



RTXC Quadros Real-time Operating System Technical Summary

Quadros Systems, Inc.

Real-time Operating Systems for Convergent Processing

www.quadros.com

Table of Contents

1	Introduction	3
1.1	Dual-kernel Architecture	3
1.2	Scalability	3
2	Feature Summary.....	3
2.1	RTXC/ss	3
2.2	RTXC/ms.....	4
3	Structure and Organization	4
3.1	Operating Zones.....	4
3.2	RTXC/ss	4
3.3	RTXC/ms.....	5
3.4	RTXC/dm.....	6
4	System Configuration with RTXCgen.....	6
5	System Components	6
5.1	User Space Components	6
5.1.1	<i>User Startup Code.....</i>	<i>6</i>
5.1.2	<i>System Initialization Code.....</i>	<i>7</i>
5.1.3	<i>Thread or Task Code</i>	<i>7</i>
5.1.4	<i>RTXC API Library.....</i>	<i>7</i>
5.1.5	<i>User Workspace.....</i>	<i>7</i>
5.1.6	<i>Configuration Tables</i>	<i>7</i>
5.2	System Space Components	7
5.2.1	<i>RTXC Module.....</i>	<i>7</i>
5.2.2	<i>System Workspace</i>	<i>7</i>
5.3	Host Environment.....	7
5.3.1	<i>Programming Language.....</i>	<i>8</i>
5.3.2	<i>Development Tools</i>	<i>8</i>
6	Object Class Overview	8
6.1	Threads	8
6.1.1	<i>Thread Scheduling</i>	<i>9</i>
6.1.2	<i>Thread Gates</i>	<i>9</i>
6.1.3	<i>Environment Arguments.....</i>	<i>9</i>
6.2	Tasks.....	9
6.2.1	<i>Task States</i>	<i>9</i>
6.2.2	<i>Scheduling Policies</i>	<i>10</i>
6.3	Semaphores	10
6.4	Queues.....	10
6.5	Mailboxes and Messages	10
6.6	Pipes	11
6.7	Event Sources	11
6.8	Counters.....	11
6.9	Alarms	11
6.10	Memory Partitions.....	11
6.11	Mutexes.....	12
6.12	Exceptions.....	12
7	Interrupt Servicing.....	12
7.1	Normal Interrupts.....	12
7.2	Private Interrupts	12
7.3	Hybrid Interrupts	12
8	More Information.....	12

1 Introduction

RTXC Quadros represents a new generation in real-time operating systems, designed from the ground up to address the demands of today's sophisticated embedded systems and new, convergent processing applications. RTXC Quadros was designed using the knowledge and experience of more than 25 years of commercial operating system experience over a broad range of real-time applications.

At the heart of the RTXC Quadros operating system is a unique, dual-kernel architecture that can be configured in four different frameworks. The result is an RTOS that is equally suited for 8- and 16-bit MCUs, 32-bit microprocessors, high performance digital signal processors and the rapidly emerging hybrid and multi-core processors. This scalable architecture provides a reliable foundation for present and future programs as the same API is utilized across all four frameworks. A developer can begin using the basic scheduler for an 8-bit project today and then easily move to one of the other frameworks when the next design calls a higher powered microprocessor.

1.1 Dual-kernel Architecture

The two major architectural components of RTXC Quadros are RTXC/ss (a thread-based architecture) and RTXC/ms (a task-based architecture). RTXC/ss features a single stack model with a very low-latency scheduler and a small footprint, making it ideally suited for applications requiring high frequency interrupt processing, such as in digital signal processing. RTXC/ms provides a multiple independent stack model for a classic pre-emptive multitasking architecture with a rich set of kernel services well suited to deterministic, hard real-time system requirements.

1.2 Scalability

RTXC Quadros is highly scalable. The user may select the basic framework of RTXC/ss or RTXC/ms alone, or RTXC/ss in combination with RTXC/ms. RTXC Quadros is further scalable by the selection of kernel object classes and the services that operate on them. RTXC Quadros also includes a graphical configuration tool, RTXCgen, to assist the user in scaling the selected RTXC framework as well as defining the RTXC kernel objects required by the application.

RTXC Quadros is written in ANSI C but it permits the user to develop applications using C or C++. Each distribution of RTXC is ported to a specific processor and bound to C compilers that support the target processors, making access to kernel services convenient for the developer.

2 Feature Summary

RTXC Quadros features support real-time, multitasking applications using either RTXC/ss or RTXC/ms, or RTXC/dm, a combination of RTXC/ss and RTXC/ms which allows the application to utilize the features of both frameworks.

2.1 RTXC/ss

RTXC/ss meets the needs of applications requiring a small code footprint, minimum use of RAM by the RTOS and very fast response to events. It makes use of lightweight application code elements called threads, which along with the RTOS make use of a single common stack. RTXC/ss supports the following features:

- Multiple levels of thread priority
- Fixed thread priorities within a level
- No context saved or restored for threads operating at same level
- Pre-emptive scheduling of threads between levels
- Multi-thread processing with selectable scheduling policies by level:
 - Priority
 - Round robin
- Static kernel objects:
 - Threads
 - Pipes
 - Event Sources
 - Counters
 - Alarms
 - Exceptions
- Single stack for all operations
- Small RAM and ROM usage
- Standard programmer interface in C or assembly language

2.2 RTXC/ms

RTXC/ms is intended for applications where the RAM budget is not as tight as in an application using RTXC/ss. RTXC/ms application code elements are called tasks and each task must have its own independent stack. The RTXC/ms multitasking scheduler allocates ownership of the CPU to various tasks according to their priorities and in response to external or internal events. RTXC/ms supports the following features:

- Multitasking with selectable scheduling policies
 - Preemptive
 - Round robin
 - Time-sliced
- Multiple stack model (one stack per task)
- Static and dynamic kernel objects
 - Tasks
 - Semaphores
 - FIFO Queues
 - Mailboxes and messages
 - Memory Partitions
 - Mutexes
 - Pipes
 - Event Sources
 - Counters
 - Alarms
 - Exceptions
- Fixed or dynamic task priorities
- Intertask communication and synchronization
 - Semaphores
 - Mailboxes and messages
 - Queues
 - Pipes
- Efficient management for “ticks” from internal or external sources
- Alarm options for many services
- Partitioned RAM management
- Priority inversion handling option for mutexes
- Efficient interrupt servicing
- Fast context switch
- Small RAM and ROM requirements
- Standard programmer interface in C on all processors
- Highly flexible configuration to permit custom fit to the application

3 Structure and Organization

RTXC Quadros is designed to operate in an embedded system, which is typically intended to perform a set of operations with little or no human intervention. RTXC Quadros makes no assumptions about the configuration of the target system. It is the responsibility of the application designer to define the target environment and to insure that all necessary devices have program support.

3.1 Operating Zones

Most real-time applications have operations that take place at different priority zones, the priority being associated with the importance of the operation. RTXC Quadros operates in three distinct priority zones. Interrupt servicing occupies the highest priority, Zone 1 and takes precedence over all operations in the other zones. RTXC Quadros supports interrupt nesting provided the hardware permits it. Most operations within RTXC/ss occur in Zone 2, the middle priority. Kernel operations of the RTXC/ms component also run in Zone 2. Zone 3 operations occur when there are no Zone 1 or Zone 2 operations that need attention. Typically, operations at Zone 3 are classified as control functions. In the RTXC/ss environment, Zone 3 operations are usually associated with a user-defined background loop. When the RTXC/ms component is present, Zone 3 operations are reserved for tasks.

3.2 RTXC/ss

The RTXC/ss component of RTXC Quadros closely meets the needs of applications requiring minimal RTOS footprint and RAM use along with very fast response time to events. It is a thread-based design in which the RTXC/ss Scheduler schedules threads as the result of requests by interrupt service routines or other threads. Typical use of RTXC/ss would be the execution of a DSP algorithm, such as an FFT, which processes a block of data in a loop fashion without stopping, until the data block is consumed.

While well suited for use in DSP processing, the RTXC/ss component is by no means limited to that class of application. The structure, size and performance of RTXC/ss, plus a rich set of services associated with the six supported classes of objects provide data communication, event synchronization and thread management, make RTXC/ss an excellent RTOS choice for a variety of other applications such as instrumentation, control, or device management.

RTXC/ss supports six classes of objects:

- o Threads
- o Event Sources
- o Counters
- o Alarms
- o Pipes
- o Exceptions

Each class is associated with a set of services that enables the developer to effect certain operational behaviors of the system.

Figure 1 depicts a typical topology for an application based on RTXC/ss. Note that the application exists in three priority zones: Zone 1 for interrupt processing is the highest priority, as it normally demands immediate attention. Zone 2 contains RTXC/ss and the threads it schedules for operations generally associated with interrupts. Zone 3 operates only when all processing in Zones 1 and 2 is complete. System startup occurs in Zone 3. After startup, operations in Zone 3 are user defined and do not interact with RTXC/ss. Zone 3 may be referred to as the Background Zone.

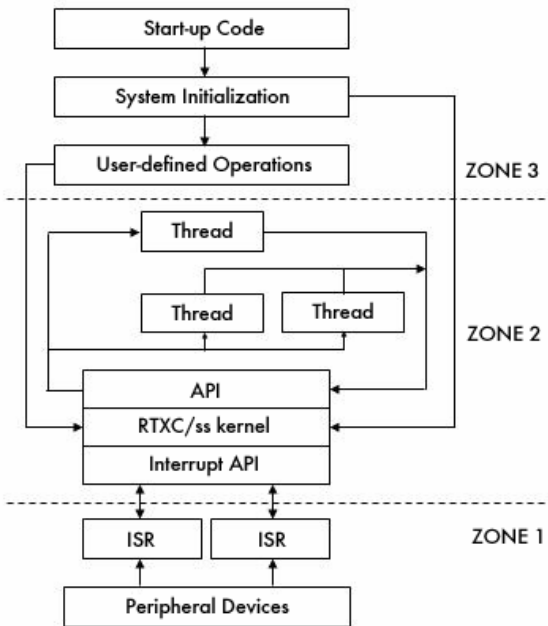


Figure 1. RTXC/ss topology

3.3 RTXC/ms

The flexible design of RTXC/ms supports a wide range of applications, including communications, automotive, process control, and instrumentation systems. It is ideal for systems that require fast interrupt response time and rapid, deterministic switching between tasks. RTXC/ms also provides a robust set of services for intertask communication and event synchronization, combined with RAM management and efficient execution, permitting it to meet the requirements for most real-time systems designs.

RTXC/ms supports 11 kernel object classes, each with a set of kernel services. The object classes include:

- o Tasks
- o Semaphores
- o Queues
- o Mailboxes and Messages
- o Memory Partitions
- o Mutexes (Exclusive Access Semaphores)
- o Pipes
- o Event Sources
- o Counters
- o Alarms
- o Exceptions

Figure 2 shows a system topology using RTXC/ms only. RTXC/ms uses the same three zones of processing as RTXC/ss. Zone 1 is reserved for interrupt servicing. Zone 2 is limited to RTXC/ms kernel services. Task operations under RTXC/ms occur in Zone 3 when there is no activity in either Zone 1 or Zone 2. Zone 3 operations may cause activity to occur at Zone 2.

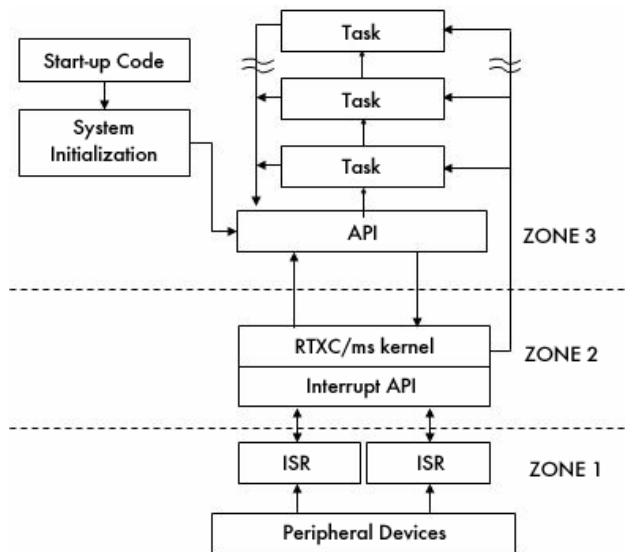


Figure 2. RTXC/ms topology

3.4 RTXC/dm

Figure 3 depicts RTXC/dm a dual-mode design using both RTXC/ss and RTXC/ms.

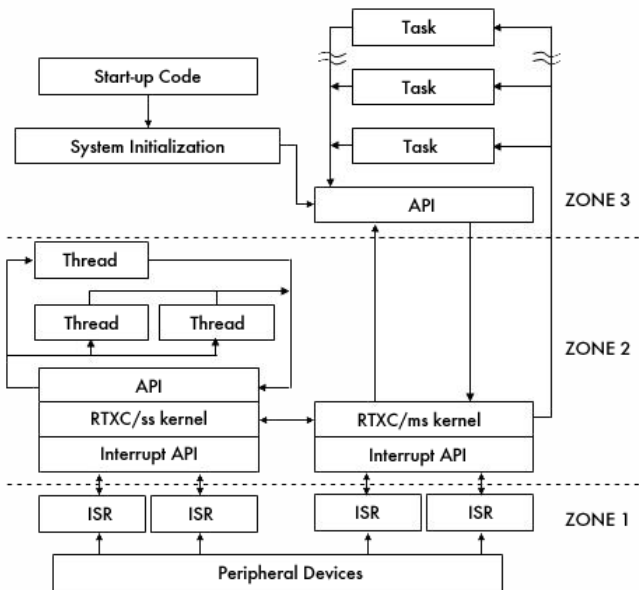


Figure 3. RTXC/dm topology

4 System Configuration with RTXCgen

Accurate configuration of the system is a vital part of any successful application. There are two parts to the configuration process:

- Configuration of the RTOS with respect to the basic kernel configuration, selection of object classes and included kernel services
- Specification of the static objects and their properties and enumeration of dynamic objects as needed by the application

RTXCgen is host-based configuration program with an intuitive graphical interface that assists the developer in system configuration. It gives the user an easy way to specify the RTOS framework and included Object Classes with point-and-click editing. It also allows the developer to define the application properties of static objects.

The graphical presentation makes it easy to enter the configuration data and to view the entered data in context with other entries. Along with its intuitive interface, RTXCgen uses color-coding to indicate the status of each item at a glance—whether it has been changed since the last save or since the last code was generated.

Once all configuration parameters are defined, the RTXCgen tool automatically generates error-free C source code, representing the selected RTOS configuration and object specifications. The developer includes these source code modules (.c and .h files) when compiling RTXC source code and application source code modules.

5 System Components

RTXC-based applications contain two distinct spaces that contain components of the application and system. These spaces are called User Space and System Space.

5.1 User Space Components

User Space components are those elements which lie outside the direct control of or use by RTXC and include at least the following:

- User Startup Code
- System Initialization Code (Zone 3 operation including *main()*)
- Thread and Task Code
- RTXC API Library
- User Workspace
- Configuration Tables (from RTXCgen)

5.1.1 User Startup Code

Startup code is the code that executes whenever the system is RESET. It contains the sequence of instructions necessary to set up the CPU and specific hardware to the configuration required by the application. It is usually written in assembly language, although there is no requirement that it must be so. Chip selects, module initialization, diagnostics, and so on, are some of the functions the developer might put into this body of code. In general, startup code is independent of the RTXC configuration being used.

5.1.2 System Initialization Code

In an application written in some higher order language than assembler, the startup code usually ends by calling the mainline routine that begins operation in the realm of the compiled code. In C, for example, the mainline function has the name *main()* which is, in actuality, a function with several major sub-functions. The operation of *main()* includes initializing RTXC kernel objects and System Space as well as initializing any application specific data. In systems using RTXC/ss only, this code may also include the background operations that execute whenever there is no thread operation in Zone 2. In an application based on RTXC/ms, the initialization code normally becomes the Null Task (also called the Idle Task) upon completion of the initialization operations. The major sub-functions of this code are the application tasks.

5.1.3 Thread or Task Code

Depending on which RTXC Quadros framework the user chooses, the embodiment of the application is in the thread or task code components. These code modules are very application specific and are solely the responsibility of the developer. To achieve the desired behavior of threads and tasks, the code can make use of RTXC services by making requests in the form of function calls to various RTXC API Library services.

5.1.4 RTXC API Library

The RTXC API Library contains the functions that constitute the interface between the application code and the RTXC Kernel Services. RTXC/ss threads make direct calls to Kernel Services while an RTXC/ms task makes calls to API functions that form the proper calling environment for the Kernel Services and then invokes the RTXC kernel. The RTXC Software Developer's Kit contains the RTXC API Library.

5.1.5 User Workspace

The User Workspace is that part of RAM that is normally managed and used by the application code outside the direct control of RTXC. However, some RTXC kernel objects have portions that make use of User Workspace to minimize System RAM requirements. Objects in this category make use of RAM areas of a user-defined size such as that found in task stacks, queues, memory partitions, and pipes. The developer should never allow User Workspace and RTXC Workspace to overlap.

5.1.6 Configuration Tables

The Configuration Tables are produced by RTXCgen and contain the properties for the system, the various object classes, and even the individual static objects. The developer compiles or assembles and links them with the object code of the application tasks. During a startup, System Initialization code uses the Configuration Tables to initialize the selected kernel object classes and the various static objects in System RAM. The use of static objects eliminates the need to write application code to create them, thereby removing a potential source of error.

5.2 System Space Components

System Space contains several elements, most of which may be found in either ROM or RAM. One component, System Workspace, must be in RAM. These components are either a standard part of the RTXC distribution or are produced by RTXCgen during the system generation procedure. The primary System Space components are:

- o RTXC Module
- o System Workspace

5.2.1 RTXC Module

The RTXC Module contains the RTXC Initialization, Kernel Service Library, and whatever other functions may be part of System Space. The RTXC Module can be located in either RAM or ROM, its address being assigned during link/locate operations. The application accesses the RTXC Module contents through API function calls.

5.2.2 System Workspace

System Workspace must be in RAM because it contains all of the data needed by RTXC to control the various kernel objects. RTXC configures System Workspace during the system initialization operations and manages it during system operation. It is inaccessible to the developer.

5.3 Host Environment

The host environment is the development workstation used for program development. Source code editing, linking, system configuration (RTXCgen) and debugging are all performed in the host environment. RTXC Quadros supports Windows 98/2000/NT/XP and Solaris hosts.

5.3.1 Programming Language

Real-time system developers need to be able to write application code in the language that is best able to meet system performance goals. RTXC Quadros permits the user to develop applications using C or C++. It is also permissible to mix languages.

5.3.2 Development Tools

The code development process typically employs compilers, assemblers and debuggers. These are commonly provided in an integrated development environment (IDE). Each distribution of RTXC Quadros is ported to a specific processor and bound to a specific compiler. RTXC Quadros provides bindings for many popular code development systems.

6 Object Class Overview

RTXC Quadros operates on a framework of data structure templates called *object classes*. From these, a developer creates specific objects that implement the application architecture. Each object class has properties unique to its class while supporting four primary and two secondary (or optional) functions of an RTOS:

- Primary
 - Management of program code execution
 - Synchronization with events
 - Passing data
 - Managing time- or event-based alarms
- Secondary
 - Management of memory (RAM)
 - Exclusive access

Each object instantiated within a class has two kinds of properties—those it inherits from the class and those unique to its usage in the application.

RTXC Quadros supports several different kernel object classes, each with a set of kernel services.

- *Levels, Threads* and *Tasks* exist to manage the execution of program code.
- *Semaphores* allow tasks to synchronize with events.
- *Queues, Mailboxes* and *Messages*, and *Pipes* support the passing of data between producers and consumers.
- *Event Sources, Counters*, and *Alarms* are the basis for managing time or other types of “tick-based” inputs.
- *Memory Partitions* permit a user to manage RAM without fragmentation.

- *Mutexes* serve to ensure that exclusive access to an entity is possible.
- *Exceptions* permit programmatic attachment of interrupt vectors to interrupt servicing routines.

RTXC Quadros objects are either statically or dynamically defined. Static definition takes place before program execution while dynamic definition occurs at runtime. RTXC Quadros supports applications that use static objects only, dynamic objects only or a combination of the two. Dynamic objects are the key requisite for being able to download applications to a running system and having the new code operate without causing a system upset.

6.1 Levels and Threads

A *Thread* is a code module that is granted control of the CPU by the RTXC/ss Scheduler. Each thread exists within a level that defines the thread’s priority. The primary use of threads is to provide a low-latency means of executing code related to external interrupts or high frequency processes. The code model of a thread is that of a function, which the RTXC/ss Scheduler calls, passing it two arguments: a user-defined parameter and the pointer to the thread’s environment arguments, if any. The thread has no context upon entry and must perform any required data initialization upon entry. When its operations are complete, the thread returns to the RTXC/ss Scheduler without context.

A thread has a two-dimensional execution priority defined as a combination of its priority level and an order number within the level. Therefore, threads may exist at multiple levels of priority and there may be multiple threads at a given priority level. Once defined, a thread’s scheduling order within the priority is fixed and cannot be changed programmatically.

When a thread begins execution, it is in control of the CPU and must run to completion. This rule is partly responsible for the high performance nature of threads because RTXC/ss does not save or restore any context when exiting or entering the thread.

While the design of a thread must permit it to run from its entry point to a point of completion, RTXC/ss supports multiple priority levels of threads, permitting pre-emptive scheduling of threads. Thus, a thread scheduled to execute at a higher priority level may pre-empt a thread currently running at a lower priority level. This feature permits the RTXC/ss framework to be employed in applications needing deterministic scheduling.

When thread preemption occurs, RTXC/ss saves the context of the preempted thread before starting the preempting thread at the higher priority level. When it is time for RTXC/ss to return processor control to the preempted thread, it restores the context of the preempted thread before resuming it.

6.1.1 Thread Scheduling

The RTXC/ss Scheduler schedules another thread for execution only at a point of rescheduling, that is, when the current thread's operation is complete. Within a given priority level, the RTXC/ss Scheduler supports two different methods of scheduling threads:

- Priority
- Round Robin

1.1.1.1 Priority Scheduling

In Priority Scheduling, threads at a given level are scheduled according to their order number. At each point of rescheduling, the Scheduler selects the thread with the highest order number from among all the ready threads in the current priority level. Even though other threads with lower order numbers may have been ready for a longer time, the Scheduler activates the ready thread having the highest order number.

1.1.1.2 Round Robin Scheduling

Round Robin scheduling activates threads in descending order within the current level at each point of scheduling. There is no priority implied by the thread's order number within the level.

6.1.2 Thread Gates

A configurable property of the thread class is that of a *gate*. If the property exists within the class, then each thread has a gate that can be useful in scheduling the thread. The gate is simply a datum associated with the thread that allows an interrupt service routine (ISR), another thread, or possibly a Zone 3 task to perform an operation on the gate. Permissible thread gate operations are incrementing, decrementing, setting bits (logical OR), and clearing bits (logical NAND).

When the gate is *opened* by one of these actions, RTXC/ss schedules the associated thread to run. Operationally, when a thread's gate is the object receiving an increment or a logical OR operation, the content of the gate becomes non-zero and the associated thread is scheduled. Conversely, when a gate is decremented or has one or more bits cleared by a logical NAND operation such that the content of the gate becomes zero, the associated thread is scheduled.

These operations on the thread's gate give rise to many combinations of behavior such as waiting to schedule a thread until certain events, each associated with a bit in the gate, have occurred. Services exist that permit a thread to read its gate to determine how and why it was scheduled.

6.1.3 Environment Arguments

Environments are a user-definable property of the thread class that permits variables to be stored in a structure associated with the thread without occupying any stack space. A thread can store and retrieve data into Environment Arguments structure as a means of preserving data between execution cycles such as a state variable, a buffer pointer or a scalar datum.

The RTXC/ss Scheduler passes the address of the thread's environment argument block to the thread each time it activates the thread. The block address serves as a base pointer through which the thread can easily access elements of the environment argument block.

6.2 Tasks

Tasks contain the code that implements an application running in Zone 3 under the RTXC/ms Scheduler. Tasks exist independently, each having a program stack and context. After the application tasks are defined, the user develops the task program code. Tasks are coded as functions but receive no arguments from the RTXC/ms Scheduler and return no arguments. During operation, the RTXC/ms Scheduler maintains the status of all tasks and activates them according to a defined scheduling policy.

6.2.1 Task States

A task is always in one of two states: *ready* or *blocked*. A ready task is said to be ready to execute when there are no impediments to its execution other than gaining control of the CPU. A ready task exists in a list of other tasks, according to its priority with respect to other tasks in the same state. The Scheduler maintains the list of ready tasks in a descending order of priority. The most important ready task is always at the head of the list.

A blocked task is one that is not capable of gaining control of the CPU for reasons that may include its need to synchronize with an event, the need for some resource to become available, or by its own action. While a task is blocked, it consumes no CPU cycles.

6.2.2 Task Scheduling

The RTXC/ms Scheduler determines when a task gains control of the CPU. The Scheduler supports three scheduling methods that may be mixed in whatever combination the developer requires. RTXC/ms supports task scheduling using any of three different policies:

- Preemptive
- Round Robin
- Time-Sliced

1.1.1.3 Preemptive Scheduling

Preemptive scheduling is the default scheduling policy in RTXC/ms as it is quite suitable for most embedded applications. Preemptive scheduling in conjunction with task priorities provides the basis for a responsive, deterministic system design. Preemptive scheduling simply means that the highest priority *ready* task in RTXC/ms is the task that receives control of the CPU

1.1.1.4 Round Robin Scheduling

Round Robin scheduling is a polling protocol in which each task executes until it explicitly blocks or voluntarily yields control of the CPU to another task at the same priority.

1.1.1.5 Tick-Sliced Scheduling

RTXC Quadros supports the concept of *tick-sliced* scheduling, which allows for any tick-based event, such as time, to be the independent variable that governs how long a task gets to run. This gives the developer additional application flexibility since he could utilize either a tick based on a periodic tick (time) or an aperiodic tick such as flow rate or volume through a meter.

Tick-sliced scheduling is a variant of round robin scheduling and is similar in every respect except that a tick-sliced task can only execute for a predefined quantum of ticks. A tick-sliced task remains in control of the CPU until its tick quantum expires or until it yields control or blocks. If the tick quantum expires, the Scheduler forces the task to yield CPU control temporarily to another task at the same priority.

6.3 Semaphores

Semaphores serve as a means to synchronize tasks with internal or external events. RTXC/ms uses a counting semaphore model that contains information about an event and the tasks waiting for it.

RTXC/ms also permits a task to wait on events associated with a set of semaphores, using a logical *OR* condition for task resumption. By this technique, a task can monitor many events in a deterministic manner without the need for cycle-wasting polling schemes. When the RTXC/ms kernel identifies a signal to a semaphore in such a set, it resumes the waiting task and provides the identity of the signaled semaphore.

6.4 Queues

Queues allow an object to support data passing in a FIFO manner, preserving its chronological nature. Within a given queue, all data packets are of the same size. Services for the Queue object class copy data from a source buffer into the queue and from the queue into a destination buffer. Queues are thus circular, with RTXC/ms maintaining their status and ensuring proper operation on EMPTY and FULL conditions.

RTXC/ms also permits semaphores to be associated with certain transitional conditions of a queue. In normal operation, there is no need for these semaphores as the normal queue services provide the necessary synchronization. However, it is sometimes advantageous to use queue event semaphores for special synchronization purposes. Almost without exception, queue semaphores are used in connection with RTXC/ms services that operate on a set of semaphores.

6.5 Mailboxes and Messages

A Mailbox is an exchange point for messages sent by a sending task to a receiving task. Only the sender and the receiver tasks know the form and content of the message. Message services do not copy the data into a mailbox. Instead, they move the address of the message. The result is a clean, efficient means for passing data of variable size.

Options exist for passing data in a FIFO manner or in one of two priorities, Urgent or Normal. The receiver task gets Urgent priority messages before those of Normal priority.

A task can send Messages synchronously or asynchronously. For the former, the sending task sends the message and then waits for the receiver to acknowledge its receipt, after which the sender can proceed. It is possible to associate an alarm with the receipt of the acknowledgement so that the waiting condition can be limited.

For asynchronous message transfers, the sending task sends the message but continues without waiting for an acknowledgement. The task can choose to wait at a later time for an acknowledgement.

6.6 Pipes

Pipes are data passing mechanisms in RTXC Quadros that permit rapid, unidirectional transfers from a single producer to a single consumer. The producer or the consumer may be an ISR or a thread. Pipes are ideally suited to communications applications where data is acquired and buffered at a high frequency and then must be processed rapidly while the next block is being acquired.

Functionally, pipes are buffers of data that are filled by the producer and processed by the consumer. The RTXC Quadros pipe services perform the necessary functions of managing the buffers in the pipe while the actual reading and writing of the data in the pipe buffers is the responsibility of the consumer and producer, respectively. In operation, the producer allocates a free buffer from the pipe, fills it with data, puts it back into the pipe. The consumer gets the full buffer, processes the data in it, and then frees the empty buffer back to the pipe where it can be re-used.

Normally, pipe buffers are allocated sequentially and processed in the same order. However, RTXC/ss also permits the producer to put a full buffer at the front of the pipe instead of at the tail, causing the consumer to process it next.

Pipes can also be associated with thread gates in such a manner that certain pipe operations can result in operations on associated thread gates. Such a capability permits close synchronization between pipe producers and consumers and minimizes the amount of RAM required for pipe buffers.

6.7 Event Sources

Event Sources are objects that count recurring events, whether synchronous or asynchronous. Such events are referred to as *ticks* and have no data content other than their sequencing with respect to other ticks of the same type. A tick can occur for an external event, usually an interrupt, or an internal event associated with the application. For example, an event source may be the interrupt of a periodic timer used for the system time base or an interrupt from a shaft encoder on an axle indicating angular position.

Event Sources count the ticks as they occur in a free-running accumulator. Event Sources also serve as the parent class of Counters and thus, associate counters with the ticks.

6.8 Counters

Counters increment an associated accumulator each time the counter receives a user-defined number of event ticks from its parent event source. Counters receive event ticks at the same rate as they occur at the event source but are *divided down* through a user-defined count modulus specific to the counter. When the counter receives the modulus number of ticks from the source, it adds one counter tick to its accumulator.

In addition to counting, counters also serve as the parent class of Alarms and provide the basis for permitting alarms to exist in RTXC Quadros. Whenever a counter increments its accumulator for a counter tick, the alarms associated with a given counter are also examined.

6.9 Alarms

Alarms provide the ability to cause a desired action to occur when the alarm expires. The basis of the alarm may be time or some other system-specific parameter such as angular position. Because alarms are associated with counters, they necessarily take on dimensions of the associated counter. If the counter accumulates time, then the alarm expiry is measured in the same units of time.

RTXC Quadros permits different types of actions to occur upon expiration of an alarm. In RTXC/ss stand-alone mode, these actions affect threads or could possibly be treated as an internal event associated with a source object. In RTXC/ms, the user may associate a semaphore with the expiration of a given alarm so that it can be used in a set of semaphores on which a task waits for one of the associated events to occur.

6.10 Memory Partitions

A Memory Partition is a collection of RAM blocks that permits the allocation and freeing of RAM without the possibility of fragmentation, a common problem in some embedded applications. RTXC/ms supports the existence multiple memory partitions. Different memory partitions can have different block sizes but all the blocks within a given memory partition must be the same size. Services exist to allocate and free a block of memory. The RTXC/ms kernel handles the EMPTY condition automatically.

6.11 Mutexes

It is quite common in a multitasking system to have different tasks use a common entity or resource. Because an event driven design permits the preemption of a task by one of higher priority, it is necessary to provide a mechanism for preventing uncontrolled access to such a common resource. RTXC/ms provides such a mechanism in an exclusive access semaphore called a *mutex*. By proper use of the services for this class, the developer can guarantee exclusive use of the associated resource at any one time.

The use of mutexes in a preemptive scheduling, multitasking system naturally brings about the possibility that a higher priority task may preempt a lower priority task that currently owns a resource the preempting task wants to use. Such a situation is commonly referred to as a *priority inversion* because the lower priority task is holding up the execution of the higher one—an obvious twisting of the scheduling rules.

When a priority inversion situation is likely for a given mutex, the developer may selectively choose to invoke a procedure known as *priority inheritance* to correct the problem. In this method, RTXC/ms raises the priority of the task owning the resource to that of the higher priority task waiting to use it. When the owner of the mutex completes the required use of the resource, it releases ownership of the mutex and the kernel restores the task to its original priority while granting ownership of the mutex to the higher priority task that was waiting for it.

6.12 Exceptions

RTXC Quadros also provides for a generalized interrupt service scheme. Because ISR code is specific to both the particular device and the method of use in the application, the developer must provide it. See section 7 for more information on writing ISRs.

7 Interrupt Servicing

RTXC Quadros provides a generalized interrupt service scheme for which the rules are quite simple. While the hardware specifics of interrupt recognition and acknowledgment vary from CPU to CPU, software handling of interrupts is more consistent. In RTXC Quadros, there are three basic parts to an ISR:

- Prologue
- Device servicing
- Epilogue

The prologue begins the processing of the interrupt. The device servicing section deals with the particular device. The epilogue is the end action performed to finish interrupt processing and continue with the application.

RTXC Quadros supports a concept of *Normal*, *Private* and *Hybrid* interrupt service routines.

7.1 Normal Interrupts

Normal interrupts are those that make use of the RTXC ISR Macro, which processes the interrupt request according to the three-steps above. The prologue saves the interrupted process context and calls the device-servicing function that deals with the device and announces the event to the kernel as it ends its operation. When the device-servicing function completes, it returns to the epilogue, which then resumes the interrupted process. A device servicing function has access to a set of RTXC kernel services specifically intended for ISR usage.

7.2 Private Interrupts

For users who want to exercise complete control over the servicing of interrupts, a Private interrupt service routine may be desired. Private ISRs do not use the standard RTXC ISR Macro processing sequence, thus they may be used where speed of service or latency is of utmost importance. A Private ISR must exist outside of the RTXC Quadros application and there can be no connection between the two, leading to the rule that the user must write all elements of the Private ISR – prologue, device-servicing and epilogue.

7.3 Hybrid Interrupts

RTXC Quadros also supports a third method for treating interrupts – the Hybrid interrupt service routine. The Hybrid ISR is a combination of the features of the Private and the Normal ISR servicing methods. It allows the user to process the interrupt with very tailored code for higher performance than in possible with Normal interrupts but also to connect the interrupt servicing to the application so its occurrence can be announced through the use of the RTXC Kernel Services.

8 More Information

For more information on RTXC Quadros, supported processors and other QSI products, including an embedded file system, an embedded database and embedded networking stacks, please contact Quadros Systems at sales@quadros.com or call your local sales representative.