the notification function needs.

3.8.5.2 Cancelling Outstanding Lightweight Timer Requests

A task can cancel an outstanding lightweight timer request by calling **_lwtimer_cancel_timer()** with the address of the **LWTIMER_STRUCT**.
A task can cancel all the timers on a lightweight timer queue by calling **_lwtimer_cancel_period()** with the address of the **LWTIMER_PERIOD_STRUCT**.

3.8.6 Watchdogs

Most embedded systems have a hardware watchdog timer. If the application does not reset the timer within a certain time (perhaps because of deadlock or some other error condition), the hardware generates a reset operation. As such, a hardware watchdog timer monitors the entire application on a processor; it does not monitor individual tasks.

Note	To optimize code and data memory requirements on some target platforms, the Watchdog component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS dfor more details.

The MQX RTOS watchdog component provides a software watchdog for each task. If a single task starves or runs beyond certain timing constraints, the watchdog provides a way to detect the problem. Initially, the task starts its watchdog with a specific time value, and if the task fails to stop or restart the watchdog before that time expires, MQX RTOS calls a processor-unique, application-supplied expiry function that can initiate error recovery.

Table 3-43. Summary: Using Watchdogs

Watchdogs use certain structures and constants, which are defined in *watchdog.h.*	Watchdogs use certain structures and constants, which are defined in *watchdog.h.*
_watchdog_create_component	Creates the watchdog component.
_watchdog_start	Starts or restarts the watchdog (time is specified in milliseconds).
_watchdog_start_ticks	Starts or restarts the watchdog (time is specified in ticks).
_watchdog_stop	Stops the watchdog.
_watchdog_test	Tests the watchdog component.

3.8.6.1 Creating the Watchdog Component

Before a task can use the watchdog component, the application must explicitly create it by calling **_watchdog_create_component()** with the interrupt vector of the periodic timer device and a pointer to the function that MQX RTOS calls if a watchdog expires.

3.8.6.2 Starting or Restarting a Watchdog

A task starts or restarts its watchdog by calling either:

- **_watchdog_start()** with the number of milliseconds, before the watchdog expires.
- **_watchdog_start_ticks()** with the number of ticks, before the watchdog expires.

If the task does not restart or stop its watchdog before the watchdog expires, MQX RTOS calls the expiration function.

3.8.6.3 Stopping a Watchdog

A task can stop its watchdog with **_watchdog_stop()**.

3.8.6.4 Example: Using Watchdogs

A task creates the watchdog component on the periodic timer interrupt vector and specifies the expiry function (**handle_watchdog_expiry()**). Then it starts a watchdog that expires after two seconds. To prevent its watchdog from expiring, the task must either stop or restart the watchdog within two seconds.

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



```
{
 /* Task Index, Function, Stack, Priority, Name, Attributes,       Param, Time Slice*/
  { MAIN_TASK, main_task, 2000,  8,       "Main", MQX_AUTO_START_TASK, 0,  0},
  { 0 }
};

/*FUNCTION*------------------------------------------------------
*
* Function Name : handle_watchdog_expiry
* Returned Value : none
* Comments      :
*     This function is called when a watchdog has expired.
*END*-------------------------------------------------------- */
void handle_watchdog_expiry(void * td_ptr)
{
    printf("\nwatchdog expired for task: %p", td_ptr);
}

/*FUNCTION*------------------------------------------------------
*
* Function Name : waste_time
* Returned Value : input value times 10
* Comments      :
*     This function loops the specified number of times,
*     essentially wasting time.

*END*-------------------------------------------------------- */
_mqx_uint
   waste_time (
      _mqx_uint n
   )
{
   _mqx_uint        i;
   volatile _mqx_uint result;
   result = 0;
   for (i = 0; i < n; i++) {
      result += 1;
   }
   return result*10;
}

/*TASK*----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*   This task creates a watchdog, then loops, performing
*   work for longer and longer periods until the watchdog fires.
*END*-------------------------------------------------------- */
void
   main_task
   (
      uint32_t initial_data
   )
{
   MQX_TICK_STRUCT ticks;
   _mqx_uint      result;
   _mqx_uint      n;
   _time_init_ticks(&ticks, 10);

   result = _watchdog_create_component(BSP_TIMER_INTERRUPT_VECTOR,
      handle_watchdog_expiry);
   if (result != MQX_OK) {
```

```
      printf("\nError creating watchdog component");
      _mqx_exit(0);

   }
   n = 100;
   while (TRUE) {
      result = _watchdog_start_ticks(&ticks);
      n = waste_time(n);
      _watchdog_stop();
      printf("\n %d", n);
   }
}
```

3.8.6.4.1 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   `mqx\examples\watchdog`

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.

When the watchdog expires, the Main task prints a message to the output device.

3.9 Handling Interrupts and Exceptions

MQX RTOS handles hardware interrupts and exceptions with interrupt service routines (ISRs). An ISR is not a task; it is a small, high-speed routine that reacts quickly to hardware interrupts or exceptions. ISRs are usually written in C. The duties of an ISR might include:

- servicing a device
- clearing an error condition
- signaling a task

When MQX RTOS calls an ISR, it passes a parameter, which the application defines, when the application installs the ISR. The parameter might, for example, be a pointer to a configuration structure that is specific to the device.

Note	The parameter should not point to data on a task's stack, because this memory might not be available to the ISR.

The ISR might run with some interrupts disabled, depending on the priority of the interrupt being serviced. Therefore, it is important that the ISR performs a minimal number of functions. The ISR usually causes a task to become ready. It is the priority of this task that then determines, how quickly the information gathered from the interrupting device can be processed. The ISR can ready a task in a number of ways: through lightweight events, events, lightweight semaphores, semaphores, messages, lightweight message queues or task queues.

MQX RTOS provides a kernel ISR, which is written in assembly language. The kernel ISR runs before any other ISR, and does the following:

- It saves the context of the active task.
- It switches to the interrupt stack.
- It calls the appropriate ISR.
- After the ISR has returned, it restores the context of the highest-priority ready task.

When MQX RTOS starts, it installs the default kernel ISR (**_int_kernel_isr()**) for all possible interrupts.

When the ISR returns to the kernel ISR, the kernel ISR performs a task dispatch operation if the ISR readied a task that is of higher priority, than the one that was active at the time of the interrupt. This means that the context of the previously active task is saved, and the higher-priority task becomes the active task.

The following diagram shows, how MQX RTOS handles interrupts.

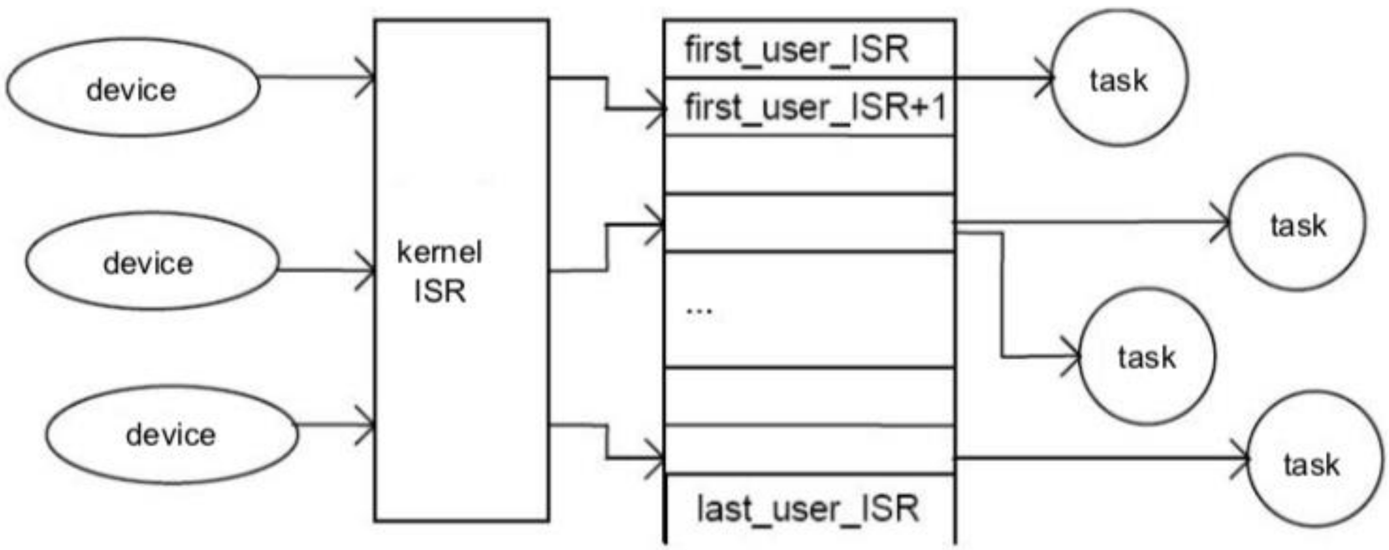

Figure 3-3. Handling Interrupts

Table 3-44. Summary: Handling Interrupts and Exceptions

_int_disable	Disables hardware interrupts.
_int_enable	Enables hardware interrupts.
_int_get_isr	Gets the ISR for a vector number.
_int_get_isr_data	Gets the data pointer associated with an interrupt.
_int_get_isr_depth	Gets the current ISR nesting depth.
_int_get_kernel_isr	Gets the kernel ISR for an interrupt.
_int_get_previous_vector_table	Gets a pointer to the interrupt vector table that is stored when MQX RTOS starts.
_int_get_vector_table	Gets a pointer to the current interrupt vector table.
_int_install_isr	Installs an application-defined ISR.
_int_install_kernel_isr	Installs a kernel ISR.
_int_install_unexpected_isr	Installs _int_unexpected_isr() as the default ISR.
_int_kernel_isr	The default kernel ISR.
_int_set_isr_data	Sets the data associated with a specific interrupt.
_int_set_vector_table	Changes the location of the vector table.

3.9.1 Initializing Interrupt Handling

When MQX RTOS starts, it initializes its ISR table, which has an entry for each interrupt number. Each entry consists of:

- A pointer to the ISR to call.
- Data to pass as a parameter to the ISR.
- A pointer to an exception handler for that ISR.

Initially, the ISR for each entry is the default ISR **_int_default_isr()**, which blocks the active task.

3.9.2 Installing Application-Defined ISRs

With **_int_install_isr()**, an application can replace the ISR with an application-defined, interrupt-specific ISR, which MQX RTOS calls, when the interrupt occurs. The application should do the replacement before it initializes the device.

The parameters for **_int_install_isr()** are:

- interrupt number
- pointer to the ISR function
- ISR data
- An application-defined ISR usually signals a task, which can be done by:
- Setting an event bit (**_event_set()**).

- Posting a lightweight semaphore (**_lwsem_post()**).
- Posting a non-strict semaphore (**_sem_post()**).
- Sending a message to a message queue. An ISR can also receive a message from a system message queue (**_msgq_send** family).

Note	The most efficient way to allocate a message from an ISR is to use **_msg_alloc()**.

- dequeuing a task from a task queue, which puts the task in the task's ready queue. Task queues let you implement signaling methods that are customized for your application (**_taskq_resume()**).

3.9.3 Restrictions on ISRs

The following table contains information about ISR restrictions.

3.9.3.1 Functions That the ISR Cannot Call

MQX RTOS returns an error, if the ISR calls any of the following functions.

Table 3-45. Functions That the ISR Cannot Call

Component	Function
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 message queue	**_lwmsgq_send()** (when LWMSGQ_SEND_BLOCK_ON_FULL or LWMSGQ_SEND_BLOCK_ON_SEND flags used) _lwmsgq_receive()
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() _timer_cancel()**
Watchdogs	**_watchdog_create_component()**

3.9.3.2 Functions That ISRs should not call

ISRs should not call MQX RTOS functions that might block or take a long time to run. These include:

- most functions from the **_io_** family
- **_event_wait** family
- **_int_default_isr()**
- **_int_unexpected_isr()**
- **_klog_display()**
- **_klog_show_stack_usage()**
- **_lwevent_wait** family
- **_lwmsgq_send()** (when LWMSGQ_SEND_BLOCK_ON_FULL or LWMSGQ_SEND_BLOCK_ON_SEND flags used)
- **_lwmsgq_receive()**
- **_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
- **_timer_start** family

3.9.3.3 Non-Maskable Interrupts

Non-Maskable Interrupts (NMI) are defined as interrupts that cannot be disabled (masked) by software. It is possible to use such interrupts in MQX RTOS applications, but NMI service routines must be installed directly to vector table as kernel ISRs (use **_int_install_kernel_isr()** instead of _int_install_isr()). The NMI service routines are not allowed to call any MQX RTOS API function.

Note that _int_install_kernel_isr() call is only enabled if the vector table is located in RAM memory (see MQX_ROM_VECTORS configuration option in section Configuring MQX RTOS at Compile Time).

3.9.3.4 MQX_HARDWARE_INTERRUPT_LEVEL_MAX Configuration Parameter

On some processor platforms an internal concept of disabling "all interrupt levels" may be re-configured in a way that only interrupt levels up to the MQX_HARDWARE_INTERRUPT_LEVEL_MAX (field in the MQX_INITIALIZATION_STRUCT) are disabled. This effectively enables critical interrupt requests above that maximum level to be serviced asynchronously to MQX RTOS kernel execution and with minimum possible latency. From the MQX RTOS perspective, such an interrupt is considerred as a non-maskable interrupt and the same restrictions as for NMI apply.

Tables below summarize values written into the SR/BASEPRI register when switching to the task withthe defined priority, considering the value of the MQX_HARDWARE_INTERRUPT_LEVEL_MAX.

As an example for ColdFire platform, when MQX_HARDWARE_INTERRUPT_LEVEL_MAX is set to 7 switching to the task with the priority of 4 causes the SR register is loaded by the value of 2. It means that this task cannot be interrupted by the interrupts with the priority lower than 3.

Table 3-46. SR Register Values for Different Task Priorities and Different Values of MQX_HARDWARE_INTERRUPT_LEVEL_MAX valid for ColdFire platforms

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	Task Priority							
0	1	2	3	4	5	6	7	

0	NOT ALLOWED. EFFECTIVELY CHANGES TO MQX_HARDWARE_INTERRUPT_LEVEL_MAX=1							
1	0	0	0	0	0	0	0	0
2	1	0	0	0	0	0	0	0
3	2	1	0	0	0	0	0	0
4	3	2	1	0	0	0	0	0
5	4	3	2	1	0	0	0	0
6	5	4	3	2	1	0	0	0
7	6	5	4	3	2	1	0	0
8	NOT ALLOWED. EFFECTIVELY CHANGES TO MQX_HARDWARE_INTERRUPT_LEVEL_MAX=70							

On Cortex®-M4® and Cortex®-A5® core based platforms, the MQX RTOS interrupt processing is designed this way. Kinetis K family MCUs support 16 hardware interrupt priority levels. Internally MQX RTOS maps even levels (0, 2, 4, .., 14) for MQX RTOS applications while odd levels (1, 3, .., 15) are used internally. MQX RTOS application interrupt levels are 0 to 7, the mapping from MQX RTOS application levels 0 to 7 to hardware priority levels (0, 2 to 14) is implemented in the _bsp_int_init() function.

To install an MQX RTOS application defined ISR on Kinetis K, use the following code:

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

vector - number of non-core vector (for example, 37 for LLWU, defined in IRQInterruptIndex in the MCU header file).

priority - priority of the interrupt source. Allowed values: any integer between MQX_HARDWARE_INTERRUPT_LEVEL_MAX and 7 (including both values), the lower number, the higher priority is expected.

subpriority - omitted on Kinetis K.

enable - TRUE to enable the interrupt vector source in NVIC.

To install a kernel ISR on Kinetis K (to bypass MQX RTOS), use the following code:

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

vector - number of non-core vector (for example, 79 for FTM1, defined in IRQInterruptIndex in the MCU header file).

priority - priority of the interrupt source. Allowed values: 0 (for the highest priority interrupt) up to 7.

subpriority - omitted on Kinetis K.

enable - TRUE to enable the interrupt vector source in NVIC.

Notice that due to the ARM® hardware interrupt stacking feature, the kernel isr can be any C function with declaration void my_kernel_isr(void).

ARM Cortex®-M4 BASEPRI register values for different task priorities and different values of MQX_HARDWARE_INTERRUPT_LEVEL_MAX are shown in the image below. Note the most significant nibble is used to set-up the priority. Refer the ARM Reference Manual for BASEPRI register description.

Example: BASEPRI=0x20, the most significant nibble is 0x2, which means only interrupt with hardware priority level 1 or 0 can interrupt this task.

Table 3-47. SR Register Values for Different Task Priorities and Different Values of MQX_HARDWARE_INTERRUPT_LEVEL_MAX valid for ARM Cortex®-M4 core based platforms

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	Task Priority							
0	1	2	3	4	5	6	7	
0	NOT ALLOWED. EFFECTIVELY CHANGES TO MQX_HARDWARE_INTERRUPT_LEVEL_MAX=1							
1	0x20	0x40	0x60	0x80	0xA0	0xC0	0xE0	0
2	0x40	0x60	0x80	0xA0	0xC0	0xE0	0	0
3	0x60	0x80	0xA0	0xC0	0xE0	0	0	0
4	0x80	0xA0	0xC0	0xE0	0	0	0	0
5	0xA0	0xC0	0xE0	0	0	0	0	0
6	0xC0	0xE0	0	0	0	0	0	0
7	0xE0	0	0	0	0	0	0	0
8	NOT ALLOWED. EFFECTIVELY CHANGES TO MQX_HARDWARE_INTERRUPT_LEVEL_MAX=70							

ARM Cortex-A5 interrupt priority mask register (GICC_PMR – GIC register) values for different task priorities and different values of MQX_HARDWARE_INTERRUPT_LEVEL_MAX are shown in the following table. Note the most significant nibble is used to set-up the priority. Refer to the ARM Generic Interrupt Controller Architecture Specification for GICC_PMR register description.

Table 3-48. SR Register Values for Different Task Priorities and Different Values of MQX_HARDWARE_INTERRUPT_LEVEL_MAX valid for ARM® Cortex®-A5 core based platforms

MQX_HARDWARE_INTERRUPT_LEVEL_MAX	Task Priority							
0	1	2	3	4	5	6	7	
0	NOT ALLOWED. EFFECTIVELY CHANGES TO MQX_HARDWARE_INTERRUPT_LEVEL_MAX=1							
1	0x20	0x40	0x80	0xA0	0xC0	0xE0	0xFF	0xFF
2	0x40	0x80	0xA0	0xC0	0xE0	0xFF	0xFF	0xFF
3	0x80	0xA0	0xC0	0xE0	0xFF	0xFF	0xFF	0xFF
4	0xA0	0xC0	0xE0	0xFF	0xFF	0xFF	0xFF	0xFF
5	0xC0	0xE0	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
6	0xE0	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
7	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF	0xFF
8	NOT ALLOWED. EFFECTIVELY CHANGES TO MQX_HARDWARE_INTERRUPT_LEVEL_MAX=7							

For NXP PowerPC® devices and ARM® Cortex®-M0+ devices, there is no support for automatic switching of interrupt levels based on priority of running task and all peripheral interrupts are always disabled by *_int_disable* regardless of MQX_HARDWARE_INTERRUPT_LEVEL_MAX setting.

3.9.4 Changing Default ISRs

When MQX RTOS handles an interrupt, it calls **_int_kernel_isr()**, which calls a default ISR with the interrupt number, if either of these conditions is true:

- The application has not installed an application-defined ISR for the interrupt number.
- The interrupt number is outside the range of the ISR table.

The application can get a pointer to the default ISR with **_int_get_default_isr()**.

The application can change the default ISR as described in the following table.

Table 3-49. Default ISRs

Default ISR	Description	Modify or install with

_int_default_isr	MQX RTOS installs it as the default ISR, when MQX RTOS starts. It blocks the task.	To modify: **_int_install_default_ isr()**
_int_exception_isr	Implements MQX RTOS exception handling.	To install: **_int_install_exception_ isr()**
_int_unexpected_ isr	Similar to **_int_default_isr()**, but also prints a message to the default console, identifying the unhandled interrupt.	To install: **_int_install_unexpected_ isr()**

3.9.5 Handling Exceptions

To implement MQX RTOS exception handling, an application should call **_int_install_exception_isr()**, which installs _int_exception_isr() as the default ISR. Thus, **_int_exception_isr()** is called, when an exception or unhandled interrupt occurs. The function **_int_exception_isr()** does the following when an exception occurs:

- If the exception occurs when a task is running and a task exception ISR exists, MQX RTOS runs the ISR; if a task exception ISR does not exist, MQX RTOS aborts the task by calling **_task_abort()**.
- If the exception occurs when an ISR is running and an ISR exception ISR exists, MQX RTOS aborts the running ISR and runs the ISR's exception ISR.
- The function walks the interrupt stack looking for information about the ISR or task that was running before the exception occurred.

Note	If the MQX RTOS exception ISR determines that the interrupt stack contains incorrect information, it calls **_mqx_fatal_error()** with error code **MQX_CORRUPT_INTERRUPT_STACK**.

3.9.6 Handling ISR Exceptions

An application can install an ISR exception handler for each ISR. If an exception occurs while the ISR is running, MQX RTOS calls the handler and terminates the ISR. If the application has not installed an exception handler, MQX RTOS simply terminates the ISR.

When MQX RTOS calls the exception handler, it passes:

- current ISR number
- data pointer for the ISR
- exception number
- address on the stack of the exception frame

Table 3-50. Summary: Handling ISR Exceptions

_int_get_exception_handler	Gets a pointer to the current exception handler for the ISR.
_int_set_exception_handler	Sets the address of the current ISR exception handler for the interrupt.

3.9.7 Handling Task Exceptions

A task can install a task-exception handler, which MQX RTOS calls, if the task causes an exception that is not supported.

Table 3-51. Summary: Handling Task Exceptions

_task_get_exception_handler	Gets the task-exception handler.
_task_set_exception_handler	Sets the task-exception handler.

3.9.8 Example: Installing an ISR

Install an ISR to intercept the kernel timer interrupt. Chain the ISR to the previous ISR, which is the BSP-provided periodic timer ISR.

```
/* isr.c */
#include <mqx.h>
#include <bsp.h>
#define MAIN_TASK     10
extern void main_task(uint32_t);
extern void new_tick_isr(void *);
const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function, Stack, Priority, Name, Attributes,      Param, Time Slice*/
  { MAIN_TASK, main_task, 2000,  8,       "Main", MQX_AUTO_START_TASK, 0,  0},
  { 0 }
};

typedef struct
{
   void *        OLD_ISR_DATA;
   INT_ISR_FPTR  OLD_ISR;
   _mqx_uint     TICK_COUNT;
} MY_ISR_STRUCT, * MY_ISR_STRUCT_PTR;

/*ISR*-----------------------------------------------------------
*
* ISR Name : new_tick_isr
* Comments :
*   This ISR replaces the existing timer ISR, then calls the
*   old timer ISR.
*END*----------------------------------------------------- */
void
   new_tick_isr
   (
      void * user_isr_ptr
```

```
   )
{
   MY_ISR_STRUCT_PTRisr_ptr;
   isr_ptr = (MY_ISR_STRUCT_PTR)user_isr_ptr;

   isr_ptr->TICK_COUNT++;
   /* Chain to previous notifier */
   (*isr_ptr->OLD_ISR)(isr_ptr->OLD_ISR_DATA);
}

/*TASK*-----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*   This task installs a new ISR to replace the timer ISR.
*   It then waits for some time, finally printing out the
*   number of times the ISR ran.
*END*----------------------------------------------------- */
void
   main_task
   (
      uint32_t initial_data
   )
{
   MY_ISR_STRUCT_PTRisr_ptr;

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

3.9.8.1 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   `mqx\examples\isr`

2. See the *MQX™ RTOS Release Notes document* for instructions on how to build and run the application.

Main task displays the number of times the application ISR was called.

3.10 Instrumentation

Instrumentation includes the following components:

- logs

- lightweight logs
- kernel log
- stack usage utilities

3.10.1 Logs

Many real-time applications need to record information about significant conditions, such as events, state transitions, or function entry and exit information. If the application records the information as it occurs, you can analyze the sequence to determine whether the application processed conditions correctly. If each piece of information has a timestamp (in absolute time), you can determine, where the application spends processing time, and therefore, which code should be optimized.

Note	To optimize code and data memory requirements on some target platforms, the Log component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

With the log component, you can store data into and retrieve it from a maximum of 16 logs. Each log has a predetermined number of entries. Each entry contains a timestamp (in absolute time), a sequence number, and application-defined data.

Table 3-52. Summary: Using Logs

Logs use certain structures and constants, which are defined in *log.h.*	Logs use certain structures and constants, which are defined in *log.h.*
_log_create	Creates a log.
_log_create_component	Creates the log component.
_log_destroy	Destroys a log.
_log_disable	Disables logging.
_log_enable	Enables logging.
_log_read	Reads from a log.
_log_reset	Resets the contents of a log.
_log_test	Tests the log component.
_log_write	Writes to a log.

3.10.1.1 Creating the Log Component

You can explicitly create the log component with **_log_create_component()**. If you do not explicitly create it, MQX RTOS creates it the first time an application creates a log or kernel log.

3.10.1.2 Creating a Log

To create a log, a task calls **_log_create()** and specifies:

- Log number, in range of zero through 15.
- Maximum number of **_mqx_uint** quantities to be stored in the log (this includes headers).
- What happens when the log is full. The default behavior is that no additional data is written. Another behavior is that new entries overwrite the oldest ones.

3.10.1.3 Format of a Log Entry

Each log entry consists of a log header (**LOG_ENTRY_STRUCT**), followed by application-defined data.

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

The fields are described in MQX RTOS Reference Manual.

3.10.1.4 Writing to a Log

Tasks write to a log with **_log_write()**.

3.10.1.5 Reading From a Log

Tasks read from a log by calling **_log_read()**, and specifying, how to read the log. Possible ways to read the log are:

- To read the newest entry.
- To read the oldest entry.
- To read the next entry from the previous one read (used with read oldest).
- To read the oldest entry and delete it.

3.10.1.6 Disabling and Enabling Writing to a Log

Any task can disable logging to a specific log with **_log_disable()**. Any task can subsequently enable logging to the log with **_log_enable()**.

3.10.1.7 Resetting a Log

A task can reset the contents of a log to its initial state of no data with **_log_reset()**.

3.10.1.8 Example: Using Logs

```
/* log.c */
#include <mqx.h>
#include <bsp.h>
#include <log.h>
#define MAIN_TASK 10
#define MY_LOG    1
extern void main_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
{
 /* Task Index, Function, Stack, Priority, Name, Attributes,      Param, Time Slice*/
  { MAIN_TASK, main_task, 2000,  8,       "Main", MQX_AUTO_START_TASK, 0,  0},
  { 0 }
};

typedef struct entry_struct
{
   LOG_ENTRY_STRUCT HEADER;
   _mqx_uint        C;
   _mqx_uint        I;
} ENTRY_STRUCT, * ENTRY_STRUCT_PTR;

/*TASK*----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*   This task logs 10 keystroke entries then prints out the log.
*END*------------------------------------------------------- */
void
   main_task
   (
      uint32_t initial_data
   )
{
   ENTRY_STRUCT entry;
   _mqx_uint   result;
   _mqx_uint   i;
   uchar       c;
   /* Create the log component. */
   result = _log_create_component();
   if (result != MQX_OK) {
      printf("Main task - _log_create_component failed!");
      _mqx_exit(0);

   }
   /* Create a log */
   result = _log_create(MY_LOG, 10 * (sizeof(ENTRY_STRUCT)/sizeof(_mqx_uint)), 0);
   if (result != MQX_OK) {
```

```
      printf("Main task - _log_create failed!");
_     _mqx_exit(0);

   }
   /* Write data into the log */
   printf("Please type in 10 characters:\n");
   for (i = 0; i < 10; i++) {
      c = getchar();
      result = _log_write(MY_LOG, 2, (_mqx_uint)c, i);
      if (result != MQX_OK) {
         printf("Main task - _log_write failed!");
      }
   }
   /* Read data from the log */
   printf("\nLog contains:\n");
   while (_log_read(MY_LOG, LOG_READ_OLDEST_AND_DELETE, 2,
     (LOG_ENTRY_STRUCT_PTR)&entry) == MQX_OK)
   {
      printf("Time: %ld.%03d%03d, c=%c, i=%d\n",
         entry.HEADER.SECONDS,
         (_mqx_uint)entry.HEADER.MILLISECONDS,
         (_mqx_uint)entry.HEADER.MICROSECONDS,
         (uchar)entry.C & 0xff,
         entry.I);
   }
   /* Delete the log */
   _log_destroy(MY_LOG);
_  _mqx_exit(0);

}
```

3.10.1.8.1 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   `mqx\examples\log`

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.
3. Type ten characters on the input console.

The program logs the characters, and displays the log entry on the console.

3.10.2 Lightweight Logs

Lightweight logs are similar to logs (see Logs), but with the following differences:

- All entries in all lightweight logs are the same size.
- You can create a lightweight log at a particular memory location.
- Lightweight logs can be timestamped in tick time or second/millisecond time, depending on how MQX RTOS was configured at compile time (for more

information, see Configuring MQX RTOS at Compile Time).

Note	To optimize code and data memory requirements on some target platforms, the LWLog component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Table 3-53. Summary: Using Lightweight Logs

Lightweight logs use certain structures and constants, which are defined in *lwlog.h.*	Lightweight logs use certain structures and constants, which are defined in *lwlog.h.*
_lwlog_calculate_size	Calculates the size needed for a lightweight log with a specified maximum number of entries.
_lwlog_create	Creates a lightweight log.
_lwlog_create_at	Creates a lightweight log at a location.
_lwlog_create_component	Creates the lightweight log component.
_lwlog_destroy	Destroys a lightweight log.
_lwlog_disable	Disables logging to lightweight logs.
_lwlog_enable	Enables logging to lightweight logs.
_lwlog_read	Reads from a lightweight log.
_lwlog_reset	Resets the contents of a lightweight log.
_lwlog_test	Tests the lightweight log component.
_lwlog_write	Writes to a lightweight log.

3.10.2.1 Creating the Lightweight Log Component

You can explicitly create the lightweight log component with **_lwlog_create_component()**. If you do not explicitly create it, MQX RTOS creates it the first time an application creates a lightweight log or kernel log.

3.10.2.2 Creating a Lightweight Log

A task can create a lightweight log at a particular location (**_lwlog_create_at()**), or let MQX RTOS choose the location (**_lwlog_create()**).

With either function, the task specifies:

- Log number in the range of one through 15 (zero is reserved for kernel log).
- Maximum number of entries in the log.
- What happens when the log is full. The default behavior is that no additional data is written. Another behavior is that new entries overwrite the oldest ones.

In the case of **_lwlog_create_at()**, the task also specifies the address of the log.

3.10.2.3 Format of a Lightweight Log Entry

Each lightweight log entry has the following structure.

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

The fields are described in MQX RTOS Reference Manual.

3.10.2.4 Writing to a Lightweight Log

Tasks write to a lightweight log with **_lwlog_write()**.

3.10.2.5 Reading From a Lightweight Log

Tasks read from a lightweight log by calling **_lwlog_read()** and specifying, how to read the log. Possible ways to read the log are:

- To read the newest entry.
- To read the oldest entry.
- To read the next entry from the previous one read (used with read oldest).
- To read the oldest entry and delete it.

3.10.2.6 Disabling and Enabling Writing to a Lightweight Log

Any task can disable logging to a specific lightweight log with **_lwlog_disable()**. Any task can subsequently enable logging to the lightweight log with **_lwlog_enable()**.

3.10.2.7 Resetting a Lightweight Log

A task can reset the contents of a lightweight log to its initial state of no data with **_lwlog_reset().**

3.10.2.8 Example: Using Lightweight Logs

```
/* lwlog.c */
#include <mqx.h>
#include <bsp.h>
#include <lwlog.h>
#define MAIN_TASK10
#define MY_LOG   1
extern void main_task(uint32_t initial_data);
const TASK_TEMPLATE_STRUCT   MQX_template_list[] =
{
 /* Task Index, Function, Stack, Priority, Name, Attributes,      Param, Time Slice*/
  { MAIN_TASK, main_task, 2000,  8,       "Main", MQX_AUTO_START_TASK, 0,  0},
  { 0 }
};

/*TASK*----------------------------------------------------------
*
* Task Name : main_task
* Comments :
*   This task logs 10 keystroke entries in a lightweight log,
*   then prints out the log.
*END*----------------------------------------------------- */
void
   main_task
   (
      uint32_t initial_data
   )
{
   LWLOG_ENTRY_STRUCT entry;
   _mqx_uint         result;
   _mqx_uint         i;
   uchar             c;
   /* Create the lightweight log component
   */ result = _lwlog_create_component();
   if (result != MQX_OK) {
      printf("Main task: _lwlog_create_component failed.");
     __mqx_exit(0);

   }
   /* Create a log */
   result = _lwlog_create(MY_LOG, 10,
   0); if (result != MQX_OK) {
      printf("Main task: _lwlog_create failed.");
      _mqx_exit(0);

   }
   /* Write data to the log */
   printf("Enter 10
   characters:\n"); for (i = 0;
   i < 10; i++) {
      c = getchar();
      result = _lwlog_write(MY_LOG, (_mqx_max_type)c,
         (_mqx_max_type)i, 0, 0, 0, 0, 0);
      if (result != MQX_OK) {
```

```
         printf("Main task: _lwlog_write failed.");
      }
   }
   /* Read data from the log */
   printf("\nLog contains:\n");
   while (_lwlog_read(MY_LOG,
      LOG_READ_OLDEST_AND_DELETE, &entry) == MQX_OK)
   {
      printf("Time: ");
#if MQX_LWLOG_TIME_STAMP_IN_TICKS
      _psp_print_ticks((PSP_TICK_STRUCT_PTR)&entry.TIMESTAMP);
#else
      printf("%ld.%03ld%03ld", entry.SECONDS, entry.MILLISECONDS,

         entry.MICROSECONDS);
#endif
      printf(", c=%c, I=%d\n", (uchar)entry.DATA[0] &
         0xff, (_mqx_uint)entry.DATA[1]);
   }

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

}
```

3.10.2.8.1 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   ```
   mqx\examples\lwlog
   ```

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.
3. Type ten characters on the input console.

The program logs the characters and displays the log entry on the console.

3.10.3 Kernel Log

Kernel log lets an application log any combination of:

- Function entry and exit information for all calls to MQX RTOS functions.
- Function entry and exit information for specific function calls.
- Context switches.
- Interrupts.

Note	To optimize code and data memory requirements on some target platforms, the KLog component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file, and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Performance tool uses kernel log data to analyze, how an application operates and how it uses resources. For more information, see the MQX RTOS Host Tools User's Guide .

Table 3-54. Summary: Using Kernel Log

Kernel log uses certain structures and constants, which are defined in *log.h, lwlog.h,* and *klog.h.*	Kernel log uses certain structures and constants, which are defined in *log.h, lwlog.h,* and *klog.h.*
_klog_control	Control kernel logging.
_klog_create	Creates kernel log.
_klog_create_at	Creates kernel log at a specific location.
_klog_disable_logging_task	Disables kernel logging for the specified task.
_klog_enable_logging_task	Enables kernel logging for the specified task.
_klog_display	Displays an entry in kernel log.

3.10.3.1 Using Kernel Log

To use kernel log, an application follows these general steps.

1. Optionally create the lightweight log component as described on page Creating the Lightweight Log Component.
2. Create kernel log with **_klog_create()**. This is similar to creating a lightweight log, which is described on page Creating the Lightweight Log Component. You can also create kernel log at a specific location with **_klog_create_at()**.
3. Set up control for logging by calling **_klog_control()**, and specifying any combination of bit flags, as described in the following table.

Table 3-55. Logged Functions Overview

Select flags for:		
• MQX RTOS component	Select for:	These functions are logged:
	Errors	For example, **_mqx_exit(), _task_set_error(), _mqx_fatal_error()**.
	Events	Most from the **_event** family.
	Interrupts	Certain ones from the **_int** family.
	LWSems	The **_lwsem** family.
	Memory	Certain ones from the **_mem** family.
	Messages	Certain ones from the **_msg, _msgpool,** and **_msgq** families.
	Mutexes	Certain ones from the **_mutatr** and **_mutex** families.

	Names	The **_name** family.
	Partitions	Certain ones from the **_partition** family.
	Semaphores	Most from the **_sem** family.
	Tasking	The **_sched, _task, _taskq**, and **_time** families.
	Timing	The **_timer** family; certain ones from the **_time** family.
	Watchdogs	The **_watchdog** family.
• Specific tasks only (task qualified)	For each task to log, call one of: **_klog_disable_logging_task()** **_klog_enable_logging_task()**	For each task to log, call one of: **_klog_disable_logging_task()** **_klog_enable_logging_task()**
• Interrupts	• Interrupts	• Interrupts
• Periodic timer interrupts (system clock) • Context switches	• Periodic timer interrupts (system clock) • Context switches	• Periodic timer interrupts (system clock) • Context switches

3.10.3.2 Disabling Kernel Logging

Kernel logging can make your application use more resources and run slower. After you have tested and verified the application, you might want to create a version that does not include the ability to log to kernel log. To remove kernel logging for any part of MQX RTOS, you must recompile MQX RTOS with the **MQX_KERNEL_LOGGING** option set to zero. For more information, see MQX RTOS Compile-Time Configuration Options. The complete procedure for recompiling MQX RTOS is described in Rebuilding NXP MQX RTOS.

3.10.3.3 Example: Using Kernel Log

Log all calls to the timer component and all periodic timer interrupts.

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



```
  { 0 }
};

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

   /* Create kernel log */
   result = _klog_create(4096, 0);
   if (result != MQX_OK) {
      printf("Main task - _klog_create failed!");

     _mqx_exit(0);
   }
   /* Enable kernel log */
   _klog_control(KLOG_ENABLED | KLOG_CONTEXT_ENABLED |
      KLOG_INTERRUPTS_ENABLED|
      KLOG_SYSTEM_CLOCK_INT_ENABLED |
      KLOG_FUNCTIONS_ENABLED | KLOG_TIME_FUNCTIONS |
      KLOG_INTERRUPT_FUNCTIONS, TRUE);
   /* Write data into kernel log
   */ for (i = 0; i < 10; i++) {
      __time_delay_ticks(5 * i);
   }
   /* Disable kernel log */
   _klog_control(0xFFFFFFFF, FALSE);
   /* Read data from kernel log */
   printf("\nKernel log contains:\n");
   while (_klog_display()){
   }
__mqx_exit(0);

}
```

3.10.3.3.1 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   `mqx\examples\klog`

2. See the *MQX™ RTOS Release Notes* for instructions on how to build and run the application.

After about three seconds, **Main_task()** displays the contents of kernel log.

3.10.4 Stack Usage Utilities

MQX RTOS offers core utilities that let you examine and refine the size of the interrupt stack and the size of each task's stack.

Table 3-56. Summary: Stack Usage Utilities

To use these utilities, you must have configured MQX RTOS with MQX_MONITOR_STACK. For more information, see MQX RTOS Compile-Time Configuration Options. The complete procedure for recompiling MQX RTOS is described in Rebuilding NXP MQX RTOS.	To use these utilities, you must have configured MQX RTOS with MQX_MONITOR_STACK. For more information, see MQX RTOS Compile-Time Configuration Options. The complete procedure for recompiling MQX RTOS is described in Rebuilding NXP MQX RTOS
_klog_get_interrupt_stack_ usage	Gets the interrupt stack boundary and the total amount of stack used.
_klog_get_task_stack_usage	Gets the stack size and the total amount of the stack used for a specific task.
_klog_show_stack_usage	Calculates and displays the amount of stack used by each task and the interrupt stack.

3.11 Utilities

Utilities include:

- queues
- name component
- run-time testing
- additional utilities

3.11.1 Queues

The queue component lets you manage doubly linked lists of elements.

Note	To optimize code and data memory requirements on some target platforms, the Queue component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file and recompile the MQX RTOS PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Table 3-57. Summary: Using Queues

_queue_dequeue	Removes the element that is at the start of the queue.
_queue_enqueue	Adds the element to the end of the queue.
_queue_get_size	Gets the number of elements in the queue.

_queue_head	Gets (but doesn't remove) the element that is at the start of the queue.
_queue_init	Initializes the queue.
_queue_insert	Inserts the element in the queue.
_queue_is_empty	Determines, whether the queue is empty.
_queue_next	Gets (but doesn't remove) the next element in the queue.
_queue_test	Tests the queue.
_queue_unlink	Removes the specific element from the queue.

3.11.1.1 Queue Data Structures

The queue component requires two data structures, which are defined in *mqx.h*:

- **QUEUE_STRUCT**- keeps track of the size of the queue, and pointers to the start and end of the queue. MQX RTOS initializes the structure, when a task creates the queue.
- **QUEUE_ELEMENT_STRUCT**- defines the structure of a queue element. The structure is the header structure of an application-defined object that the task wants to queue.

3.11.1.2 Creating a Queue

A task creates and initializes a queue by calling **_queue_init()** with a pointer to a queue object and the maximum size of the queue.

3.11.1.3 Adding Elements To a Queue

A task adds an element to the end of a queue by calling **_queue_enqueue()** with pointers to the queue and to queue element object, which is the header structure of the object that the task wants to queue.

3.11.1.4 Removing Elements From a Queue

A task gets and removes an element from the start of a queue by calling **_queue_dequeue()** with a pointer to the queue.

3.11.2 Name Component

With the name component, tasks can associate a 32-bit number with a string or symbolic name. MQX RTOS stores the association in a names database that all tasks on the processor can use. The database avoids global variables.

Note	To optimize code and data memory requirements on some target platforms, the Name component is not compiled in the MQX RTOS kernel by default. To test this feature, you need to enable it first in the MQX RTOS user configuration file and recompile the RTOS MQX PSP, BSP, and other core components. See Rebuilding NXP MQX RTOS for more details.

Table 3-58. Summary: Using the Name Component

The name component uses certain structures and constants, which are defined in *name.h*	The name component uses certain structures and constants, which are defined in *name.h*
_name_add	Adds a name to the names database (a name is a NULL-terminated string, max length 32 characters, including NULL).
_name_create_component	Creates the name component.
_name_delete	Deletes a name from the names database.
_name_find	Looks up a name in the names database and gets its number.
_name_find_by_number	Looks up a number in the names database and gets its name.
_name_test	Tests the name component.

3.11.2.1 Creating the Name Component

An application can explicitly create the name component with **_name_create_component()**. If you do not explicitly create it, MQX RTOS creates it with default values the first time an application uses the names database.

The parameters and their default values are the same as for the event component, which is described on page Creating the Event Component.

3.11.3 Run-Time Testing

MQX RTOS provides core run-time testing that tests the integrity of most MQX RTOS components.

A test determines, whether the data that is associated with the component is valid and not corrupted. MQX RTOS considers the data in a structure valid, if the structure's **VALID** field is a known value. MQX RTOS considers data in a structure corrupted, if its **CHECKSUM** field is incorrect or pointers are incorrect.

An application can use run-time testing during its normal operation.

Table 3-59. Summary: Run-Time Testing

_event_test	Events
_log_test	Logs
_lwevent_test	Lightweight events
_lwlog_test	Lightweight logs
_lwmem_test	Lightweight memory with variable-size blocks
_lwsem_test	Lightweight semaphores
_lwtimer_test	Lightweight timers
_mem_test	Memory with variable-size blocks
_msgpool_test	Message pools
_msgq_test	Message queues
_mutex_test	Mutexes
_name_test	Name component
_partition_test	Memory with fixed-size blocks (partitions)
_queue_test	Application-implemented queue
_sem_test	Semaphores
_taskq_test	Task queues
_timer_test	Timers
_watchdog_test	Watchdogs

3.11.3.1 Example: Doing Run-Time Testing

The application uses all MQX RTOS components. A low-priority task tests all the components. If it finds an error, it stops the application.

```
/* test.c */
#include <mqx.h>
#include <fio.h>
#include <event.h>
#include <log.h>
#include <lwevent.h>
#include <lwlog.h>
#include <lwmem.h>
#include <lwtimer.h>
#include <message.h>
#include <mutex.h>
#include <name.h>
#include <part.h>
#include <sem.h>
#include <timer.h>
#include <watchdog.h>
extern void background_test_task(uint32_t);
const TASK_TEMPLATE_STRUCT  MQX_template_list[] =
```

```
{
 /* Task Index,Function,       Stack,Prio,Name,Attributes,       Param,Time Slice */
 { 10      , background_test_task,2000, 8,
           "Main",MQX_AUTO_START_TASK,0,  0},
 { 0 }
};

/*TASK*----------------------------------------------------------
*
* Task Name : background_test_task
* Comments :
*   This task is meant to run in the background testing for
* integrity of MQX RTOS component data structures.
*END*------------------------------------------------------- */
void background_test_task
   (
      uint32_t parameter
   )
{
   _partition_id partition;
   _lwmem_pool_id lwmem_pool_id;
   void *       error_ptr;
   void *       error2_ptr;

   _mqx_uint     error;
   _mqx_uint     result;
   while (TRUE) {
      result = _event_test(&error_ptr);
      if (result != MQX_OK){
         printf("\nFailed _event_test: 0x%X.", result);
         _mqx_exit(1);
      }
      result = _log_test(&error);
      if (result != MQX_OK){
         printf("\nFailed _log_test: 0x%X.", result);
         _mqx_exit(2);
      }
      result = _lwevent_test(&error_ptr, &error2_ptr);
      if (result != MQX_OK){
         printf("\nFailed _lwevent_test: 0x%X.", result);
         _mqx_exit(3);
      }
      result =
      _lwlog_test(&error);
      if (result != MQX_OK){
         printf("\nFailed _lwlog_test: 0x%X.", result);
         _mqx_exit(4);
      }
      result = _lwsem_test(&error_ptr, &error2_ptr);
      if (result != MQX_OK){
         printf("\nFailed _lwsem_test: 0x%X.", result);
         _mqx_exit(5);
      }
      result = _lwmem_test(&lwmem_pool_id, &error_ptr);
      if (result != MQX_OK){
         printf("\nFailed _lwmem_test: 0x%X.", result);
         _mqx_exit(6);
      }
      result = _lwtimer_test(&error_ptr, &error2_ptr);
      if (result != MQX_OK){
```

```
        printf("\nFailed _lwtimer_test: 0x%X.", result);
        _mqx_exit(7);
    }
    result = _mem_test_all(&error_ptr);
    if (result != MQX_OK){
        printf("\nFailed _mem_test_all,");
        printf("\nError = 0x%X, pool = 0x%X.",
        result,(_mqx_uint)error_ptr);
        _mqx_exit(8);
    }
    /*
    ** Create the message component.
    ** Verify the integrity of message pools and message queues.
    */
    if (_msg_create_component() != MQX_OK){
        printf("\nError creating the message
        component.");
        _mqx_exit(9);
    }
    if (_msgpool_test(&error_ptr, &error2_ptr) != MQX_OK){
        printf("\nFailed _msgpool_test.");
        _mqx_exit(10);
    }
    if (_msgq_test(&error_ptr, &error2_ptr) != MQX_OK){
        printf("\nFailed _msgq_test.");
        _mqx_exit(11);
    }
    if (_mutex_test(&error_ptr) != MQX_OK){
        printf("\nFailed _mutex_test.");
        _mqx_exit(12);
    }
    if (_name_test(&error_ptr, &error2_ptr) != MQX_OK){
        printf("\nFailed _name_test.");
        _mqx_exit(13);
    }
    if (_partition_test(&partition, &error_ptr, &error2_ptr)
        != MQX_OK)
    {
        printf("\nFailed _partition_test.");
        _mqx_exit(14);
    }
    if (_sem_test(&error_ptr) != MQX_OK){
        printf("\nFailed _sem_test.");
        _mqx_exit(15);
    }
    if (_taskq_test(&error_ptr, &error2_ptr) != MQX_OK){
        printf("\nFailed _takq_test.");
        _mqx_exit(16);
    }
    if (_timer_test(&error_ptr) != MQX_OK){
        printf("\nFailed _timer_test.");
        _mqx_exit(17);
    }
    if (_watchdog_test(&error_ptr, &error2_ptr) != MQX_OK){
        printf("\nFailed _watchlog_test.");
        _mqx_exit(18);
    }
    printf("All tests passed.");
```

```
        _mqx_exit(0);
    }
}
```

3.11.3.1.1 Compiling the Application and Linking it with MQX RTOS

1. Go to this directory:

   ```
   mqx\examples\test
   ```

2. See the *MQX™ RTOS Release Notes document* for instructions on how to build and run the application.

3.11.4 Additional Utilities

Table 3-60. Summary: Additional Utilities

_mqx_bsp_revision	Revision of the BSP.
_mqx_copyright	Pointer to the MQX RTOS copyright string.
_mqx_date	Pointer to the string that indicates, when MQX RTOS was built.
_mqx_fatal_error	Indicates that an error has been detected that is severe enough that MQX RTOS or the application can no longer function properly.
_mqx_generic_revision	Revision of the generic MQX RTOS code.
_mqx_get_counter	Gets a processor-unique 32-bit number.
_mqx_get_cpu_type	Gets the processor type.
_mqx_get_exit_handler	Gets a pointer to the MQX RTOS exit handler, which MQX RTOS calls when it exits.
_mqx_get_kernel_data	Gets a pointer to kernel data.
_mqx_get_system_task_id	Gets the task ID of System task descriptor.
_mqx_get_tad_data	Gets the **TAD_RESERVED** field from a task descriptor.
_mqx_idle_task	Idle task.
_mqx_io_revision	I/O revision for the BSP.
_mqx_psp_revision	Revision of the PSP.
_mqx_set_cpu_type	Sets the processor type.
_mqx_set_exit_handler	Sets the address of the MQX RTOS exit handler, which MQX RTOS calls, when it exits.
_mqx_set_tad_data	Sets the **TAD_RESERVED** field in a task descriptor.
_mqx_version	Pointer to the string that indicates the version of MQX RTOS.
_mqx_zero_tick_struct	A constant zero-initialized tick structure that an application can use to initialize one of its tick structures to zero.
_str_mqx_uint_to_hex_string	Converts an **_mqx_uint** value to a hexadecimal string.

_strnlen	Calculates the length of a limited-length string.

3.12 Embedded Debugging

There are several ways to debug MQX RTOS-based applications:

- Using plain debugger environment, which is not aware about the MQX RTOS operating system. This simple approach may work well, when using breakpoints and single-stepping through application code.
- Using operating system awareness in the debugger (so called task-aware debugger or TAD). This approach helps to see the debugged code in the context of individual tasks. It also helps to examine the internal MQX RTOS data structures in a user-friendly way.

3.13 Configuring MQX RTOS at Compile Time

MQX RTOS is built with certain features that you can include or exclude by changing the value of compile-time configuration options. If you change any configuration value, you must recompile MQX RTOS and relink it with your target application.

As the MQX RTOS library may also depend on some MQX RTOS configuration options, it must be typically recompiled as well.

Like MQX RTOS library, there are also other code components that use the MQX OS services (for example RTCS, MFS, USB). These components need to be re-compiled after MQX RTOS.

Note	Comparing with original ARC versions, NXP MQX RTOS introduces a different method of compile-time configuration of the MQX OS and other components. Original method used the compiler command-line -D options or *source\psp\platform \psp_cnfg.asm* file. In NXP MQX RTOS, there is a central user configuration file *user_config.h* in the *config/mcu/<mcu>* directory, which can be used to override default configuration options. The same configuration file is used by other system components like RTCS, MFS, or USB.

3.13.1MQX RTOS Compile-Time Configuration Options

This section provides a list of MQX RTOS configuration options. The default value of any of these options can be overridden in the *config/<board>/user_config.h* file.

The default values are defined in the *mqx/source/include/mqx_cnfg.h* file.

Note	Do not change the *mqx_cnfg.h* file directly. Always use the board-specific or project-specific *user_config.h* file in your *config* directory.

3.13.1.1 MQX_COMPONENT_DESTRUCTION

Default is one.

One: MQX RTOS includes the functions that allow MQX RTOS components (such as the semaphore component or event component) to be destroyed. MQX RTOS reclaims all the resources that the component allocated.

3.13.1.2 MQX_DEFAULT_TIME_SLICE_IN_TICKS

Default is one.

One: Default time slice in the task template structure is in units of ticks.

Zero: Default time slice in the task template structure is in milliseconds.

The value also affects the time-slice field in the task template, because the value is used to set a task's default time slice.

3.13.1.3 MQX_EXIT_ENABLED

Default is one.

One: MQX RTOS includes code to allow the application to return from the **_mqx()** call.

3.13.1.4 MQX_HAS_TIME_SLICE

Default is one.

One: MQX RTOS includes code to allow time-slice scheduling of tasks at the same priority.

3.13.1.5 MQX_HAS_DYNAMIC_PRIORITIES

Default is one.

One: MQX RTOS includes code to change task priorities dynamically by **_task_set_priority()** call or by priority inheritance or priority boosting.

3.13.1.6 MQX_HAS_EXCEPTION_HANDLER

Default is one.

One MQX RTOS includes code to handle exceptions (see psp/<psp>/int_xcpt.c) and to set/get task **exception** handler routine by using the **_task_set_exception_handler** and**_task_get_exception_handler** calls.

3.13.1.7 MQX_HAS_EXIT_HANDLER

Default is one.

One: MQX RTOS includes code to execute task exit handler before the task exits. Also the **_task_set_exit_handler** and **_task_get_exit_handler** calls are included.

3.13.1.8 MQX_HAS_HW_TICKS

Default is one.

One: MQX RTOS includes support for hardware ticks and associated calls: **_time_get_hwticks**, **_time_get_hwticks_per_tick** and **_psp_usecs_to_ticks**. Note that hardware ticks also need to be supported by the BSP.

3.13.1.9 MQX_HAS_TICK

Default is one. It is recommended to leave this option enabled.

One: MQX RTOS includes support for tick time and all related functionality of delaying tasks, waiting for synchronization objects with timeout etc.

3.13.1.10 MQX_TD_HAS_TEMPLATE_INDEX

Default is one.

One: The MQX RTOS task descriptors maintain the original index value coming from the TASK_TEMPLATE_ STRUCT array. This value is maintained for backward compatibility only and is not used by MQX RTOS kernel.

3.13.1.11 MQX_TD_HAS_STACK_LIMIT

Default is one.

One: The MQX RTOS task descriptors maintain the task limit value which is needed by various stack overflow checking calls like **_task_check_stack**.

3.13.1.12 MQX_INCLUDE_FLOATING_POINT_IO

Default is zero.

One: **_io_printf()** and **_io_scanf()** include floating point I/O code.

3.13.1.13 MQX_IS_MULTI_PROCESSOR

Default is one.

One: MQX RTOS includes code to support multiprocessor MQX RTOS applications.

3.13.1.14 MQX_KERNEL_LOGGING

Default is one.

One: Certain functions in each component write to kernel log, when they are entered and as they exit. The setting reduces performance, only if you enable logging for the component. You can control, which component is logged with **_klog_control()**.

3.13.1.15 MQX_LWLOG_TIME_STAMP_IN_TICKS

Default is one.

One: Timestamp in lightweight logs is in ticks.

Zero: Timestamp is in seconds, milliseconds, and microseconds.

3.13.1.16 MQX_MEMORY_FREE_LIST_SORTED

Default is one.

One: MQX RTOS sorts the freelist of memory blocks by address. This reduces memory fragmentation, but increases the time MQX RTOS takes to free memory.

3.13.1.17 MQX_MONITOR_STACK

Default is one.

One: MQX RTOS initializes all task and interrupt stacks to a known value, so that MQX RTOS components and debuggers can calculate how much stack is used. The setting reduces performance, only when MQX RTOS creates a task.

You must set the option to one in order to make use of:

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

3.13.1.18 MQX_MUTEX_HAS_POLLING

Default is one.

One: MQX RTOS includes code to support the mutex options **MUTEX_SPIN_ONLY** and **MUTEX_LIMITED_SPIN**.

3.13.1.19 MQX_PROFILING_ENABLE

Default is zero.

One: Code to support an external profiling tool is compiled into MQX RTOS. Profiling adds to the size of the compiled image, and MQX RTOS runs slower. You can use profiling, only if the toolset that you are using supports profiling.

3.13.1.20 MQX_RUN_TIME_ERR_CHECK_ENABLE

Default is zero.

One: Code to support an external run-time error-checking tool is compiled into MQX RTOS. This adds to the size of the compiled image, and MQX RTOS runs slower. You can use run-time error checking, only if the toolset that you are using supports it.

3.13.1.21 MQX_ROM_VECTORS

Default is zero.

One: The interrupt vector table is not copied into RAM. The ROM-based table is set up correctly to handle all interrupts by the default MQX RTOS interrupt dispatcher. The application is still able to install interrupt service routine by using the **_int_install_isr** call. However, the **_int_install_kernel_isr** call cannot be used to install the low-level interrupt service routines directly in the vector table.

3.13.1.22 MQX_SPARSE_ISR_TABLE

Default is zero.

One: The MQX RTOS interrupt service routine table is allocated as an "array of linked lists" instead of linear array. This option is independent on the MQX_ROM_VECTORS as it deals with the "logical" table managed by the interrupt dispatcher in MQX RTOS. With the sparse ISR table, only the ISRs installed by the **_int_install_isr** call consume RAM memory. Interrupt latency increases as MQX RTOS needs to walk the list to find user ISR to be invoked.

3.13.1.23 MQX_SPARSE_ISR_SHIFT

Default is 3.

When MQX_SPARSE_ISR_TABLE is defined as 1, this MQX_SPARSE_ISR_SHIFT option determines the number of bits the vector number is shifted to get index of ISR linked list root. For example, with 256 potential interrupt sources and with shift value of 3, it makes 256>>3=32 lists each with maximum depth of eight ISR entries. Shift value of 8 would yield one big linked list of all ISR entries.

3.13.1.24 MQX_TASK_CREATION_BLOCKS

Default is one. The option applies to multiprocessor applications only.

One: A task blocks, when it calls **_task_create()** to create a task on another processor. The creating task blocks, until the new task is created and an error code is returned.

3.13.1.25 MQX_TASK_DESTRUCTION

Default is one.

One: MQX RTOS allows tasks to be terminated. As a result, MQX RTOS includes code that frees all the MQX RTOS-managed resources that terminated tasks own.

3.13.1.26 MQX_TIMER_USES_TICKS_ONLY

Default is zero.

One: Timer task processes periodic-timer and one-shot timer requests using tick time for timeout reporting, rather than second/millisecond time.

3.13.1.27 MQX_USE_IDLE_TASK

Default is one.

One: the kernel creates the idle task which executes when no other tasks are ready, otherwise, the processor stops when there are no tasks to run.

3.13.1.28 MQX_USE_IO

Default is one.

One: MQX RTOS implements the I/O subsystem calls needed by I/O drivers. Without the I/O subsystem, no driver can be installed or used and tasks are not able to use stdin/

stdout/stderr handles.

3.13.1.29 MQX_USE_LWMEM_ALLOCATOR

Default is zero.

One: Calls to the **_mem** family of functions are replaced with calls to the corresponding function in the **_lwmem** family.

3.13.1.30 MQXCFG_ENABLE_FP

Default value depends on the **MQXCFG_MEM_COPY_NEON**. If **MQXCFG_MEM_COPY_NEON** is set, default value is 1. Otherwise, default value is 0.

If it is set, enables FPU support in MQX RTOS. Scheduler stores and restores the FPU context and provides API for float point support in tasks and interrupts.

3.13.1.31 MQX_SAVE_FP_ALWAYS

Default value depends on the **MQXCFG_MEM_COPY_NEON**. If **MQXCFG_MEM_COPY_NEON** is set, default value is 1. Otherwise, default value is 0.

Enables the **MQX_FLOATING_POINT_TASK** flag to be set at each task. MQX RTOS stores and restores the FPU context in the scheduler. FPU context is stored in the interrupt prologue and restored in the interrupt epilogue. The user cannot disable FPU context storing during run time.

3.13.1.32 MQX_INCLUDE_FLOATING_POINT_IO

The default value is 0.

Enables floating point types, such as `printf` and `scanf`, in the MQX RTOS I/O function and enables float point conversion API.

3.13.1.33 MQXCFG_MEM_COPY

Default value is 0.

If it is set, it enables MQX RTOS to have a unique memory copy. Otherwise, it uses `memcpy` from the compiler library.

3.13.1.34 MQXCFG_MEM_COPY_NEON

Default value is 0.

If it is set, MQX RTOS uses special memory copy implementation with NEON instructions. This feature requires FPU to be supported in MQX RTOS. The options **MQXCFG_ENABLE_FP**, **MQX_SAVE_FP_ALWAYS** are set to 1.

3.13.2 Recommended Settings

The settings you choose for compile-time configuration options depend on the requirements of your application.

Note	The MQX RTOS build process and its compile-time configuration is specific for given target board (set in *config/<board>/user_config.h* directory). You may want to create your own configurations, specific to the custom board or even the application.

The following table shows common settings you can use as you develop your application.

Table 3-63. Compile-time Configuration Setting

Option	Default	Debug	Speed	Size
MQX_ALLOW_TYPED_MEMORY	1	1	0	0,1
MQX_COMPONENT_DESTRUCTION	1	0*, 1	0*	0*
MQX_DEFAULT_TIME_SLICE_IN_TICKS	0	0, 1	1	1
MQX_EXIT_ENABLED	1	0, 1	0	0
MQX_HAS_DYNAMIC_PRIORITIES	1	0, 1	0	0
MQX_HAS_EXIT_HANDLER	1	0, 1	0	0
MQX_HAS_TASK_ENVIRONMENT	1	0, 1	0	0
MQX_HAS_TIME_SLICE	1	0, 1	0	0
MQX_INCLUDE_FLOATING_POINT_IO	0	0, 1	0	0
MQX_IS_MULTI_PROCESSOR	1	0, 1	0	0
MQX_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_ROM_VECTORS	0	0, 1	0, 1	1
MQX_RUN_TIME_ERR_CHECK_ENABLE	0	1	0	0
MQX_SPARSE_ISR_TABLE	0	0, 1	0	1
MQX_SPARSE_ISR_SHIFT (in range 1-8)	3	any	lower	higher
MQX_TASK_CREATION_BLOCKS (for multiprocessor applications)	1	1	0	0, 1
MQX_TASK_DESTRUCTION	1	0, 1	0	0
MQX_TD_HAS_STACK_LIMIT	1	0, 1	0	0

MQX_TD_HAS_TEMPLATE_INDEX	1	0, 1	0	0
MQX_TIMER_USES_TICKS_ONLY	0	0,1	1	1
MQX_USE_IDLE_TASK	1	0, 1	0, 1	0
MQX_USE_LWMEM_ALLOCATOR	0	0, 1	1	1
MQX_VERIFY_KERNEL_DATA	1	1	0	0

Chapter 4 Rebuilding NXP MQX RTOS

4.1 Why Rebuild MQX RTOS?

Starting at version 4.0, the factory-precompiled libraries are not available within MQX RTOS distribution. To start working with the MQX RTOS you have to build all necessary MQX RTOS libraries first. Read this chapter to find out how to do that and what are the necessary steps.

In general, building or re-building the MQX RTOS libraries is required when you do any of the following:

- After installing a fresh MQX RTOS package without factory-precompiled libraries.
- If you change compiler options (for example optimization level).
- If you change MQX RTOS compile-time configuration options in the *config/<board>/user_config.h* file.
- If you develop a new BSP (for example by adding a new I/O driver).
- If you incorporate changes that you made to MQX RTOS source code.

4.2 Before You Begin

Before you compile or build MQX RTOS:

- Read the *NXP MQX™ RTOS Release Notes* that accompany NXP MQX RTOS, to get information that is specific to your target environment.
- Ensure you have the required tools for your target environment:
- compiler
- assembler
- linker
- librarian
- Be familiar with the MQX RTOS directory structure and re-build instructions, as they are described in the *Getting Started with NXP MQX™ RTOS* document and also the instructions provided later in this section.

Note	NXP MQX RTOS can be conveniently built by using one of the supported development environments.

4.3 NXP MQX RTOS Directory Structure

The following figure shows the directory structure of a typical NXP MQX RTOS distribution, however, since MQX is delivered as a package specific to a particular processor and development tools, the layout of the package that you receive may vary slightly.

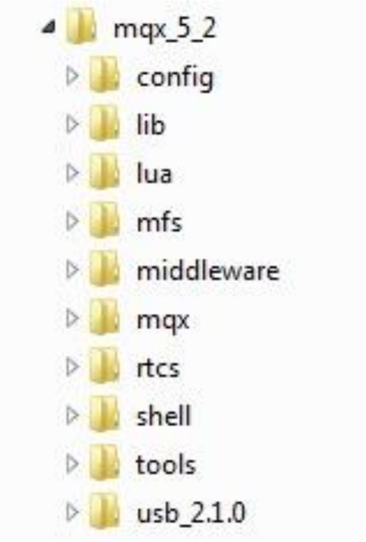

Figure 4-1. Directory Structure of NXP MQX RTOS

4.3.1 MQX RTOS Directory Structure

The following figure shows the directory structure of the MQX RTOS component located in the top-level *mqx* directory in more detail. In the build folder there is a sub-folder called 'mcux'. This contains the project files for building the BSP and PSP with the MCUXpresso Development Tools. However, if your distribution of MQX contains project files for a different set of development tools the the 'mcux' sub-folder would be replaced with a sub-folder with a name representative of your set of development tools. For example, a sub-folder called 'iar' would be located here if you were using the IAR Embedded Work Bench development tools.

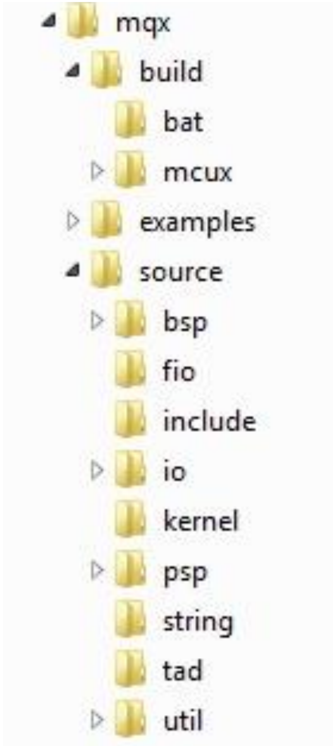

Figure 4-2. MQX RTOS Directory Structure

4.3.2 PSP Subdirectories

The *mqx\source\psp* directory contains the platform-dependent code of the PSP library. For example, a cortex_m subdirectory contains the MQX RTOS kernel parts specific to the NXP processors that are based on the ARM Cortex M architecture (core initialization, register save/restore code for interrupt handling, stack handling, cache control functions, etc.). This directory also contains processor definition files for the supported processor derivative.

4.3.3 BSP Subdirectories

The subdirectories in *mqx\source\bsp* typically follow the name of the board, and contain low-level startup code, processor, and board initialization code. The BSP also contains data structures used to initialize various I/O drivers in a way suitable for a given board.

This code compiles (together with the I/O drivers code) into the BSP library.

4.3.4 I/O Subdirectories

Subdirectories in the *mqx\source\io* contain source code for MQX RTOS I/O drivers. Typically, source files in each I/O driver directory are further split to device-specific and device-independent. The I/O drivers, suitable for given board, are part of the BSP build project, and are compiled into the BSP library.

4.3.5 Other Source Subdirectories

All other directories in the source contain generic parts of the MQX RTOS. Together with the platform-dependent PSP code, the generic sources are compiled into the PSP library.

4.4 NXP MQX RTOS Build Projects

All necessary build projects are located in the *mqx\build\<compiler>* directory. For each board, there are two build projects available, PSP and BSP. The BSP project contains board-specific code, while PSP is platform-specific (for example Cortex M) only. The PSP project does not contain any board-specific code. Despite this, both projects refer to the board name in their file names, and both also generate the binary output file into the same board-specific directory *lib\<board>.<compiler>*.

The board-independent PSP library is also compiled to board-specific output directory because the compile-time configuration file is taken from board-specific directory *config \<board>*. In other words, even if the PSP source code itself does not depend on the board features, the user may want to build a different PSP for different boards.

4.4.1 PSP Build Project

The PSP project is used to build the PSP library, which contains the platform-dependent parts from *mqx\source\psp* and also contains generic MQX RTOS code.

4.4.2 BSP Build Project

The BSP project is used to build the BSP library, which contains the board-specific code from *mqx\source\bsp\<board>* and also the selected I/O drivers from *mqx\source\io* directory.

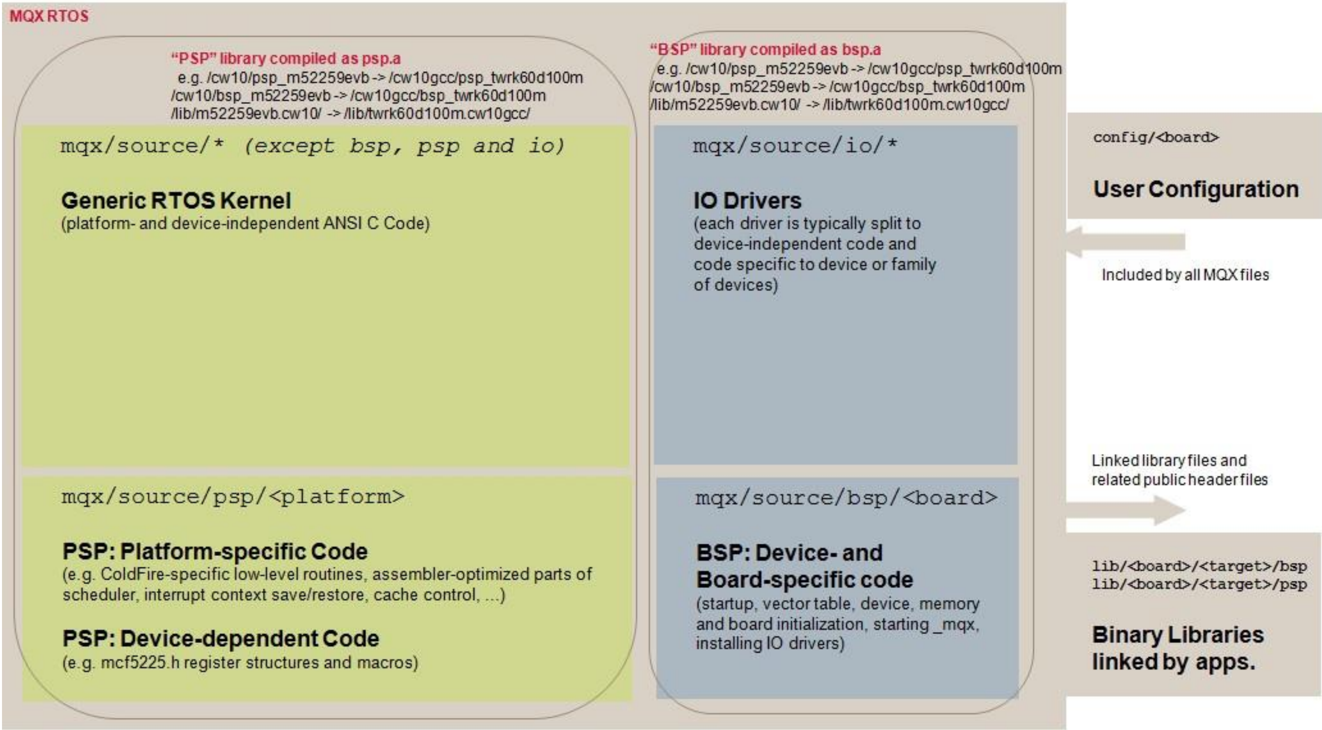

Figure 4-3. BSP Build Project

4.4.3 Post-Build Processing

All build projects are configured to generate the resulting binary library file in the top-level *lib\<board>.<compiler>* directory.

Both BSP and PSP build projects are also set up to execute the post-build batch file, which copies all the public header files to the destination *lib* directory. This makes the output */lib* folder the only place accessed by the MQX RTOS application code. The MQX RTOS application build projects do not need to make any reference to the MQX RTOS source tree at all.

4.4.4 Build Targets

All supported development environment enables you to have multiple build configurations, called build targets.

- **Debug** target - Compiler optimizations are set low to enable easy debugging. Libraries built using this target are copied into the respective folder of the *lib \<board>.<compiler>\debug* directory.

- **Release** target - Compiler optimizations are set to maximum, to achieve the smallest code size and fast execution. The resulting code is very hard to debug. Libraries built using this target are copied into the respective folder of the *lib\<board>.<compiler>\release* directory

4.5 Rebuilding NXP MQX RTOS

Rebuilding the MQX RTOS libraries is a simple task that involves opening the proper build projects for PSP and BSP in the development environment and building them. Do not forget to select the proper build target to be built or build all targets.

For specific information about rebuilding MQX RTOS and the examples that accompany it, see the release notes document in the MQX RTOS installation directory.

Chapter 5 FAQs

6.1 General

My application stopped. How do I tell if MQX RTOS is still running?

If the time is being updated, MQX RTOS is processing the periodic timer interrupt. If Idle task is running, MQX RTOS is running.

6.2 Events

Two tasks use an event group. The connection works for one task, but not for the other. Why?

The tasks are probably sharing the same global connection, rather than having their own local, individual connection. Each task should call **_event_open()** or **_event_open_fast()** to get its own connection.

6.3 Global Constructors

I need to initialize some global constructors, which use the 'new' operator, before I call 'main'; that is, before I start MQX RTOS. The 'new' operator calls malloc(), which I redefine to call the MQX RTOS function _mem_alloc(). How do I do this?

Initialize the constructors from **_bsp_pre_init()** (in *init_bsp.c*), which MQX RTOS calls after it initializes the memory management component.

6.4 Idle Task

What happens if Idle task blocks because of an exception?

If Idle task blocks, System task, which is really a system task descriptor that has no code, becomes the active task. System task descriptor sets up the interrupt stack, then re-enables interrupts. As a result, the application can continue to run.

6.5 Interrupts

An interrupt comes at periodic intervals that my application must respond to very quickly - quicker than MQX RTOS allows. What can I do?

Call **_int_install_kernel_isr()** to replace the kernel ISR (**_int_kernel_isr()**). Your replacement ISR must:

- Save all registers on entry, and restore them on exit.
- It must not call any MQX RTOS functions.
- Pass information to other tasks (if required) by an application-implemented mechanism (usually ring buffers with head and tail pointers and total size fields).

My application consists of several tasks that should run only when a certain signal comes in by an interrupt. How can my ISR that handles the interrupt communicate to the appropriate tasks?

If the target hardware allows it, set the priority of the interrupt to be higher than what MQX RTOS uses, when it disables interrupts (see the **MQX_HARDWARE_INTERRUPT_LEVEL_MAX** field in the **MQX_INITIALIZATION_STRUCT**). If you do so, the interrupt is able to interrupt an MQX RTOS-critical section. For example, on an ARCtangent processor, MQX RTOS can be configured to never disable level-2 interrupts and to use only level-1 interrupts to disable/enable in critical sections.

If the target hardware does not allow you to set the priority of the interrupt as described in the preceding paragraph, use the event component to send a signal from the ISR to several tasks. The tasks open connections to an event group, and one of the tasks gives the ISR the connection. Each task calls **_event_wait_any()** or **_event_wait_all()** and blocks. The ISR calls **_event_set()** to unblock the tasks.

When I save, and then restore an ISR for a specific interrupt, how do I get the value of the data pointer that was associated with the original ISR?

Call **_int_get_isr_data()** before you install the temporary ISR. This function returns a pointer to the data of the specific vector that you pass to it.

6.6 Memory

How does a task transfer a memory block that it does not own?

Although the task that owns the memory is the one that usually transfers it, a non-owner can do so with **_mem_transfer()**.

My task allocates a 10-byte memory block, but it always gets more. Why?

When MQX RTOS allocates a memory block, it aligns the block to the appropriate memory boundary and associates an internal header with the block. It also enforces a minimum size.

Can a task allocate a memory block for another task?

No. Tasks allocate their own memory. However, a task can subsequently transfer the memory to another task.

If _partition_test() detects a problem, does it try to repair the problem?

No. This indicates that memory is corrupted. Debug the application to determine the cause.

When I extend the default memory pool, must the additional memory be contiguous with the existing end of the pool?

No. The additional memory can be anywhere.

What does _mem_get_highwater() return, if I extend the default-memory pool with non-contiguous memory?

The highwater mark is the highest memory location, from which MQX RTOS has allocated a memory block.

I have tasks on several processors that need to share memory. How can I provide mutual exclusion to the memory?

Depending on your hardware, you might be able to use a spin mutex to protect the shared memory. Spin mutexes call **_mem_test_and_set()**, which is multiprocessor safe, when the hardware supports locking shared memory.

6.7 Message Passing

How can I guarantee that target message queue IDs are associated with the correct task?

Create one task that uses the names database to associate each message queue number with a name. Each task then gets the queue number by specifying the name.

Can I send messages between a PC and my target hardware?

Yes. Create a program to run on your PC that sends and receives data packets to/from the application either serially, over PCI, or over ethernet. As long as the packets are formatted correctly, MQX RTOS passes on any that it receives.

My task successfully calls _msgq_send() several times with a newly allocated message each time. Eventually _msgq_send() fails.

You have probably run out of messages. Each time you allocate a new message to send, check whether the return is NULL. If it is, the receiving task is probably not freeing the messages, or is not getting an opportunity to run.

6.8 Mutexes

What happens, when the task that owns a mutex data structure is destroyed? Do tasks that are waiting to lock the mutex wait forever?

No. All components have cleanup functions. When a task is terminated, the cleanup function determines what resources the task is using and frees them. If a task has a mutex locked, MQX RTOS unlocks the mutex when it terminates the task. A task should not own the mutex structure memory; it should create the structure as a global variable or allocate it from a system memory block.

6.9 Semaphores

What happens if I "force destroy" a strict semaphore?

If the force destroy flag is set when you destroy a strict semaphore, MQX RTOS does not destroy the semaphore, until all the waiting tasks get and post the semaphore. (If the semaphore is non-strict, MQX RTOS immediately readies all the tasks that are waiting for the semaphore.)

Two tasks use a semaphore. The connection works for one task, but not for the other. Why?

The tasks are probably sharing the same global connection, rather than having their own local, individual connection. Each task should call **_sem_open()** or **_sem_open_fast()** to get its own connection.

6.10 Task Exit Handler Versus Task Exception Handler

What is the difference between the two?

MQX RTOS calls the task exit handler when a task calls **_task_abort()**, or when a task returns from its task body. If MQX RTOS exception handling is installed, MQX RTOS calls the task exception handler, if the task causes an exception that is not supported.

6.11 Task Queues

My application puts several tasks of the same priority in a priority task queue? How are they ordered?

Tasks are in FIFO order within a priority.

6.12 Tasks

Do I always need at least one autostart task?

Yes. In an application, at least one autostart application task is required in order to start the application. In a multiprocessor application (the application can create tasks remotely), each image need not have an autostart application task; however, each image must include IPC task as an autostart task in the task template list. If no application task is created on a processor, Idle task runs.

One autostart task creates all my other tasks and initializes global memory. Can I terminate it without affecting the child tasks?

Yes. When MQX RTOS terminates the creator, it frees the creator's resources (memory, partitions, queues, and so on) and stack space. The resources of the child tasks are independent of the creator and are not affected.

Does the creator task own its child task?

No. The only relationship between the two is that the child can get the task ID of its creator. The child has its own stack space and automatic variables.

What are tasks, and how are they created?

Tasks share the same code space, if they execute the same root function. A task always starts executing at the entry point of the root function even if the function is its creator's root function. This is not the same behavior as **fork()** in UNIX.

Can I move a created task to another processor?

No.

6.13 Time Slices

How does MQX RTOS measure a time slice? Is the time slice absolute or relative? That is, if a task has a 10 ms time slice and starts at time = 0 ms, does it give up the processor at time = 10 ms, or does it give up the processor after 10 ms of execution?

With a 10 ms time slice, MQX RTOS counts the number of periodic timer interrupts that have occurred, while the task is active. If the equivalent of ten or more milliseconds have expired, MQX RTOS effectively runs **_sched_yield()** for the task. As a result, a task does not get 10 ms of linear time since higher-priority tasks will preempt it. Also, if the task calls a scheduling function (for example **_task_block()** or **_sched_yield()**), MQX RTOS sets the task's time-slice counter back to zero.

As with timeouts, the time that MQX RTOS allocates is plus or minus **BSP_ALARM_FREQUENCY** ticks per second.

6.14 Timers

My application is on more than one processor. I have a master processor that sends a synchronization message to the other processors that causes them to reset their time. How can I make sure that the reset messages don't interfere with the timers that the application uses?

So that timers are not affected by changes to absolute time (**_time_set()**), start timers with relative time (**TIMER_ELAPSED_TIME**), rather than absolute time (**TIMER_KERNEL_TIME_MODE**).

What happens if _timer_start_oneshot_at() is given an expiry time that is in the past?

MQX RTOS puts the element in the timer queue. When the next periodic timer interrupt occurs, MQX RTOS determines that the current time is greater than, or equal to the expiry time, so the timer triggers and MQX RTOS calls the notification function.