RTXC Kernel User's Guide, Volume 1

Levels, Threads, Exceptions, Pipes, Event Sources, Counters, and Alarms

Disclaimer

Quadros Systems, Inc. makes no representations or warranties with respect to the contents or use of this manual, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Quadros Systems, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Quadros Systems, Inc. makes no representations or warranties with respect to any Quadros software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Quadros Systems, Inc. reserves the right to make changes to any and all parts of Quadros software, at any time, without any obligation to notify any person or entity of such changes.

Trademarks

Quadros is a registered trademark of Quadros Systems, Inc. **RTXC**, **RTXC Quadros**, and **RTXC DSP** are trademarks of Quadros Systems, Inc.

Other product and company names mentioned in this document may be the trademarks or registered trademarks of their respective owners.

Quadros Systems, Inc.
10450 Stancliff, Suite 110
Houston, TX 77099-4336
USA

RTXC Kernel User's Guide, Volume 1
Part Number: RTXC-UGV1-0602
June 2002
RTXC Kernel, Version 1.0

Contents

List of Examples

List of Figures

CHAPTER 1

Introduction

In This Chapter

We introduce the **RTXC/ss** component of the **RTXC** RTOS and describe the contents of this book.

The RTXC Kernel

The **RTXC** Kernel is the heart of the **RTXC** RTOS, a multitasking real-time operating system (RTOS) for the development of embedded applications. It comes in two variants, the **RTXC DSP** Kernel with support for using digital signal processors (DSP), and the **RTXC Quadros** Kernel for non-DSP processors.

The **RTXC** Kernel provides a software framework for real-world, real-time systems, consisting of two major components, **RTXC/ss** and **RTXC/ms**, each of which can schedule the use of the CPU according to the demands of the application. The **RTXC/ss** component features a single stack model with a low-latency thread scheduler and a small footprint, making it ideally suited for applications requiring high frequency interrupt processing, such as in digital signal processing. The **RTXC/ms** component provides a multiple independent stack model for a fully pre-emptive multitasking scheduler with a rich set of kernel services well suited to deterministic, hard real-time system requirements.

The **RTXC** Kernel is highly scalable in that the user may select the basic framework of either the **RTXC/ss** component or the **RTXC/ms** component alone, or both components combined. It is further scalable by the selection of various properties within each kernel object class and the services that operate on those classes. The **RTXC** RTOS includes a configuration utility program, **RTXCgen**, to assist the user in configuring the kernel with the set of resources and features most suitable to the needs of the application.

The **RTXC** RTOS permits the user to develop applications using assembly language, C, or C++. Each distribution of the **RTXC** RTOS is ported to a specific processor and bound to the application source language, making access to kernel services convenient for the developer. The **RTXC** Kernel has an implementation history dating from 1978, and thus provides a sound foundation for development of software over a broad range of real-time applications.

The **RTXC** Kernel consists of a library of functions that provide a rule-based architecture for the design and implementation of embedded real-time systems. A set of object classes and the kernel services that operate on them are the embodiment of the architecture of the **RTXC**

Kernel. Users gain access to the RTOS by calling various kernel services through a comprehensive application program interface (API) to achieve desired system behavior. This API library uses kernel service names that help make the product easy to learn and easy to use. The programmer can spend less time dealing with system matters and more time on developing the application.

The **RTXC** Kernel software should be used as any other software library. You do not need to know *how* the **RTXC** Kernel operates internally. Rather, you need to know only *which* **RTXC** Kernel service to use to achieve a desired result. Thus, the **RTXC** Kernel becomes much like a large-scale integrated circuit hardware component.

Users of the **RTXC** RTOS have access to this book and the other volumes of the **RTXC** Library to learn more about the product and to resolve technical issues.

RTXC Kernel Features

The **RTXC** Kernel features support real-time, multitasking applications using either **RTXC/ss** or **RTXC/ms**, or a combination of **RTXC/ss** and **RTXC/ms**. General features of the **RTXC** Kernel include:

- Three levels of code and data scalability for optimized configurations:
  - Class
  - Class Properties
  - Kernel Services
- Standard programmer interface in C language on all processors

RTXC/ss Features

The **RTXC/ss** component supports the following features:

- Multi-thread processing with selectable scheduling methods:
  - Preemptive between Levels
  - Priority within the same Level
  - Round robin
- Static kernel objects:
  - Levels and Threads
  - Pipes
  - Event Sources, Counters, and Alarms
  - Exceptions
- Multiple thread priority levels
- Fixed thread priorities within a level
- No context saved or restored for threads operating at same level
- Pre-emptive scheduling of threads between levels
- Single stack for all operations
- Low latency for fast processes
- Small RAM and ROM usage

How to Use This Book

Note: The *RTXC Kernel User's Guide, Volume 1* contains information needed by users of both the Single Stack and the Dual Mode configurations of the **RTXC** Kernel. If you purchase the Single Stack configuration of the **RTXC** Kernel, you receive only Volume 1 of this book, and you can ignore references in this text to the **RTXC/ms** Kernel component.

If you purchase the Dual Mode configuration, you receive both Volume 1 and Volume 2.

The *RTXC Kernel User's Guide, Volume 1* assumes the reader has fundamental knowledge about multitasking real-time kernels and expands on that knowledge by explaining the inputs and outputs of the **RTXC** Kernel as a software component of an embedded application. This book focuses on **RTXC/ss**, the Single Stack component of the **RTXC** Kernel, and includes the following chapters and appendixes:

Chapter 1, "Introduction," describes the contents of the volume.

Chapter 2, "Levels and Threads—Meeting Functional Requirements," discusses how the **RTXC/ss** component of the **RTXC** Kernel uses levels and threads to meet the functional requirements of the application.

Chapter 3, "Exceptions—Claiming Interrupt Vectors," discusses how the **RTXC/ss** component of the **RTXC** Kernel uses exceptions to prepare for servicing interrupts.

Chapter 4, "Pipes—Buffered Data Movement," discusses how the **RTXC/ss** component of the **RTXC** Kernel uses pipes to move data between threads.

Chapter 5, "Event Sources, Counters, and Alarms—Keeping Track of Events," discusses how the **RTXC/ss** component of the **RTXC** Kernel uses event sources, counters, and alarms to manage time- and tick-based operations of the application.

CHAPTER 2

Levels and Threads—Meeting Functional Requirements

In This Chapter

We discuss how the **RTXC/ss** component uses levels and threads to meet the functional requirements of the application. We present level and thread concepts, organizations, and properties. Then we expand on the policies and present methods of thread scheduling. Finally, we present a functional overview of the thread management capabilities of the **RTXC/ss** component.

Introducing Levels

Levels are a special class within the design of the **RTXC/ss** component. Levels have properties but no associated kernel services other than for initialization of the class properties. The purpose of the Levels class is to organize the operation of its child class, Threads. The architecture of **RTXC/ss** component is based on a design philosophy that uses a single stack, permitting a low latency code execution model based on threads. Threads are more completely defined later in this chapter.

Levels represent the priorities at which threads execute. An application based on the **RTXC/ss** component can employ one or more levels at which to execute threads. Execution of threads is based upon a scheduling policy using the priority of each level. Threads associated with levels that have a high priority execute before threads at lower priority levels. Within a level, all threads operate at a fixed priority and cannot preempt one another. However, a thread operating at one level can preempt another thread operating at a lower priority level.

The following rule applies:

Rule: In a system using the RTXC/ss component, there must be at least one level.

Level Definition

During system generation for a system configuration using the **RTXC/ss** component, you may define from one to 16 total levels.

A level handle must be within the range of the total number of levels defined for the application. Where a reference to a level is applicable, you may refer to it by its handle, which is a LEVEL type datum. There are two rules applicable to level definition:

Rule: All levels must be statically defined.

Rule: The RTXC Kernel **does not support dynamic levels.**

Level Organization

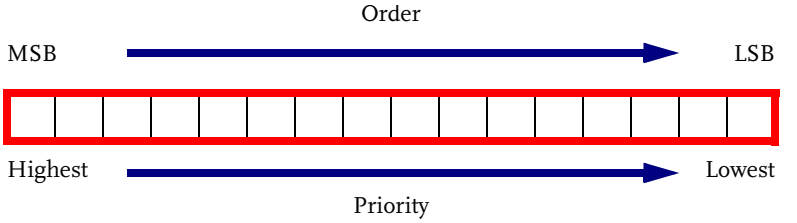
A level consists of two main parts: its data structure, called a Level Control Block (LCB), and a set of pointers to its associated threads. The level's handle is the LCB's index in the LCB array. The **RTXC/ss** Scheduler uses the information in the LCB to control the execution of threads associated with that level. The key element in the LCB is the Ready Table.

Ready Table

The Ready Table is a single datum that contains one bit for each thread associated with the level. A thread becomes ready to receive control of the CPU when application code, an interrupt handling routine, another thread, or a task, if the **RTXC/ms** component is present, schedules its execution. In scheduling the thread, the kernel sets a bit in the Ready Table corresponding to the thread. A thread's bit being set does not necessarily mean that the thread immediately gains control of the CPU. Other conditions must also be present, as explained later in this chapter.

Each bit in the Ready Table has an order number that associates it with a particular thread. The order numbers begin at 1, starting from the most significant bit, as illustrated in Figure 2-1, and represent the priorities of the associated threads. Thus, a thread's execution priority consists of two parts: its level and its order number.

Figure 2-1. Ready Table Layout



The actual size of the datum used for the Ready Table is a function of the processor. Depending on the target processor, it size could be 8, 16, 24, or 32 bits.

Level Properties

Level objects have several properties, all of which you define during the system configuration process using **RTXCgen**. Once set, there are no kernel services available in the **RTXC/ss** component to modify these properties under program control. The **RTXC** Kernel defines a `LEVELPROP` properties structure for use during the initialization process. The members of the structure represent the properties and have the organization shown in Example 2-1.

Example 2-1. Level Properties Structure

```
typedef struct
{
   KATTR attributes;     /* attributes */
   KCOUNT n_static;      /* number of static threads */
   KCOUNT n_dynamic;     /* number of dynamic threads */
} LEVELPROP;
```

Level Attributes

A level has a single value for the *attributes* property denoting the method by which the **RTXC/ss** Scheduler grants control of the CPU to threads whose corresponding bits in the Ready Table are set to 1. The value denotes either *Priority* or *Round_Robin* scheduling. The default value is *Priority* scheduling.

Number of Static Threads

The *n_static* property specifies the number of statically defined threads the user has defined for the priority level. The **RTXCgen** program automatically determines the value of *n_static*. An application using only the **RTXC/ss** component can employ only static threads.

Number of Dynamic Threads

The *n_dynamic* property, which specifies the number of dynamic threads, applies only if the application configuration includes the **RTXC/ss** *and* **RTXC/ms** components. Dynamic threads are not available for use in a system having only the **RTXC/ss** component.

Only Zone 3 operations using **RTXC/ms** component services can create and destroy dynamic threads.

Level Priority

During the system configuration process using **RTXCgen**, the user defines each level in the application and implicitly defines the level's priority. Unlike all other **RTXC** Kernel classes where the object's handle implies no priority, the handle of a level does. The level's handle represents its index in the array of LCBs and the index, a level's position in the hierarchy of levels, defines its priority with respect to thread execution. Levels share the same inverse priority model as Zones: thread and task priorities decrease as the numerical value of the priority increases. Therefore, the first level has the highest priority, the second level has the next lower priority, and so on.

Note: The handle of a level is equal to its priority. The handle of a level is equivalent to its index in the array of LCBs. For example, if three levels exist, their indexes, and therefore their priorities, are 1, 2, and 3, respectively. The level with index 1 has the highest priority. Index 2 is the next highest priority, and so on.

Introducing Threads

In a real-time embedded system, the system designer decomposes the application's functional requirements into a suite of functional entities. In applications based on the **RTXC/ss** component, these entities are called *threads*, a code design and execution technique that features minimum RAM requirements and minimum system overhead. The **RTXC/ss** component provides a simple model for executing threads, which are workhorse program elements. The nature of each thread is, of course, application-dependent, and is left to the imagination of the system designer. Threads implement the design policies concerning management of the application processes and solving the application's functional requirements.

References to the **RTXC** Kernel in this chapter mean any configuration of the **RTXC** Kernel that contains the **RTXC/ss** component.

Thread Definition

Each thread is specific to a given level. During system generation, you may define, for each specified level, the threads that execute at the priority represented by that level. You may define only static threads, the number of which must be less than or equal to the size (in bits) of the Ready Table.

Because the size of the Ready Table governs the maximum number of threads at a given level, **RTXCgen** associates each thread with a bit in the Ready Table for its level. Therefore, the assignment of bits refers to the thread's order, as depicted in Figure 2-1 on page 10, proceeding from the MSB to the LSB of the Ready Table.

A thread handle must be within the range of the total number of threads defined for all levels in the application. The application program code refers to a thread by its handle, which is a `THREAD` type datum. A thread handle of zero (0) is legal in a kernel service for the Thread object class and is a shorthand definition for the *Current Thread*, which is the thread currently in control of the CPU.

Thread Organization

In the **RTXC** Kernel, a thread consists of two parts: its program code and a data structure called a Thread Control Block (ThCB). Each thread requires a ThCB, and it is the ThCB's index in the ThCB array that constitutes the thread's handle. The **RTXC/ss** Scheduler controls the execution of the thread code by managing the data in the ThCB.

The **RTXC** Kernel treats the code for a thread like a function. Consequently, threads should be written as a function called by the **RTXC/ss** Scheduler and returning to it as well. One difference between a **RTXC** thread and a **RTXC** task is that the task never returns to its caller.

There is one code model for **RTXC** threads as shown in Example 2-2 on page 14. It receives two possible arguments in the calling sequence: its argument and a pointer to its environment arguments.

These arguments and the conditions for their use are more fully explained later in this chapter.

After gaining control of the CPU, the thread performs its required operations and when finished, returns to the **RTXC/ss** Scheduler.

Example 2-2. Thread Code Model

```
void threadname (thread argument, environment argument pointer)
{
   ... Data declarations
   ... Thread initialization

   ... Thread operations

   return;
}
```

Thread States

The **RTXC** Kernel maintains a state for each thread in an application. A thread is always in one of the following states:

Ready	The thread is available for execution as evidenced by its order bit in the appropriate level's Ready Table being set to 1.
Not Ready	The thread is not scheduled and is not capable of receiving CPU control.
Running	The thread has CPU control.

During startup, the kernel initializes all threads to the Not Ready state and also receives the definition of each static thread. Later, at the request of an interrupt handling routine, another thread, or a task (if the **RTXC/ms** component is present), the kernel schedules a thread's execution using the `XX_ScheduleThread` or `XX_ScheduleThreadArg` kernel services. At that point, the requested thread's state becomes Ready. In the Ready state, there are no impediments to the thread's execution, other than gaining control of the CPU.

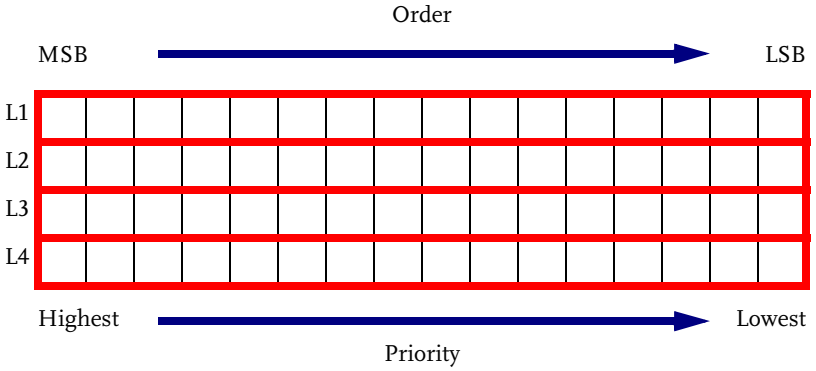
When a thread gains control of the CPU, the **RTXC/ss** Scheduler changes the thread's state to Running. When the thread returns

control of the CPU to the **RTXC/ss** Scheduler, the thread's state becomes Not Ready unless it has been rescheduled by an interrupt handling routine or a thread executing at a higher priority level.

Readying Threads for Execution

The key to running threads in the **RTXC/ss** component is the set of Ready Tables for all defined levels. Together they assume the characteristics of a two-dimensional array where the rows represent levels (priority) and the columns are bits (order) representing the threads assigned to each level. Figure 2-2 depicts such an array using four levels. Level 1 is the highest priority level.

Figure 2-2. Ready Table Array for Four Levels



The Ready Tables, taken as an array, compose an instantaneous, ordered representation of threads that are ready to get control of the CPU. Threads become Ready at varying rates and move into the Ready Tables as they become scheduled. Consequently, the Ready Tables constantly change as the **RTXC/ss** Scheduler gives CPU control to Ready threads while threads at higher priority levels or exception handling routines schedule more threads. The rules regarding levels are:

Rule: The highest priority level that has a thread in a ready state becomes the Current Level.

Rule: The scheduling policy of the level determines which thread receives control of the CPU at the Current Level.

The **RTXC/ss** Scheduler must determine which ready thread is the next one to receive control of the CPU. To do so, it must first determine the highest priority level that has threads in a Ready state. When it determines the level, it selects the thread to run next in accordance with the scheduling policy of the level. For information about thread scheduling policies, see "Thread Scheduling Protocols" on page 21. After selecting the appropriate thread, the **RTXC/ss** Scheduler changes the thread's state to Running and gives it control of the CPU. The rules for thread execution are:

Rule: The Current Thread is the thread in control of the CPU.

Rule: The Current Thread must run to completion.

Rule: The Current Thread cannot suspend its execution and return to the point of suspension.

Rule: The Current Thread must execute using the common stack.

Rule: The Current Thread has no context on entry.

Rule: The Current Thread leaves no context upon exit.

Rule: Threads at a lower priority level will not run until all threads at higher priority levels are in a Not Ready state.

Thread Properties

Each thread in an application serves a defined purpose represented by the thread's code and properties. The Thread object class has a set of properties and individual threads have properties. Together, those properties define the information the **RTXC** Kernel needs to manage threads.

Through **RTXCgen**, the developer defines thread properties for static threads. The caller, another thread or a task, can define a thread's properties through the `XX_DefThreadProp` service, where `XX_` is either `TS_` (Zone 2) or `KS_` (Zone 3). The kernel service passes the values of the thread's properties in a `THREADPROP` structure to define the thread. When a thread retrieves information about its own or another thread's properties, the `XX_GetThreadProp` service returns the information in a `THREADPROP` structure. Example 2-3 on page 17 shows the organization of the `THREADPROP` structure.

Note: The XX_ in the service name indicates that the service has variants for use in more than one operating zone. Depending on the service, XX_ may stand for two or more of the following actual prefixes: IS_ (Zone 1), TS_ (Zone 2), or KS_ (Zone 3).

Example 2-3. Thread Properties Structure

```
typedef struct
{
   KATTR attributes            /* thread attributes /*
   TLEVEL level                /* thread base level */
   TORDER order                /* thread order (priority) */
   void (*threadentry)(void *, void *);  /* entry point */
} THREADPROP;
```

attributes	Contains attributes of the thread.
level	The level at which the thread runs, that is to say, its priority level. The level of a thread as defined in the THREADPROP structure is also called its *base level*.
order	Corresponds to the bit number in the level's Ready Table. If the scheduling policy of the level is *Priority*, the order element of THREADPROP also represents the thread's priority with respect to other threads at the same level. If the level's scheduling policy is *Round_Robin*, there is no priority implied by the value of the order element.
threadentry	A pointer to the thread's entry point. The address can be anywhere in the User Code Space.

The following rule about thread properties applies:

Rule: The definition of a thread's properties may only occur when the thread's state is Not_Ready.

Violating the preceding rule may produce unpredictable results. It is permissible, however, for a thread to read its own properties or modify existing properties through the use of **RTXC** Kernel services.

Optional Properties

The **RTXC** Kernel supports optional properties for the Thread object class. Using the system configuration utility, **RTXCgen**, the user selects the set of optional Thread class properties that suits the needs of the application. Selection of these properties controls how the **RTXC** Kernel configures the Thread Control Block and the code that supports the options. When selected, they permit the developer to make use of the thread properties through the use of specific kernel services.

The optional Thread class properties are:

- Thread Arguments
- Environment Arguments
- Thread Gates

To change the way the **RTXC** Kernel treats threads with respect to these optional properties, the user must use **RTXCgen** to change the optional property selection state and then recompile the **RTXC** code.

Thread Arguments

The Thread Arguments property controls the first argument in a thread's calling sequence. The thread's argument can be a scalar or a pointer as determined by the user. If the optional Thread Arguments property is enabled through **RTXCgen**, calls to either the `XX_DefThreadArg` or `XX_ScheduleThreadArg` kernel services control the value of a thread's argument. If the Thread Arguments property is disabled, the **RTXC/ss** Scheduler always treats the first calling parameter to every thread as a NULL pointer (`((void *)0)`).

If the optional Thread Arguments property is enabled, the **RTXC/ss** Scheduler passes the thread's argument (or a pointer to the thread's argument) as it was defined by the last use of the `XX_DefThreadArg` or `XX_ScheduleThreadArg` kernel service for that thread. Until a code entity invokes either of these services, the **RTXC** Kernel maintains that thread's argument as a NULL pointer.

Example 2-4 shows how a thread receives its thread argument as a pointer to a structure. The example uses data from the structure as input into a procedure. In this example, the thread does not use environment arguments and the calling parameter is ignored.

Example 2-4. Using Thread Arguments

```
#define SELF (THREAD)0

struct muxdata
   {
      int *dataset;   /* pointer to dataset */
      int setsize;    /* amount of data in dataset */
   };

void threadname (struct muxdata *args, (void *)0)
{
... Data declarations

   ... Thread operations
   /* pass pointer to dataset and dataset size to number */
   /* crunching function */

   crunchnumbers (args->dataset, args->setsize);

   return;
}
```

Environment Arguments

In **RTXCgen**, the optional Thread class Environment Arguments property controls the ability of threads to use environment arguments. If the user enables the property through **RTXCgen**, the **RTXC/ss** Scheduler passes the existing value of the thread's environment arguments pointer to the thread whenever the thread gains control of the CPU.

The pointer to the thread's environment arguments is the second parameter in the calling sequence to a thread. The default value of the pointer in the thread's ThCB is a NULL pointer (`(void *)0`) and the **RTXC** Kernel maintains it as such until it is otherwise defined by a call to the `XX_DefThreadEnvArg` kernel service.

Environment arguments exist to permit multiple threads to share a common body of code or as a place for a thread to keep intermediate

results between execution cycles. When threads share a common code body, it is necessary to distinguish one from another. Each thread using the common code can specify and make use of a separate structure that contains information the thread needs to define its runtime environment.

An environment arguments structure can also be of value to the thread that does not share code. Quite often, the thread may need to know the value of some variable created or modified during a previous cycle of the thread, the value of a state variable, or the port identity for some input or output operation. The **RTXC** Kernel imposes no restriction on the form or content of the environment arguments. The **RTXC** Kernel only uses pointers to the structure; therefore, only the defining thread and the using thread know its organization. The following rule applies:

> **Rule: The environment argument structure for a thread can be located anywhere in the User RAM space.**

The **RTXC** Kernel provides the `XX_DefThreadEnvArg` service, where `XX_` is the zonal prefix `TS_` or `KS_`, to define the address of the structure to the object thread. The `XX_GetThreadEnvArg` service, where `XX_` is the zonal prefix `TS_` or `KS_`, returns the address of the structure.

Example 2-5 on page 21 shows how to access members of an environment arguments structure. The example uses the port and channel numbers in the thread's environment argument structure to acquire the channel status. While the channel status is not `IDLE`, the thread does some operations and then terminates when the channel status becomes `IDLE`.

Thread Gates

The **RTXC** Kernel allows the definition of an optional set of values collectively called the *thread gate*. When enabled by the inclusion of this scalable property during system configuration using **RTXCgen**, the ThCB is extended to contain two additional variables: the *Thread Gate* and the *Thread Gate Preset*.

Example 2-5. Accessing Thread Environment Arguments Structure

```
struct myargs
   {
      int port;       /* port number */
      int chnl;       / channel number */
   };

void threadname ((void *)0, struct myargs *args)
{
... Data declarations

   int chnl_stat;             /* channel status /

   ... Thread operations
   /* pointer to environment arguments structure passed in as call */
   /* parameter to the thread. Second call parameter is NULL. */

   while ((chnl_stat = getchnlstat (args->port, args->chnl))
          != IDLE)
   {
      ...do something
   }

   return;
}
```

The purpose of a thread gate is to establish conditions for scheduling a thread. When used, the thread gate must assume a particular value before the **RTXC/ss** Scheduler can make the thread ready. The thread gate preset serves as a value to use in atomically resetting the value of the thread gate as a result of certain operations on the thread gate. Kernel services exist that perform operations on the thread gate and thread gate preset values for program entities in Zones 1, 2 and 3.

For more information about using thread gates, see “Using Thread Gates” on page 28.

Thread Scheduling Protocols

The **RTXC/ss** component accomplishes the policy of multitasking by the method it uses to schedule threads for operation. As previously stated, the **RTXC** basic rules do not enforce any specific thread scheduling protocol. They only state general rules regarding preemption, CPU control, and Current Thread definition.

The thread scheduling methods used by the **RTXC/ss** Scheduler are specific to the level at which a thread runs. During system configuration, you specify a scheduling policy for each level. The **RTXC** Kernel supports the following methods (or protocols) for scheduling threads within an overall multitasking policy:

- Round robin
- Priority

Some general rules apply to thread scheduling regardless of the scheduling policy in use at the Current Level.

Rule: A new thread cannot be granted control of the CPU while another thread at the same level is running.

Rule: Once in control of the CPU, the Current Thread must run to completion

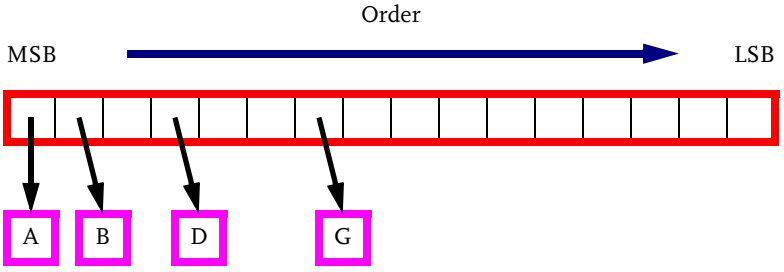
Rule: A thread that becomes ready at a level with higher priority than that of the Current Level preempts the Current Thread and gains control of the CPU, becoming the new Current Thread.

Rule: A preempted thread eventually becomes the Current Thread again and resumes operation at the point where it was preempted.

Round Robin Scheduling

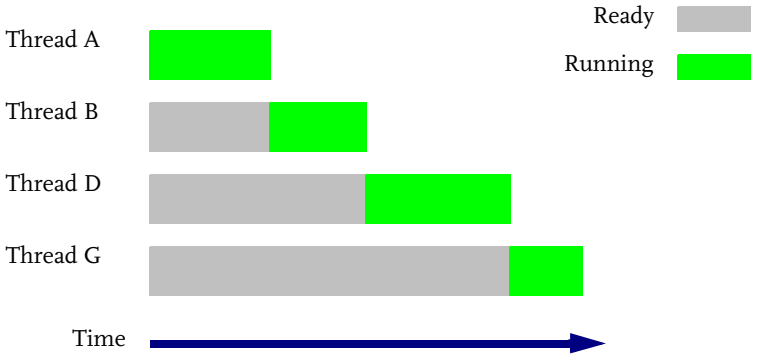
Round robin scheduling is probably the oldest and simplest of the multitasking methods. Threads receive control of the CPU from the **RTXC/ss** Scheduler beginning at the order number of the Current Thread and going to the order number of the next ready thread in the same Ready Table. To illustrate, consider Figure 2-3 on page 23 in which threads A, B, D and G have descending order values and are associated with the bits of the Ready Table pointed to by the blocks containing their names.

Figure 2-3. Thread Order for Scheduling Examples



In the first round robin example, all four threads are ready. The **RTXC/ss** Scheduler gives control of the CPU to Thread A, then B, then D and finally, thread G, assuming no other thread was scheduled in the interim. Figure 2-4 demonstrates this time sequence of events.

Figure 2-4. Round Robin Time Sequence for First Example



But consider a second example where only threads B and G are ready. The **RTXC/ss** Scheduler first gives control to thread B. While thread B is running, an interrupt handling routine schedules thread A, making it ready. Even though thread A has a higher order number, the next thread to get CPU control will be thread G because it is the next ready thread whose order number is lower than the Current Thread, B. Figure 2-5 on page 24 shows the time sequence for the events.

Figure 2-5. Round Robin Time Sequence for Second Example

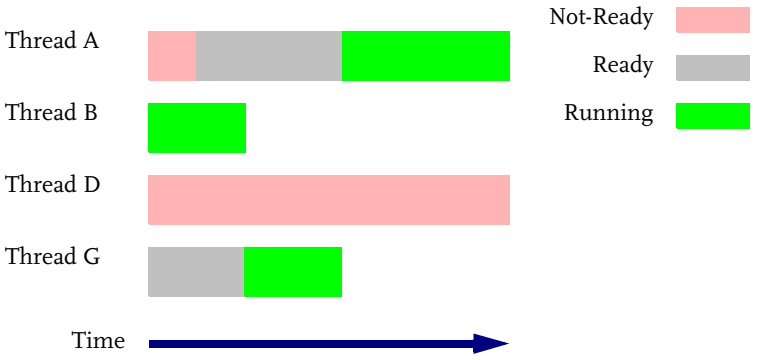


Figure 2-5 also demonstrates what happens when there are no more Ready threads in the Ready Table whose order number follows that of the Current Thread. In the example, no Ready thread follows thread G. However, thread A is Ready; so the **RTXC/ss** Scheduler restarts the round robin at the MSB of the Ready Table, the order number corresponding to thread A.

Priority Scheduling

For a level using the priority scheduling policy, the **RTXC/ss** Scheduler grants control of the CPU to threads within the level according to their order number. The main rule of priority scheduling applies:

> **Rule: Higher order numbered threads run before those of lower order number within the same level.**

Figure 2-6 on page 25 depicts the priority scheduled time sequence of thread executions of the four threads from the second example. Threads A, B, D and G, are being scheduled by interrupt handling routines. The time sequence begins with thread B as the Current Thread.

Figure 2-6. Priority Time Sequence for Second Example

From the time sequence, it is easy to see the application of the general rules of thread scheduling at work. Thread B begins the sequence. Thread A becomes ready shortly after while B is still running. Notice there is no preemption. Also note that immediately after B terminates, thread A becomes Current Thread even though thread G was ready before thread A. Thread A is selected instead of thread G because thread A has a higher order number (priority) than thread G (see Figure 2-3 on page 23) within the priority level. Thread G eventually gets CPU control after A completes. While G is running, thread A and D are scheduled and become ready. Neither thread A nor D can get control of the CPU yet. When thread G completes, thread A runs again. During the execution of thread A, thread B becomes ready. When A completes, thread B runs followed by thread D, even though D has been waiting in a ready state since thread G last executed. And so on.

Thread Contexts

A thread has no context such as that for a task. Because of the nature of the **RTXC/ss** component's single stack design, a thread cannot block or wait for some other process to cause it to continue. Consequently, a thread receives control of the CPU at its entry point

but without any defined processor context. It is the responsibility of the thread to establish on the system stack any local variables it needs during its execution cycle. Any information needed from one cycle to the next is best maintained either globally or in a structure defined as the thread's environment arguments.

On completion of its execution cycle, the thread must clean up any local variables it put on the system stack before it returns to the **RTXC/ss** Scheduler. There is no processor context relative to the thread that is saved upon completion of the thread's execution cycle.

Using Threads

The **RTXC** Kernel provides a large complement of services for use in managing threads. Some of them have been mentioned in previous paragraphs in this chapter. This manual does not try to explain all of them because most of them have functionality that is self-evident. However, the following topics deserve special mention.

Thread Definition

Before a thread can execute, the application must define it to the system along with all of its properties. Use the `XX_DefThreadProp` service to define a thread's properties before using it. The **RTXC** Kernel supports static threads and manages them through a Thread Control Block (ThCB). **RTXC** Kernel services reference a thread by its handle, which is a `THREAD` type datum.

Thread execution must follow this rule:

Rule: A thread can only begin execution at its entry address.

Thread Scheduling

When a process schedules a thread, the thread only has its state changed to Ready by virtue of its order bit set in the Ready Table of its assigned level. The thread does not necessarily begin its execution cycle at that moment. Figure 2-6 on page 25 depicts the difference between a thread being scheduled (its Ready state), and when it receives control of the CPU (its Running state). The thread does not receive control of the CPU until the **RTXC/ss** Scheduler determines

that it has met all conditions necessary to make it the Current Thread according to the scheduling policy for its assigned level.

Using the Thread Argument

After the definition of the thread's properties, an interrupt handling routine, another thread, or a task may use it in a kernel service. The `XX_ScheduleThread` or `XX_ScheduleThreadArg` services make the thread ready to run by setting the bit corresponding to the thread's order number in the Ready Table of its assigned level. The `XX_` prefix for those services represents the zonal prefixes `IS_` (Zone 1), `TS_` (Zone 2), or `KS_` (Zone 3). The difference between the two services is the use of the thread's optional Argument property. That property is defined as a `void *` so that it can be a scalar datum or a pointer. The Argument property allows the scheduling program to pass data easily to the thread.

The value passed by the `XX_ScheduleThreadArg` service is maintained in the thread's ThCB until such time as the thread becomes the Current Thread. At that time, the **RTXC/ss** Scheduler passes the value to the thread as the first of two arguments in accordance with the prototype for a thread.

Warning: If another `XX_ScheduleThreadArg` service executes before the thread receives CPU control as the result of a previous `XX_ScheduleThreadArg` request, the value of the first argument is overwritten. This condition may lead to unpredictable results.

In some applications, it is not necessary to schedule a thread with an argument each time the thread needs to execute. If so, the designer has two choices. One is to use no argument at all. The second is to define the argument one time using the `XX_DefThreadArg` kernel service. That service causes the defined argument to be maintained in the thread's ThCB. From there, the **RTXC/ss** Scheduler uses the argument repeatedly when making the thread the Current Thread, until the argument needs to be redefined. In both cases where the argument is not defined at the time of scheduling, the designer uses the `XX_ScheduleThread` kernel

service to make the thread ready. The **RTXC/ss** Scheduler passes the thread the argument as previously defined, or a NULL pointer as the case may be, when the thread receives control of the CPU.

Using Thread Environment Arguments

When the **RTXC/ss** Scheduler passes control to a thread, the second parameter in the calling sequence is a pointer to the thread's environment arguments or a NULL pointer. The thread's environment arguments is a structure containing information that this invocation of the thread needs to use as it executes. Typically, use of environment structures accompanies the use of shared code entities, with the values of the various elements providing the information about the specific environment of the thread. Example 2-5 on page 21 shows an example of how to use environment arguments for a thread.

Using Thread Gates

Thread gates permit the designer to achieve very sophisticated control over the scheduling of a thread. The thread gate is simply a numeric, unsigned value that the thread, as well as other processes, can operate on to achieve a desired behavior of the thread. There are four ways in which to operate on a thread gate:

1. If the value of the thread gate is initially zero, the `XX_ORThreadGateBits` service logically ORs one or more bits into the thread gate, making the value greater than zero and simultaneously causing the thread state to become Ready. The value of the thread gate retains the result of the operation, allowing the thread code to read it and interpret the meaning assigned to the bits that were set.
2. If the value of the thread gate is initially zero, the `XX_IncrThreadGate` kernel service increments the thread gate, making the value greater than zero and simultaneously causing the thread state to become Ready. The value of the thread gate retains the result of the operation, allowing the thread code to read it and use it as a counter of the number of times the thread was scheduled.

3. If the value of the thread gate is initially non-zero, the `XX_DecrThreadGate` kernel service decrements the thread gate by one. If the resulting value of the thread gate is zero, the thread's state immediately becomes *Ready* and the value of the thread gate is simultaneously changed to the value of the thread gate preset. Use of this method allows a thread to be scheduled only when a certain number of events has occurred.
4. If the value of the thread gate is initially non-zero, the `XX_ClearThreadGateBits` kernel changes the value by clearing one or more bits in it, reducing the value according to the value of the bits being cleared. If the resulting value of the thread gate is zero, the thread's state immediately becomes Ready and the value of the thread gate is simultaneously changed to the value of the thread gate preset. Use of this method allows a thread to be scheduled only when a set of specific events has occurred.

Methods 1 and 2, in which the thread gate value changes from zero to non-zero, schedule the associated thread with the new thread gate value. When the thread executes, it can read the thread gate value to determine the circumstances that caused the execution cycle. Two kernel services read the thread gate value.

The `XX_GetThreadGate` kernel service reads the thread gate value without modifying it. This service is an information retrieval tool only and can be called by any thread, including the Current Thread, as well as by tasks from Zone 3.

The second kernel service, `TS_GetThreadGateLoadPreset`, is available only to the Current Thread. This service returns the current value of the thread gate and also resets the thread gate value to the value of the thread gate preset property. Thus, if other processes perform additional thread gate operations on the thread between the time it is scheduled and the time it reads the thread gate value, it can detect those operations and take appropriate action.

Null Thread

The *Null Thread* is a special process in the **RTXC/ss** component. It is not an actual thread associated with a level because it only runs when all threads on all levels are in a Not_Ready state. It operates logically in Zone 3 and must use the system stack for any local variables. In a system using only the **RTXC/ss** component, the Null Thread is user-defined. It may be a simple spin loop or it can perform more complex operations particular to the application.

When it begins running, the Null Thread stays in control of the CPU until it or an interrupt handling routine schedules a thread. Because any such thread will, by definition, be of higher priority than the Null Thread, the **RTXC/ss** Scheduler preempts the Null Thread and gives control of the CPU to the higher priority thread. When the **RTXC/ss** Scheduler once again grants CPU control to the Null Thread, it continues from the point of its preemption.

If the system includes both the **RTXC/ss** and **RTXC/ms** components, the Null Thread functions are assumed by the **RTXC/ms** component, providing a very powerful tool to perform Zone 3 operations.

CHAPTER 3

Exceptions—Claiming Interrupt Vectors

In This Chapter

We discuss how the **RTXC** Kernel handles interrupts through the Exception object class. We first present the basic principles, rules, and organization of Exceptions and how the kernel uses them to prepare for servicing interrupts. Then we present some general usage concepts and more detailed information on Exceptions and interrupt servicing.

Introducing Exceptions

An embedded system usually has a relationship with an external process that it may be monitoring or controlling. The external process commonly requires servicing, sometimes at varying rates or periods. Devices connected to or associated with the process can make demands upon the system to take some action with respect to the process. These demands take the form of exceptions to the normal flow of processing. For each such exception or interrupt source, there may be a dedicated portion of code called an interrupt service routine (ISR) required to handle the demand.

The **RTXC** Kernel provides a generalized interrupt service scheme using the Exception object class. An exception object specifies the connection between an interrupt source and the application code that services it. When an interrupt or exception occurs, the **RTXC** Kernel uses that connection to transfer control from the interrupted process to the interrupt servicing procedure specific to the particular device causing the interrupt.

This chapter deals only with how to use the **RTXC** Kernel to claim interrupt vectors to establish the relationship between the exception source and the code that processes the exception request.

The exception object associated with each interrupt contains properties that direct the transfer of CPU control to the associated interrupt service routine. The exception properties do not dictate the technique of servicing the associated interrupt or whether the interrupt is even known to or processed by the **RTXC** Kernel. The user must make that specification by the nature of the interrupt service code. The following rule applies to all interrupts falling under control of the **RTXC** Kernel:

> **Rule: Every interrupt service routine defined or controlled to any degree by the Kernel must have an associated exception kernel object.**

An exception kernel object contains only data that represents the defined properties of the exception. The main purpose for the Exception as a class is that it promotes the possibility of having device drivers that are loadable at runtime by permitting the

interrupt vectors associated with the devices to be claimed while the system is in operation. There are few **RTXC** Kernel services associated with the Exception class. Except for services for specifying the properties of an exception, the services in this class are primarily associated with the use of dynamic exceptions.

As with all **RTXC** objects, exception kernel objects must reside in RAM.

Exception Definition

The kernel refers to exceptions by their handle, which is an `EXCPTN` type value. An exception handle must be within the range of the total number of exceptions defined for the application. During system generation, **RTXCgen** supports any combination of static and dynamic exceptions up to a total dependent on the size of a datum of the `EXCPTN` type.

There is no difference between the handle of a static exception and a dynamic exception. An exception handle of zero (0) is illegal if used in a kernel service for the exception object class.

Exception Properties

The Exception has several properties that determine the path of interrupt processing. The **RTXC** Kernel defines an `EXCPTNPROP` structure for use in claiming a vector for the application. The members of the `EXCPTNPROP` property structure represent the properties to which the developer has direct access. The `EXCPTNPROP` structure is organized as shown in Example 3-1.

Example 3-1. Exception Properties Structure

```
typedef struct
{
   KATTR attributes;      /* reserved for system use */
   unsigned char level;   /* interrupt level */
   unsigned char vector;  /* vector # */
   void (*handler)(void); /* address of interrupt service prologue */
} EXCPTNPROP;
```

The XX_DefExceptionProp service defines the properties of an exception using the values for the elements in the EXCPTNPROP structure. The XX_GetExceptionProp service, available only in Zone 2 and 3, reads the properties of a given exception and puts the property values into an EXCPTNPROP structure.

Exception Attributes

The *attributes* property of the exception object is reserved for internal system use.

Priority Level

The value of the exception *level* property represents the hardware interrupt priority level (IPL) at which the processor recognizes the interrupt request and begins interrupt processing. This property may not apply to all processors using the **RTXC** Kernel. Consult the target processor's reference manual.

Interrupt Vector

The kernel associates each exception with an interrupt vector location in memory. The vector property contains the vector number, which is an index into the processor's vector table. Depending on the processor, the vector may contain the address of the prologue of the interrupt servicing code or a branch or jump to the prologue. The **RTXC** Kernel permits all interrupt vectors to be resident in either RAM or ROM. The user makes the choice of interrupt vector memory type during system configuration. That choice determines how the **RTXC** Kernel claims interrupt vectors.

The use of RAM or ROM vectors has implications that are dependent on the processor. Consult the *Binding Manual* for the target processor for specific information on vector setup. One rule applies to vector claiming due to the way different processors treat interrupt vectors:

> **Rule: If a user writes a routine to claim an interrupt vector, it must match the method the Kernel uses.**

ISR Prologue Address

The *handler* property specifies the beginning memory address of the prologue segment of the interrupt service routine.

The prologue begins an ISR by saving the interrupted context of the processor to the extent the ISR requires.

Exception Vectors

The principal use of the Exception class is to provide a way of associating an interrupt vector with code that performs the interrupt servicing. A secondary use is to allow a designer to employ dynamic exceptions that associate a device with an interrupt vector at runtime, making it possible to have device drivers that the system can load dynamically.

There are no design or specific use methods for the **RTXC** Exception class because the exception object is merely an associative object. The details about the code in interrupt servicing routines is fully covered in the **RTXC** *Kernel I/O and Device Driver Development Guide*.

However, the user should understand the handling of vectors so as to define an exception properly.

The **RTXC** Kernel allows the placement of interrupt vectors in either RAM or ROM. The developer makes the decision where to place them during system generation as a configuration choice. The `XX_DefExceptionProp` service not only associates the exception with a particular vector but also sets up a pointer to the exception's interrupt servicing routine.

When using RAM vectors, all vectors are in an unknown state with undefined content at the time of system reset. At some time, the application code uses `XX_DefExceptionProp` to make the necessary exception property definitions. The exception definition procedure includes establishing a direct reference to the beginning of the exception's interrupt processing code, usually considered as the beginning of the prologue. RAM vectors are generally the most efficient in that the processor usually does not execute any extra instructions to get to the prologue. Some processors, however, do not

treat the vector content as an address of the prologue but rather as a jump, or other branch control instruction, to the prologue. To determine which method is appropriate, you should refer to your processor's reference manual. Refer to the *Binding Manual* for specific information about RAM vectors for the target processor.

The **RTXC** Kernel also supports the use of ROM-based interrupt vectors. ROM vectors require the user to set the content of interrupt vectors to be a direct or indirect reference to the interrupt service prologue in the manner described for the particular processor. With the vector in ROM, the `XX_DefExceptionProp` service cannot change the contents of the vector. ROM vectors do not cause any increase in interrupt latency compared to RAM vectors.

CHAPTER 4

Pipes—Buffered Data Movement

In This Chapter

We discuss the use of pipes as one of three data movement methods supported by the **RTXC** Kernel. We first present the basic principles, rules and organization of pipes. Then, to help you understand how to use this object class, we present some general usage concepts supported by extensive examples.

Introducing Pipes

The **RTXC** Kernel provides a method of moving data between program entities executing in different zones. The Pipe object class allows a zone 1 interrupt handler to pipe data to a thread or task, running at zones 2 and 3, respectively. A thread in zone 2 can pipe data to another thread, a task, or an interrupt handler. A zone 3 task can pipe data to another task, a thread or an interrupt handler. Therefore, pipes serve as a medium of data transfer that can operate in both the **RTXC/ss** and **RTXC/ms** components.

A pipe is an intervening object providing a standard interface between a producer and a consumer. Conceptually, a pipe is a pair of circular lists, one that holds empty buffers and one that contains full buffers. The producer and consumer may each be an interrupt handler, a thread, or a task. The producer puts data into the pipe using a buffer and the consumer gets it from the pipe as a buffer. Pipes are useful for handling such operations as stream input/output or other type of operations where data buffering is useful.

In an application, a pipe is generally employed with a single producer and a single consumer. However, the **RTXC** pipe model allows more than one task to insert data into a pipe (multiple producers) and more than one task to remove data from a pipe (multiple consumers). This capability leads to the following rule:

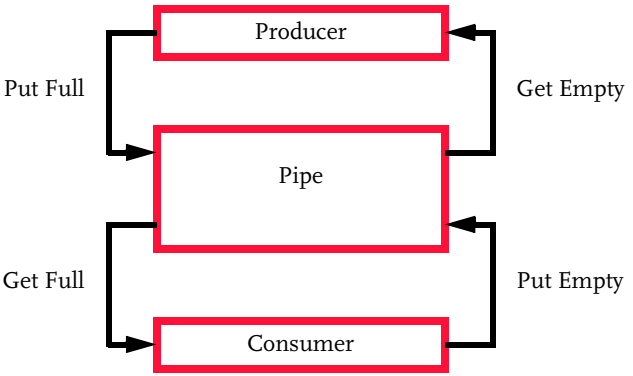
> **Rule: Any thread or task may put data into or get data from any pipe.**

By definition, all pipes use a buffer allocation model where the pipe content represents the chronological order of buffer entry into and extraction from the pipe. However, there are **RTXC** Kernel services that permit last-in-first-out entries and extractions when necessary. Like queues, there is no priority to the entries into a pipe.

Figure 4-1 on page 39 shows the four basic services the **RTXC** Kernel provides to circulate empty and full buffers through the pipe. No kernel services exist for the purpose of moving data into or out of a buffer. Copying of data into an allocated buffer as well as accessing data in a full buffer is the responsibility of the producer and consumer application programs, respectively. Additional **RTXC**

Kernel services provide combinational variants of the basic four pipe operations.

Figure 4-1. Basic Pipe Operations



Pipe Definition

Pipe handles are `PIPE` type numerical values. You can define, during system generation, any combination of static and dynamic pipes up to a total number dependent on the capacity of a datum of the `PIPE` type. You also define the sizes of each static pipe at that time.

A pipe handle must be within the range of the total number of pipes defined for the application. There is no difference between the handle of a static pipe and the handle of a dynamic pipe. A pipe handle of zero (0) is illegal if used in a kernel service for the Pipe object class.

Pipe Organization

A pipe has two parts: the Pipe Control Block (PiCB) and the set of data buffers it controls, both of which must reside in RAM. The PiCB must reside in System RAM. It contains information the **RTXC** Kernel services use to move buffers into and out of the pipe correctly. The set of pipe buffers must be located in User RAM.

Pipe Properties

RTXC pipes have several properties that can be defined and accessed by the developer. The **RTXC** Kernel defines a `PIPEPROP` properties structure for use in operations involving pipe properties. The members of the pipe property structure represent the properties to which the developer has direct access. The `PIPEPROP` structure has the organization shown in Example 4-1.

Example 4-1. Pipe Properties Structure

```
typedef struct
{
   KATTR attributes;
   KCOUNT numbufs;       /* number of buffers managed by pipe */
   ksize_t bufsize;      /* max useful size of buffer */
   void * base;
   void ** fullbase;     /* pointer to address of full buffer list */
   void ** freebase;     /* pointer to address of free buffer list */
   int * sizebase;       /* pointer to list of full buffer sizes; *
} PIPEPROP;
```

The `XX_DefPipeProp` service defines the properties of a pipe using values for the elements in the `PIPEPROP` structure. The `XX_GetPipeProp` service returns a pointer to the `PIPEPROP` structure of a given pipe.

The following rules apply to pipe properties:

Rule: A pipe must be defined with a maximum buffer size property (*bufsize*) that is greater than zero.

Rule: A pipe must be defined with a number of buffers property (*numbuf*) that is greater than zero.

Rule: A pipe must be defined with non-NULL pointers for the *fullbase*, *freebase*, and *sizebase* list pointers.

Pipe Attributes

Use of the pipe attribute property is currently restricted to and reserved for internal use by the **RTXC** Kernel. Future extensions to the **RTXC** Kernel will make use of this property and will be user-accessible.

Number of Buffers

The *numbuf* property defines the number of buffers the pipe is to manage.

Maximum Buffer Size

The *bufsize* property defines the maximum useful size of a buffer in the pipe. The user should ensure that all buffers managed by the pipe have a size greater than or equal to the value of *bufsize*. Proper specification of this property allows the use of buffers of unequal size so long as each meets or exceeds this specification.

Address of Pipe

Each pipe requires a pointer, *base*, to the User RAM area used as the body of the pipe. During system generation, the user can specify whether the kernel is to create the defined number of buffers or if the application code is to do so. For static pipes defined as being automatically created, the linker assigns the area for the pipe body and makes the value of *base* non-zero. The product of *numbuf* times *bufsize* defines the amount of User RAM necessary to hold the buffers for the pipe. If the user wishes to define the buffers at runtime, the **RTXCgen** program defines *base* as a NULL pointer. For dynamic pipes, the application determines and defines the pipe body area and its address during system operation.

Pointer to Full Buffer List

Each pipe requires a pointer, *fullbase*, to the User RAM area used as the list of pointers to the full buffers of the pipe. The producer puts full buffers into the pipe after having filled them with data. The full buffer list contains a set of *numbuf* entries, each of which points to a full buffer in the pipe.

Pointer to Free Buffer List

Each pipe requires a pointer, *freebase*, to the User RAM area used as the list of pointers to the free (empty) buffers of the pipe. The producer gets empty buffers from the pipe before filling them with

data. The free buffer list contains a set of *numbuf* entries, each of which points to an empty buffer in the pipe.

When the initialization process initializes static pipes using the `XX_DefPipeProp` kernel service, it uses the value of *base* to determine how to complete the definition. If *base* is a non-zero pointer, the kernel service divides up the allocated buffer area into *numbuf* blocks, each being *bufsize* long. As each buffer's address is determined, the kernel service also puts the pointer to the buffer into the free buffer list.

Note: Because the number of buffers is user-determined, the `XX_DefPipeProp` service is non-deterministic when used to allocate buffers and create the free buffer list contents automatically.

If the user's choice is to have the application code assign the free buffers to the free buffer list, as indicated by *base* being a NULL pointer, the `XX_DefPipeProp` service only defines the pipe's properties. It is the responsibility of the user to allocate and assign buffers to the free buffer list using the `XX_PutEmptyPipeBuf` service. It is the further responsibility of the application code to ensure that *numbuf* buffers are allocated, each of which has a size of at least *bufsize*.

Note: Because the buffers are not allocated when the `XX_DefPipeProp` service is used when *base* is a NULL pointer, the operation of the kernel service is deterministic. Each call to the `XX_PutEmptyPipeBuf` service to define a free buffer is also deterministic. This method may require more code than the automatic assignment method but each kernel service is deterministic. However, the overall execution time of this method may actually require more time than the automatic allocation method previously described.

Pointer to Buffer Size List

Each pipe requires a pointer, *sizebase*, to the User RAM area used as the list of sizes of full buffers in the pipe. The **RTXC** pipe model allows the user to fill a buffer with less than *bufsize* entries of data. Thus, when the producer puts full buffers into the pipe, it is necessary to define the amount of the buffer actually containing data so that the consumer knows how much data to process. The full buffer size list contains *numbuf* entries.

The **RTXC** Kernel permits a task to read the properties structure of a pipe at any time using the `XX_GetPipeProp` service.

Pipe States

The **RTXC** Kernel maintains a record of the available free and full buffers in the pipe at all times. The pipe does not have a single state but rather one defined by the condition of the free and full buffer lists. Therefore, the **RTXC** Kernel automatically maintains the pipe's state. There are no user-accessible pipe states.

Optional Properties

The **RTXC** Kernel supports no optional properties for the Pipe object class.

Using Pipes

Pipes provide an easy method of moving buffered data from one point to another in an **RTXC** Kernel-based application. Generally, processing of the buffered data takes place in chronological order but there are circumstances that require LIFO order. The use of pipes in an application involves the use of two code entities, a producer and a consumer. It is the job of the producer to allocate an empty buffer, fill it with data, and then put the full buffer and its size specification into the pipe. The consumer has to get a full buffer and its size specification from the pipe, process the data therein, and then put

the now-empty buffer back into the pipe. Thus the flow of buffers is circular from the producer to the consumer and back.

Pipe operations fall into two basic categories: managing empty buffers and managing full buffers. For each type of buffer there are two basic operations: *getting* buffers from pipes and *putting* buffers into pipes. The **RTXC** Kernel provides one basic kernel service for each basic pipe operation, plus some variants of each.

Note: This section uses the terms *free buffers* and *empty buffers* interchangeably. Both terms mean a buffer that no longer contains information that either the producer or the consumer needs.

Producer Operations

The basic pipe operations of a producer are to get an empty buffer, fill it with data, and put the full buffer into the pipe. Before a producer can put data into a buffer, it must first acquire an empty buffer. To do so, the producer uses the `XX_GetEmptyPipeBuf` service, which returns a pointer to the next available free buffer in the pipe's free buffer list. If a free buffer is not available, the kernel service returns a NULL pointer and the producer must deal with the failure of the request.

When the producer has the pointer to the free buffer, it is free to write data into the buffer in whatever manner is appropriate to the application. Having filled the buffer with data, the producer then puts the buffer into the pipe using the `XX_PutFullPipeBuf` kernel service, along with a specification about how much data it wrote into the buffer. The full buffer and its size specification then become part of the full buffer list of the pipe.

After putting the full buffer into the pipe, the producer may acquire a new empty buffer or it may defer that operation until its next execution cycle. The `XX_PutFullGetEmptyPipeBuf` service allows the producer to combine the operations of putting the full buffer into the pipe and getting a new empty buffer. The combined operations are intended to reduce the amount of overhead required

in making two separate kernel service requests. However, the following rules apply to using a combined operation service:

Rule: When using a combined operation service for pipes, both operations must be successful for the service to complete successfully.

Rule: When using a combined operation service for pipes, both operations apply to the same pipe.

For example, the `XX_PutFullGetEmptyPipeBuf` kernel service requires there to be a place in the pipe's full buffer list and an empty buffer available in the pipe's free buffer list at the time the application code places the request. If either condition is not true, the service fails and the producer has to take appropriate action. A failure is likely to be the result of improper specification of the number of buffers needed in the pipe.

Example 4-2 on page 46 shows a code fragment of an interrupt handling routine as a producer putting data into the `PIPEXYZ` pipe and then scheduling a consumer, the `THREADXYZ` thread, when the buffer is full. Note that the example relies on an external initialization of the four global variables: `bufptr`, `bufbase`, `bufcount`, and `maxbufsize`.

Example 4-2. Producer Putting Data into Pipe

```
#include "rtxcapi.h"
#include "kproject.h"
#include "kpipe.h"   /* defines PIPEXYZ */
#include "kthread.h" /* defines THREADXYZ */

/*   Global Variables for Use by Producer */
 void * bufptr;         /* working pointer to current buffer */
 void * bufbase;        /* base pointer to current buffer */
 int bufcount;          /* working counter */
 int maxbufsize;        /* max value for counter */

/*************************************************************/
/*         Interrupt Handler for Device XYZ                  */
/*************************************************************/
void deviceXYZhandler ()
{
   ...service the device and get the data

   if (bufbase == (void *)0)
   {
      if ((bufbase = IS_GetEmptyPipeBuf (PIPEXYZ)) == (void *)0)
      {
         ...no buffers available, no place to store data
         return;  /* data missed because consumer is too slow */
      }
      bufptr = bufbase; /* set up working pointer to buffer */
   }
   /* a buffer is available */
   ...store data in buffer using bufptr
   bufcount++;

   if (bufcount == maxbufsize)       /* test for end of buffer */
   {
      /* buffer is full. Send it to pipe and setup next buffer */
      IS_PutFullPipeBuf (PIPEXYZ, bufbase, bufcount);
      bufbase = IS_GetEmptyPipeBuf (PIPEXYZ);/* get empty buffer */
      bufptr = bufbase;       /* setup working pointers and counts */
      bufcount = 0;
     IS_ScheduleThread (THREADXYZ);/* schedule thread to process data
*/
   }
   return;   /* end of interrupt handler */
}
```

Example 4-3 on page 48 is also a producer. Again, an external code entity has initialized the working variables in the same manner as the previous example. It is permissible to have `bufbase` initialized to a NULL pointer as the producer determines if it is necessary to acquire an empty buffer from the pipe. The code that verifies the existence of a valid buffer pointer protects the producer from the situation where the `XX_PutEmptyGetFullPipeBuf` kernel service fills the full buffer list and has no empty buffer available to allocate for the next empty buffer. That situation is an indication that the producer is outrunning the consumer. You may choose to omit this extra code if it is certain that the consumer can keep up with the producer. However, even if that is the case, it may be safer to use it just to avoid the possibility of the producer not having a buffer in which to store data.

Example 4-3. Producer Putting Data into Pipe Using Combined Operations

```
#include "rtxcapi.h"
#include "kproject.h"
#include "kpipe.h"   /* defines PIPEXYZ */
#include "kthread.h" /* defines THREADXYZ */

/*   Global Variables for Use by Producer */
 void * bufptr;         /* working pointer to current buffer */
 void * bufbase;           /* base pointer to current buffer */
 int bufcount;                            /* working counter */
 int maxbufsize;                    /* max value for counter */

/*************************************************************/
/*         Interrupt Handler for Device XYZ                  */
/*************************************************************/
void deviceXYZhandler ()
{
   ...service the device and get the data

   if (bufbase == (void *)0)
   {
      if ((bufbase = IS_GetEmptyPipeBuf (PIPEXYZ)) == (void *)0)
      {
         ...no buffers available, no place to store data
         return;
      }
      bufptr = bufbase; /* set up working pointer to buffer */
   }
   /* a buffer is available */
   ...store data in buffer using bufptr
   bufcount++;

   if (bufcount == maxbufsize)       /* test for end of buffer */
   {
      /* buffer is full. Send it to pipe and setup next buffer */
      bufbase = IS_PutFullGetEmptyPipeBuf (PIPEXYZ, bufbase,
bufcount);
      bufptr = bufbase;   /* setup working pointers and counts */
      bufcount = 0;
      IS_ScheduleThread (THREADXYZ);/* schedule thread to process data
*/
   }
   return;   /* end of interrupt handler */
}
```

Consumer Operations

The basic pipe operations of a consumer are to get a full buffer, process the data in it, and return the empty buffer to the pipe. Before a producer can process data in a buffer, it must first acquire a full buffer. To do so, the producer uses the `XX_GetFullPipeBuf` kernel service, which returns a pointer to the next available full buffer in the pipe's full buffer list along with its size specification. If a full buffer is not available, the kernel service returns a NULL pointer and the producer must deal with the failure of the request.

When the producer has the pointer to the full buffer, it can process the data in the buffer in whatever manner is appropriate to the application. It is not uncommon for the consumer on one pipe to be the producer of another pipe. This situation often arises when the consumer processes data from a pipe by reducing it and sending it to another pipe.

Having acquired the full buffer and processed its data, the consumer then frees the buffer to the pipe's free buffer list using the `XX_PutEmptyPipeBuf` service. The empty buffer then becomes part of the free buffer list of the pipe.

After putting the free buffer into the pipe, the consumer may acquire a new full buffer or it may defer that operation until its next execution cycle. The `XX_PutEmptyGetFullPipeBuf` service allows the consumer to combine the operations of putting the empty buffer into the pipe and getting a new full buffer. The combined operations reduce the amount of overhead required compared to making two separate kernel service requests. The same rules about combined operations, as previously stated, apply to consumer operations as well.

Example 4-4 on page 50 shows a code fragment of a thread, `THREADXYZ`, acting as a pipe consumer. It contains two parts, an initialization function with an entry at `threadxyz`, and a processing function whose entry is `XYZentryA`. The purpose of the initialization function is to initialize the producer, which is the interrupt service routine in either Example 4-2 or Example 4-3. In this example, the initialization function gets an empty buffer from the `PIPEXYZ` pipe and initializes with its address. According to the

design in Example 4-2 and Example 4-3, the producer would work equally well if the initialization function of THREADXYZ set the global variable *bufbase* to a NULL pointer ((void *)0).

Example 4-4. Consumer Getting Data from Pipe

```
#include "rtxcapi.h"
#include "kproject.h"
#include "kpipe.h"   /* defines PIPEXYZ */
#include "kthread.h" /* defines THREADXYZ */

/*   Global Variables for Use by Producer */
extern void * bufptr;      /* working pointer to current buffer */
extern void * bufbase;     /* base pointer to current buffer */
extern int bufcount;       /* working counter */
extern int maxbufsize;     /* max value for counter */

/*   Environment Arguments for THREADXYZ  */
struct
{
   int state;
   int value;
}myenvargs;

void XYZentryA (void *, void *); /* function for processing */
                                 /* buffers                 */

/*****************************************************************/
/*             ThreadXYZ initialization function                 */
/*****************************************************************/
void threadxyz ((void *)0, (void *)0) /* no arguments passed */
{
struct myenvargs * myargs;
PIPEPROP xyzprops;

   TS_DefThreadEnvArg (SELFTHREAD, myargs);
   myargs->state = 0;   /* initialize the state of the thread */

   TS_GetPipeProp (PIPEXYZ, &xyzprops); /* get pipe properties */
   maxbufsize = xyzprops.bufsize;  /* set up maximum buffer size */
   bufbase = TS_GetEmptyPipeBuf (PIPEXYZ); /* get empty buffer */
                                          /* pointer */

   bufptr = bufbase;  /* initialize working pointers for use by  */
                      /*   by exception handler */

   /* at end of initialization, setup new entry point for thread */
   TS_DefThreadEntry (SELFTHREAD, XYZentryA);
```

```
   ...then enable device XYZ

   /* after the following return, further processing in this */
   /* thread commences at entry point XYZentryA */

   return;
}

/*****************************************************************/
/*                 ThreadXYZ processing function.                */
/*     Processes data in PIPEXYZ according to thread's state     */
/*****************************************************************/
void XYZentryA ((void *)0, (struct myenvargs *)myargs)
{
int actualsize;   /* actual size of full buffer */
char * newbuf;    /* pointer to full buffer */

   switch (myargs->state)
   {
      case 0
         newbuf = TS_GetFullPipeBuf (PIPEXYZ, &actualsize);

         ...process the data in the buffer

         TS_PutEmptyPipeBuf (PIPEXYZ, newbuf);
         break;
      case ???   /* other cases for other states if needed */
   }   /* end of switch statement
   return;
}
```

Jamming Data into a Pipe

For the producer, the normal mode of putting a full buffer into a pipe is to append it to the tail of the full buffer list, preserving the chronological nature of the data. Under some conditions, the producer may need to put a buffer into the pipe that is not in chronological order. The **RTXC** pipe model supports the `XX_JamFullPipeBuf` services to *jam* the buffer into the pipe, not at the tail, but at the head of the pipe. When the producer jams a buffer into a pipe, that buffer becomes the new head of the pipe's full buffer list. As such, the consumer retrieves that buffer because the consumer's request for a full buffer always is filled from the head of the pipe.

There is also the XX_JamFullGetEmptyPipeBuf service that combines jamming operations with getting an empty buffer. Except that the buffer is put at the head of the full buffer list, the service works in the same manner and carries the same restrictions as the XX_PutFullGetEmptyPipeBuf kernel service.

Pipe Actions and Conditions

The **RTXC** Kernel provides ways for pipes to act on threads as a result of putting a full or empty buffer into a pipe. Putting an empty buffer into a pipe makes it available to a producer. Conversely, putting a full buffer into a pipe makes it available for a consumer. A thread, unlike a task, cannot wait for something to occur. If the producer is a thread, it cannot wait for an empty buffer to become available. When it runs, the empty buffer must be available or the producer may fail to pass on critical data. The **RTXC** Kernel supports actions that can occur as a result of freeing an empty buffer or putting a full one into a pipe. These actions permit threads to use pipes effectively even though a thread cannot wait for a buffer in the pipe to become available. The XX_DefPipeAction service exists for this purpose. It allows a task or thread to set up one of two basic actions to take when a application program puts either an empty or a full buffer into a given pipe. Once defined, the definition remains in place until it is changed by another call to XX_DefPipeAction to change the action or to specify no action.

The specified action takes place only when the program uses a kernel service to put a buffer into a pipe. The action does not occur when the application gets a buffer from the pipe. Putting a buffer into a pipe means use of any of the kernel services that put empty and full buffers into a pipe. The kernel service may put the buffer in normally or jam it in. The put operation may be a combination operation that also gets a buffer from the same pipe.

If an interrupt handler calls the kernel service that puts the buffer into the pipe, then the resulting action must be performed from that same zone. The situation is identical when a thread calls a buffer put kernel service. Because there are two zones from which the two basic actions can take place, the user must specify the zone in which the

action is to occur. Hence, the kernel services have to allow for four actual action definitions.

The two basic actions are:

SCHEDULETHREAD	Schedule a thread
DECRTHREADGATE	Decrement a thread's thread gate

The XX_DefPipeAction kernel service requires the specification of when the action is to occur. The two possible conditions are:

PUTEMPTY	When putting an empty buffer into the pipe.
PUTFULL	When putting a full buffer into the pipe.

When the action is properly defined, a kernel service that puts a buffer into the specified pipe according to the given condition will execute the internal functions to perform one of these two basic services. The internal functions are dependent on the zone of the caller to the kernel service that performs the buffer putting operation.

One action, scheduling a thread, has direct effect. The specified thread is scheduled and has only to wait until the **RTXC/ss** Scheduler gives it control of the CPU. If the specified action is to decrement the thread's thread gate, the thread may or may not become ready, depending on the value of the thread gate.

Example 4-5 on page 54 shows how a producer and a consumer can use pipe actions to create an effective synchronization of their operations. In the example, the producer is an interrupt handler and the consumer is a thread, THREADXYZ.

Example 4-5. Pipe Action when Putting Full Buffers into Pipe

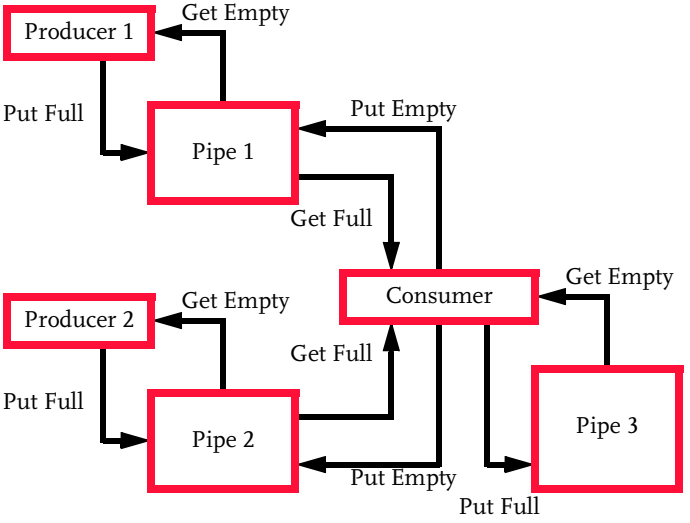
```
#include "rtxcapi.h"
#include "kproject.h"
#include "kpipe.h"   /* defines PIPEXYZ */
#include "kthread.h" /* defines THREADXYZ */

/*   Global Variables for Use by Producer */
 void * bufptr;         /* working pointer to current buffer */
 void * bufbase;           /* base pointer to current buffer */
 int bufcount;                            /* working counter */
 int maxbufsize;                    /* max value for counter */

/*   Environment Arguments for THREADXYZ  */
struct
{
   int state;
   int value;
}myenvargs;

void XYZentryA (void *, void *);   /* function for processing buffers
*/


/*************************************************************/
/*         Interrupt Handler for Device XYZ                  */
/*************************************************************/
void deviceXYZhandler ()
{
   ...service the device and get the data

   if (bufbase == (void *)0)
   {
      if ((bufbase = IS_GetEmptyPipeBuf (PIPEXYZ)) == (void *)0)
      {
         ...no buffers available, no place to store data
         return;  /* data missed because consumer is too slow */
      }
      bufptr = bufbase; /* set up working pointer to buffer */
   }
   /* a buffer is available */
   ...store data in buffer using bufptr
   bufcount++;

   if (bufcount == maxbufsize)        /* test for full of buffer */
   {
      /* buffer is full. */
      /* Send it to pipe, schedule THREADXYZ, and get new buffer */
      bufbase = IS_PutFullGetEmptyPipeBuf (PIPEXYZ, bufbase,
                bufcount);
```

```
      bufptr = bufbase;     /* setup working pointers and counts */
      bufcount = 0;
   }
   return;   /* end of interrupt handler */
}

/*************************************************************/
/*             ThreadXYZ initialization function             */
/*************************************************************/
void threadxyz ((void *)0, (void *)0) /* no arguments passed */
{
struct myenvargs * myargs;
PIPEPROP xyzprops;

   /* define action producer takes when putting full buffer */
   TS_DefPipeAction (PIPEXYZ, SCHEDULETHREAD, THREADXYZ, PUTFULL);

   TS_DefThreadEnvArg (SELFTHREAD, myargs);
   myargs->state = 0;   /* initialize the state of the thread */

   TS_GetPipeProp (PIPEXYZ, &xyzprops); /* get pipe properties */
   maxbufsize = xyzprops.bufsize;  /* set up maximum buffer size */
   bufbase = TS_GetEmptyPipeBuf (PIPEXYZ); /* get empty buffer */
                                           /* pointer */
   bufptr = bufbase;  /* initialize working pointers for use by */
                      /*  exception handler */

   /* at end of initialization, setup new entry point for thread */
   TS_DefThreadEntry (SELFTHREAD, XYZentryA);

   ...then enable device XYZ

   /* after the following return, further processing in this */
   /* thread commence at entry point XYZentryA */

   return;
}

/*************************************************************/
/*                 ThreadXYZ processing function.            */
/*     Processes data in PIPEXYZ according to thread's state */
/*************************************************************/
void XYZentryA ((void *)0, (struct myenvargs *)myargs)
{
   int actualsize;   /* actual size of full buffer */
   char * newbuf;    /* pointer to full buffer /

   switch (myargs->state)
   {
      case 0
```

```
            newbuf = TS_GetFullPipeBuf (PIPEXYZ, &actualsize);

            ...process the data in the buffer

            TS_PutEmptyPipeBuf (PIPEXYZ, newbuf); /* no pipe action */
                                                  /* here */
            break;

         case ???   /* other cases for other states if needed */

      }   /* end of switch statement
      return;
}
```

Now consider a situation where two producers feed data into two pipes with a single consumer getting data from both pipes, operating on it, and then putting the combined data into a third pipe. For simplicity, the example assumes the consumer (`threadxyz`) of the two pipes can keep up with the two producers. The consumer cannot be scheduled until it has a full buffer in Pipe 1 and Pipe 2 plus an empty buffer from Pipe 3 for which it is also the producer. Figure 4-2 shows the organization of the example.

Figure 4-2. Multiple Pipe, Single Consumer Organization



The key to making this example work efficiently is the pipe action to decrement a thread gate. Both producers do that whenever they put a full buffer into their respective pipes. Similarly, the consumer of Pipe 3 (not shown) causes a decrement of the thread gate for the multiple pipe consumer (shown) whenever it puts an empty buffer into Pipe 3. By that sequence of pipe actions, the process stays synchronized and efficient. Example 4-6 shows the code fragments for this organization.

Example 4-6. Pipe Actions with Multiple Producers and Single Consumer

```
#include "rtxcapi.h"
#include "kproject.h"
#include "kpipe.h"   /* defines PIPE1 and PIPE2 and PIPE3*/
#include "kthread.h" /* defines THREADXYZ */

/*   Global Variables for Use by Producer 1 */
 void * buf1ptr;         /* working pointer to current buffer */
 void * buf1base;           /* base pointer to current buffer */
 int buf1count;                          /* working counter */
 int maxbufsize1;                   /* max value for counter */

/*   Global Variables for Use by Producer 2 */
 void * buf2ptr;         /* working pointer to current buffer */
 void * buf2base;           /* base pointer to current buffer */
 int buf2count;                          /* working counter */
 int maxbufsize2;                   /* max value for counter */

/*   Environment Arguments for Consumer (THREADXYZ)  */
struct
{
   int state;
   int maxbufsize3;
}myenvargs;

void XYZentryA (void *, void *);   /* function for processing buffers
*/

/*************************************************************/
/*         Interrupt Handler Producer for PIPE1               */
/*************************************************************/
void deviceXYZhandler ()
{
   ...service the device and get the data
   ...store data in buffer using buf1ptr
   buf1count++;
```

```
   if (buf1count == maxbufsize1)     /* test for full of buffer */
   {
      /* buffer is full. */
      /* Send it to pipe, decr THREADXYZ thread gate, */
      /*   get new buffer */
      buf1base = IS_PutFullGetEmptyPipeBuf (PIPE1, buf1base,
buf1count);
      buf1ptr = buf1base;  /* setup working pointers and counts */
      buf1count = 0;
   }
   return;   /* end of interrupt handler */
}

/*************************************************************/
/*         Exception Handler Producer for PIPE2              */
/*************************************************************/
void deviceQRSexception ()
{
   ...service the device and get the data
   ...store data in buffer using buf2ptr
   buf2count++;

   if (buf2count == maxbufsize2)     /* test for full of buffer */
   {
      /* buffer is full. */
      /* Send it to pipe, decr THREADXYZ thread gate, */
      /*   get new buffer */
      buf2base = IS_PutFullGetEmptyPipeBuf (PIPE2, buf2base,
                                            buf2count);
      buf2ptr = buf2base;  /* setup working pointers and counts */
      buf2count = 0;
   }
   return;   /* end of exception handler */
}

/*************************************************************/
/*             ThreadXYZ initialization function             */
/*************************************************************/
void threadxyz ((void *)0, (void *)0) /* no arguments passed */
{
struct myenvargs * myargs;
PIPEPROP xyzprops;

   /* define action producers take when putting full buffer */
   TS_DefPipeAction (PIPE1, DECRTHREADGATE, THREADXYZ, PUTFULL);
   TS_DefPipeAction (PIPE2, DECRTHREADGATE, THREADXYZ, PUTFULL);
   TS_DefPipeAction (PIPE3, DECRTHREADGATE, THREADXYZ, PUTEMPTY);

   TS_DefThreadEnvArg (SELFTHREAD, myargs);
   myargs->state = 0;   /* initialize the state of the thread */
```

```
   TS_GetPipeProp (PIPE1, &xyzprops); /* get properties of pipe 1*/
   maxbufsize1 = xyzprops.bufsize; /* set up maximum buffer size */
   buf1base = TS_GetEmptyPipeBuf (PIPE1); /* get empty buffer */
                                          /*   pointer */
   buf1ptr = buf1base;  /* initialize working pointers for use */
                        /*   by exception handler */

   TS_GetPipeProp (PIPE2, &xyzprops); /* get properties of pipe 2*/
   maxbufsize2 = xyzprops.bufsize; /* set up maximum buffer size */
   buf2base = TS_GetEmptyPipeBuf (PIPE2); /* get empty buffer */
                                          /*   pointer */
   buf2ptr = buf2base;  /* initialize working pointers for use */
                        /*   by exception handler */

   TS_GetPipeProp (PIPE3, &xyzprops); /* get properties of pipe 3*/
   myargs->maxbufsize3 = xyzprops.bufsize;/* maximum buffer size */

   /* at end of initialization, setup new entry point for thread */
   TS_DefThreadEntry (SELFTHREAD, XYZentryA);

   /* set thread gate and thread gate preset to 3 */
   TS_SetThreadGate (SELFTHREAD, (GATEKEY)3);

   ...then enable interrupts on devices XYZ and QRS

   /* after the following return, further processing in this */
   /*   thread commences at entry point XYZentryA */

   return;
}

/*****************************************************************/
/*                 ThreadXYZ processing function.                */
/*     Processes data in PIPEXYZ according to thread's state     */
/*****************************************************************/
void XYZentryA ((void *)0, (struct myenvargs *)myargs)
{
int truesize1;     /* actual size of full buffer */
char * newbuf1;    /* pointer to full buffer /

int truesize2;     /* actual size of full buffer */
char * newbuf2;    /* pointer to full buffer /

char * buf3base;
int buf3count;

   switch (myargs->state)
   {
      case 0
```

```
         /* get the two full buffers from Pipes 1 and 2 */
         newbuf1 = TS_GetFullPipeBuf (PIPE1, &truesize1);
         newbuf2 = TS_GetFullPipeBuf (PIPE2, &truesize2);

         /* get the empty buffer from Pipe 3 */
         buf3base = TS_GetEmptyPipeBuf (PIPE3);

         for (...loop conditions)
         {
            ...now process the data in the full buffers and put
               results into the buffer from Pipe 3
            buf3count++; /* increment results buffer size */
            ...continue this processing loop until done
         }

         /* done, release now empty buffers back to Pipes 1 & 2 */
         TS_PutEmptyPipeBuf (PIPE1, newbuf1); /* no pipe action */
         TS_PutEmptyPipeBuf (PIPE2, newbuf2); /* no pipe action */

         /* then put results buffer into Pipe 3 */
         TS_PutFullPipeBuf (PIPE3, buf3base, buf3count);
              /* no pipe action */

         break;   /* then quit and wait until next cycle */

      case ???   /* other cases for other states if needed */

   }   /* end of switch statement
   return;
}
```

CHAPTER 5

Event Sources, Counters, and Alarms—Keeping Track of Events

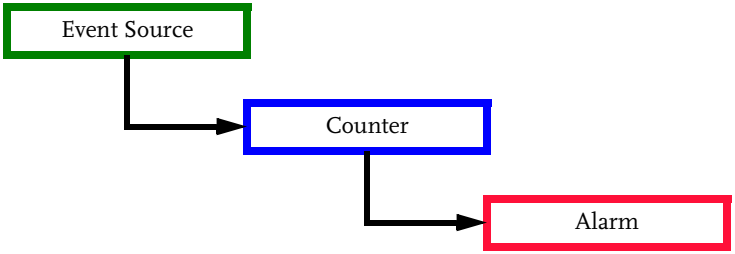
In This Chapter

We first discuss the basic principles of event management hierarchy in the **RTXC** Kernel. Next, we present the guidelines for each of the classes in the hierarchy, Event Sources, Counters, and Alarms. Then we present some related concepts and basic rules of event counting within the **RTXC** Kernel. Last, we discuss the usage of all three classes and present some examples.

The Event Management Hierarchy

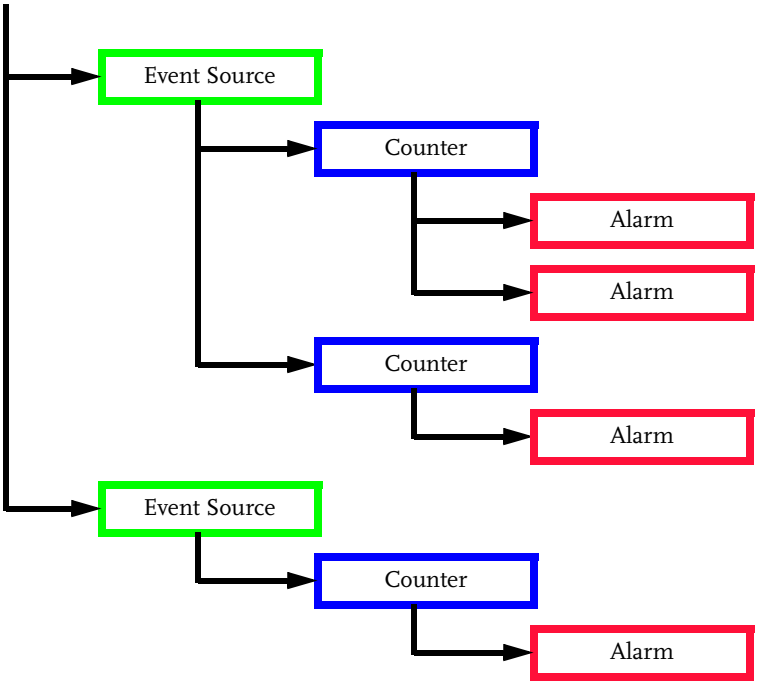
Events occur in a real-time system in various ways; some are periodic while others are not. Some events are points of synchronization and others need only to be counted. Synchronization events are generally associated with **RTXC** Semaphores. Counted events must be counted, but they can be used to initiate actions when the accumulated count reaches a predefined value. To track events that need to be counted, the **RTXC** Kernel incorporates the event management hierarchy using three object classes: Event Sources, Counters, and Alarms. An example of a type of event source is an interrupt that occurs periodically from a system clock, sometimes called a time base. Figure 5-1 shows the event management hierarchy and the relationships between the three classes

Figure 5-1. Event Management Hierarchy



From Figure 5-1, Event Sources are the parent object of Counters, which are in turn the parent of Alarms. An application using an **RTXC** Kernel configuration that includes the Event Source class may use one or more Event Source objects. Each Event Source object may have one or more associated Counter objects. Finally, each of those Counter objects may have one or more associated Alarm objects. Figure 5-2 on page 63 depicts a more realistic event management and counting hierarchy.

Figure 5-2. Event Management Hierarchy, Realistic Example



This chapter deals with each of these classes in order of their precedence in the hierarchy. Because event sources, counters and alarms are accessible to tasks, threads and, sometimes, interrupt handlers, we indicate mixed-use kernel services by the `XX_` prefix, where `XX_` represents the zonal prefixes `IS_` (Zone 1), `TS_` (Zone 2), and `KS_` (Zone 3). To determine the legal prefixes for a particular kernel service, see the kernel service descriptions in the **RTXC Quadros** and **RTXC DSP** *Kernel Service Reference* manuals.

Introducing Event Sources

The Event Source class is the basis of a general purpose counting and alarm model that enables threads and tasks to operate in conjunction with or in response to alarms based on counted events. The Event Source class allows the accumulation of counts and division of event counts into lower order counters to permit the use of alarms.

Consequently, the actual source of an event is immaterial. The event may originate externally and present itself to the system as an interrupt, or it may be an internal event generated by the application software. In either case, there is no distinction between such events as they relate to the event counting and alarm model.

Event Counting

In the **RTXC** Kernel, event sources and semaphores both have the ability to count events. However, semaphores count events only when there is no task waiting for the event. Therefore, the semaphore's counter serves as a record of how many occurrences of the event the kernel has detected since the last time a task synchronized with the event. If there is an accumulation of event occurrences, the synchronization process reduces the value of the semaphore's count each time the task attempts to wait on the event. As a result, the semaphore's count is eventually reduced to zero, an indication that all occurrences have been accounted for.

Event sources accumulate a count of event occurrences (event *ticks*) in a free-running manner. That is to say, the count increases with each event occurrence and rolls over to zero when it reaches a maximum value. However, the primary purpose of an event source is to establish the set of counter objects associated with the event source.

It is permissible to treat any type of event as an event source for the purposes of counting. A typical type of event used for counting is one that represents the system time base. Such an event is normally a periodic interrupt. Other types of events can represent the rotation of a number of degrees on a shaft or axle, the number of switch closures or openings, and so on.

Event Source Definition

The kernel refers to an event source by its handle, which is an `EVNTSRC` type value. You can define, during system generation, any combination of static and dynamic event sources up to a total dependent on the size of a data word of the `EVNTSRC` type.

An event source handle must be within the range of the total number of event sources defined for the application. There is no difference between the handle of a static or a dynamic event source.

An event source handle of zero (0) has special meaning. It defines the event source for the application's time base.

Event Source Properties

Event sources have two properties that are available to the developer. Each event source object has an accumulator property that collects the number of event ticks. There is also an attributes property that specifies whether accumulation of event ticks is enabled or disabled, and if the event source is associated with the system time base periodic event tick.

A task or thread can access the definable alarm properties through an `EVNTSRCPROP` structure, which is organized as shown in Example 5-1.

Example 5-1. Event Source Properties Structure

```
typedef struct
{
   KATTR attributes;   /* Enable/Disabled, Use for System Time Base */
   TICKS accumulator;  /* initial time count */
} EVNTSRCPROP;
```

Event Source Attribute

The event source *Counting_State* attribute defines whether or not the event source can accumulate counts of event ticks. By definition, the kernel sets this attribute to `ATTR_EVENT_COUNTING_ENABLED`. To disable event tick accumulation temporarily or permanently, a task or thread sets the attribute to `ATTR_EVENT_COUNTING_DISABLED`.

Event Count Accumulator

The event tick *accumulator* contains the number of event ticks counted while the *Counting_State* attribute is set to `ATTR_EVENT_COUNTING_ENABLED`. The event tick accumulator is a free-running accumulator. The initialization code defines the initial

value of the accumulator to be zero for all static event sources objects. A task establishing a dynamic event source, or a task that needs to redefine the *accumulator* property for an existing event source, may set the initial value of *accumulator* to any legal value of the `TICKS` type using the `XX_SetEventSourceAcc` kernel service.

Using Event Sources

Event Sources are the basis for establishing a system by which tasks and threads activate in response to expiration of alarms. Event Sources may be static or dynamic and are globally accessible to tasks and threads. **RTXC** Kernel services exist to define, enable or disable, update, read, set accumulators for, and perform all the updating of accumulators for event sources as well as processing the event for all child counter objects and their alarms.

The key to the entire event management hierarchy is the `XX_ProcessEventSourceTick` service. That kernel service takes care of all of the processing required to update the entire event management hierarchy for a given event source.

Introducing Counters

Counters accumulate counter ticks in a free-running manner. That is to say, the counts increase with each counter tick and roll over to zero when they reach a maximum value for the accumulator data type. Each counter tick represents a user-defined ratio of the number of event ticks the counter's parent event source receives. For example, if the ratio is 100 event ticks per counter tick on a given counter, the counter gets one tick added to it for each 100 event ticks.

Besides accumulating counter ticks, a counter can also be the parent of a set of alarm objects. The **RTXC** Kernel manages alarms for tasks and threads in relation to counter tick accumulations. Only when a counter tick occurs for a given counter does the `XX_ProcessEventSourceTick` service test for expiry on the alarms associated with that counter.

Counter Definition

The kernel refers to a counter by its handle, which is a `COUNTER` type value. You can define, during system generation, any combination of static and dynamic counters up to a total dependent on the size of a data word of the `COUNTER` type.

An counter handle must be within the range of the total number of counters defined for the application. There is no difference between the handles of static or dynamic counters.

Counter Properties

Counters have three properties that are available to the developer. Each counter object has an accumulator property that collects the number of counter ticks. There is also an *attributes* property that specifies whether accumulation of counter ticks is enabled or disabled, and if the counter is the system time base.

A task or thread can access the definable counter properties through a `COUNTERPROP` structure, which is organized as shown in Example 5-2.

Example 5-2. Counter Properties Structure

```
typedef struct
{
   KATTR attributes;   /* Enable/Disabled, Use for System Time Base */
   EVNTSRC evntsrc;    /* Event Source associated with this counter /
   KMODULUS modulus;   /* modulus (ratio) for dividing event ticks */
} COUNTERPROP;
```

Counter Attribute

The counter *Enable/Disable* attribute defines whether or not the counter can accumulate counter ticks. By definition, the kernel sets this attribute to a default value of `ATTR_COUNTER_ENABLED` (specifically, `~(ATTR_COUNTER_DISABLED)`). A task or thread may disable counter tick accumulation temporarily or permanently by setting the attribute to `ATTR_COUNTER_DISABLED`. Besides using the `XX_DefCounterProp` kernel services to define all the properties, including the attributes, of a counter, the

`XX_SetCounterAttr` service allows a task or a thread to set the counter's attributes directly. The `XX_ClearCounterAttr` service allows a task or thread to clear specific attribute settings directly.

Event Source

The *evntsrc* property defines the counter's parent event source. The counter receives event ticks from the defined parent event source and reduces those event ticks to get counter ticks.

Event Tick Modulus

The *modulus* property contains the number of event ticks to count for each counter tick. The counter's modulus is essentially a divider to be applied to the stream of event ticks from the parent event source. If the modulus has a value of 10, the counter must receive 10 event ticks before it counts one counter tick.

Tick Count Accumulator

There is an implicit property of every counter, the counter tick *accumulator*, which contains the number of counter ticks counted while the attribute is set to `ATTR_COUNTER_ENABLED`. The counter tick accumulator is free-running and rolls over to zero when it reaches the maximum value for it's defined data type plus one. The `XX_DefCounterProp` service defines the initial value of the accumulator to be zero. A task can define the counter's *accumulator* property to begin counting from a specified base value by setting the initial value of *accumulator* to any legal value of the `TICKS` type using the `XX_SetCounterAcc` service.

Tick Conversion

When using counters and alarms, it is quite common to encounter the need to convert a value in engineering units into a number of ticks or vice versa. The methods to do both conversions are quite simple as each tick represents a fixed number of engineering units.

Consider the conversion of real time to **RTXC** counter ticks of the application time base. Take the real-time value, expressed in some convenient units such as seconds, milliseconds, or microseconds, and simply divide it by the real-time duration of a single clock tick on

the counter defined as the application time base. For example, if the application time base operates at 200 Hz, each tick of that counter represents an interval of 5 milliseconds. Thus, a real time period of 500 milliseconds is equivalent to 100 ticks of the application time base counter (500 msec divided by 5 msec per tick).

To calculate the value of a number of counter ticks in engineering units, reverse the previous process. Multiply the number of counter ticks by the value of each counter tick in engineering units. To illustrate, consider a gas meter that has accumulated 1200 ticks. If each tick represents 1.5 cubic feet of gas, then the conversion becomes 1200 ticks multiplied by 1.5 cubic feet/tick, yielding a volume of metered gas of 1,800 cubic feet.

These methods, while correct, can lead to problems should a change occur to the specification of the counter tick frequency of the time base counter. It is better to associate a symbol with the value of the number of engineering units per tick such that it can be used in the application code without regard to the actual value. It is the responsibility of the user to make the definition of such a conversion symbol. However, a single exception exists. For the application time base counter, **RTXCgen** defines a standard symbol, `CLKTICK`.

RTXCgen calculates `CLKTICK`, the value of a tick in the application time base counter, and puts it in the **`kproject.h`** header file. The previous example of converting time to ticks on the application time base counter serves as a good illustration. In that example, **RTXCgen** defines `CLKTICK` to have a value of 5, representing 5 msec/tick. The following expression converts a 500 msec period to counter ticks on the application time base counter:

```
500/CLKTICK
```

This method of reduction makes application code more robust. First of all, the conversion occurs at compile time and does not require any runtime cycles. Second, if the specification of the number of engineering units per tick changes, it is necessary only to recompile the application code to adjust any real-world values.

Application Time

Using **RTXCgen**, the user can specify one counter in the system to be the application time base whose parent event source should be a periodic event. Other than that specification, the counter is a normal **RTXC** counter in all respects. The counter's accumulator is a value representing the current application time.

Because the **RTXC** Kernel makes no stipulation about which counter the user can specify for the application time base, a special construct allows the user to develop program code independently of the counter definition. The special construct is a counter handle of zero (0), which instructs an **RTXC** Kernel service to use the user-defined counter handle for the operation. **RTXCgen** defines a macro for this special construct in the **`rtxcapi.h`** file as follows:

```
#define TIMEBASE (COUNTER)0;
```

If you build library routines that use current time but do not know the actual definition of the application time base counter, you should include **`rtxcapi.h`** in your code modules. Then refer to the counter for current time as `TIMBASE`.

System Time

Some applications require certain time management at a resolution much lower than one tick of the application time base counter. Quite often, it is desirable to measure alarm periods in seconds, or minutes or even hours or days. Time of this magnitude is usually called system time or real time, or even clock time. The difference between application time and system time is simply the granularity of the tick for each type.

There is no specific provision in the **RTXC** Kernel for maintaining system time. However, creating a counter to serve that purpose is quite easy. Simply define another counter using the same periodic event source parent as used for the application time base counter. Set the modulus property of the second counter to a value that ratios the event ticks to give a counter tick at the desired frequency. When a real time period is needed for a time period calculation, simply make

reference to the second counter, using one of the conversion methods already described.

For example, if the application requires a system time counter that has a resolution of 1 Hz, and the event source provides event ticks at a frequency of 10,000 Hz, the modulus of the system time counter needs to be 10,000 to deliver one counter tick per second, or every 10,000 event ticks. Such a counter serves as an effective and accurate calendar.

Functions exist in most C libraries that can convert a calendar date and time-of-day into a value equivalent to the number of seconds since a base date, usually Base Universal Time, which begins January 1, 1970. Using such a routine to calculate that time difference as a number of seconds provides a very deterministic method of maintaining a calendar with one-second accuracy. To seed the system time counter with such a value, make the necessary conversion and pass it to the system time counter using the `XX_SetCounterAcc` service.

Converting Calendar Date To System Time

ANSI C specifies the `mktime` library function for converting a `struct_tm` type value to a `time_t` type value using a base date of January 1, 1970. Any valid date on or after January 1, 1970 and before March 2038 yields a correct value in 32-bits.

Note: The **RTXC** Kernel does not require the calendar to be defined with a date and time to operate properly.

Converting System Time To Calendar Date

ANSI C also defines the `gmtime` and `localtime` library functions and other functions for converting `time_t` type values to `struct_tm` type values. These functions convert the system time date to a structure containing the calendar date and time-of-day.

Using Counters

Counters are the second element in the event management hierarchy establishing a system by which tasks and threads activate in response to expiration of alarms. Each counter has a parent event source and several counters may share the same parent. Counters may be static or dynamic. **RTXC** Kernel services exist to define, enable or disable, read or set the accumulator for, perform all the updating of the accumulator for, and process the associated alarms. Counters are globally accessible to tasks and threads.

Reading Counter Ticks

The **RTXC** Kernel allows the user to gain access to a counter's accumulator through the `XX_GetCounterAcc` service. The `XX_GetCounterAcc` kernel service returns a `TICKS` type value representing the current value of the counter's tick accumulator. The service does not change the content of the counter's accumulator.

Elapsed Ticks

There are many applications that need to know the number of counter ticks that occur between two events. The **RTXC** Kernel can easily provide that information. The operation requires two kernel service calls. At the first event, the first kernel call sets up a variable that contains the value of the specified counter's tick accumulator at that instant. At the second event, a second kernel service subtracts the counter ticks at the first event from the current value of the counter's tick accumulator. The `XX_GetElapsedCounterTicks` service returns the difference to the calling task and also updates the variable to the current value of the counter's tick accumulator to prepare for the next event. The computed tick difference is accurate to less than one tick's equivalent of the real interval. Example 5-3 on page 73 shows a code model for using the system time base counter, `TIMEBASE`, to measure elapsed time between successive occurrences of an event associated with the `XEVENT` semaphore.

Example 5-3. Computing Elapsed Time between Two Events

```
#include "rtxcapi.h"    /* defines TIMEBASE */
#include "kproject.h"   /* defines CLKTICK */
#include "ksema.h"      /* defines XEVENT  */

TICKS cticks, diff;
int elapsed_time;

...initialize the task

 do something

/* then wait for first event */
KS_TestSemaW (XEVENT);     /* use XEVENT sema */

/* got first event, now initialize tick counter */
KS_GetElapsedCounterTicks (TIMEBASE, &cticks); /* ignore return value
*/

for (;;)  /* loop for 2nd & successive events */
{
   /* wait for next event */
   KS_TestSemaW (XEVENT);

   /* got next event, now compute tick difference /
   diff = KS_GetElapsedCounterTicks (TIMEBASE, &cticks);

   elapsed_time = diff * CLKTICK;   /* calculate elapsed time */
   do something with the elapsed time
}
```

Note: For the second and subsequent events in Example 5-3, each call to `XX_GetElapsedCounterTicks` returns the elapsed time between the current and previous events.

Introducing Alarms

RTXC alarms are the lowest level of the event counting hierarchy and also the closest of the three classes to threads and tasks. Counters accumulate counter ticks and alarms represent the points at which certain values of accumulated counter ticks cause some thread or

task action to happen. If a counter accumulates ticks from a periodic source, the counter can represent time and associated alarms are time-based. Other counters perhaps accumulate a count of irregular events and the units of the counter, and of the associated alarms, are more process specific.

Application code uses **RTXC** Kernel services that operate directly on the alarms to establish points of action with respect to the counter accumulator. These are called *general* alarms. Other kernel services make use of the counters to set up internal alarms for use as a limitation component of their function. These alarms are referred to as *internal* alarms, also called *tickout* alarms. When the parent counter counts time, internal alarms are called time-outs. However, other types of counts can be used with equal ease for the same purpose if the counter is not time-based.

The **RTXC** Kernel employs a generalized scheme using one-shot and cyclic alarms. The kernel can manage multiple alarms simultaneously and one or more alarm points can occur at the same counter accumulation value. Kernel services for scheduling and canceling alarms are an integral part of the kernel service library. Regardless of their number, the time to service an active alarm is fixed and, therefore, deterministic.

Alarm Management

The sources of events that eventually become counter ticks on the various counters are particular to the implementation of the target system. They may be external, such as an interrupt from the system time event source, or an internal event within the application code.

The **RTXC** Kernel manages all alarm operations in units of ticks. If the alarm's parent counter ticks occur at a regular frequency, they represent a passage of time. The user defines the value in engineering units of one tick.

The basic rule of alarm management in the **RTXC** Kernel is as follows:

> **Rule: All RTXC alarms are measured in counter ticks with respect to their parent counter.**

Counter ticks serve three purposes in conjunction with **RTXC** Kernel-based alarms:

General purpose alarming	Synchronizes a task or schedules a thread with an event that occurs after a certain amount of ticks occur on a particular counter.
Internal alarming	Permits certain kernel services to limit blockage of the calling task for a specific number of counter ticks.
Elapsed tick counting	Permits the **RTXC** Kernel to compute the number of ticks that occur between two events.

It should be noted when using one-shot, general purpose, or internal alarms, especially those used with synchronous counters, that there is a possible error that can result. The problem occurs because the alarm is practically never activated at the moment its parent counter receives a tick. Instead, activation normally occurs at some indeterminate point between the last tick received and the next counter tick, resulting in an actual duration of the one-shot alarm period that is less than expected. An alarm period of one tick readily illustrates this problem.

Figure 5-3 shows the relationship, using time, between the point in time that a one-tick alarm goes active (that is, the alarm is armed) and the occurrence of the next tick of the counter.

Figure 5-3. Possible Duration of a 1-Tick Alarm Period, Case A

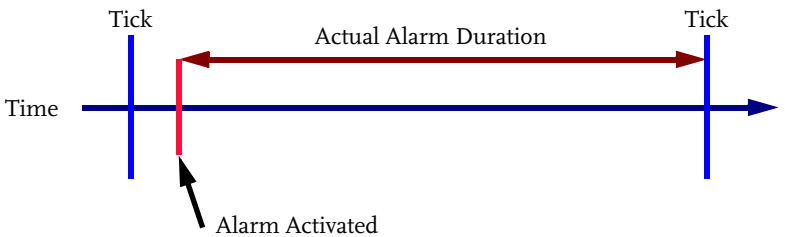
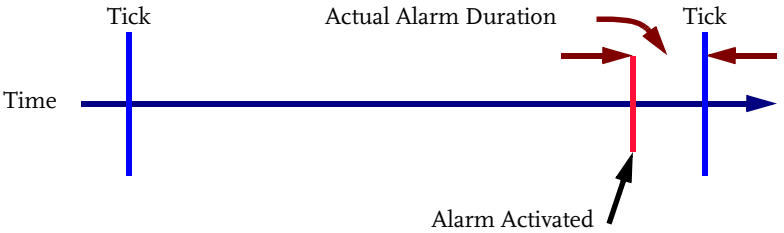


Figure 5-4 on page 76 depicts the same one-tick alarm count but for a different arming point between the two ticks.

Figure 5-4. Possible Duration of a 1-Tick Alarm Period, Case B



In both figures, the desired (or expected) alarm duration is one tick but the actual alarm duration is less than that. In Figure 5-3, the duration is almost one full tick but not quite. In Figure 5-4, the actual duration of the alarm is very much less than one tick. This is an end-point consideration of which the user should be aware when defining a one-shot alarm period, whatever the duration. If the specified duration of the alarm is greater than one tick, the possible error only occurs at the first tick. Subsequent ticks are not in error. Therefore, the following rule applies:

> **Rule: A one-shot alarm period is not necessarily synchronous with the parent counter and may result in an error or undesirable results.**

Alarm Definition

The kernel refers to an alarm by its handle, which is an `ALARM` type value. You can define, during system generation, any combination of static and dynamic alarms up to a total number allowed by a datum of the `ALARM` type.

An alarm handle must be within the range of the total number of alarms defined for the application. There is no difference between the handle of a static alarm and a dynamic alarm.

Internal alarms have no identity with respect to the application and no kernel services exist with which to manipulate or directly access internal alarms. The **RTXC** Kernel automatically manages the creation and destruction of internal alarms.

One main rule applies to defining alarms:

Rule: Internal alarms are not declared during the system generation procedure.

Alarm Properties

Alarms have four properties that are available to the developer. In addition to a property specifying its parent counter, each alarm object contains two tick values: one to define the expiry point of the first alarm after the alarm becomes active (the initial alarm), and another to define the next alarm point increment if it is a cyclic alarm. There is also an *attributes* property that the kernel uses for special modes of operation.

While not directly available to the developer, an alarm has three more important properties: its state, the active alarm expiry point, and the list of tasks waiting for the alarm's expiration. The Alarm object class properties permit the inclusion of up to two optional semaphores. One allows a semaphore association with the alarm expiration. The second associates a semaphore with an alarm abort operation.

A task or thread can access the definable counter properties through an `ALARMPROP` structure, shown in Example 5-4.

Example 5-4. Alarm Properties Structure

```
typedef struct
{
   KATTR attributes;   /* reserved for future use */
   COUNTER counter;    /* Handle of alarm's parent counter */
   TICKS initial;      /* initial expiry point of alarm */
   TICKS recycle;      /* recycle count if cyclic alarm */
} ALARMPROP;
```

When defining an alarm with the `XX_DefAlarmProp`, the kernel maintains the value of *initial* to compute the alarm's initial point of expiry whenever it becomes an active alarm.

The `XX_GetAlarmProp` kernel service reads the current developer-accessible properties of the given alarm and puts them in an `ALARMPROP` structure.

An alarm having no cyclic tick counts (*recycle* = 0), is a one-shot alarm. An alarm having a non-zero value for *recycle* is a cyclic alarm. In a cyclic alarm, the value of *initial* may or may not be equal to the number of ticks in *recycle*. The first cycle of the alarm after it is armed uses the *initial* value unless it is zero (0), then subsequent alarm cycles use the *recycle* value. The following rule applies to alarms:

Rule: Defining an alarm's properties establishes the relationship of the alarm with a parent counter.

Rule: Defining the properties of a alarm object does not start the alarm.

Figure 5-5 and Figure 5-6 show the relationship of *initial* and *recycle* properties to alarm expiry points, C_i, for typical values of *initial* and *recycle*. Figure 5-5 shows a one-shot alarm starting at the counter accumulator value of C_0 with a non-zero value of *initial* and a *recycle* value of 0. The point of expiry of the alarm is C_1.

Figure 5-5. One-Shot Alarm

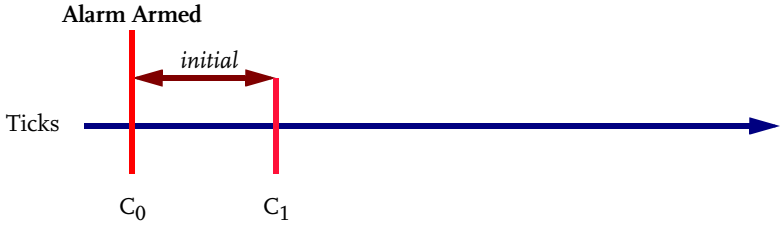
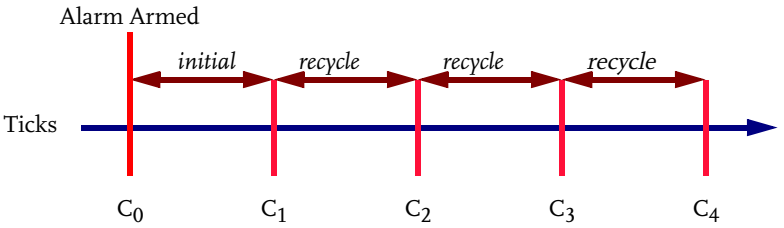


Figure 5-6 on page 79 shows a cyclic alarm starting at the counter accumulator value of C_0 with non-zero values for both *initial* and *recycle*. The point of expiry of the first alarm is C_1 followed successively by alarms at C_2, C_3, C_4, and so on.

Figure 5-6. Cyclic Alarm



Alarm Attributes

The alarm *Waiter_Mode* attribute defines how the **RTXC** Kernel manages tasks waiting on an alarm's expiration. By definition, the kernel sets this attribute to `ATTR_FIFO_ORDER` and adds tasks in chronological order to the list of tasks waiting for the expiration of the alarm. This setting offers the most efficient for processing because all waiting tasks receive notice upon expiration of the alarm.

The alarm *Waiter_Mode* attribute has no effect on **RTXC** threads.

Counter

The *counter* property defines the handle of the alarm's parent counter. The kernel calculates all points of expiry for the given alarm using the counter ticks from the parent counter's accumulator and either the *initial* or *recycle* property.

Initial Tick Count

If the value of *initial* is non-zero, it represents the increment of ticks to add to the value of the parent counter's accumulator to establish the point of expiry of the first alarm after the alarm becomes armed (active). If the value of *initial* is zero, the kernel does not use it but uses the value of *recycle* instead.

Recycle Tick Count

For one-shot alarms, the value of the *recycle* property should be zero. If the alarm is cyclic, this property contains a non-zero value that the kernel uses to determine the next point of expiry. When the current alarm expires, the kernel uses the value of *recycle* and the parent

counter's accumulator to compute the point of expiry of the next alarm.

Alarm State

A alarm must exist in one of two states, *Alarm_Active* or *Alarm_Inactive*. All alarms initialize as being *Alarm_Inactive*. When application code arms an alarm, the alarm's state becomes *Alarm_Active* and remains so until it expires or a task or thread stops or aborts it with a call to the `XX_AbortAlarm` or `XX_CancelAlarm` kernel services. The state of a one-shot alarm becomes *Alarm_Inactive* when it expires. After `XX_AbortAlarm` or `XX_CancelAlarm` stops an active cyclic alarm, the alarm's state becomes *Alarm_Inactive*. The following rule applies to alarms:

Rule: Application code (thread or task) must not manipulate any properties of an active alarm.

Optional Properties

Each general purpose alarm inherits the optional *Semaphores* property of the Alarm object class as defined by the user during the system generation procedure. If the class allows it, an alarm object supports up to two semaphores: one associated with the *Alarm_Expiration* (AE) event and the other related to the *Alarm_Abort* (AA) event. The `KS_DefAlarmSema` kernel service associates these semaphores with the alarm. A task may define an alarm semaphore without regard to the state of the alarm.

Alarm_Expiration Semaphore

The *Alarm_Expiration* (AE) semaphore, if defined, receives a signal when the alarm expires, in support of the following rule:

Rule: Expiration of a general purpose alarm is an event.

In typical operation, it is not necessary to associate a semaphore with an alarm expiration to synchronize one or more tasks with the alarm. The **RTXC** Kernel handles synchronization of alarm expiration with tasks waiting on the event without requiring a semaphore. The AE semaphore is typically used in conjunction with a task waiting on multiple semaphores associated with various events.

The AE semaphore also finds utility as an easy means of setting up a software watchdog to rearm a one-shot alarm to prevent its expiration. Tasks in the application use the `XX_RearmAlarm` kernel service to give the watchdog alarm a new point of expiration tick count before the alarm has a chance to expire. Should the watchdog alarm expire, the signal to the AE semaphore indicates there may be something amiss in the system. The task detecting the AE event must deal with any special recovery or system restart operations. One rule applies:

Rule: Only a task can receive notification that an alarm expiration (AE) event has occurred.

Alarm_Abort Semaphore

The *Alarm_Abort* (AA) semaphore, if defined, receives a signal if the alarm is prematurely stopped by a call to the `XX_AbortAlarm` kernel service. A task can use the `KS_TestSemaW` kernel service as part of a simple synchronization with the AA event. A task may also include the AA semaphore in a group of semaphores on which the task uses the `KS_TestSemaMW` kernel service to wait for any event in the group to occur. One rule applies:

Rule: Only a task can receive notification that an alarm abort (AA) event has occurred.

Using Alarms

General purpose alarms are useful for managing events with respect to the number of ticks accumulated on the alarm's parent counter. Alarms may be static or dynamic and one-shot or cyclic. **RTXC** Kernel services exist to start them, restart them, and stop them. The **RTXC** Kernel supports the ability of both tasks and threads to use alarms. Tasks can use alarms for task activation using either one-shot or cyclic alarms. The kernel permits a task or thread to use more than one timer at the same time. Threads can only activate alarms. Unlike a task, threads cannot wait for an alarm because a thread is not allowed to wait. However, through the use of thread gates, you can schedule threads as a result of alarm expiration.

Alarm Creation

The **RTXC** Kernel requires that an alarm exist before application code can use it. Specifying a static alarm and defining its properties with **RTXCgen** during system initialization creates the static alarm.

On the other hand, a task must specifically create a dynamic alarm before using it. If the *Dynamics* attribute is enabled for the Alarm class, a task creates a dynamic alarm by opening it and then defining its properties. When opening a dynamic alarm, the task may or may not assign a name to the alarm. Assigning a name is sensible when the name can be used by other tasks. However, if the alarm is to be used solely within the scope of the requesting task, assigning a name has little consequence other than to require memory space.

Because a task can create and use more than one alarm concurrently, it is good practice to have the task open all of the dynamic alarms it needs before starting the main body of the task. The task may use the alarm handle in subsequent alarm management kernel services. Any task using alarms should maintain the handle of each dynamic alarm until such time as the alarm is closed by the `KS_CloseAlarm` kernel service.

Allocating and defining dynamic alarms early typically ensures they will be available when their first use occurs. However, even in a design where dynamic alarms are created early, a condition may arise where no dynamic alarm is available when a task attempts to open one. Should this occur, the task must handle the situation and take corrective action. An unsuccessful alarm opening may indicate the presence of a problem elsewhere in the system. However, the absence of available dynamic alarms usually results from improper configuration.

Example 5-5 on page 83 shows a code fragment in which a task opens and defines a dynamic one-shot alarm without a name. The alarm's parent counter is the application time base, `TIMEBASE`, and the initial period is 500 msec.

For static alarms, the process is much simpler. The user defines the properties of the alarm using **RTXCgen**. During system startup, the initialization procedure defines the properties of all static alarms through repetitive calls to the `XX_DefAlarmProp` kernel service.

Subsequently, all that is required to use a static alarm is to arm it at the appropriate point in the application code.

Example 5-5. Creating a Dynamic Alarm

```
#include "rtxcapi.h"    /* defines TIMEBASE */
#include "kproject.h"   /* defines CLKTICK */

ALARM dynalarm;    /* gets handle of dynamic alarm */
static ALARMPROP aprop;

...task operations

/* open new dynamic alarm and verify it exists */
if ((KS_OpenAlarm ((char *)0, &dynalarm)) != RC_GOOD)
{
   ...failure to open alarm. deal with it
}
else   /* alarm opened successfully. */
{
   /* now define its properties */
   aprop.attributes = 0;
   aprop.counter = TIMEBASE;
   aprop.initial = (TICKS)500/CLKTICK;
   aprop.recycle = (TICKS)0;
   KS_DefAlarmProp (dynalarm, &tprop);

   /* alarm okay to use now */
}
```

Arming an Alarm

After the application code defines an alarm's properties, it may use the alarm by calling the `XX_ArmAlarm` kernel service. Arming an alarm changes its state to *Alarm_Active* and sets up the alarm's initial expiry point using the value of the *initial* property as previously described. The following rule applies to arming an alarm:

Rule: Application code may not arm an active alarm.

To extend Example 5-5 to include the task starting the dynamic periodic alarm, or to activate a static alarm, simply add the following statement at the appropriate point in the task's code:

```
KS_ArmAlarm (dynalarm);
```

If a thread is activating the alarm, use the `XX_ArmAlarm` Zone 2 kernel service.

Rearming an Alarm

The `XX_RearmAlarm` kernel service redefines the values of the *initial* and *recycle* properties of a given alarm and establishes a new point of expiry of the alarm based on the redefined value of *initial*. Rearming an alarm is possible without regard to the alarm's state and has no effect on the alarm's `AE` or `AA` semaphores, if defined.

Consider that the dynamic one-shot alarm created in Example 5-5 on page 83 is actually a watchdog alarm (WDT). Assuming another task knows the alarm's handle, it can reset the WDT by a single **RTXC** Kernel service. Example 5-6 shows the code model for rearming the WDT.

Example 5-6. Rearming a Software Watchdog Alarm

```
#include "rtxcapi.h"
#include "kproject.h"    /* defines CLKTICK */

ALARM dynalarm;
ALARMPROP aprop;

/* alarm has already been created and armed */

if (KS_RearmAlarm (dynalarm,(TICKS)500/CLKTICK,(TICKS)0) == (TICKS)0)
{
   ...alarm had already expired. do something
}
else
   ...WDT restarted. Continue
```

Alarm Expiration

Alarm expiration is an event on which one or more tasks may synchronize. The **RTXC** Kernel supports a design that allows

multiple tasks to wait for the same alarm to expire using the KS_TestAlarmW kernel service. However, unlike some other object classes, the following rule applies:

Rule: The kernel unblocks all tasks waiting on a alarm's expiration whenever expiration occurs.

In Example 5-7, a task arms the TIMERX static cyclic alarm and uses its expiration to synchronize the task's normal operation. The possibility exists that other tasks can cancel or abort the alarm.

Example 5-7. Waiting for Alarm Expiration with Possibility of Alarm Cancel or Abort

```
#include "rtxcapi.h"
#include "kalarms.h"

...task operations

KS_ArmAlarm (TIMERX);     /* arm cyclic alarm */

/* do the following forever */
for (;;)
{
   /* wait for alarm to expire */
   if (KS_TestAlarmW (TIMERX, (TICKS *)0) != RC_GOOD)
   {
      ...alarm was inactive or was aborted. Treat it specially
   }
   else
      /* alarm expired. do task operations */
}
```

In Example 5-7, a task arms the TIMERX static cyclic alarm and uses its expiration to synchronize the task's normal operation. In Example 5-8 on page 86, there is no other task that can cancel or abort the alarm.

Example 5-8. Waiting for Alarm Expiration without Possibility of Alarm Cancel or Abort

```
#include "rtxcapi.h"
#include "kalarms.h"

...task operations

KS_ArmAlarm (TIMERX);     /* arm cyclic alarm */

/* do the following forever */
for (;;)
{
   /* wait for alarm to expire */
   KS_TestAlarmW (TIMERX, (TICKS *)0);

   /* alarm expired. do task operations */
}
```

Aborting an Alarm

Sometimes a task needs to stop an active alarm prematurely. The `XX_AbortAlarm` kernel service cancels the specified alarm and signals the alarm's `AA` semaphore, if defined. This function, if successful, makes the alarm inactive and returns the number of ticks remaining until the alarm would have reached its point of expiration. If the task attempts to stop an inactive alarm with `XX_AbortAlarm`, the kernel service returns a value of zero (0) to indicate the alarm's state.

Freeing Alarms

A task may determine that it no longer needs a dynamic alarm. The `KS_CloseAlarm` kernel service releases the handle of the alarm and frees the RAM used by the alarm object. The following rule applies:

Rule: A task can free only dynamic alarms.

Reading Ticks Remaining on a Alarm

The **RTXC** Kernel provides two ways of determining the number of ticks remaining on the current alarm. One such service is `XX_GetAlarmTicks` and the other is `KS_TestAlarm`. Both kernel services read the number of counter ticks remaining on the active

alarm and return the number to the caller. The second service however, also determines the state of the alarm and returns a corresponding indication. If the `KS_TestAlarm` service returns an indicator that the alarm is inactive and the remaining time is non-zero, the alarm was stopped previously by a call to the `XX_AbortAlarm` or `XX_CancelAlarm` kernel service.

Actions Taken at Alarm Expiry

The **RTXC** Kernel provides ways for alarms to act on threads as a result of alarm expiry. These actions permit threads to use alarms effectively even though a thread cannot wait, as a task can, for the alarm to expire. Two kernel services exist for this purpose. The first, `XX_DefAlarmAction` allows a task or thread to set up one of two basic actions to take when the alarm expires. Once defined, the definition remains in place until it is changed by another call to `XX_DefAlarmAction` or until the alarm becomes inactive.

Detection of alarm expiry occurs in the zone that calls `XX_ProcessEventSourceTick`. If an exception handler calls the kernel service, then the resulting action must be called from that same zone. The situation is identical when a thread calls the `XX_ProcessEventSourceTick` kernel service. Because there are two zones from which the two basic expiry actions can take place, the kernel services have to allow for four actual action definitions.

The two basic actions are:

- Schedule a thread.
- Decrement a thread's thread gate.

When the action is defined, the `XX_ProcessEventSourceTick` kernel service executes the internal functions to perform one of these two basic services upon determining the alarm has expired. The internal functions are dependent on the zone of the caller to the `XX_ProcessEventSourceTick` kernel service.

One action, scheduling a thread, has direct effect. The specified thread is scheduled and has only to wait until the **RTXC/ss** Scheduler gives it control of the CPU. If the specified action is to decrement the thread's thread gate, the thread may or may not become ready, depending on the value of the thread gate.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X