SECTION 1
INTRODUCTION
Table of Contents

SECTION 1

INTRODUCTION

BACKGROUND

RTXC, the **R**eal-**T**ime e**X**ecutive in **C**, is an efficient software framework with which to develop real-time embedded systems on a broad range of micro-processors, microcontrollers, and DSP processors. The RTXC Application Program Interface (API) has understandable Kernel Service names which make it easy to learn and easy to use. That ease of use translates to less time involved with system matters and more time to spend on developing the application.

Based on concepts developed by Dr. E.W. Dijkstra in the mid-1960s with an implementation history dating from 1978, RTXC provides a sound foundation for the solution of complex real-time systems. It is based on the concept of preemptive multitasking which permits a system to make efficient use of both time and system resources.

As a system developer, however, it is our belief that you cannot be in control of your product without access to the most fundamental software in the system, the real-time kernel. Therefore, it is our policy to license and distribute RTXC in source code form only.

If you use RTXC in a product in compliance with the terms of E.S.P.'s software license agreement, you may do so without payment of royalties for your continued use of it regardless of the number of instances of use or the number of products into which it is integrated.

FEATURES

RTXC provides many features which are designed to support real-time, multitasking systems. These features include:

- Multitasking with preemptive task scheduling
- Round Robin and Time-Sliced scheduling within same priority level
- Support for static and dynamically created tasks
- Fixed or dynamically changeable task priorities
- Intertask communication and synchronization via semaphores, messages, and queues
- Efficient timer management
- Timeouts on many services
- Management of memory
- Resource management
- Fast context switch
- Small RAM and ROM requirements
- Standard programmer interface on all processors
- Highly flexible configuration to permit custom fit to the application

USER'S MANUAL SUMMARY

This User's Manual is not intended to be a tutorial on real-time kernels in general. We assume that you know the fundamentals of multitasking. We will try to flesh out that knowledge by explaining the "inputs" and "outputs" of RTXC as a software component of your application.
The RTXC User's Manual is divided into nine (9) separate sections which are summarized below.

Section 1 of this manual is an overview of RTXC.

Section 2 gives the RTXC Theory of Operations.

Section 3 details the organization and content of the various RTXC control and data structures.

Section 4 is a brief tutorial on using RTXC.

Section 5 is the RTXC Kernel Service Reference giving a description of each Kernel Service.

Section 6 deals with the use of RTXCgen to generate and configure RTXC systems.

Section 7 discusses device drivers with emphasis on interrupt service routines.

Section 8 introduces the system level debug utility RTXCbug.

Section 9 is for Application Notes which may be sent to you periodically.

RTXC AS A SOFTWARE COMPONENT

RTXC is furnished as a set of C language source code files. In its distribution form, it is not executable. You must compile it and link it with the object files of your application programs, the system configuration files, as well as the object form of any device drivers peculiar to the application.

You should treat RTXC as any other software library. It is not necessary that you know *how* RTXC performs its functions internally. Rather, you need only know *what* Kernel Services of RTXC to use to achieve a desired result. Thus, RTXC becomes much like a large scale integrated circuit component in the hardware. Knowledge of what inputs produce what outputs is all that is needed to use the part successfully.

Unlike a chip, however, RTXC users have access to its source code to supplement their usage knowledge or to resolve technical issues about RTXC internals. With the source code, users may even wish to extend the functionality of RTXC by the addition of new services. The result is that an RTXC user can exercise complete control over the application.

RTXC LIBRARY CONFIGURATIONS

RTXC is distributed in three source code configurations defined by the set of Kernel Services embodied in each. The different configurations are available to meet the real needs of the embedded systems marketplace where there is a wide diversity of functional capabilities required in a real-time kernel. RTXC allows you to license the source code library that most closely fits your needs. If you need more capabilities later on, there is a simple upgrade path.

The three source code libraries, Basic, Advanced, and Extended, are compatible with each other. All of the services in the Basic Library are included in the Advanced Library. And all of the Advanced Library is part of the Extended Library. If you have obtained a license for the Basic Library, you may upgrade to either the Advanced or Extended Library without changing the application programs developed with the Basic Library.

The Kernel Service descriptions in Section 5 will indicate to which RTXC configuration each Kernel Service belongs. The method is explained in the following paragraphs.

BASIC LIBRARY

The Basic Library, RTXC/BL, consists of the fundamental operations you need to be able to use all classes of RTXC system components, tasks, messages, mailboxes, queues, semaphores, memory partitions, and timers. In the Kernel Services Reference in Section 5, you will see the following symbol for a Kernel Service in the Basic Library. Note that the three check marks indicate that the service is to be found in all three RTXC source code configurations.

ADVANCED LIBRARY

RTXC/AL, the RTXC Advanced Library, augments the RTXC Basic Library with additional Kernel Services. Most of the additional functions are related to allowing a task to perform some synchronous operation with some system resource. The following symbol indicates that the corresponding Kernel Service is part of the Advanced and Extended Libraries only. It is not found in the RTXC/BL configuration.

EXTENDED LIBRARY

The Extended Library, RTXC/EL, contains the full complement of RTXC services. The additional Kernel Services offered in the Extended Library implement the services with timeouts. The single check in the symbol below indicates that an associated Kernel Service is only in the Extended Library. It is not in either the Basic or Advanced Libraries.

TARGET ENVIRONMENT

RTXC is designed to operate in an embedded system, that is, one which is intended to perform a defined set of jobs with little or no operator intervention. No assumptions are made about the configuration of the target system. It is the responsibility of the user to define the target environment and to insure that all necessary devices have program support.

You will also find that there is a test application included as part of the standard RTXC distribution package. The test software, furnished in source code form, provides you with a good example of how to construct an application using RTXC.

GLOSSARY OF TERMS

API	See **Application Program Interface**.
Application Program Interface	The rules and syntax defining how an application program accesses the **kernel** and its services.
Blockage	A setting of a task's status which prevents the task from executing. In order to get control of the **CPU**, a **task** must not have anything blocking it.
CPU	**C**entral **P**rocessing **U**nit. Also a shorthand term referring to a computer.
Clock Tick	One single **event** of the system's time base indicating that a predetermined amount of time has elapsed. Clock ticks occur at regular intervals.
Clock Driver	That portion of the system which is dedicated to servicing **Clock Ticks** for the purpose of time management as required by various **Kernel Services**.
Critical Region	A section of program within which the program is vulnerable to undesirable interruption or corruption.
Current Task	The task which is currently in control of the **CPU** and is, by definition, the highest priority task in the **READY List**.

Doubly Linked List	A **Threaded List** which uses a forward link to point to the next node in the list and a backwards link pointing to the previous node in the list. End nodes are treated specially.
Executive	See **Kernel**.
Event	An occurrence of something in the process requiring a response or invocation of some action, e.g., a **Clock Tick**.
FIFO	See **First-In-First-Out**.
First-In-First-Out	A method of serving a chronological set of items, e.g., a line of waiters at a box office. Also known as "first come, first served".
Free Pool	A set of unused elements or **kernel objects** mapped by some mechanism which permits rapid removal from and insertion into the set.
General Timer	A timer used for general purpose timing and having a **Handle**.
Handle	An identifier used to indicate a particular RTXC **kernel object**.
ISR	See **Interrupt Service Routine**.

Interrupt Service Routine	A program module which deals exclusively with servicing an exception to the normal flow of processing caused by a particular **event**.
Kernel	The **multitasking** control and library of services for use by the system designer to promote an orderly implementation of the application.
Kernel Object	A data structure or object used by the kernel for purposes of task control, memory management, timer control, exclusive access, or intertask communication and synchronization.
Kernel Service	A function of the **Kernel** which performs a particular operation affecting **multitasking**.
Mailbox	A **kernel object** which serves as a repository for **messages** sent by one or more tasks and intended for another **task** or tasks.
Message	A data structure composed of a **Message Envelope** and a **Message Body** all of which is deposited into a mailbox for pickup by a recipient **task**.
Message Body	That part of a **Message** which contains application oriented information.
Message Envelope	A **kernel object** and part of a **Message** which contains information specific to RTXC necessary to route the message and provide a return path.

Message Priority	An arbitrary value which can be used to assign relative importance to a message at runtime.
Microcontroller	A single semiconductor package containing a combination of a **CPU** and several related peripheral devices which form a complete computer system capable of operation with minimal external circuitry. Microcontrollers are ideally suited to embedded systems.
Microprocessor	A **CPU** which may or may not have some basic peripheral devices combined into a single part. Usually microprocessors are perceived to be more powerful than microcontrollers.
Partition	An RTXC **kernel object** which plays a part in managing **Free Pools** of **RAM**.
Queue	An RTXC **kernel object** which operates as a FIFO buffer for chronological processing of data.
RAM	Acronym for **R**andom **A**ccess **M**emory. RAM is volatile memory and is only valid when power is supplied. It may be either static, i.e., not requiring refreshing, or dynamic, which requires periodic refreshing to maintain content. RAM is usually used to store variable data which must be both written and read.
READY List	An RTXC **Doubly Linked List** which links the

	TCBs of each **task** having no **Blockage**. The first entry in the list is the TCB of the **Current Task**.
ROM	Acronym for **R**ead **O**nly **M**emory. ROM is non-volatile memory when power is not present. It may be read but not written under program control. ROM is most useful for storing programs.
Resources	An RTXC **kernel object** used to designate some data element, structure, device, or other entity for which one task at a time may need exclusive access for brief periods.
Semaphore	An RTXC **kernel object** which is associated with an **Event** and whose content yields information about the state of the event.
Signal	An action applied by a **Kernel Service** to a **Semaphore** to indicate the occurrence of the associated **Event**.
Singly Linked List	A **Threaded List** consisting of a single link to the next node in the thread.
TCB	See **Task Control Block**.
Task	The fundamental program element, usually represented as a C function, in a system operating under RTXC. An application may be composed of one or more tasks.

Task Control Block	An RTXC **kernel object** containing everything about a task needed to permit any supported multitasking scheduling discipline.
Task Number	A number used to identify a **Task Control Block** and, by implication, the task associated with that **TCB**.
Task Priority	A number defining the relative importance of the task with respect to all other tasks in the application. The higher the priority, the more time-critical the task.
Threaded List	A data construct in which one data structure, or node, is linked to another data structure by one or more address pointers. The list is headed by a pointer to the first node. The first node contains the address of the second node in the list, etc. If the thread pointer is NULL, usually zero, the thread is said to be EMPTY.
Waiter	A **task** which is waiting on the availability of a **kernel object**, RTXC data element, or an **Event**.

SECTION 2
THEORY OF OPERATION
Table of Contents

SECTION 2
THEORY OF OPERATION

WHAT IS A REAL-TIME KERNEL?

A real-time kernel, also called a real-time executive, is a program which implements a set of rules and policies about allocation of a computer system's resources. Policies are those principles which guide the design. Rules implement those policies and resolve policy conflicts. Neither can be violated without indeterminate or catastrophic results to the system's operation.

An example of a policy would be that the design must be deterministic, i.e., predictable. A rule example might be that threaded lists permitting random order of node insertion and/or deletion shall be implemented as doubly linked lists. This is an implementation of the policy of deterministic design. The double links permit direct access to a node during the deletion process, thus making it a predictable procedure.

The rules permit software processes to operate and gain access to various system resources in an orderly manner. Access to the kernel's services may take several forms but is usually one of calls to subroutines or higher language functions. The kernel's services embody and enforce these rules to ensure orderliness in the application processes which use them.

RTXC POLICIES

In order to understand much of what is to follow in this manual, an explanation of the policies of RTXC is in order. If your application design conforms, you should produce an efficient system design.

Policy.....	RTXC should contain a sufficient number and types of services to make it useful to a variety of real applications.

Policy.....	RTXC should employ a multitasking design in order to achieve maximum CPU efficiency.

Policy.....	RTXC multitasking should be driven in response to system events, whether of internal or external origin.

Policy.....	The executive should support an application design composed of a set of separate but interrelated tasks each having a priority indicative of its relative scheduling importance.

Policy.....	RTXC performance should be deterministic to the greatest extent possible.

Policy.....	RTXC should have a small RAM requirement for kernel operations.

Policy	RTXC should be written in such a manner that it imposes minimal overhead to the application tasks it is governing.

RTXC BASIC RULES

The following rules attempt to implement some of the RTXC policies above. An understanding of these rules will enable you to resolve questions about how the kernel is operating.

Rule	The Current Task (i.e., the task which is currently in control of the CPU) is the highest priority task in the system which is not otherwise blocked (Ready).

Rule	The Current Task maintains control of the CPU until it runs to completion (i.e., termination), voluntarily yields, becomes blocked by unavailability of a needed resource, or is preempted.

Rule	If a task of higher priority than the Current Task becomes Ready, it preempts the lower priority task and becomes the Current Task.

Rule	The RTXC Kernel is interruptible, but not reentrant.

Rule	An Interrupt Service Routine (ISR) must not issue a Kernel Service request except for those specifically permitted in an ISR.

Rule	The Null Task is always the lowest priority task and whose priority must never be changed.

Rule.....The Null Task must always be Ready and MUST NEVER be blocked.

SYSTEM RESOURCES

The design of the kernel must be concerned with the management of certain system resources which include the CPU, memory, and implicitly, time. Each must be shared among the competing processes in such a manner that the overall function of the system is accomplished.
Sharing memory is obviously essential as it is a finite resource in the system. The CPU must be shared to increase its efficiency because it is usually much faster than the physical process it is controlling or monitoring. To have it wait on a slow process would be inefficient, thereby violating a basic system policy.

Time is the most difficult of the resources managed by the kernel as it is the most unforgiving. The design and code of Kernel Services must be such that they require minimal execution time yet are predictable. Execution speed of the various services determines the responsiveness of the system to changes in the physical process. But speed alone is not sufficient. It is equally important that each service be predictable with respect to time.

Without the predictability, a system designer would have no assurance that the timing constraints of the physical process would be met.

MULTITASKING

Multitasking is one of the major policies implemented in a modern executive. Real-time kernels of today generally make use of some part of the work done by Dr. E.W. Dijkstra in the early and mid-1960s. While multitasking was an acknowledged concept before then, it is his work which has had the most impact as it formulated a set of constructs and rules for implementing such a design.

Multitasking appears to give the computer the ability to perform multiple operations concurrently. Obviously, the computer cannot be doing two or more things at once as it is a sequential machine. However, with the functions of the system decomposed into different tasks, the effect of concurrency can be achieved.

In multitasking, each task, once given operating control of the CPU, either runs to completion, or to a point where it must wait for an event to occur, for a needed resource to become available, or until it is interrupted. Because the computer is usually much faster than the events in the physical process, efficient use of the computer can be obtained by using the time a task might wait for an event to occur to run another task.

This switching from one task to another forms the basis of multitasking. The result is the appearance of several tasks being executed simultaneously.

READY LIST

The key to multitasking is the READY List. This list is constantly being changed by various Kernel Services which insert runnable tasks or remove those which are blocked and temporarily not able to run. The READY List is actually a doubly linked list containing those tasks which are runnable (Ready) in descending order of priority. Thus, the Current Task is always the first task in the thread.

A couple of rules about the READY List are:

Rule ..... The READY List MUST never be a NULL (empty) list.

Rule ..... The Null Task must terminate the READY List.

TASKS

In RTXC, a task is a program module, a process, which exists to perform a defined function or set of functions as part of an overall application. An application usually consists of several tasks. A task is independent of other tasks but may establish relationships with other tasks. These relationships may exist in the form of data structures, input, output, or other constructs.

A task executes when the RTXC task scheduler determines that the resources required by the task are available and that no other task of higher priority is also ready to run. Once it begins running, the task has control of all of the system's resources. But as there are other tasks in the system, a running task cannot be allowed to control all of the resources all of the time. Thus, RTXC implements the policy of multitasking.

NULL TASK

The Null Task is a special task in RTXC and performs a vital service. During system initialization, the Null Task is inserted into the READY List as the first Ready task, and having the lowest possible priority. The Null Task acts as a list terminator because the RTXC Task Dispatcher knows that there is always at least one Ready task in the READY List. All other tasks will be of higher priority than the Null Task; therefore, when they become Ready, their position in the READY List will be higher than that of the Null Task.
There are more rules regarding the Null Task:

Rule	It MUST always be the lowest priority task in the system.

Rule	The Null Task MUST never become blocked for any reason.

PRIORITY AND PREEMPTION

A multitasking real-time executive promotes an orderly transfer of control from one task to another such that efficient usage of the computer's resources is achieved. Orderly transfers require that the executive keep track of the needed resources and the execution state of each task so that they can be granted to each task in a timely manner.

The key word is *timely*. A real-time system which does not perform a required operation at the correct time has failed. That failure can have consequences which range from the benign to the catastrophic. Response time to a need for executive services and the execution time of such services must be sufficiently fast and predictable. With such an executive, application code can be designed such that no need goes undetected.

Real-time systems usually consist of several processes, or tasks, which need to have control of the system resources at varying times due to the occurrence of external events. These tasks are at various times competing for system resources such as memory, execution time, or peripheral devices. They range from being compute bound to I/O bound.

Tasks which are I/O or compute bound cannot be allowed to monopolize a system resource if a more important function requires the same resource. There must be a way of interrupting the operation of the task of lesser importance and granting the needed resource to the more important task.

One way to achieve timeliness is the assignment of a priority to each task. The priority of a task is then used to determine its place within the sequence of execution of other runnable tasks. Tasks of low priority may have their execution preempted by a task of higher priority so that the latter can perform some time critical function.

EVENT DRIVEN OPERATION

An event can be any stimulus which requires a reaction from the executive or a task. Examples of an event would include a timer interrupt, an alarm condition, or a keyboard input. Events may originate externally to the processor or internally from within the software. An executive which responds to these events as the stimuli for allocating system resources is said to be event driven.

If the response time of the system to any event occurs within a period of time which can be accurately predicted and guaranteed, the executive can be said to be deterministic. By these definitions, RTXC is a deterministic, event driven, multitasking, real-time executive.

The RTXC construct associated with an event is the Semaphore. An RTXC semaphore is not a counting semaphore as defined by Dijkstra nor is it a simple binary event flag. An RTXC semaphore is a tri-state device capable of containing information about its associated event and the task waiting on the event. This points out the major rules regarding the use of RTXC semaphores.

Rule.....A semaphore can be associated with only one event at a time.

Rule.....Any event used for task synchronization must be associated with a semaphore.

Rule	Only one task at a time may use a semaphore for synchronization with the associated event.

It is considered a design error if a task attempts to synchronize with an event using a semaphore which is already in use by another task for the same purpose. The offending task receives an indication of the error and it is up to the task to handle the situation. Rather than spending programming time to adjust for the error, a better solution would be to adjust the design of the task to prevent the error.

TASK SCHEDULING

The policy of multitasking in RTXC is realized by the manner in which the various tasks are scheduled for operation. As previously stated, the RTXC Basic Rules do not enforce any specific task scheduling protocol. They only state general rules regarding preemption, CPU control, and Current Task definition.

Over many years of real-time systems development there have been three basic means (or protocols) of scheduling tasks within a multitasking policy. In fact, it could be said that there are actually only two methods with one of them having a variant. These protocols are usually called Round Robin, Time-Sliced, and Preemptive scheduling.

Round Robin

Round Robin scheduling is probably the oldest of the three multitasking methods and is also very simple in that it is essentially a polling protocol. As RTXC grants control to each such task, it is the responsibility of the task to determine if the conditions are correct for it to run and for how far. Once the Round Robin task determines that it can progress no farther due to the unavailability of some system element, it must yield control of the CPU or become blocked. If it becomes blocked, RTXC removes it from the READY List. If it yields, it can yield control only to another task of the same priority. This gives rise to the fundamental rule about Round Robin scheduling.

Rule ..... A set of tasks using a Round Robin scheduling protocol must have the same priority.

It is permissible in RTXC to have only some of the tasks in the application using Round Robin scheduling. Those that use the protocol must follow the rule above, but those not using it may have different priorities. While there may be a mixture of scheduling protocols, the RTXC Basic Rules regarding preemption still apply to Round Robin tasks.

It is important to note that because of the way Round Robin scheduling is performed, any task of lower priority cannot gain control of the CPU while Round Robin tasks are in the READY List. Therefore, the possibility exists that a task so positioned in the READY List might never gain control of the CPU.

Consider the scenario where the READY List contains four tasks, A, B, C, and D, where A, B, and C are Round Robin tasks having the same priority, and task D is lower priority. RTXC grants task A control and, following the second Basic RTXC Rule, controls the CPU until it voluntarily yields control to task B. Since there is no higher priority task in the READY List, the possibility of preemption is eliminated.

Even though it is yielding control, task A remains in a READY state and is thus reinserted into the READY List at a position following the last task,

task C, having the same priority. The READY List then contains tasks B, C, A, and D respectively.

Continuing with the scenario, task B runs and yields to task C leaving the READY List containing tasks C, A, B, and D. Task C gains control, runs for a while, and then yields control to task A. The READY List resumes its original form, which is tasks A, B, C, and D. From here on the cycle repeats.

Throughout the process, task D never gets a chance to execute because it never becomes the highest priority task in the READY List. The situation will persist until all of the Round Robin tasks, A, B, and C, become blocked.

It would be logical to assume that, at some point, only one of the Round Robin tasks, task B for example, will remain in the READY List with task D. Task D would still not be scheduled when task B attempts to yield control because of the second rule concerning Round Robin scheduling.

Rule.....The Current Task attempting to yield control will remain the Current Task unless it and the next task in the Ready List have the same priority.

By this rule, task B will remain the Current Task forcing task D to remain waiting for CPU control until tasks A, B, and C are blocked and no longer in the READY List.

The use of Round Robin scheduling, while quite simple, has important ramifications in a real-time system and should be used judiciously. Of primary importance is the fact that the tasks execute sequentially because they lack a priority differentiation. The time through any Round Robin cycle varies according to the amount of code executed in each task. Similarly, the time from when an event occurs until it is serviced or used is unpredictable because it varies according to which task is executing at the time of the event's occurrence. Thus, this method of task scheduling can be very nondeterministic. In hard real-time applications, this method should be used cautiously.

The advantage of Round Robin scheduling, however, also lies in its simplicity. With the complexity of preemption eliminated, the relationships between tasks are usually predictable.

Time-Sliced

Time-Sliced scheduling can be considered a variant of Round Robin scheduling. The difference between the two protocols is that the Time-Sliced task may only execute for some predefined quantum of time. If the task remains in control of the CPU long enough for the time quantum to expire, RTXC automatically forces the task to yield. The task may also voluntarily yield (or block) prior to the expiration of the time quantum.

RTXC does not enforce a global specification of a time quantum for all tasks. Instead, RTXC time quantums follow the rule below.

Rule.....Each task using Time-Sliced scheduling must have its own non-zero time quantum.

The amount of each time quantum can be tuned for the specific task thus making the overall system response better than for a single global specification.

Because Time-Slicing is a variant of Round Robin scheduling, RTXC has a similar rule regarding task priorities.

Rule.....A set of tasks using a Time-Sliced scheduling protocol must have the same priority.

All tasks at the same priority are not necessarily scheduled by a Time-Sliced protocol. RTXC permits there to be a mix of Time-Sliced tasks and Round-Robin tasks at the same priority. Using a mixture of scheduling protocols within the same priority level can have some pitfalls and should be employed with care.

For example, consider two tasks, A and B, to be equal priority. Task A uses Time-Slicing but task B does not. When task A exhausts its Time-Slice quantum or voluntarily yields, RTXC passes control to task B. It is possible in this example that task B may never pass control back to task A. This is a conscious decision in the design of RTXC to allow

for such a case. It is deemed the designer's choice to make, not the kernel's.

Rule	The Current Task being forced to yield control will remain the Current Task unless it and the next task in the READY List have the same priority.

This rule functions exactly as it does for Round Robin scheduling. Whether the yield is made voluntarily or is forced by RTXC, the above rule applies.

Rule	If a Time-Sliced task is preempted or blocked, the amount of time remaining on the current quantum is preserved.

The time quantum for a Time-Sliced task is cleared when the task is initially executed. Time-Sliced scheduling of a task is enabled by a Kernel Service which sets the time quantum to a non-zero value. Whenever the time-slice timer expires during Time-Sliced operation, it is reset to the current time quantum amount. If the task is preempted or blocked as the result of an RTXC Kernel Service request, RTXC does not subtract the duration of the preemption from the task's time quantum. Instead, the remaining time is simply preserved at the time of preemption or blockage, and that same amount of time is given to the task when it resumes.

Rule	It is permitted to change the time quantum of a task while the task is in the READY List.

If a task's time quantum is changed from a non-zero value to zero, Time-Slicing is disabled for that task effective the next time the task is granted CPU control. If a time quantum is changed from zero (disabled) to non-zero (enabled), then Time-Slicing is enabled with the new time slice value the next time the task is scheduled. If a time quantum is changed from a non-zero value to a different non-zero value, the new time quantum value is not effective until the old value expires. If an immediate time quantum change is required, change the time quantum value to zero, and then change it to the desired value.

Like Round Robin, Time-Sliced scheduling carries some of its own caveats. Time-Slicing should be used when it is well suited to the physical process of the application. Proper usage of Time-Sliced scheduling requires a thorough understanding of the physical processes of the application and how the various tasks in the system operate on the process.

The ability to tune the time quantum on each task can be an important element in a successful application implementation, but it can also be easily abused. Each time a Time-Sliced task's time quantum expires it requires some activity in RTXC necessary to process the forced yield. Proper selection of time quanta based on a knowledge of the process can produce a responsive system capable of producing good results even though it cannot be said to be strictly deterministic.

It is quite common to try to improve the responsiveness of individual tasks by selectively adjusting upstream time quanta, usually by making them smaller. However, if those time quanta are improperly chosen and become too small, the amount of time spent in RTXC servicing expired time quanta can become excessive, and overall system performance can degrade. This is one of the fundamental behavior characteristics of Time-Sliced scheduling.

Preemptive

Most users of RTXC will select the Preemptive protocol as the preferred method of scheduling tasks. While it supports both Round Robin and Time-Sliced scheduling, the design of RTXC's suite of Kernel Services primarily supports Preemptive task scheduling as the normal protocol. Through the use of task priorities and event driven operation, RTXC provides the basis for successful, responsive, and deterministic system design.

Unless specifically noted, the descriptions in this manual of the various functions of RTXC and its support services imply applicability to usage within a Preemptive scheduling protocol.

KERNEL SERVICES

Except for the selection and dispatching of the ready tasks, as performed by the RTXC Task Dispatcher, most of the code in RTXC is that necessary for Kernel Services. The Kernel Services are the various functions which RTXC performs when requested by an application task.

RTXC Kernel Services exist as routines which are executed by the Kernel Service Dispatcher. When a task needs some function which the kernel performs, it makes a Kernel Service Request. A Kernel Service Request takes the form of a C function call to a function which resides in the RTXC Application Program Interface Library. The purpose of the API Library is to structure the function arguments and to call the Kernel Services Dispatcher. Once there, the requested service is determined, and the corresponding kernel library function is executed to perform the requested operation.

After completing the function, control usually returns to the requesting task. However, there are circumstances during the course of performing the service where a higher priority task becomes Ready, or some system element needed by the Current Task is unavailable. If so, the Kernel Service functions may preempt the Current Task or block it and make another task the Current Task. After such an occurrence, RTXC grants control to the new Current Task instead of the one which made the Kernel Service Request.

Tasks become Ready at varying rates and are inserted into the READY List as they do so. Once there, they execute in accordance with their respective priorities. Higher priority tasks are run ahead of those of lower priority. Just as they become Ready, tasks also are removed from the READY List when they become blocked. Thus, the scheduling of tasks is very dynamic and closely related to the functions performed by the various Kernel Services.

DATA MOVEMENT

RTXC supports two primary methods of moving data from task to task: chronological and with respect to priority. Both methods require intervening constructs to provide a standard interface between the sending and receiving tasks.

For chronological data movement, the interface construct is a FIFO Queue. For movements with respect to priority, RTXC provides for bi-directional message transmission.

FIFO Queues

Queues are circular buffers which hold data entries of one or more bytes. This gives rise to the rules about queues.

Rule.....A queue has a capacity (Depth) of a predefined number of entries.

Rule.....Within a given queue, an entry has a predefined size (Width).

Rule.....When a queue has no entries, it is in an EMPTY state.

Rule	When a queue has all entries occupied, it is in a FULL state.

Entries are put into a queue by moving the data from the source into an entry slot in the queue. RTXC keeps track of the available free entry slots in the queue as it must know whether the queue is EMPTY, FULL, or in between. As the insertion procedure is chronological, the newest entry is at the end of the queue while the oldest entry is at the head of the queue. Attempting to put an entry into a FULL queue causes a condition which requires attention by the requesting task or by RTXC.

Removal of an entry from a queue involves locating the oldest entry in the queue and moving the data therein to a given destination location supplied by the requesting task. An attempt to remove data from an EMPTY queue requires extraordinary action by either the Kernel Service or by the requesting task.

Mailboxes

The interface between a message sender and the receiver task is a Mailbox. A Mailbox is a construct which promotes the orderly accumulation of messages from various senders. RTXC supports a variable number of independent mailboxes capable of containing mail from multiple senders. RTXC mail is always in the form of a message.

While more than one task may send messages to a given mailbox, the mailbox should be considered to be owned by a single receiver task. The analogy would be similar to your personal mailbox. You receive mail from many senders, but only you read your mail. By the same token, you don't look in your neighbor's mailbox.

Therefore, the rules about mailboxes are:

Rule.....Any task can send mail to any mailbox.

Rule.....A task may own none, one, or many mailboxes.

Rule.....Only one task should receive mail from a given mailbox.

Rule.....A message added to a mailbox is inserted into the mailbox's linked list according to the message's priority.

Messages

Messages are unlike queues in that the data in the messages is not moved about. Instead, pointers to the data are passed. This makes for a very efficient way to move about large volumes of data without actually having to load and store individual bytes or words of data.

Thus, the rules concerning Messages are:

Rule	A task may be both a message sender and a message receiver.

Rule	A message may be sent synchronously or asynchronously.

Rule	Message acknowledgement is an Event.

Rule	Each message has two parts: an associated envelope and a message body, both of which must be located in RAM.

Rule	Each message has a user-defined priority.
Rule	There is no defined format for a message body other than that upon which the sender and the receiver agree.

Synchronous Transmission

Messages may be sent synchronously, that is, with an automatic wait until there is an acknowledgement response from the receiver. When a task wants to send a message synchronously, it must specify a semaphore number as one of the arguments in the Kernel Service request. The reference associates the semaphore with a message acknowledgement performed by the receiver.

Once the message is linked into the specified mailbox, RTXC blocks the sender by changing its

state and removes it from the READY List. With the removal of the sender task (which was the Current Task) from the READY List, the next task in the READY List becomes the Current Task.

The receiver removes the message from the mailbox and processes it according to the content of the message body. When the receiver no longer needs the data in the body of the message, it acknowledges the message thereby making the sender task runnable again and allowing it to continue its operation.

The body of the message can also be used by the receiver to return a response to the sender. This is a very efficient way of passing data bi-directionally between two tasks with little overhead. The mechanism is quite simple.

The sender sends the message and waits for the receiver to acknowledge the message. The receiver task receives the message and, at some point in its processing, inserts a response into the message body. It acknowledges the message at an appropriate point. When the acknowledgement occurs, the sender task resumes and examines the response information in the message body as returned from the receiver. The sender then continues with its processing based on the indicated response.

Asynchronous Transmission

If the sending task does not wish to wait on the action of the receiver or if there is no response

required, it may send a message without waiting for receipt or completion of processing to be acknowledged. This makes it possible for a sender to send multiple messages to a receiver, or, simply do something else while the receiver processes the message.

Even though a task sends a message without waiting for the response, a semaphore can still be associated with the message. Doing so makes it possible for the sender to wait for the message acknowledgement event at some point subsequent to the send operation.

If the receiver completes use of the message by the time the sender waits for that event, the sending task continues operation without interruption. If the receiver has not yet completed processing of the message, the sender must wait for the event to occur. When it does occur, the sender's operation is resumed.

As for synchronous transmissions, the message body may also be used to transfer information bi-directionally between the sender and receiver tasks.

TIME

Managing time is fundamental to a real-time kernel. In RTXC, time is evidenced by the receipt of periodic interrupts from a system time base. The interrupts are referred to as *ticks* and constitute the period granularity of the device which generates the interrupts. Timer granularity, specified during system generation, may be fixed or configurable. However, once configuration of the timer device takes place during system initialization, it must not change during RTXC operation.
Timer ticks serve three purposes in RTXC:

- General purpose timing
- Timeout timing
- Elapsed time counting

General purpose timing serves to synchronize a task with an event which takes place after a certain amount of time passes.

Timeout timing permits tasks using certain Kernel Services to be blocked for a limited amount of time. This facility is quite useful in certain applications where it may be necessary to ensure that a task is not blocked for a long period of time.

Elapsed time counting permits RTXC to provide the elapsed time between any two events. There may be any number of elapsed time intervals being counted at any given moment.

The basic rules of time management in RTXC are as follows:

Rule	All RTXC time is measured in timer ticks.

Rule	The period between timer ticks is fixed once RTXC is initialized.

Rule	The period between timer ticks is configurable if permitted by the physical timer device.

Rule	Expiration of a general purpose timer is an Event.

Timer Devices

The device which generates the interrupts is particular to the hardware implementation. It may be an external timer or a timer which is "on-chip". Whatever the source, the device must provide an interrupt (a tick) at a fixed interval.

Timer Ticks

Timer ticks represent time since they occur at a fixed frequency. By counting ticks, one may calculate time with an error of less than one tick. Actual time may be reduced to RTXC ticks by a simple multiplication or division depending on the granularity of the time

base device.

MEMORY SHARING

RTXC manages RAM memory through a mapping scheme which employs a system of memory partitions. RTXC can support any number of static or dynamic memory partitions. Static memory partitions must be fully defined while dynamic partitions need only be enumerated during system generation. Dynamic memory partitions reside in a free pool until such time as they are allocated for use.

When in use, each partition is composed of any number of blocks. Within a single partition, all blocks are of the same size. While different partitions will likely employ different size blocks, more than one partition may use the same size block.

The blocks in a memory partition are initially threaded together in a singly linked list. RTXC allocates a block from a memory partition by unlinking it from the thread. The reverse process is used when freeing a block by inserting it back into the linked list.

The purpose of such a memory management scheme is to prevent fragmentation of RAM. Fragmentation is a situation which results when arbitrarily sized amounts of memory are allocated and freed from the heap. If this is permitted, at some point the heap will become so fragmented that there will not be enough contiguous memory available to fulfill a request. At that point, the system becomes non-deterministic if the request is to be fulfilled.

By definition, if a process is not predictable, it is not deterministic. The routine to reform the heap may take an indeterminable amount of time depending of the severity of the fragmentation. If a time critical process were waiting for that memory space to be allocated, an event could be missed with adverse consequences.

RTXC cannot prevent an application task from using all of the blocks in a map and asking for more. However, RTXC does provide Kernel Services which return an indication that there are no blocks available. It then becomes the responsibility of the programmer or system designer to provide the program steps which deal with the situation.

Rule.....All blocks within a given memory partition must be the same size.

Rule.....A block within a memory partition can be no smaller than the size of a pointer.

Rule.....A dynamically defined memory partition must be allocated from the pool of free partitions before it can be used.

EXCLUSIVE ACCESS

Exclusive access to some physical device, application construct, or system element is permitted by RTXC through the use of RTXC Resources. These kernel objects are similar to Dijkstra's *mutex* semaphores and perform the same function. The first rule concerning exclusive access defines a resource.

Rule ..... A resource is a logical construct associated with some entity.

Such a broad definition allows anything to be treated as a resource. Because the resource is a logical construct, there need be no physical means of seizing the entity during the period in which exclusive access is required. This introduces the concept of ownership of the entity and another rule.

Rule ..... Only the owner of a resource can access the associated entity to the exclusion of other users.

In RTXC, exclusive access to an entity is granted to a task; therefore, the owner of a resource is a task. In a multitasking environment, it is quite likely that two or more tasks may attempt to gain exclusive access to an entity. Assuming that the associated resource is unowned, ownership will be granted to the task whose request occurs first, regardless of its priority with respect to other requesting tasks.

Rule ..... A resource can be owned by only one task at any given time.

Rule.....A task may own more than one resource at any given time.

Rule.....Ownership of a resource remains with the task until such time as it voluntarily releases it.

However, assuming that the resource is owned when ownership requests are made by two or more tasks, the possibility exists that the designer may wish the tasks to wait for access to the entity before continuing. At some point, the owning task will release the resource, and RTXC will grant ownership to one of the waiting tasks according to the following rule.

Rule.....If two or more tasks are waiting to gain ownership of a resource that is already owned, the next owner will be the waiting task having the highest priority.

The rules concerning resources describe a software protocol to gain exclusive access to and to release the entity associated with the resource. A task needing an entity must first become owner of its associated resource. During the period of ownership, the entity can be used exclusively by its owner. When its need for exclusive access is finished, the owning task must then release the resource.

SECTION 3

RTXC FUNCTIONAL OVERVIEW

Table of Contents

SECTION 3

RTXC FUNCTIONAL OVERVIEW

INTRODUCTION

In the previous section, the policies and rules which constitute the theory of operation of RTXC were presented. This section puts those theories into the context of actual system function by presenting how RTXC uses its various control and kernel objects. These control and kernel objects are data structures which serve as interfaces between the kernel and the application. Knowledge of how they work is fundamental to building real-time application systems around RTXC.

This section describes these kernel objects and their interrelationships. The descriptions will include:

- TASKS
- SEMAPHORES
- MAILBOXES

- MESSAGES
- QUEUES
- TIMERS
- SYSTEM TIME

Mention will also be made of the RTXC Kernel Services which deal with these data structures. A complete presentation of the Kernel Services is found in Section 5.

TASKS

In a real-time embedded system, the system designer decomposes the overall function of the application into smaller functional entities called tasks. The nature of each task is, of course, application dependent and left to the imagination of the system designer. Tasks are the workhorse program elements as they implement the design policies about management of the application processes.

The primary purpose of a real-time kernel is to serve those tasks. The kernel provides a set of services so that tasks may react to or synchronize with events and pass data between each other. RTXC provides a complete set of functions for dealing with tasks, from their definition to the various Kernel Services on through to system level debugging.

Task Definition

Before a task may execute, it must be defined to the system along with all of its attributes. RTXC supports both static and dynamic tasks. Static tasks are those whose attributes are known before the system executes and which remain fixed for the life of the configuration. Dynamic tasks are those whose TCBs are allocated and whose attributes are defined as the result of some situation in the process which requires their existence.

Static Tasks

RTXC employs the concept of predefinition of most kernel objects among which are the various static tasks constituting all, or part, of the application. For static tasks, TCB allocation and task definition occur through use of the system generation utility, RTXCgen

With RTXCgen, the user defines a new static task or changes a characteristic or attribute of an existing static task. Information about each static task includes the various task attributes which are put into several tables. Among these tables is a task definition block which RTXC uses to build a Task Control Block when a *KS_execute()* request is made to execute a given static task. RTXC uses the TCB to manage the task while it is executing.

In addition to the task information needed for TCB, RTXCgen also permits the user to specify whether or not the task is to be started automatically and to specify its position in the starting sequence. The starting sequence number is not related to the task's identifier number or its priority. The user may also specify whether the task requires an extended context.

Dynamic Tasks

In applications where the behavior of the process requires tasks to be created or defined dynamically, the attributes of such tasks are not known before the system is generated. Instead, such tasks are created "on-the-fly" by another task, or tasks, which also specifies via RTXC Kernel Services the task's attributes and environment.

For dynamic tasks, TCB allocation and attribute definition occurs under program control. To use dynamic tasks, the user must first employ RTXC Kernel Services to allocate a Task Control Block. Because dynamic task's TCBs are allocated from a pool of free TCBs with the *KS_alloc_task()* Kernel Service, such a task may use one TCB in one instance and a different TCB in another.

After allocating a TCB for the dynamic task, the user must define the task's attributes through the RTXC Kernel Service, *KS_deftask()*. Once the attributes are defined, execution of the task may be invoked by *KS_execute()* in the same manner as for a static task.

Number of Tasks

The number of tasks which RTXC supports is determined by the system designer. Through definition of the storage quantum used for data of type *TASK*, the user defines the maximum possible number of tasks permitted in the system. Thus, in a large system, *TASK* may define a 16-bit entity theoretically permitting up to 32,766 tasks. On the other end of the scale, a microcontroller may use an 8-bit field which permits up to 126 tasks in a single system.

Due to the manner in which RTXC kernel objects are accessed, it is necessary during system generation to specify the number of dynamically defined tasks that the application is required to accommodate. RTXCgen uses that number to allocate an equal number of additional Task Control Blocks which form the pool of free TCBs from which dynamic allocations are made.

Having the actual number of static tasks, *NTASKS*, in the application and the number of possible dynamically defined tasks, *DNTASKS*, RTXCgen automatically adds the two to derive
Control Blocks needed in the system.

Task Organization

RTXC treats a task as though it were a C function. Consequently, tasks should be written as a function called by the RTXC Task Dispatcher. There is one main difference between an RTXC and a C function, however. In RTXC, the task (i.e., function) never returns to its caller.

There are two basic code models for RTXC tasks. In the first, a task begins execution at its entry point after being invoked by a *KS_execute()* function, performs its required operations, and terminates using the *KS_terminate()* Kernel Service. This "once-only" design assumes the following code model:

```
void taskname(void)
{
   ... Data declarations
   ... Task initialization

   ... Task operations


   KS_terminate(SELF);
}
```


In the second model, a task never terminates but executes forever in a loop architecture. When using a loop architecture, a task assumes the following code model. Notice that there is no request to terminate the task as in the first example.

```
void taskname(void)
{
   ... Data declarations

   ... Task initialization

   for (;;)
   {
      ... Task operations
   }
}
```

Task Attributes

Each task has a purpose which is application specific thereby making it unique. However, in order to provide a consistent interface between the programmer and the operating environment, all tasks must share a common set of attributes. These attributes define all the information about a task the kernel needs to manage it properly. They include:

- Task Identifier
- Priority
- Task Control Block

- Entry Point
- Stack
- Processor Context
- Extended Context (optional)

Task Identifier

Each task is identified by a numerical identifier which is a number from 1 to the maximum number of tasks, *NTASKS*, inclusively. For example, if you have defined the system to have 12 tasks, all task numbers must be between 1 and 12 inclusively. The task identifier, or number, provides a reference during executive operations and is associated with the TCB. The task number serves no purpose other than as a means of determining which task is being referenced.

Task numbers for statically defined tasks will range from 1 to the number of static tasks, *NTASKS*, as defined during system generation. If dynamically allocatable TCBs are defined, their numbering begins at the number of static tasks plus 1 (*NTASKS+1*). They also have a maximum number of *DNTASKS+NTASKS*, where *DNTASKS* is the number of dynamic tasks.

Task Priority

The priority of a task is indicative of the relative importance of the task with respect to the other tasks and, indirectly, to time. Normally, each task has a unique priority but RTXC also allows multiple tasks to have the same priority.

The Task is considered an signed number. It may be any value between 1, the highest priority, and one less than the largest possible number in a data quantum of type *PRIORITY*. If a 16-bit value, the maximum task priority is 32,766. If an 8-bit number, the maximum priority is 126. Whatever the size of the *PRIORITY* type data definition, the largest value (all one bits) is the priority reserved for the Null Task. Remember that a low numerical value of the task priority number is a high priority.

A task's priority is inversely related to the numeric value of the priority. The lower the value of the priority number, the higher the task's priority. A task having priority 1 is executed before a task at priority 2 which is executed prior to a task at priority 3 and so on. The higher the priority, the more critical the timely execution of the task when it is Ready.

Execution control is granted in descending order of priority only to those tasks which are Ready. To reiterate the Rule previously stated, the Current Task, by definition, is the highest priority Ready task in the system. A task having a higher priority than the current task may exist, but if it is not Ready, it cannot be considered for execution.

Task Control Block

The task's state table is commonly referred to as a Task Control Block (TCB). A TCB in RTXC is located entirely in RAM and contains those data about the execution state of the task. All of the TCBs in a system are kept in an array which allows direct access to the data based on the task number. This makes for very quick access without wasting time searching a linked list for a task name match.

The TCB contents include the following data about the task:

- the **Execution State** containing a number which the kernel interprets as the state of the task. A value of zero ($00) indicates that the task is Ready, or runnable. Any nonzero content in the task's execution state indicates the task is blocked and will prevent it from running.
- the **Task Number** (identifier)
- the **Task Priority**
- the **Initial Entry Point** specifying the address where the task is to begin executing.
- the **Stack Pointer** containing the address of the task's current top-of-stack.
- the **Environment Arguments Pointer** containing the address, if any, of a structure holding parameters which define the task's

runtime environment. This member of the TCB is most often associated with dynamic tasks.

Task Stack

The policy of multitasking requires that each task have a stack on which are stored local variables, return addresses from subroutine calls, and the context of the preempted task. The base address of the stack is stored when the task is created for execution.

For static tasks, you must specify the size of the stack when you define the system configuration. Stack sizes of dynamic tasks are defined when the *KS_deftask()* Kernel Service is invoked. The size of each task's stack is dependent on many things such as the maximum depth of nested subroutines calls, the maximum amount of working space needed for temporary variables, and the size of any stack frames used by the task. At minimum, the size of the stack must allow for the storage of a complete processor context.

In addition to the stacks needed by the tasks, there is also the need for a stack for the kernel. This system, or kernel, stack must have sufficient space to handle the processor contexts for the maximum number of interrupts possible at any given time, less one context.

The sum of all of the stack requirements must not be allowed to exceed available RAM.

Processor Context

The amount of space required to store a processor context varies between processor types and models. You should consult reference manuals pertaining to your processor to determine the size of a processor context. The stack frame structure is completely defined in the **RTXSTRUC.H** file in the **KERNEL** directory of the RTXC distribution.

Extended Context

Some tasks, regardless of type, may make use of an extended context involving more than the standard processor registers. A common example would be the use of a math coprocessor which contains its own set of registers. The extended context, like the basic context, also needs to be preserved in certain task preemption conditions. The definition of the task as one which employs an extended context causes storage to be allocated for that purpose.

Environment Arguments

Dynamically created tasks are often an instance of another task already running. In order to distinguish one from another, RTXC uses a structure to contain information that the task needs to define its runtime environment. Hence, the name, Environment Arguments.

RTXC makes no specification about the organization or content of the Environment Arguments structure. RTXC only uses pointers to the structure; thus, its organization needs to be known only by the defining task and the using task. RTXC provides a Kernel Service, *KS_deftask_arg()*, to define the address of the structure to the object task. An additional Kernel

Service, *KS_inqtask_arg()*, is available to the using task to retrieve the address of the structure.

Task Execution

A task begins execution only when it is instructed to do so by the automatic startup procedure or upon command. Because a task is a function to the RTXC Task Dispatcher, a task can only begin execution from its starting address.

Static Tasks

Execution of a static task begins when the task is made runnable and is inserted into the READY List by another task using a *KS_execute()* function. Static tasks do not necessarily need Environment Arguments. However, RTXC permits static tasks to have defined Environment Arguments if they are defined prior to execution of the task.

Dynamic Tasks

Execution of a dynamically created task must follow a particular sequence in order for it to run. The sequence is:

1. Allocation of the Task Control Block
2. Definition of Attributes
3. Definition of Environment Arguments (if any)
4. Execution

Allocation of the TCB assigns to the task the next available TCB from the free pool with the

KS_alloc_task() function. Having the TCB, the task creating the dynamic task must use the *KS_deftask()* Kernel Service to define the task's attributes. Next, the created task's Environment Arguments, if any, may be defined. The *KS_deftask_args()* Kernel Service is used to set up the pointer to the task's Environment Arguments. Finally, the task may be invoked by the *KS_execute()* Kernel Service.

READY List

The READY List is a doubly linked list linking together the TCBs of those tasks which are capable of execution once they gain access to the CPU. The list is ordered in descending order of task priority. Thus, the highest priority task capable of receiving CPU control is always at the head of the READY List. The RTXC Task Dispatcher never needs to search for the highest priority task.

A task may share the same priority with one or more other tasks. If there is at least one more task at the same priority, the second TCB is inserted after the first task at that priority, the third after the second, and so on.

When a task becomes blocked, for whatever reason, it is no longer capable of receiving control of the CPU. Thus, a blocked task must be removed from the READY List.

Task States

A task is always in one of two basic states: **runnable** or **blocked**.

When runnable, a task has been readied for execution either by the automatic startup procedure or by request from another task. There are no impediments to its execution other than its gaining control of the CPU. A runnable task is always placed in the READY List at a position relative to its priority and that of the other tasks in the READY List.

A blocked task is not found in the READY List. It is not capable of receiving CPU control as it is waiting for some external event to occur which will remove the blocking condition. RTXC blockages occur for the following conditions:

- *INACTIVE* - Inactive (Idle)
- *QUEUE_WAIT* - Waiting on a queue condition (*Queue_not_Full* or *Queue_not_Empty*) to occur
- *SEMAPHORE_WAIT* - Waiting for a semaphore to be signaled
- *MSG_WAIT* - Waiting to receive mail
- *BLOCK_WAIT* - Blocked by RTXCbug

- *RESOURCE_WAIT* - Waiting for a resource to become available
- *DELAY_WAIT* - Waiting for a delay period to expire
- *PARTITION_WAIT* - Waiting for a memory partition to have a block available
- *SUSPFLG* - Suspended

Task Termination

A task should never execute a *return* statement, explicitly or implicitly. The proper way to terminate execution of an RTXC task is through use of the *KS_terminate()* Kernel Service.

The following code model constitutes improper coding of an RTXC task and should, therefore, be avoided.

```
void taskname(void)
{
   ... Data declarations
   ... Task initialization

   ... Task operations

}
```

INTERTASK COMMUNICATION AND SYNCHRONIZATION

The policy of having an event driven multitasking system requires flexible means of intertask communication and synchronization. The capability of RTXC to synchronize tasks with events and to move data from task to task is at the heart of system functionality.

RTXC provides a rich set of services whereby two or more tasks can synchronize or communicate with one another. There are three major mechanisms through which this is accomplished:

- SEMAPHORES
- MESSAGES and MAILBOXES
- FIFO QUEUES

Since some events are likely to be of an external origin, another important system capability is its handling of interrupts.

SEMAPHORES

There are several forms that a semaphore may take in the design of a real-time kernel. RTXC uses a semaphore construct that is known to handle most events and is low in operational overhead and RAM requirements.

RTXC semaphores are the primary mechanism of synchronizing a task with an event. Each semaphore contains information about the state of the associated event and any task trying to synchronize with the event. RTXC provides several Kernel Services to deal with processing events using semaphores. Such Kernel Services are associated with one of the possible state transitions of a semaphore.

The use of an RTXC semaphore is quite simple. For instance, one task may need to wait for the other to reach a certain point before continuing. Input/output operations are examples of this type of synchronization.

Consider a driver task which inputs data from an external device. The device driver task must wait for the input event to occur. When the input operation happens and causes an interrupt, the device driver's interrupt service routine reads the device and signals that the event has occurred. Signaling the semaphore causes the waiting device driver task to resume, presumably to process the input data. The synchronization of the task with the event is done with the use of a semaphore.

Semaphore Definition

The system designer defines all semaphores via RTXCgen during the system generation process. Semaphores are assigned names which equate to numbers. The semaphore name or number is its identifier. The semaphore number is assigned in the order of its appearance in the list of all semaphores. No special significance is implied by a semaphore's identifier. It is simply an index into the RTXC semaphore table defined during the system generation process. RTXC expects all semaphore references to be by the semaphore identifier.

Semaphore Identifiers

The user specifies the size of the data quantum needed for a semaphore identifier through definition of data type *SEMA*. The size of the defined data type determines the maximum number of semaphores that are possible. An 8-bit definition has a maximum of 126 semaphores while a 16-bit definition limits a design to 32,766.

Semaphore States

An RTXC semaphore contains a value representing one of the three possible states in which it can exist. These states are:

- **PENDING**
- **WAITING**
- **DONE**

A **PENDING** state indicates that the event associated with the semaphore has not yet occurred and is therefore pending.

The **WAITING** state shows that not only has the event not yet occurred, but a task is waiting for it to happen.

The **DONE** state tells that the event has occurred.

RTXC startup code initializes all semaphores to the **PENDING** state.

State Transitions

RTXC semaphores have a very strict state transition protocol which is automatically managed by RTXC. The permissible state changes are shown below in Figure 3-1.

Figure 3-1
Semaphore State Transitions

Using Semaphores

RTXC semaphores provide the fundamental tools for providing a means of intertask synchronization. The basic use of semaphores is that of a "handshake" in which one task waits for a signal and another provides the signal. While there are also indirect uses of semaphores in RTXC, as in messages and timers, all RTXC semaphore usage reduces to this simple relationship.

Because semaphores are always associated with an event, the use of semaphore and event are interchangeable. In fact, it sometimes makes for a better explanation to speak in terms of events rather than of semaphores, as that more closely corresponds to the real world.

Event Waiting

For a task to synchronize with an event it must first wait for the event to occur. To do this, the task waits on an RTXC semaphore using one of three Kernel Services, *KS_wait()*, *KS_waitm()*, or *KS_waitt()*. Each will change the state of a given semaphore according to its content at the time of the wait request.

If a task attempts to wait on a semaphore in the **PENDING** state, the state of the semaphore is changed to **WAITING**. The Current Task will be blocked with *SEMAPHORE_WAIT* and removed

from the READY List. Additionally, the task's execution is suspended until the event occurs.

If the Current Task attempts to wait on a semaphore which is in the **DONE** state, the wait does not occur (since the desired event has already happened), and the task is immediately resumed. RTXC automatically changes the semaphore state back to **PENDING**.

An attempt to wait on a semaphore which is already in the **WAITING** state can cause unpredictable results and should not be attempted. Although the Kernel Service *KS_wait()* returns an indication in this situation, it should be considered a design error.

Event Signaling

Signaling a semaphore constitutes the second action in the handshake. The occurrence of a specific event can be indicated by signaling the semaphore associated with that event. For tasks performing a signal, RTXC provides the Kernel Services, *KS_signal()* and *KS_signalm()* for that purpose. It is also possible to signal one or more semaphores from an interrupt service routine.

Thus, a signal may originate in either a task or in an interrupt service routine. Regardless of the signal origin, the state transition of the semaphore and any further action taken by RTXC depends on the state of the semaphore after the signal.

When signaling a semaphore in a **PENDING** state, the semaphore goes to the **DONE** state. As this action does not concern any task, RTXC takes no further action. If signaled from the Current Task, it remains in control and continues processing.

However, the signaling procedure gets more complex if the semaphore is in a **WAITING** state. The state of the semaphore does not go to **DONE** but instead returns to **PENDING**. This action saves the application software from the chore of maintaining the state of the semaphore.

Next, the RTXC signaling function determines the identity of the waiting task and unblocks it by removing the *SEMAPHORE_WAIT* condition. Once the waiting task is unblocked, and found to be runnable, RTXC inserts it into the READY List. If it is of higher priority than the signaling task, it becomes the new Current Task. Thus, synchronization of a task to an event can occur with a simple semaphore.

Signaling a semaphore which is already in the **DONE** state indicates that the previous event has not been processed. It is also indicative that no task has issued a wait request for that event since its last occurrence. Simply put, there is something wrong because the system is not able to keep up with events.

State Forcing

The third set of Kernel Services dealing with semaphores are those intended to preset the state of one or more semaphores, *KS_pend()* and *KS_pendm()*. These Kernel Services force the state of a semaphore to **PENDING**. As the system maintains semaphore states automatically, there is little use for these services except in very specific circumstances.

There are times when it may be necessary to ensure that a wait will occur. If you are uncertain about the state of the semaphore, simply precede the wait request by a call to one of the semaphore *pend* Kernel Services above.

MAILBOXES

Mailboxes are the interface between tasks which send messages to each other. Consequently, it is not necessary for a sender task to know anything about a receiver task's internal structure, or vice versa. This promotes a very clean and efficient mechanism for passing data.

Mailbox Definition

Definition of each mailbox occurs during system generation. You define a symbolic name for each mailbox, and that name becomes the mailbox identifier. The name is equated to a number based upon the position of the mailbox in the list of mailboxes. There is no priority inherent in the mailbox name or number.

Mailbox References

Mailboxes are identified by a number which has a value between 1 and the maximum number of mailboxes specified in the system configuration. determines the number of specified mailboxes and defines it as *NMBOXES*. A mailbox identifier is a data value of type M*BOX*. You should define *MBOX* in accordance with your system needs. Definition of *MBOX* should be either an 8-bit or 16-bit value.

Mailbox Structure

A mailbox resides in RAM and includes the head

link of a singly linked list. The list threads together all of the messages currently in the mailbox in descending order of message priority as defined by the senders. The head link in a mailbox contains the address of the highest priority message waiting to be received by the mailbox owner.

The highest priority message is linked to the next highest priority message, and it in turn is linked to the third highest priority message, and so on until the end of the thread. The last message in a mailbox will contain a NULL link.

If no message is waiting in the mailbox, the head link contains a NULL.

The mailbox also contains an element which optionally defines a semaphore. RTXC does not make an assignment of a semaphore to the mailbox but does provide a Kernel Service, *KS_defmboxsema()*, for doing so.

Using Mailboxes

Mailbox usage is directly associated with the transmission of messages between tasks. The description of message transmission will also serve to show how mailboxes operate.

Using a Mailbox Semaphore

The structure of a mailbox includes an element for a semaphore assignment. The semaphore has an

implicit association with the event occurring when a message arrives at an empty mailbox. In normal operation where a task uses a conditional or unconditional message receive, this semaphore is not necessary. RTXC performs all of the necessary functions to ensure that a waiting receiver task becomes runnable when the message arrives.

However, a task may need to receive mail from multiple mailboxes or need to synchronize with other events as well as the arrival of mail. To accomplish such a feat, the task must know not only when an event occurs but also the identity of the event.

RTXC provides two Kernel Services which accomplish this quite easily. The task must first assign a semaphore to each event on which it must wait. A special Kernel Service, *KS_defmboxsema()*, associates a semaphore with mail arriving at an empty mailbox.

Having defined a null terminated list of semaphores on which it is to wait, the task invokes the *KS_waitm()* Kernel Service using a list of semaphores. All of the semaphores in the list are set to a **WAITING** state if they are all **PENDING**. Any semaphore found in a **DONE** state will cause the immediate resumption of the task. If all are PENDING, RTXC blocks the task and removes it from the READY List.

When an event associated with one of the semaphores in the list occurs, RTXC resumes the

waiting task and returns the identity of the semaphore which was signaled. In this manner, the task knows the identity of the event and takes action accordingly.

For example, *KS_waitm(),* used with a list of mailbox semaphores, will block the Current Task if all of the mailbox semaphores are **PENDING**. When mail arrives at any of the mailboxes associated with the listed semaphores, the Current Task resumes. *KS_waitm()* returns the number of the semaphore associated with the event which occurred. Having the semaphore number, it is quite simple to derive the identity of the mailbox with mail. The task would then use a *KS_receive()* request to receive the mail directly from the specific mailbox.

A code model for handling multiple mailboxes through the use of mailbox semaphores is illustrated below.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"
#include "cmbox.h"

void taskname(void)
{
   RTXCMSG *msg;
   SEMA cause;
   SEMA semalist[] =
   {
      MBXSEMA1,           /* Mailbox 1 semaphore */
      MBXSEMA2,           /* Mailbox 2 semaphore */
      0                   /* null terminator */
   };

   /* Define semaphore for mailboxes */
   KS_defmboxsema(MBOX1,MBX1SEMA);
   KS_defmboxsema(MBOX2,MBX2SEMA);

   for (;;)
   {
      /* wait for either of 2 events */
      cause = KS_waitm(semalist);
      switch(cause)
      {
         case MBX1SEMA:
           msg = KS_receive(MBOX1, (TASK)0);
            ... process msg ...
            break;

         case MBX2SEMA:
           msg = KS_receive(MBOX2, (TASK)0);
            ... process msg ...
            break;
      }  /* end of switch */
      KS_ack(msg)
   }  /* end of forever */
}
```

MESSAGES

Messages are one of the means by which data moves from a sender to a receiver task. Every task running under RTXC is capable of being both a message sender and receiver. Message transmission involves the transfer of data packets from one task to another via mailboxes. Messages are transmitted from a task by being placed in a mailbox used by a receiving task.

RTXC does not actually move the content of a message from the sender to the receiver. Instead, RTXC puts the address of the message into a singly linked list found in the receiving mailbox. Placement of messages in the mailbox list is in a descending order of message priority. The sender assigns the message priority. When the receiver requests receipt of the next message, RTXC returns the address of the message which has the highest priority of all current mail in the mailbox.

It is possible, however, to temporarily suspend this order of receipt by requesting only those messages from a particular sender task. This can be useful when it is desirable not to mix messages on a shared resource, for example, a printer.

Message Structure

A message is a two-part construct residing in RAM and consisting of a message envelope and the message body. RTXC maintains the content of the message envelope. The task is responsible for the message body. The message body may be of any

format recognizable by the sender and receiver. Using the message body, data may be passed in either direction between sender and receiver.

The message body is contiguous to the envelope. The message body may be a simple pointer to another area located in either RAM or ROM. It may also be part of a single message structure enclosing both the message envelope and the message body. To reiterate, the content of the message can be anything mutually agreed upon by the sender and the receiver.

Message Priority

Each message has a priority assigned by the sender task when the message is sent. The message priority has no explicit relationship with the sender's task priority. It is simply a number between 1 and the maximum priority inclusively. However, a message may be sent with a priority of zero (0) which causes RTXC to assign the message a priority equal to the sender's task priority. RTXC uses the message priority as the key in inserting the message into the thread of the specified mailbox. Different tasks may use the same message priority without problem.

Note that if all senders use a fixed priority for all messages sent to a given mailbox, the result is a FIFO.

Using Messages

Messages are sent from one task and received by another. Like any postal service, RTXC takes the message from the sender and puts it into a mailbox. The mailbox is known by the sender to be used by the receiver. There is a direct analogy with a letter being mailed.

The letter's sender puts the letter (the message body) into an envelope and puts the recipient's address on the envelope. The letter is then posted (sent) to the postal service. The postal service delivers the letter to the mailbox at the address given on the envelope. At some time subsequent to delivery, the recipient checks the mailbox and retrieves the letter.

If the recipient were especially anxious to receive the mail, he might have checked the mailbox before the letter was delivered only to find the mailbox was empty. This corresponds to the situation of a receiver task checking a mailbox by trying to receive mail only to find the empty mailbox. Like the anxious recipient, the task must decide what to do if the mailbox is empty.

Since the recipient is a proper soul, he acknowledges receipt of the letter by sending a reply by a similar process. The sender of the original letter receives the reply acknowledging his letter. The transaction is then complete.

Sending Messages

RTXC provides two ways to send a message - asynchronously and synchronously. When sending a message synchronously, the sender sends the message and does not proceed until it gets an acknowledgment from the receiver task. In sending asynchronously, the message is sent and the sender task proceeds without waiting for an acknowledgment. However, the asynchronous sender may later choose to wait for an acknowledgment.

Asynchronous Messages

Asynchronous message transmission uses the *KS_send()* Kernel Service. The use of *KS_send()* may result in a context switch if the receiving mailbox has a task waiting for mail, and that task is of higher priority than the Current Task. Whether or not there is a context switch, the sender of an asynchronous message always remains in the READY List. When the message is sent, the task resumes processing immediately following the *KS_send()* Kernel Service request.

It may be the design of the task to continue processing after sending the message. If so, the task may choose synchronization with the message acknowledgment at a later time. To accomplish that, the task should simply invoke the *KS_wait()* Kernel Service using the message semaphore named in the *KS_send()* call.

An example is given below of a code model for a task using a loop architecture and sending asynchronous messages.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"            /* defines MSGSEMA */
#include "cmbox.h"          /* defines MAILBOX3 */

void taskname(void)
{
   struct {
   RTXCMSG msghdr;    /* Message envelope (req.) */
   char data[10];     /* Message body
   } mymessage;

   for(;;)
   {

      ... set up content of the message body

      KS_send(MAILBOX3, &mymessage.msghdr,
              (PRIORITY)4, MSGSEMA);

      ... do some more processing and then wait
          for the message acknowledgment

      KS_wait(MSGSEMA);         /* wait for ack */

      ... finish processing within the loop
   }
}
```


Synchronous Unconditional Messages

Tasks sending synchronous messages use either *KS_sendw()* or *KS_sendt()*. These two Kernel Services are functionally equivalent to *KS_send()* immediately followed by *KS_wait()*. A context switch always occurs with the use of *KS_sendw()* or *KS_sendt()* because the Current Task becomes blocked while waiting to synchronize with the message acknowledgment.

KS_sendw() will wait unconditionally until it receives the message acknowledgment. The following code example shows a task model using a loop architecture while sending synchronous messages with *KS_sendw()*.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"
#include "cmbox.h"

void taskname(void)
{
   struct {
   RTXCMSG msghdr;    /* Message envelope (req.) */
   char data[10];     /* Message body
   } mymessage;

   for(;;)
   {

      ... set up content of the message body

      KS_sendw(MAILBOX3, &mymessage.msghdr,
              (PRIORITY)4, MSGSEMA);

      ... continue processing after ack

   }
}
```


Synchronous Conditional Messages

Like *KS_sendw()*, the other synchronous message sending Kernel Service, *KS_sendt()*, also waits for receipt of the message acknowledgment. However, it also starts a timeout timer within which the task expects to receive the acknowledgment. If not, the timeout expires and the task will have to execute code to deal with the situation. An example of a task sending messages in this manner follows.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"
#include "cmbox.h"
#include "cclock.h"

void taskname(void)
{
   TICKS timeout = 250/CLKTICK;      /* 250 msec */

   struct {
      RTXCMSG msghdr;   /* Message header (req.) */
      char data[10];    /* message body */
      } mymessage;

   for(;;)
   {

      ... set up content of the message body

      if (KS_sendt(MAILBOX3, &mymessage.msghdr,
                   (PRIORITY)4, GRAFSEMA,
                   timeout) == RC_TIMEOUT)
      {
         ... message not completed within timeout
             period. Deal with it with special code
      }
      ... message sent and acknowledged
   }
}
```


Receiving Messages

The Kernel Services *KS_receive()*, *KS_receivew()*, and *KS_receivet()* will fetch mail from a mailbox if present. If there is mail present when a receive request is made, all of the RTXC Kernel Services for receiving mail are identical. Each of the functions returns a pointer to the retrieved message envelope of the requesting task.

However, if no mail is present, the functions will either report the empty condition or will block the Current Task until mail arrives. A receiver task attempting to receive mail always has to deal with the problem of what to do if the mailbox is empty. Depending on the Kernel Service used in the attempt to receive the message, RTXC will:

a) notify the receiver task that the mailbox is empty and let the task deal with it through special program logic, or

b) block the receiving task until a message is sent to the mailbox, or

c) block the receiving task until either a message is sent to the mailbox or a defined period of time elapses.

Polled Receipt

The first case is obvious. The task polls the mailbox using the *KS_receive()* Kernel Service. If the mailbox is empty, it is up to the system designer as to how to proceed at that point. An example of a task receiving mail in a loop-based task architecture follows.

```
include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"             /* defines MYMAIL */

void taskname(void)
{
   RTXCMSG *msg;

   for (;;)
   {
      /* receive next message from any task */
      while (msg = KS_receive(MYMAIL,(TASK)0) == (RTXCMSG *)0 )
      {
         ... Deal with empty mailbox with special
             logic here
      }

      ... message received, process it

      KS_ack(msg);  /* ack completion of message */
   }
}
```

Unconditional Receive

In the second case, the receiver task uses the *KS_receivew()* Kernel Service. It will remain blocked until another task sends a message to the empty mailbox. That event causes the waiting receiver task to become runnable again and inserted into the READY List. RTXC returns the address of the message to the receiver task which continues operation.

A code model for a task using *KS_receivew()* follows.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"             /* defines MYMAIL */

void taskname(void)
{
   RTXCMSG *msg;

   ... task initialization

   for(;;)
   {
      /* receive next message from any task */
      msg = KS_receivew(MYMAIL, (TASK)0);

      ... process the message

   }
}
```

Conditional Receive

In the third case, the task uses the *KS_receivet()* Kernel Service which combines elements of the first two receiving functions. Like *KS_receivew()*, RTXC blocks the receiver task but only until a message arrives or the timeout elapses. If the former, it is treated exactly as in the second case for *KS_receivew()*. However, if the timeout expires, the system designer must provide special code to handle it. The procedure to follow, as in the first case, is up to the system designer.

A code example of a task using *KS_receivet()* follows.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"          /* defines MYMAIL */
#include "cclock.h"        /* defines CLKTICK */

void taskname(void)
{
   RTXCMSG *msg;
   TICKS timeout = 500/CLKTICK;      /* 500 msec */
   KSRC ccode;

   ... Task initialization

   for(;;)
   {
      /* receive next message from any task */
      while( (msg = KS_receivet(MYMAIL, (TASK)0,
                                timeout, &ccode)) ==
                                 (RTXCMSG *)0 )
      {
         ... timeout occurred or there were no
             timer blocks available. Look at ccode
             to find out and then deal with the
             situation here.
      }

      ... message received, process it.

      KS_ack(msg);           /* ack the message */
   }
}
```

Message Acknowledge

A sender task may need to synchronize with the receiver task's receipt or processing of a message. RTXC makes it easy to do this through the synchronous message sending services, *KS_sendw()* and *KS_sendt()*. These two services automatically block the sending task by performing an implicit *KS_wait()* using the message semaphore. In the use of *KS_send()* to send a message asynchronously, the sending task is not blocked but continues processing. It may eventually issue an explicit *KS_wait()* using the message semaphore.

In the scenarios above, the sender task, having assigned a message semaphore, sends the message to the receiver and then, implicitly or explicitly, waits for the message to be acknowledged. The wait occurs in association with the given message semaphore.

When the receiver receives or completes processing of the message, it acknowledges the message using the *KS_ack()* Kernel Service. This action amounts to signaling the message semaphore. Thus, the handshake with the waiting sender task is complete. RTXC removes the *SEMAPHORE_WAIT* block on the sender task to make it runnable again and puts it back in the READY List.

Message Responses

If it had been necessary, the receiver task could have stored a response in the message body. By doing so, RTXC permits a simple but rapid means of passing data bi-directionally between two tasks. This feature makes it possible for two tasks to alternate the roles of sender and receiver.

When returning a response to the message sender, the receiving task should put the response in the message body prior to invoking RTXC to acknowledge the message.

QUEUES

A third technique whereby two tasks can communicate and synchronize is via FIFO queues. Queues are usually used to handle such operations as character stream input/output or other data buffering. RTXC provides a simple way of putting data into and getting data from a queue.
RTXC queues differ from messages in that the actual data rather than an address is entered or removed from the queue. By definition, all RTXC queues use a FIFO model. Thus, the queue content represents the chronological order of data entry and extraction. There is no priority considered with respect to the order of entry as is the case with messages.

The system designer determines the number of queues needed for the application as well as the sizes of each. Each RTXC queue may be defined as having a single or multiple bytes per entry. RTXC queues support a model allowing more than one task to put data into a queue (Queuesmultiple producers) and more than one task to remove data from a queue (Queuesmultiple consumers).

The queueing techniques used by RTXC involve the copying of data from a producer task into a FIFO queue and thence to a consumer task. Two basic Kernel Services are supplied, and each has two possible variants. RTXC performs any necessary synchronization between a queue's producer and consumer tasks.

Queue Definition

The system designer defines all queues during the system generation process using RTXCgen. Like other system elements, queues are assigned names which equate to numbers. The queue number is its position in the list of all queues. There is no special significance given to a queue identifier.

Queue Identifiers

The system designer specifies the size of the data quantum needed for a queue identifier. Queue identifiers are numerical values of type *QUEUE*. The size of a value of type *QUEUE* defines the maximum theoretical number of queues in a system. An 8-bit signed quantity permits up to 127 queues.

Queue Structure

An RTXC queue has two parts: the header and the body. Both parts of a queue must reside in RAM. The queue header contains information needed by the RTXC Kernel Services to move data into and out of the queues properly. The queue body is simply an area of RAM which is organized as an array.

The queue body array contains a specified number of entries having a specified size. All of the entries in a given queue are the same size.

The organization of the queue header includes two elements which are defined during system configuration:

- Width, queue entry size (in bytes)
- Depth, maximum length (in entries)

The size of the queue body is determined from the Width and Depth definitions. The other elements of the queue header are maintained internally by RTXC. The queue header should never be manipulated by a task.

Queue States

Each queue must always exist in one of three possible states:

- *Empty* - There are no entries in the queue.
- *not_Empty_not_Full* - There is at least one but less than the maximum number of entries in the queue.
- *Full* - All of the possible entries in the queue are used.

RTXC initializes all queues to the *Empty* state during system startup. Additionally, RTXC maintains the queue state automatically and provides all synchronization between producer and consumer tasks.

Using Queues

Queues provide an easy way of moving data between tasks so that the data may be processed in chronological order. Unlike messages, there is no priority assigned to a FIFO queue entry.

RTXC queue operations fall into two basic categories: putting data into queues, *enqueueing*, and getting data out of queues, *dequeueing*. RTXC provides one basic Kernel Service for each queue operation, and each of those has two possible variants.

Enqueueing Data

The basic Kernel Service for putting data into a queue is *KS_enqueue()*. The possible variants are *KS_enqueuew()* and *KS_enqueuet()*. In order for data to be put into a queue, there must be at least one entry in the queue body which is unused and able to receive the data. If the queue state is *Full*, all entries in the queue are occupied, and there is no place to put a new entry.

KS_enqueue() moves data from a source location specified by the producer task, the Current Task, and moves it into the next free entry in the queue. The Kernel Service determines where the next free entry is located by examining information in the queue header about current usage. If the queue is *Full*, the task is notified of the situation and must deal with it in whatever manner is required by the application.

KS_enqueuew() and *KS_enqueuet()* operate in exactly the same way as *KS_enqueue()* when the state of the queue is either *Empty* or *not_Empty_not_Full*. In other words, when there is room in the queue, it may receive new data. However, functional differences occur when the queue is *Full* and an attempt is made to put a new entry into the queue.

When the producer, the , attempts to put data into a full queue while using the *KS_enqueuew()* Kernel Service, RTXC will block the task. It will also remove the task from the READY List, and make it wait until a consumer task removes data from the queue thereby opening a slot to receive the new data. When the slot becomes open, the producer task is automatically returned to the READY List and allowed to continue its operation. Thus, the synchronization between the producer and consumer is performed without direct program intervention.

The use of the *KS_enqueuet()* Kernel Service is exactly like that of *KS_enqueuew()* except that the duration of the wait is limited by a user defined period of time. The blocked producer task will remain blocked until either a free slot becomes available or until the specified time period elapses. If the timeout occurs, the application program will be so notified and must deal with the situation in a manner consistent with the system design.

Dequeueing Data

RTXC provides one main Kernel Service to get data from a queue, *KS_dequeue()*, and two possible variants, *KS_dequeuew()*, and *KS_dequeuet()*. All of

these services operate in the same manner when the state of the queue is either *Full* or *not_Empty_not_Full*. The function locates the oldest entry in the queue and moves it to a destination specified in one of the calling arguments.

When the state of the queue is *Empty*, the dequeueing functions operate slightly differently. *KS_dequeue()* returns a function value to indicate the *Empty* state situation. The consumer task must recognize that return value and handle the situation with program logic.

The variant, *KS_dequeuew()*, acts much like the basic service except when the queue is *Empty*. In that situation, it blocks the Current Task, removes it from the READY List, and makes it wait for a producer task to put data into the queue. When data is put into the queue by another task, the consumer task will be unblocked, reinserted into the READY List, and allowed to continue. As in the basic service, the data is moved from the queue to the destination location specified by the consumer.

KS_dequeuet() is the same as *KS_dequeuew()* except that the duration of the wait on an empty queue is limited by a time period specified in the function call. The blocked task will wait until either an entry is put into the queue or until the timeout elapses. If the entry is made within the timeout period, the Kernel Service returns a value to indicate success. If the timeout occurred, the returned value will so indicate. Application code using the *KS_enqueuet()* Kernel

Service will have to provide code to check on and handle these various return values.

Producer and Consumer Task Synchronization

Queueing operation can result in a context switch under certain circumstances. The obvious cases occur when a wait occurs as the result of using *KS_dequeuew()* or *KS_dequeuet()* on an *Empty* queue or an *KS_enqueuew()* or *KS_enqueuet()* on a *Full* queue. Less obvious cases occur when a queue is *Full* and already has a producer task waiting or when an *Empty* queue has a consumer task waiting.

Whenever a task invokes a queueing operation or is forced to wait, the task is inserted into a list of waiter tasks associated with the particular queue. The order of insertion is by descending order of their respective priorities. There may be more than one task waiting to complete some activity on the queue.

As soon as the operation occurs which removes the condition on which the waiter task is blocked, the highest priority task is taken from the list of waiters, unblocked, made Ready, and inserted into the READY List. If the waiter task is of higher priority than the , a context switch will result. In this manner, RTXC maintains synchronization between the producer and the consumer tasks when they use queues.

Synchronization with Multiple Events

There are certain cases in which the producer or consumer may wish to override the automatic RTXC synchronization methods. This is most likely to happen when the task design requires it to be synchronized with any of several events. RTXC provides the *KS_waitm()* Kernel Service to allow a task to wait on the logical **OR** condition of several events. By using *KS_waitm()*, a task may easily wait on the occurrence of any event associated with a set of semaphores.

An example of this facility might be a task which must synchronize with data arriving at any of three different queues. One possible solution would be to poll each queue periodically to determine if it has data.

Another example might be a task which needs to synchronize with an external event but also needs to know whenever a particular queue gets full. Depending on the time criticality of handling the two possible events, a possible solution might be to wait for the external event and then check the queue size. Alternatively, another solution might be to check the queue states periodically.

In an event driven system, none of these solutions is necessarily a good design. It would be better, in the first example, to have RTXC determine when data arrives and inform the task as to which queue has the data. In the second example, the kernel could determine when the queue becomes full or when the

external event occurs and inform the waiting task accordingly. Both examples would benefit from the use of the *KS_waitm()* Kernel Service. This method would free the CPU to do other chores until such time as any of the blocking conditions is removed.

Consider the following code example.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"
#include "cqueue.h"


void taskname(void)
{
   SEMA cause;
   SEMA semalist[] =
   {
      Q3NESEMA,      /* Queue 3 QNE semaphore */
      EXTEVENT,      /* External event semaphore */
      0              /* null terminator */
   };

   KS_defqsema(QUE3,Q3NESEMA,QNE)

   for (;;)
   {
      /* wait for either of 2 events */
      cause = KS_waitm(semalist);
      switch(cause)
      {
         case Q3NESEMA:
            ... process event by getting data...
                from the queue.
            break;

         case EXTEVENT:
            ... process the external event...
            break;
      }  /* end of switch */
   }  /* end of forever */
}
```


The key to making these scenarios possible is the definition of semaphores associated with different queue events and the multiple event wait Kernel Service, *KS_waitm()*.

Queue Semaphores

There are four possible queue conditions (or events) with which to associate semaphores. The purpose of these semaphores is to provide a mechanism by which task synchronization may occur as a result of certain changes in the queue state. In normal queue usage, there is no real need for synchronization with queue related events other than those already provided by RTXC. However, it is sometimes advantageous to use queue event semaphores for special synchronization purposes. Almost without exception, queue semaphores will be used in conjunction with *KS_waitm()* Kernel Service.

All four semaphores relate to the four possible changes-of-state events or conditions through which a queue may transition. These four conditions and their RTXC abbreviation codes are:

- *Queue_Empty* (**QE**) - The event which occurs when a queue state changes from *not_Empty_not_Full* to *Empty*.

- Q*ueue_not_Empty* (**QNE**) - The event which occurs when a queue state changes from *Empty* to *not_Empty_not_Full*.

- *Queue_Full* (**QF**) - The event which occurs when a queue state changes from *not_Empty_not_Full* to *Full*.

- *Queue_not_Full* (**QNF**) - The event which occurs when a queue state changes from *Full* to *not_Empty_not_Full*.

RTXC does not automatically associate any semaphores with these possible queue conditions. During system operation, the application tasks may use the *KS_defqsema()* Kernel Service to perform the associations. The associated semaphore is not predefined and may be any semaphore from those configured by the system designer.

If it becomes necessary to disassociate a queue event from a semaphore, you may do so with the *KS_defqsema()* function called with a semaphore of zero ((SEMA)0).

Queue_Empty (QE)

The **QE** semaphore is associated with the next occurrence of the transition from *not_Empty_not_Full* to *Empty*. If the state of the queue is *Empty* when the **QE** semaphore is defined, the semaphore is set to a **DONE** state. Otherwise, the **QE** semaphore will be set to a **PENDING** state.

The following example, albeit contrived, illustrates a simple use for the **QE** semaphore in a producer task. This might be useful if the consumer task is at a lower priority. Whenever the consumer removes an entry from *DATAQ*, the producer task would be allowed to put another entry in the queue. This

might lead to some undesired "thrashing" between the two tasks. The thrashing could be prevented by having the producer task fill the queue and then wait for the consumer task to empty it. The producer task could synchronize with the transition to *Empty* and then continue.

```
#include "rtxcapi.h"
#include "csema.h"
#include "cqueue.h"

char source;

KS_defqsema (DATAQ, EMPTYSEM, QE);

... new data is stored in source variable

while (KS_enqueue(DATAQ, &source) == RC_QUEUE_FULL)
{
   KS_wait(EMPTYSEM); /* task waits here until */
                      /* queue goes empty      */
}
```

Queue_not_Empty (QNE)

When an entry is put into an empty queue, the state of the queue changes from *Empty* to *not_Empty_not_Full*. This is the *Queue_not_Empty* (**QNE**) event. When *KS_defqsema()* is used to associate a semaphore with the **QNE** condition, the state of the semaphore follows the state of the queue.

If the queue is *Empty*, the **QNE** semaphore is set **PENDING**. While the queue remains *Empty*, a task making a subsequent attempt to wait on the **QNE** condition would be blocked and the semaphore changed to a **WAITING** state.

If the queue state is *not_Empty_not_Full* when the **QNE** semaphore is defined, the semaphore state is set to **DONE**. This setting indicates that the associated event has already occurred, i.e., the queue is no longer empty. A task which attempts to wait for the **QNE** event would be allowed to continue without being blocked because the event has occurred. Unlike other semaphores, however, RTXC ensures that the **QNE** semaphore remains in the **DONE** state as long as the queue is neither *Empty* nor *Full*.

The following example illustrates how a consumer task can process data from multiple queues without resorting to polling.

```
#include "rtxcapi.h"
#include "csema.h"
#include "cqueue.h"

char dest;
SEMA semalist[] = {EMPTYSM1, EMPTYSM2, 0};
SEMA cause;

KS_defqsema(DATAQ1, EMPTYSM1, QNE);
KS_defqsema(DATAQ2, EMPTYSM2, QNE);

cause = KS_waitm(&semalist);
switch(cause)
{
   case EMPTYSM1:
      KS_dequeue(DATAQ1, &dest);
      break;

   case EMPTYSM2:
      KS_dequeue(DATAQ2, &dest);
      break;
}
```


Queue_Full (QF)

The **QF** semaphore operates as a mirror image of the **QE** semaphore. The **QF** event occurs when the queue state changes from *not_Empty_not_Full* to *Full*. The *KS_defqsema()* Kernel Service defines the initial state of the semaphore. It is set to a **PENDING** state when the queue is not full or to **DONE** when it is *Full*.

Queue_not_Full (QNF)

The **QNF** semaphore operates in a mirror image fashion to the **QNE** semaphore. The **QNF** event occurs when data is removed from a full queue, thereby making it *not_Empty_not_Full*. This transition may be signaled if a **QNF** semaphore has been previously defined with the *KS_defqsema()* Kernel Service. Like the **QNE** semaphore, the **QNF** semaphore is set to **PENDING** if the queue is *Full* when the definition occurs. It is set to **DONE** if the queue state is *not_Empty_not_Full* when **QNF** is defined. If a full queue has an entry removed, the **QNF** semaphore will be signaled.

Purging a Queue

Sometimes it is necessary to reset an RTXC queue so that it is considered empty. In RTXC, this action is referred to as *purging* the queue. It is not an action done frequently, but it can be useful at times. It is accomplished by the *KS_purgequeue()* Kernel Service and it has certain ramifications which must be understood prior to use.

From the aspect of clearing the queue of all entries, it is quite efficient because setting the current size of

the queue to zero effectively clears all entries. A larger problem than simply clearing the size of the queue exists, however. The main question that comes to mind would be the disposition of any tasks waiting for any of the queue events and tasks waiting for access to the queue.

Purging a queue is the logical equivalent of removing the entire content of the queue one entry at a time until the queue is empty. It is therefore proper during a queue purge to treat the queue semaphores as though the entries were removed singly. RTXC does this in the following manner:

- **QF** always remains in its current state. It is never signaled as the result of a queue purge.

- **QNF** remains in its current state if the queue state was *Full* at the time of the purge. If the state of the queue was not *Full*, the **QNF** semaphore is not signaled.

- **QNE** always remains in its current state. It is never signaled as the result of a queue purge.

- **QE** remains in its current state if the state of the queue was *Empty* at the time of the purge. If the queue was not *Empty*, **QE** is signaled.

In addition to purging the queue and treating its queue semaphores, RTXC also handles any tasks waiting to put data into the queue. If the queue was *Empty* at the time of the purge, any waiter tasks are

those trying to get data from the queue. RTXC ignores these waiters.

However, if the queue had been *Full* at the time of the purge, any waiter tasks would have been those trying to put more data into the queue. These are the tasks with which RTXC must deal. The solution is to process each task waiting to put data into the queue by moving the data into the queue and resuming the task so that it can continue normally.

As each entry is placed into the queue, the queue semaphores, **QNE** and **QF**, are properly treated. The first entry into the queue causes a *Queue_not_Empty* event requiring that the **QNE** semaphore be signaled. If the entry makes the queue become *Full*, the **QF** semaphore must be signaled.

Thus, purging the queue still retains the integrity of the queue construct and its associated parts.

RESOURCES

RTXC permits a task to gain exclusive access to some system component or element. This is especially useful where it is necessary to guarantee that one and only one user has control of an entity. Any entity, logical or physical, may be defined as one which requires restricted access. A database, a special software function, or a printer are a few examples.

In a multitasking system, it is often necessary to have different tasks make use of a common entity. It is also common to have a requirement that no task shall be able to preempt the use of certain entities by other tasks. Since an event driven design permits the preemption of a task at any time by one of higher priority, it is necessary to provide a mechanism for preventing uncontrolled access to a common entity. RTXC provides such a mechanism, a resource, for doing this.

Use of a resource is simple. are associated with entities during the system generation process. The association is purely logical since the entity may itself be logical rather than physical. Whenever a task wants to use a common entity with a guarantee of exclusive access, it simply locks the resource. Locking the resource prevents other tasks from gaining access to the entity while another task has access control.

After locking the resource, the task may use the associated entity to whatever extent is necessary to perform its functions. When the task has completed

its use of the entity, it reverses the process by unlocking the resource. Unlocking the resource permits another task to gain access control of the entity.

Resource Definition

The system designer defines all resources during the system generation process using . Resources, like other system elements, are assigned names which equate to numbers. The resource number is its position in the list of all resources. There is no special significance given to a resource identifier.

Resource Identifiers

The system designer specifies the size of the data quantum needed for a resource identifier. These identifiers are numerical values of type *RESOURCE*. The size of a value of type *RESOURCE* defines the maximum theoretical number of resources in a system. An 8-bit signed quantity permits up to 127 resources.

Resource Structure

An RTXC resource contains two basic components, the resource state and the list of waiters. The state of the resource defines whether or not the resource is "locked" (or owned). If locked, it also contains the identity of its owner task.

Only one task at a time may be the owner of the resource. After a resource becomes owned, any other task attempting to lock the resource will be prevented from doing so regardless of the task's priority relative to that of the resource's owner.

RTXC provides one basic Kernel Service to lock a resource and one to unlock. The locking function has two possible variants, both of which involve waiting for the resource if it is owned at the time of the request. With the variants, a lock request made to a resource which is already owned will cause the requesting task to be blocked, removed from the READY List, and added to the resource's list of waiters.

The resource's waiter list is a doubly linked list in which new waiter tasks are inserted in descending priority order. The highest priority task waiting for a resource is always the first task in the list. When a locked resource is unlocked, the highest priority waiting task, if any, will gain access control of the resource.

Resource States

A resource always exists in one of two possible states, *Free* and *Locked*. A task may become the owner of a resource only when the resource is *Free*. If there are no waiters for a *Locked* resource, unlocking the resource by its owner changes the resource state to *Free*.

RTXC supports locking of a resource by its owner. Nested locks might occur if a task, which has locked a resource, calls a function in which the resource is locked again. Such a situation does not cause a conflict. However, for correct operation, RTXC expects that for each lock there is an unlock.

Using Resources

A task wanting to use an entity associated with an RTXC resource must first lock the resource. When it is finished with the resource, it must unlock it. RTXC provides a basic Kernel Service for each of these operations, *KS_lock()* and *KS_unlock()*. The *KS_lock()* Kernel Services can be augmented by two more functions which will block the requesting task until the resource becomes *Free*. Unlocking is universal and has no variants to the basic function.

Resource Locking

The basic Kernel Service used to lock a resource is *KS_lock()*. If a requesting task uses *KS_lock()* to lock a resource and the resource is *Free*, the task becomes the owner of the resource. If the resource is already in a *Locked* state, the requesting task is so informed by a value returned by RTXC. The task must have the required program segment to detect the returned value as well as deal with it according to the task's function.

If the programmer does not want to have to write that extra code segment to deal with the locked resource, RTXC has two other Kernel Services which will reduce the code burden. Both *KS_lockw()*

and *KS_lockt()* will block the requesting task if the given resource is already in a *Locked* state.

KS_lockw(), upon finding the resource to be *Locked*, unconditionally blocks the task, removes it from the READY List and inserts the task into the resource's list of waiters. The task will remain in this condition until it becomes the highest priority task waiting for the resource when its current owner unlocks the resource. The unlocking will cause the waiting task to become unblocked and to be reinserted into the READY List. The unblocked task then becomes the resource's owner.

The *KS_lockt()* Kernel Service is much like that of *KS_lockw()* except that the duration of the wait is limited by a user definable time period. The task will remain blocked until either the task gains access to the resource or the timeout occurs. Both conditions return a value which must be detected and handled by the requesting task.

Resource Unlocking

RTXC provides only one Kernel Service, *KS_unlock()*, for unlocking a locked resource. There are no variants. Besides the obvious function of unlocking the resource, it will automatically lock the resource for the highest priority waiting task, if any. The new owner task will automatically be unblocked and inserted into the READY List to resume its operation.

Priority Inversion

While use of resources is simple, the rule that only one task may own a resource can lead to an undesirable situation known as *priority inversion*. Priority inversion occurs when a lower priority task blocks execution of a higher priority task.

Consider the scenario in which there are two tasks, A and B, where task A is higher priority. Task A is temporarily blocked and task B is the Current Task. As part of its execution, task B locks Resource R and continues. The event blocking task A occurs causing task A to be returned to the READY List preempting task B, since task A is higher priority. Once it is in control, task A attempts to lock Resource R and fails because the resource is owned by task B.

The system now has a priority inversion dilemma - a lower priority task, because of its ownership of the resource, is blocking execution of a higher priority task which also needs the resource. This situation can lead to undesirable results if not handled.

RTXC provides a mechanism to handle priority inversions that may be invoked at the discretion of the system designer. Many tasks which use resources do not encounter the possibility that a priority inversion can occur. It would be counterproductive to require that such a task be forced through the priority inversion detection logic. To prevent this, each resource can be given an attribute indicating that it faces possible priority inversions or not. A

task using a resource with the attribute enabled will exercise the priority inversion detection and handling logic whenever it attempts to lock the resource.

The priority inversion attribute of a resource may be enabled or disabled at any time through the use of the *KS_defres()* kernel service. Initially, all priority inversion attributes of all resources are disabled.

Assuming that the attribute is enabled, RTXC treats priority inversions in a straightforward manner. In resolving the conflict described above, it must be noted that it is not possible to preempt task B's ownership of Resource R because it is not known into what condition task B has placed the resource. For example, Resource R may be protecting a section of a database of process information into which it has partially written some update information. The update is incomplete at the time task A preempts. For task A at the point of resumption of CPU control, the database is in an unknown state when it tries to lock Resource R. Therefore, it would not be part of the solution simply to grant ownership of the resource to task A. Rather, task B must be allowed to continue to run and to retain ownership of Resource R until it can unlock Resource R with the knowledge that the database being protected by the resource is in a known state.

RTXC handles the priority conflict by temporarily elevating the priority of task B to a level equal to that of task A. The purpose of the action is to raise task B's position in the READY List in order to

grant it more execution time. The supposition is that more execution time is all that is required for Task B to complete its use of the resource and to unlock it, making it available for the next task, presumably Task A.

When task B completes the need for the resource, it unlocks the resource by using *KS_unlock()*. RTXC then reduces the priority of task B to the level where it was when the priority inversion with task A occurred. If task A, or any other task, had attempted to lock Resource R by using either a *KS_lockw()* or a *KS_lockt()* Kernel Service, RTXC would automatically grant ownership of that resource to the highest priority task waiting to lock it.

If task A used *KS_lock()* in its attempt to gain ownership of Resource R, it would have to take partial action itself in resolving the priority inversion. RTXC will elevate task B to the same priority as task A regardless of the Kernel Service used to attempt the lock. However, task A will remain the Current Task because *KS_lock()* cannot block the requesting task if the state of Resource R is already *Locked*. Instead, RTXC will return a value of **RC_BUSY** from task A's call to *KS_lock()* and continue the execution of task A as the Current Task. In order to permit task B, in its elevated priority, to become the Current Task, task A must first give up control of the CPU.

One way it can give up control is to issue a *KS_yield()* request. The result is that the composition of the READY List will change from

task A followed by task B to task B followed by task A. The reversal makes task B the Current Task and permits it to continue executing at the elevated priority level.

The following code fragment demonstrates this method of task A's handling a priority inversion.

```
while (KS_lock(RESOURCE_R) == RC_BUSY)
{
   If(KS_yield()==RC_NO_YIELD)/* yield to Task B */
      KS_delay(5/CLKTICK);  /* delay if Task B   */
                            /* not same priority */
                            /* as Current Task   */
}
/* at this point this task owns RESOURCE_R */
```

The *while* loop is used just in case task B becomes blocked for some reason and task A once again becomes the Current Task. If so, task A will continue to yield until the external event blocking task B occurs causing it to run once again.

In the example above, it is also seen that *KS_lock()* does not block the requester on a failed attempt. Consequently, there is no waiting task to which RTXC can automatically pass ownership of the resource when it is unlocked. Ownership of the resource will not be gained until task A issues the next *KS_lock()* request at the start of the *while* loop.

There is one final note of caution regarding usage of resources. A task using a resource common to one or more other tasks should not issue Kernel Service requests to functions which result in the task becoming blocked while it has locked the common

resource. RTXC does not prevent such an occurrence, but it is not considered good design as it can lead to a resource being locked for extended periods of time and, hence, to undesirable results.

For instance, if the *KS_lockt()* Kernel Service is used by task A and the associated timeout period expires before it gains control of the resource, RTXC will automatically reduce the priority of task B to the level at which it was prior to being elevated. It also unblocks task A, inserts it into the READY List and preempts task B. Once restored to control of the CPU, task A must deal not only with priority inversion but also with the fact that the timeout occurred before the resource was unlocked. Such a situation may be indicative of problems in task B or that task B was waiting on some event that has not yet happened. In any case, code in task A will have to deal with sorting it out.

MEMORY PARTITIONS

RTXC supports a RAM memory management concept which features the use of defined memory partitions to prevent fragmentation. The system designer may specify as many memory partitions as are needed to accomplish the system's functions. Memory partitions may be statically or dynamically defined according to the needs of the application.
Each memory partition, also called a *Map*, contains one or more blocks all of which are the same size. Tasks allocate blocks of memory from various Maps as needed to perform their assigned jobs. When finished with a memory block, they release it by freeing it to the Map from which it was allocated.

RTXC provides two basic Kernel Services to perform the operations of allocation and freeing of blocks from memory partitions. Additionally, three variants of the allocation function are possible.

Memory Partition Definition

RTXC supports both static and dynamically defined Memory Partitions. Static memory partitions have their attributes defined during system generation. The number of dynamic memory partitions is also declared during system generation. In use, dynamic memory partitions are allocated and have their attributes defined during runtime. Regardless of its type, a Memory Partition must exist and all of its attributes be defined in order for a task to make use of it.

The number of blocks defined in a Memory Partition is limited only by the amount of RAM available. It is not necessary that the number of blocks in a Memory Partition be a power-of-two or any particular number. All blocks in a Memory Partition must be the same size and must be at least the size of a pointer. However, blocks whose size is an odd number of bytes may be less efficient on processors that require at least 16-bit (word) access. You should consult your processor's reference manual to determine if odd number block sizes are efficient.

Static Memory Partitions

In keeping with the concept of predefinition, the system designer defines all static Memory Partitions during system generation using RTXCgen. Through an interactive dialog with RTXCgen, the user specifies the various attributes of each static Map including its name, the number of blocks it is to contain, and its block size. RTXCgen computes how much memory to allocate and produces the C structures and memory arrays to accommodate the specification. After compiling the C code produced by RTXCgen, the linking process establishes the actual address of the RAM so that RTXC will know where it is located.

Normal usage of static Memory Partitions keep their attributes fixed during the life of the application. However, RTXC does permit attribute redefinition for a static Memory Partition through the use of the Kernel Service *KS_defpart()*. If this capability is employed, it should be done with caution.

Dynamic Memory Partitions

Dynamically defined Memory Partitions are slightly different. Some applications, while knowing the number of Memory Partitions needed, defy accurate specification of their sizes until runtime when operating conditions are known. This leads to a problem with RTXC's concept of predefinition. The solution is for the system designer to specify undefinable Memory Partitions as being dynamically defined. The number of dynamic Memory Partitions is specified via RTXCgen but no attributes about them are given at that time.

When a particular need arises during system operation for a dynamic Map, a task can create one via appropriate RTXC Kernel Services. Allocation of the Map control block is the first step in creating a dynamic Memory Partition. The task issues a *KS_alloc_part()* Kernel Service to allocate an unused Map control block from the pool of free Map control blocks.

Having successfully allocated the control block, the task then must define the Map's attributes through the *KS_defpart()* Kernel Service. The dynamic Memory Partition is then usable as though it had been statically defined.

The task may choose to combine allocation of the dynamic Map control block and attribute definition into a single Kernel Service request, *KS_create_part()*.

If a dynamic Memory Partition no longer has utility to the application, it may be released. The *KS_free_part()* Kernel Service will release the Map control block to the pool of free Map control blocks. You should exercise care to ensure that all of a Map's memory blocks are freed to the Map prior to using *KS_free_part()*. Release of a dynamic Map's control block could leave any allocated memory blocks in limbo and potentially lost.

Number of Memory Partitions

The number of RTXC Memory Partitions is determined by the system designer according to the needs of the application. You may define the size of the storage quantum of type *MAP*. An 8-bit quantum is normally sufficient, permitting up to 255 Memory Partitions regardless of type.

Through RTXCgen, you must define each static Memory Partition and the number of dynamic Memory Partitions, *DNPARTS*. RTXCgen uses *DNPARTS* to generate an equal number of Map control blocks which form the free pool from which dynamic Map allocations or creations are made.

The total number of Memory Partitions may be found by adding the number of static Maps, *NPARTS*, to *DNPARTS*.

Memory Partition Organization

A Memory Partition is an area of RAM consisting of one or more blocks of the same size. Each Memory Partition consists of a Partition Header and the Partition Array. Collectively, they are referenced by a single Memory Partition identifier.

The Partition Header contains information needed by RTXC to manage the Memory Partition including the size of a RAM block and a pointer to the next available block. The Header may also contain other information about the usage of the Partition Array.

The Partition Array contains the actual memory blocks. While Memory Partitions may be either statically or dynamically created, the organization of the Memory Partition is the same. RAM blocks in the Memory Partition are contiguous and are linked together in a singly linked list.

Memory Partition Attributes

Each Memory Partition serves some purpose needed by the application and thus has unique attributes. These attributes are stored in the Map Control Block for use by RTXC during kernel services related to the Map. The attributes include:

- Memory Partition Identifier
- Block Size

- Number of Blocks
- RAM Area Address

Memory Partition Identifier

The system designer may specify the size of the data quantum needed for a Memory Partition identifier. These identifiers are numerical values of type *MAP*. The size of a value of type *MAP* defines the maximum theoretical number of Memory Partitions in a system. An 8-bit signed quantity permits up to 127 Maps.

The Partition, or Map, number is its position in the list of all Memory Partitions. There is no special significance given to a Memory Partition identifier.

Statically defined Memory Partitions will have identifiers between 1 and *NPARTS* inclusively, where *NPARTS* is the number of static Memory Partitions. Identifiers for dynamic Maps will range from *NPARTS+1* to *NPARTS+DNPARTS* inclusively, where *DNPARTS* is the number of dynamic Memory Partitions.

Block Size

The block size of a given Memory Partition is fixed and all blocks in that Map are initialized as having that size. Once defined, the Map's block size may not be varied. RTXC imposes no restriction on the size of a block other than it must be at least the size of a data pointer.

Number of Blocks

Each Memory Partition is created with a given number of fixed-size blocks. The product of the block size and the number of blocks determine the

amount of RAM needed for the Map.

RAM Area Address

The address of the RAM area used for the blocks in a static Memory Partition is defined by the linker. The RAM area address for a dynamic Map is defined at runtime.

Using Memory Partitions

The RTXC initialization procedure links all of the blocks within each static Map. The blocks of Dynamic Memory Partitions are linked by the *KS_defpart()* or *KS_create_part()* kernel services when the Map is created. During operation, a request to allocate a memory block returns the address of the next available block in the map. When the block is released, RTXC puts it back into the list of available blocks so that it will be the next block to be allocated.

RTXC also makes provisions for empty Memory Partitions. Tasks which attempt to allocate memory from an empty Map are informed of the conflict.

RTXC provides one basic Kernel Service for allocating memory, *KS_alloc()*, and one function for releasing memory, *KS_free()*. Three possible variants of the basic allocation function, *KS_allocw()*, *KS_alloct()*, and *KS_ISRalloc()* also provide kernel level support for handling empty Map conditions. The last variant, *KS_ISRalloc()*, is used by interrupt service routines to allocate a block of memory.

Allocation of a memory block, if successful, always yields the address of the allocated block. The task uses that address as a pointer to the block. The pointer to the block also serves as an argument for the *KS_free()* Kernel Service when it is time to release the block.

When using dynamic Memory Partitions, you may use a static area for the RAM or you may choose to allocate memory from the heap at runtime. The latter case should be used with caution as improper use of the heap can cause memory fragmentation. A third technique for acquiring the RAM needed for a dynamic Memory Partition is to allocate a RAM block from an existing Map.

This last technique can be quite powerful as it permits a nested definition of a RAM block. For example, a 16K byte block from a static Memory Partition can be allocated and used to define a new dynamic Map having eight blocks of 2K bytes each. In turn, one of those 2K blocks could be allocated and used to define yet another dynamic Memory Partition having eight 256 byte blocks. The subdivision can proceed to whatever depths you need for your application without the downside of fragmentation that exists with the second technique above which uses the heap.

Allocating Memory

The basic Kernel Service to allocate a block of memory from an RTXC Memory Partition is *KS_alloc()*. If there is a block available in the given Map, RTXC will allocate it and return a pointer to it. If there are no free blocks in the Map, RTXC will

return a value indicating the empty Map condition. The task will have to recognize the special return value and then deal with the situation with an appropriate program segment. *KS_ISRalloc()* operates exactly like *KS_alloc()* except it is intended for use by interrupt service routines instead of by tasks.

The use of *KS_allocw()* operates exactly like *KS_alloc()* as long as there are free blocks to allocate. However, when the given Map is empty, the Kernel Service does not return a value but instead blocks the requesting task, removes it from the READY List, and adds it to the Map's list of waiters. Waiting tasks are inserted into the waiter list in descending order of their priorities. The Map's highest priority waiting task will remain blocked until another task frees a memory block to the Map. When a block becomes available, it is allocated to the waiting task. The task is resumed with RTXC returning the pointer to the newly allocated block.

KS_alloct() operates exactly like *KS_allocw()* except that the duration of the wait is limited by a user defined timeout period. Instead of waiting indefinitely for a block, the task will wait until either a block becomes available or until the timeout expires. If the former, *KS_alloct()* returns the pointer to the allocated block and resumes the requesting task. If the timeout occurs, the requesting task resumes with a return value from *KS_alloct()* indicating the timeout condition. The task must then recognize the condition and deal with it in a special code segment.

Freeing Memory

RTXC provides one service to release a previously allocated block of memory, *KS_free()*. There are no variants of the *KS_free()* function. The Current Task need only provide the Memory Partition identifier to which the block will be released and the pointer to the block. RTXC makes no attempt to verify that the block was originally allocated from the designated Map receiving the freed block; so care must be taken lest the maps become corrupted with blocks of different sizes.

TIMERS

An RTXC system is usually configured with a time base using some periodic interrupt on the target processor as a clock. The clock permits task control on a timed basis. RTXC uses a generalized scheme using one-shot and cyclic timers in conjunction with semaphores. Multiple timers are managed simultaneously using an ordered list of pending timer events. Regardless of their number, the time to service all active timers is fixed.

A timer for an event is inserted into the active timer list in accordance with its duration. Insertion uses a technique that puts the timer with the shortest time to expiration is at the head of the list. RTXC allows one timed event to be co-terminous with another timed event. Kernel Services for scheduling and cancelling timed events are an integral part of the executive.

Timer Definition

RTXC uses two types of timers--General Timers and Timeout Timers. General Timers time system events such as the periodicity of a task's cyclic operation or the operation of some mechanical device in the physical process. Timeout Timers are a special type of timer used in limiting the duration of certain Kernel Services in blocking the requesting task.

It is during the system generation process that the system designer defines the clock interrupt frequency and the number of timers. The number of timers defined is the number of general timers

needed by the application. Timeout timers are not included in the set of defined timers because they are allocated via a separate mechanism.

At the end of the system generation process, RTXCgen produces an array of timer blocks called the Free Timer Pool. RTXC will create a linked list of the timer blocks in the Free Timer Pool during system initialization. General Timers are allocated from the Free Timer Pool by removing the first available timer block in the list. Similarly, timer blocks are freed by inserting them at the head of the linked list.

Timer Structure

Both General Timers and Timeout Timers have a common component of the remaining time counter. General Timers have an additional component in a recycle time. The remaining time counter is the amount of time remaining before the timer expires. A recycle count, if non-zero, defines the amount of time with which to reset the timer when the current time remaining expires. RTXC time period values usually have a maximum length of 16-bits or 32-bits depending on the target CPU.

If only the initial period is defined, the timer is said to be a one-shot timer. If it has an initial period and a non-zero cyclic period defined, the timer is cyclic. The initial period may or may not be equal to the recycle time. Only General Timers may be either cyclic or one-shot. Timeout Timers are one-shot timers only.

Each General Timer also has a semaphore associated with its expiration event. The association of the semaphore to the timer expiration is made when the task issues a Kernel Service request to start the timer. The semaphore is signaled when the timer expires.

Timer TICKS

The basic time unit used internally by RTXC is a **TICK**. A **TICK** defines the amount of time between interrupts generated by the system clock, or equivalently, the period between clock interrupt service requests. The frequency and Tick granularity of the system clock is hardware dependent and is usually defined during system generation.

Timer values are equivalent to the number of clock Ticks required to form the needed amount of real time. For example, if a system clock operates at 64 Hz (15.625 msec per Tick), a one-shot timer of 2 seconds has an initial period specification of 128 TICKS (2 x 64).

All timed event operations and data structures are handled by RTXC. While a timer is active, a task should not attempt to manipulate any of its control or data structures.

Using General Timers

General Timers are suitable for general purpose timing, including such uses as timing events and

establishing periodic task activation. General Timers may be one-shot or cyclic and may be allocated, started, restarted, stopped, and freed.

RTXC requires that a timer be allocated before it can be used. Once allocated, it remains so until it is released by the owning task. While it is allocated, it may be started and restarted as many times as required by the application. And more than one timer may be allocated to the same task at the same time. When the task no longer needs a timer, it may release the timer to the Free Timer Pool where it can be reused by other tasks.

General Timer Allocation

RTXC uses a concept of allocation of timer blocks prior to their use. This provides for very deterministic operation in that a task attempting to allocate a timer knows immediately whether the operation was successful. Without preallocation, a task could attempt to perform a timer management operation at a critical point and fail because a timer was unavailable.

The preferred design for an RTXC task using timers is to have it allocate all of its needed timer blocks before starting the main body of the task. Allocation prior to use guarantees the task that the necessary system resources will be available when needed. An added benefit of timer allocation prior to main body operation is that an unsuccessful allocation attempt can indicate the presence of a problem elsewhere in the system.

Allocation of a timer is accomplished by unlinking the next available timer from the Free Timer Pool and returning its handle to the requesting task. *KS_alloc_timer()*, the RTXC timer allocation Kernel Service, performs that operation. The function does not create the timer nor does it start a timer. Instead, a successful allocation returns the handle of a timer block to the Current Task.

Having the timer handle, the task may use it in subsequent timer management Kernel Services. A task may allocate and use more than one timer concurrently. Any task using timers should maintain the handle of each timer block allocated until such time as the block is to be freed, if ever.

Even with a design which uses allocation prior to use, a condition may arise in which there are no timer blocks in the Free Timer Pool when a new timer allocation attempt is made. This condition is indicated by the function value returned from the *KS_alloc_timer()* Kernel Service. It is the responsibility of the task to deal with the situation should it occur.

Automatic Timer Allocation

The concept of timer block preallocation prior to first use is the preferred method. However, it requires an explicit request which may not be acceptable for all system designs. RTXC provides for an alternative method of timer allocation by which the allocation of a timer is implicit. Under ordinary circumstances, this technique is just as good as the preferred method. Nevertheless, there are some caveats associated with its use.

The function *KS_start_timer()* is used to start a timer whose handle is provided as an argument to the Kernel Service request. But, RTXC allows *KS_start_timer()* to be called in a manner to indicate that RTXC is to allocate and create and start the timer. This technique is referred to as automatic, or implicit, timer allocation.

If *KS_start_timer()* is successfully used in automatically allocating a timer, it will return the handle of the timer to the requesting task. Likewise, a failure to allocate a timer will cause the Kernel Service to return a function value indicating that no timer blocks were available. A task using this Kernel Service with automatic timer allocation should be able to detect successful or failed attempts. For successful attempts, the task should maintain the handle of any timer block allocated implicitly.

When using automatic timer allocation, care must be taken to prevent a task from unwittingly misusing *KS_start_timer()*. Improper use would include using the same storage variable to save an allocated timer's handle while making repeated calls to the function. Each call would cause the unrecoverable loss of the timer handle from the previous call. Eventually this kind of improper use will exhaust the Free Timer Pool as evidenced by a NULL handle being returned. If the task is not monitoring the returned value from *KS_start_timer*(), it might try to use the returned NULL handle later on.

Reading Time Remaining

RTXC provides the services to read the time remaining on any active General Timer provided the timer's handle is known. The *KS_inqtimer()* Kernel Service will return the amount of time remaining on the object timer in units of TICKS.

Stopping and Restarting

Sometimes it is necessary to abort an active timing operation prematurely. RTXC permits this through use of the *KS_stop_timer()* Kernel Service. The Current Task must provide the handle of the active timer to be stopped as part of making the request. If the task attempts to stop an inactive timer, nothing happens except that RTXC returns a value which indicates that the specified timer was inactive. The task can check for that occurrence if it is important.

Another use of the active timer's handle is found when attempting to change the expiration time of an active timer. The *KS_restart_timer()* Kernel Service performs such an operation when called with the active timer's handle and the new duration of the initial timer period. The timer is stopped at the time of the function request and the new time value replaces whatever remains in the remaining time field.

Freeing Timers

A task may determine at some point that it no longer needs a General Timer it had previously allocated. A good design philosophy is to release the unneeded timer. RTXC provides a simple Kernel Service, *KS_free_timer()*, for doing that. The task need only provide the function with the handle of the timer

block to be released and put back into the Free Timer Pool.

Using Timeout Timers

Another type of timer used by RTXC is the Timeout Timer. These are timers which are used by tasks which invoke Kernel Services using timeouts. Timeout Timers, unlike General Timers, need not be allocated and released by the tasks that use them. Instead, RTXC performs those actions automatically as part of its operations. Only one Timeout Timer will be allocated to a task at a time because a timeout may only occur for one Kernel Service at a time.

Timeout Timer Allocation

Beginning with RTXC V3.1, the handling of Timeout Timers was changed from the methods used in previous versions. In RTXC V3.1 and later versions, the processing of Timeout Timers is completely automatic and transparent to the user. The Kernel Service *KS_alloc_timeout()* will no longer be included in the API of RTXC after V3.1. If you are a user of RTXC V3.0, you may eliminate all uses of *KS_alloc_timeout()* as it is no longer in the RTXC API.

Beginning with RTXC V3.1, Timeout Timers are allocated on the stack of the Current Task when the task requests Kernel Services which need to block the task only for a limited time.

Freeing Timeout Timers

The area on the task's stack used by the Timeout Timer is released automatically by RTXC when either the expected event occurs or the timeout expires. Either situation will cause the waiting task to resume.

The *KS_free_timeout()* Kernel Service is no longer included in the API for RTXC V3.2. It was present in RTXC V3.1 only for purposes of compatibility with RTXC V3.0. It may be omitted from the RTXC API if prior compatibility is not a concern.

Timer Interrupts

When there is an active timer, each interrupt of the system clock causes the active timer values to be reduced by one TICK. When a timer expires, its timer block is removed from the Active Timer List and the semaphore associated with the timed event is signaled. A task waiting on the event will be unblocked and inserted into the READY List if it has no other blocking conditions. The timer block is set to an inactive state but is not returned to the Free Timer Pool. It remains available to the task for subsequent timer management operations. A context switch can occur if the unblocked task is of higher priority than the interrupted task.

SYSTEM TIME

RTXC maintains system time as a 32-bit datum of type *time_t* that is incremented once each second after the system is initialized. In this manner, a very deterministic means of maintaining time-of-day and date is available. The calendar may be defined with the current date and time expressed as the elapsed number of seconds since January 1, 1970. If defined with a valid date on or after January 1, 1970, the calendar is accurate through the year 2037. It is not required, however, that the calendar be defined with a date and time in order for RTXC to operate properly.

The RTXC distribution provides two functions, *date2systime()* and *systime2date()*, which can convert standard calendar data (Year, Month, and Day) and clock data (Hours, Minutes, and Seconds) to the system time of type *time_t* and back again. These functions are general utilities and are not part of the RTXC API.

Conversion to System Time from Calendar Date

Function *date2systime()* is provided to convert a calendar date and clock data to the internal system time of elapsed seconds since January 1, 1970. The function requires a single argument which is the address of a structure containing:

- Year
- Month

- Date
- Hours
- Minutes
- Seconds
- Daylight Savings Time Flag

Conversion from System Time to Calendar Date

Function *systime2date()* converts the internal system time value to the corresponding calendar date and time in terms of year, month, day, hours, minutes, and seconds. The function requires an argument that is the address of a structure of type *time_tm* containing the calendar and clock members:

- Year
- Month
- Day
- Hours
- Minutes
- Seconds
- Day-of-Week

The function returns the numerical values for the calendar as year, month (1-12), day (1-31), and day-of-week (1-7, where Monday = 1). For clock data, the function returns hours (0-23), minutes (0-59), and seconds (0-59).

INTERRUPT SERVICE

RTXC also provides for a generalized interrupt service scheme. Because Interrupt Service Routine (ISR) code is specific to both the particular device and the method of use in the application, it must be provided by the User. Fortunately, the rules for writing RTXC interrupt service routines are quite simple.

While the hardware specifics of interrupt recognition and acknowledgment vary from CPU to CPU, software handling of interrupts is more consistent. In RTXC, there are three basic parts to all ISRs:

- Prologue
- Device servicing
- Epilogue

The prologue begins the processing of the interrupt. The device servicing section deals with the particular device. The epilogue is the end action performed to finish interrupt processing and continue with the application. More complete descriptions of these sections follow in the paragraphs below.

Prologue

When the ISR is entered after acknowledgment of the interrupt, it begins a code section called the ISR prologue. The prologue is usually written in assembly language and may be either straight-line code or a macro. Whichever the case, it is a normal

part of the RTXC distribution. Unless it is necessary to perform some additional operation during an ISR prologue, this code, as distributed, should not require modification.

The purpose of the ISR prologue code is to save the processor context plus any extended context necessary to preserve the interrupted environment. The processor context is stored on the task's stack while any extended context is stored in a special area. The state of the CPU interrupt facility may or may not be enabled throughout this storage process depending on the specifics of the CPU.

Device Servicing

After storing the context, the ISR prologue transfers control to the main function of the ISR to service the interrupting device. This is usually a C function which performs some device specific operation in order to clear the source of the interrupt request. As it deals with application specific devices, this code must be furnished by the user.

All device servicing functions in RTXC require one argument, a pointer to the stack frame of the interrupted process. Because the prologue is written in assembly language and the device servicing function is written in C, the prologue code must pass the stack frame pointer in a manner consistent with the conventions of the compiler for argument passing between C and assembly language. The device service function does not use the argument in its operation but only passes it to the ISR exit logic.

As part of its operation, it is quite common that the device servicing function will need to signal one or more semaphores associated with the interrupt in order to announce the event to the application tasks.

RTXC provides two special services to deal with commonly encountered requirements of interrupt processing. The function *KS_ISRsignal()* should be called to signal a semaphore from the ISR while *KS_ISRtick()* provides RTXC required processing for a clock tick.

An ISR should not make calls to RTXC Kernel Services, because the kernel is not reentrant and calls from an ISR to kernel services other than *KS_ISRsignal()*, *KS_ISRtick()*, or *KS_ISRalloc()* will yield unpredictable results.

When its device specific operations are complete, the device servicing function indicates that fact by calling the fourth special interrupt service function, *KS_ISRexit()*.

NOTE: The function *KS_ISRexit()* serves the same purpose as function *isrc()* in previous versions of RTXC. Whatever is said about *KS_ISRexit()* in this manual may also be said about *isrc()*. However, beginning with RTXC V3.2, *isrc()* is no longer included in the RTXC distribution.

One of the arguments to the function provides a convenient way of combining the exit logic with signaling a semaphore associated with the interrupt. *KS_ISRexit()* will also arbitrate the priorities of any

tasks made Ready by that signaling. When *KS_ISRexit()* is finished, the highest priority Ready task will be at the head of the READY List. Function *KS_ISRexit()* returns a pointer to the stacked context of the highest priority Ready task. Having that datum, the next step in an Interrupt Service Routine is to enter the ISR epilogue.

Epilogue

The ISR epilogue code, like the prologue, is usually in assembly language. It sole function is to restore the context of the highest priority Ready task and grant it control of the CPU. The highest priority Ready task may or may not be the task that was interrupted. Code for the ISR epilogue is included in the RTXC distribution and should not require changing.

A crude code model of an ISR for UART input/output including the use of *KS_ISRexit()* and *KS_ISRsignal()* is shown below. In the example, **CONISEMA** and **CONOSEMA** are semaphores for the character input and output events respectively.


```
/* Interrupt service example - UART output
 * KS_ISRexit() only allows for signaling a single
 * semaphore per interrupt.
*/
/* C level UART device service function */
FRAME *uartc(FRAME *frame)
{
   /* test source of interrupt */
   if (USART_STATUS == TX_BUFF_EMPTY)
   {
      /* output DONE */
      /* clear interrupt logic goes here */

      /* exit and signal char output semaphore */
      return(KS_ISRexit(frame, CONOSEMA));
   }
   else  /* if here it is USART input */
   {
      /* do some more processing */

      KS_ISRsignal(CONISEMA); /* signal event */
      /* followed by still more processing */

      return(KS_ISRexit(frame,0)); /* get out */
   }
}
```

TICK Processing

Most real-time systems employ a device which interrupts the CPU at regular intervals to provide a time base to the system. Naturally, there are many ways to implement such a timing device, or clock, but the design is immaterial to the use of RTXC. It is sufficient to say that such a device may exist in an RTXC-based real-time system. Because of the diversity of hardware designs for such a timer, it is possible to conclude that there would be at least an equal number of ways to handle a clock interrupt.

That's the bad news. The good news is that the way that the interrupt needs to be handled with respect to the needs of RTXC is quite regular. In recognition of that regularity, the RTXC distribution includes a special purpose function, *KS_ISRtick()*, that performs all of the necessary processing required for a clock interrupt (*i.e.*, a TICK). There would be only one instance of this function's use in the system - in the ISR of the driver for the system's periodic time device.

The function requires no argument as it is dealing with known objects but it returns a single value to indicate that a timer expired or not. The user may make use of this value if desired. The clock ISR (which called *KS_ISRtick()*) should then call *KS_ISRexit()* to conclude its processing. The RTXC distribution includes a prototype driver for such a system clock.

SECTION 4

TUTORIAL

Table of Contents

TUTORIAL

INTRODUCTION

This section is intended to answer some of the questions about using a real-time kernel and about using RTXC specifically. Due to the focus on RTXC, it is not an exhaustive treatise on the subject. Most users of RTXC already have some experience with this type of system software but a number of users are not well versed in the inner workings and hidden mechanisms of real-time system design and multitasking. It is for this last group that this section is targeted. Hopefully, more experienced users may also find valuable information herein as well.

The tutorial makes the assumption that you have a working knowledge of application design using multitasking and real-time kernels. We hope that you have also read at least sections 2, 3, and 7 of this manual and have an understanding of how RTXC works. If that is not the case, you probably ought to stop right here and read those sections before continuing.

MECHANICS

What do you do after you have opened the package? So it sounds like a stupid question, but we're going to assume nothing.

Backup

The first thing you should do after opening the package is to make a backup of the RTXC distribution diskette. You may do that for archival purposes only and you should keep the backup copy in a safe place. You will need a high density 5¼" or a 3½" diskette to make the backup.

The RTXC distribution diskette is a high density medium written in MS-DOS format. You may perform the copying by using the MS-DOS *COPY* command or a third-party copying utility program. After you have made your backup copy, be sure to label the diskette and include the Diskette Serial Number (Diskette S/N) that is on the label of the RTXC distribution diskette.

Now that you are protected, you need to install RTXC on your development system.

Installing RTXC

The entire RTXC package is contained on a single high density 5.25" or 3.5" diskette. The files are not compressed, so there is no need for any special decompression programs. The utilities on MS-DOS

are sufficient. But before going too far, you should take a moment and read the **Customer Letter** that accompanied your package.

Binding Manual

One of the things the Customer Letter mentions is the RTXC Binding Manual. It is located on the RTXC distribution diskette in the root directory in a file named **BINDING.TXT**. Remember, the RTXC User's Manual is a general document that is generic to all processor and compiler combinations. It is the Binding Manual that holds specific information about the processor and compiler.

BINDING.TXT is a preformatted file with embedded form feeds so that it can be printed easily. It is formatted for a standard 8½" X 11" page size but it should also work with a standard A4 page size as well. You should print the file and examine it before proceeding with the installation.

Copying the RTXC Files

The Binding Manual provides specific information about copying the files from the RTXC distribution diskette to the working disk of your development system. You may modify the procedures in whatever way you need in order to be compatible with your particular development environment.

The normal procedure of copying assumes that the destination disk is the *C:* drive. The copy process produces a root directory entry on the *C:* drive named *RTXC*. The root directory in turn contains three subdirectories named *KERNEL*, *DEMO*, and *RTXCGEN*. They contain, respectively, the RTXC kernel source code, the source code for a

demonstration application, and the RTXC system generation utility program RTXCgen.

Make whatever changes you need for your development environment and proceed to copy the files from the RTXC distribution diskette to your hard drive.

Confidence Test

Once the RTXC files are copied to your development system, you should verify proper installation. This is a fairly simple procedure because the RTXC distribution includes Make files for each subdirectory. You should have confidence in your installation if you can build the entire RTXC system and the demonstration application without any errors. You may or may not have the proper target hardware environment on which to test the demonstration application. But if you can build it, you have set up the development environment properly so that work on your actual application should be quite smooth.

Make Files

Each of the subdirectories, *KERNEL*, *DEMO*, and *RTXCGEN*, has at least one Make file included as part of the standard RTXC distribution. The Make file is usually written to be used with the Make facility in both the Microsoft C compiler V6.0 and in Borland's C++ V3.1 compiler.

The distributed Make files assume that you have installed the RTXC files on the *C:* drive. If you have used another disk drive, you should modify the

Make files so that the paths to the RTXC files reflect your actual installation.

The Make files further assume that you have installed your C compiler in the manner described in the Binding Manual. The path of the compiler reflects the normal results of following the stated installation procedure of the compiler vendor. If your installation uses a different path, you will need to reflect those differences in the path descriptions used in the Make files.

If you are using a different Make utility that is incompatible with the distributed Make files, you must create your own Make files for whichever Make utility you are using. Once you have done so, building the distributed demonstration application should provide you with sufficient confidence that your installation is correct.

Assuming you have the proper compiler path, that you have stored the RTXC distribution files, and that the Make files are compatible with your Make facility, you can now proceed to build the RTXC demonstration application.

Of course, if you don't use a Make utility, you can create a batch file and do the same thing. The benefit of using the Make utility is that it will help you by compiling only those files that have changed. Later on, when you are deeply into your own application development, you may find that the Make utility can improve your system build turnaround time.

Building the Demo Application

You have your Make files or your batch files ready, and now you are ready to build the demonstration application. The procedure is simple. Set the working directory to the *KERNEL* subdirectory and invoke your Make utility for the Make file in that subdirectory. After that operation completes, select the *DEMO* subdirectory and make it.

If the *DEMO* Make file completes normally, you have just built the demonstration application successfully. If there is an error, you should ensure that you have the proper paths to the compiler and the RTXC files and that your Make file is indeed compatible with your Make utility. Correct any errors you find and try it again. Continue this process until you get a successful completion.

DEMONSTRATION APPLICATION

The demonstration application is a simple set of tasks that can run on some system containing the target processor you have chosen. If you have the specific target hardware, you should be able to load the demonstration application and see it run. However, if your target board is different, seeing the demonstration application actually operate may require further work.

While the demonstration application is simple, it nevertheless incorporates much of what you must have in your specific application. If you understand what is in the demonstration, you can translate that knowledge into your own particular requirements. This part of the tutorial is intended to do just that.

Directory Contents

Why have three directories for such a simple application? The RTXC distribution files are divided into the three subdirectories for the specific purposes of promoting ease of use, maintainability, and improving debugging.

The *KERNEL* subdirectory contains those files which define RTXC and how it is to be built. Once you have it configured the way you need and compiled into the RTXC Kernel Library, it does not need changing very often. Because it includes nothing about any particular application, the kernel is a reasonably constant element in an application. The segregation of the kernel also promotes its

maintenance by there being only one copy of it on the system. Maintenance reports and updates need only be made in one directory.

The *DEMO* directory contains all of the files that are specific to the given application, in this case the demonstration. The content of *DEMO* can be copied to another directory, there to serve as the starting point for a particular application. As it contains only application specific files, you need only to add new application files, edit the existing kernel object definitions, modify the existing device drivers or add new ones, and you have your own directory for a new application.

The *RTXCGEN* directory is likely to change the least of all. RTXCgen is the system generation utility, and it exists not only in source form but also as an executable file, **RTXCGEN.EXE**. It is usually sufficient to run **RTXCGEN.EXE** as distributed and never make any changes to it. However, if you want to make some changes, you have the complete source code with which to work.

Elements of the Demonstration

The function of the demonstration application is to have two tasks outputting brief messages at different times. One task outputs its message once per second. The second task performs its function once every 5 seconds. Assuming that the output is to some sort of display device, the visible output shows an endless cycle of five repetitions of the first message followed by one message from the second

task.

On the surface such functionality appears trivial but underneath there is more than meets the eye. There is the RTXC real-time kernel providing not only the library of real-time services but also the architecture for the application. And there are certainly the two tasks which perform the periodic message output. But there is more.

For example, there must be some mechanism for keeping track of time because both tasks operate cyclically with respect to time. There must also be some sort of support for a serial output port through which the various messages are transmitted. Both time and serial output involve the handling of interrupts. Thus, the supporting code must be able to respond to and process asynchronous and synchronous external interrupts.

Another piece of executable code that must be a part of the application is the startup code. This is a piece of code, often written in assembly language, whose charge it is to start up the processor and perform low level or processor specific initialization functions.

Lastly, there must also exist a set of kernel objects which define to RTXC its environment for this particular application. While these definitions are not specifically executable program code, they are produced by RTXCgen, the system generation utility, as C source code modules. They are compiled

and linked with other object modules just as though they were actual executable code.

Reduce all of these various elements to object form by compilation or assembly. Then link all of their object modules together, i.e., kernel, application tasks, device drivers, kernel objects, and startup code and you get the application as an executable module. Now you have the total picture.

Let us visit these elements and see what makes them work and what may be necessary to adapt them to your particular use.

RTXC Kernel

The RTXC Kernel Library was made during the initial building process when you did the confidence test. It was compiled according to a given set of Configuration options found in a C header file named **RTXCOPTS.H**. The header file is also found in the *KERNEL* subdirectory. But we will save a more complete discussion of this file and its role until later in the tutorial. The kernel library always has the name **LIBRTXC** but its extension may vary according to conventions used by the librarian utilities of different compilers. Usually the extension is **LIB** making the complete name of the library **LIBRTXC.LIB**. Consult your C compiler user's manual for the correct extension name.

Startup Code

The startup code is application and processor specific. It is usually found in a file named **START.***, where ***** is the file extension specific to the compiler or assembler in use. Most compilers targeted for embedded applications include startup

code that can be modified for a user's specific application. The distributed *DEMO* directory file **START.*** contains that same code, when it is available, but modified to include any RTXC specific startup requirements and definitions. If you don't find a file named **START.*** in your *DEMO* directory, the file may have another name. You may check the binding manual for the name of the startup code file in your distribution if one exists. In some cases the startup code distributed with the compiler does not require changing so it is not included in the RTXC distribution.

Kernel Objects

The demonstration application needs definitions of its environment in order for RTXC to know what to do. These definitions are known collectively as *kernel objects*. We defined a set of kernel objects specifically for the demonstration application using the system generation utility, RTXCgen. The output from that exercise is found in the *DEMO* directory in three forms: 1) kernel object definition files, 2) kernel object C files, and 3) kernel object header files.

Kernel object definition files have the extension **DEF** and contain the definition of each type of kernel object defined during system generation.

Kernel object C files contain the C data structures and related source code translations of the kernel object definition files. These files use the extension of **C**.

Kernel object header files use the file name extension of **H** and contain definitions of the names of the various static objects defined in the kernel object definition files.

The three types of kernel object files are all related, and that relationship is evidenced by the actual naming of the files. All of the files have the same file name; only the extensions change. There are seven such files and their names are:

- **CCLOCK** - Clock (periodic timer) objects
- **CMBOX** - Mailbox objects
- **CPART** - Memory partition objects
- **CQUEUE** - Queue objects
- **CRES** - Resource objects
- **CSEMA** - Semaphore objects
- **CTASK** - Task objects

Clock Driver

The ability to handle a periodic interrupt for the purpose of time management, while actually not necessary for RTXC to operate, is a customary component of a real-time system. When it is used, as in the demonstration application, it is an important component; so let's go into it in enough detail that you understand how it works and what you must do if you use a different interrupt source than that used by the demonstration.

In RTXC, this component is called a clock driver and is part of the *DEMO* directory as a file named **CLKDRV.C**. The clock driver may contain a device initialization function which sets up the clock according to the clock frequency (**CLKRATE**) and tick rate (**CLKTICK**) defined during system generation. This is not always the case as there are some microcontrollers that must have the periodic timer on the chip initialized within a specified amount of time following a reset condition. (This is one reason the clock driver is in the *DEMO* directory - it is application specific.)

The clock driver is divided into two parts: an interrupt servicing function, and an interrupt service prologue. In **CLKDRV.C**, you will find the C interrupt servicing function of the clock driver. This device servicing function for the clock ISR is named *clkc()* and is called by the interrupt service prologue. The function, *clkc()*, takes care of servicing the actual interrupt caused by the periodic timer device. That code is very hardware dependent but quite simple in its operation. Basically all it does is to clear the interrupting device and then call *KS_ISRtick()* to increment the RTXC tick counter Then, by calling *KS_ISRexit(), clkc()* returns to its caller which is the interrupt service prologue.

The interrupt service prologue is written in assembly language because it must deal with the specifics of the particular processor; e.g., saving the processor context, ensuring that the proper stack is being used, etc. You may see the prologue written as an assembler macro or it may be straight instructions.

All RTXC distributions have an assembly language file, usually named **RTXCASM.***, where the ***** represents the extension for an assembly language file compatible with the assembler being used, that contains the interrupt service prologue.

Whatever is the case on your distribution, it would be good advice not to change that code. The interrupt service prologue deals with the processor, and it is very specific. The interrupt servicing function handles the actual interrupt but is often completely independent of the processor.

To make a new clock driver for your application, you need only to change the function *clkc()* to handle whatever you are using as the source of your system's periodic interrupts.

Serial I/O Driver

A serial I/O port is often utilized by real-time applications to receive and transmit information, and a serial I/O driver is employed to handle those functions. A serial driver usually has one task for the receive (input) operations and another for transmit (output) duties. However, due to various implementations of serial ports in hardware, there may be one or two interrupt service routines required.

You should keep in mind during this tutorial and while examining the related source code that these drivers may not suit your application needs. Indeed, they are rather simple in their behavior and it is quite likely that you will need to make modifications to them. But take them for what they are, prototypes

or code models for your own drivers. Remember, the drivers you need are specific to your application's requirements, and only you know what they are. What you get with the RTXC demonstration is a good look at how a working driver operates, how it uses RTXC kernel services, how interrupts are serviced, and how the whole thing is put together to form a working component.

Let's use the output side of the serial I/O as our focus in this tutorial as we know it must be present. Although the demonstration requires only the functions of the output driver, the input driver is often included.

The organization of a serial I/O driver in the demonstration application is much like that for the clock. In fact, you will see the same organization in all RTXC device drivers and that makes it simple to write one and get it installed quickly. There is a task portion, an interrupt servicing function, and an interrupt service prologue. Because you should not be changing the interrupt service prologue code, let's concentrate on the code found in the output driver file. It will be in your *DEMO* directory under one of several names, but you can refer to your Binding Manual to find the exact name being used. For tutorial purposes, let's call it **UARTOUT.C**.

It will have an entry in the task definitions header file, **CTASK.H**, defining the task name as **UARTOUT**. Its task's entry point will be *uartout()*. The driver design is based on getting a character from the serial output queue and outputting it to the

appropriate register of the UART or serial I/O port. Once the character is output, the task waits for the character to be output which is an event associated with an RTXC semaphore. When the event is signalled, the device is clear, and another character can be obtained from the queue and the cycle repeated.

The name of the serial output queue usually follows the name of the consumer task. In our tutorial we are using **UARTOUT** as the task name so it would be logical to name the queue **UARTOUTQ**. (The name of the queue in your demonstration may be different.) To get a character from the **UARTOUTQ**, the task uses the *KS_dequeuew()* naming **UARTOUTQ** as an argument. The use of *KS_dequeuew()* ensures us that the process will operate only if there is a character in the queue which it can remove. Otherwise, RTXC will block the task and it will not be allowed to regain control of the CPU until another task puts a character into the queue.

When a character is returned to the task from the *KS_dequeuew()* request, it is output to the serializing device in whatever manner is appropriate to it. Conversion of the bits of the character loaded in parallel into a serial bit stream requires an amount of time related to the baud rate of the device. (The baud rate was selected during execution of an initialization function when the driver was initially entered.)

During the time that it takes to serialize the bits in the character, the CPU can perform other tasks. This is one of the basic tenets of multitasking architectures. Thus, **UARTOUT** executes a *KS_wait()* RTXC kernel service naming the semaphore as the one associated with the event. (Let's call that semaphore **UOUTSEMA** for our purposes here. Its name could be different but it will be shown in the source code in **UARTOUT.C**.) Executing the *KS_wait()* function allows RTXC to block **UARTOUT** and perform one or more other tasks, if Ready, until semaphore **UOUTSEMA** is signalled indicating that the character has been completely serialized.

How did **UOUTSEMA** get signalled? Answering that question goes right to the heart of how interrupts are processed in RTXC, and how context switches can occur as a result.

When the interrupt occurred, control of the CPU was transferred to the service prologue for the transmit interrupt. After saving the necessary processor and system context, the prologue code called the interrupt servicing function *cuout()* where the device is actually serviced. (The code for *cuout()* is in C and is part of the file **UARTOUT.C**.) In *cuout()*, the interrupt source is cleared and the semaphore associated with the event, **UOUTSEMA**, is signalled by passing it to the common interrupt service exit function, *KS_ISRexit()*.

That answers how **UOUTSEMA** is signalled, but there is more to the story. Following it through the complete processing sequence shows us how all interrupts are serviced. Function *KS_ISRexit()* calls another function, *postem()*, which signals semaphore **UOUTSEMA** and the waiting task, **UARTOUT**, to be made runnable. It also places the task's TCB in the READY List at a position according to its priority relative to other tasks in the READY List. Finally, it determines the Ready task having the highest priority and returns its TCB address to *KS_ISRexit()*, which in turn returns it to the interrupt servicing function for the driver, *cuout()*, which then passes it to the interrupt service epilog (common to all ISRs).

In the ISR epilog, the address of the highest priority task is used to restore the task's context. Control of the CPU is given to the task at a point indicated by the content of the processor's Program Counter (or Instruction Pointer) register of task's context.

In our example, task **UARTOUT** would have been automatically put back into the READY List and a new character output cycle begun. And that is not only how **UOUTSEMA** was signalled as a result of the interrupt, but how the interrupt is processed.

Application Tasks

Besides the driver task for the serial output, the demonstration application uses two others, **DEMO1** and **DEMO2**, as the two tasks which actually generate the messages. Let's examine how they operate and how they use RTXC to accomplish their function.

The two demonstration tasks simply run periodically and output a message. There is some synchronization at the beginning, using a semaphore handshake. Beneath the surface there are queueing operations and exclusive resource access going on as well.

Functionally, we want **DEMO1** to synchronize the startup of **DEMO2** by signaling semaphore **DEMOSEM0** and then to wait on a handshake signal by **DEMO2** to semaphore **DEMOSEM1**. Having received the handshake, **DEMO1** knows that it can proceed, and it enters a "forever" loop where it outputs a simple text message. In performing the output of the text, the task uses exclusive access to resource **CONRES** to prevent corruption of the message by output generated by **DEMO2**. This is done by passing the resource identifier to the function, *printl()*.

The RTXC distribution includes the function, *printl()*, to move text from a buffer to a queue. **DEMO1** formats the text message in a character array called **buffer** and then uses *printl()* to move the text from **buffer** to the Console Output Queue, **CONOQ**. When it is finished, **DEMO1** delays for a period of 1 second (1000 msec/**CLKTICK** ticks/msec) and then repeats the output message for the next cycle.

The code for demonstration task **DEMO1** appears below.

```
/* demo1.c */

/*
 *   RTXC    Version 3.2
 *   Copyright (c) 1986-1994
 *   Embedded System Products, Inc.
 *   ALL RIGHTS RESERVED
*/

#include "rtxcapi.h"
#include "cclock.h"   /* CLKTICK */
#include "cres.h"     /* CONRES  */
#include "cqueue.h"   /* CONOQ   */
#include "csema.h"    /* DEMOSEM0, DEMOSEM1 */

#define SELFTASK ((TASK)0)

extern int printl(char *, RESOURCE, QUEUE);

demo1()
{
   KS_signal(DEMOSEM0); /* tell task 2 to display startup message */
   KS_wait(DEMOSEM1);   /* wait for response */

   for (;;)
   {
      printl("Demo task 1: delay\n", CONRES, CONOQ);
      KS_delay (SELFTASK,(TICKS)1000/CLKTICK);
   }
}
```

Turning our attention to **DEMO2**, we can see from **CTASK.C** that it is of lower priority than **DEMO1**. We can also see that when **DEMO2** was first initialized, it waited on **DEMOSEM0** from **DEMO1**. When this event occurred, **DEMO2** signaled **DEMOSEM1** then allocated a timer block and set up a cyclic 5 second timer (5000 msec/**CLKTICK** ticks/msec) whose expiration is associated with semaphore **DEMOSEM2**. After establishing the timer, **DEMO2** enters a "forever" loop and immediately waits for **DEMOSEM2**.

The expiration of the cyclic 5 second timer will cause semaphore **DEMOSEM2** to be signaled. Due to the rules of preemption, **DEMO2** cannot get control of the CPU until **DEMO1** is blocked and this event occurs.

The code for demonstration task **DEMO2** is shown below.

```
/* demo2.c */
/*
 *   RTXC    Version 3.2
 *   Copyright (c) 1986-1994.
 *   Embedded System Products, Inc.
 *   ALL RIGHTS RESERVED
*/

#include "rtxcapi.h"
#include "cclock.h"   /* CLKTICK  */
#include "cres.h"     /* CONRES   */
#include "cqueue.h"   /* CONOQ    */
#include "csema.h"    /* DEMOSEM0, DEMOSEM1, DEMOSEM2 */

#define TMINT ((TICKS)5000/CLKTICK)

extern int printl(char *, RESOURCE, QUEUE);

demo2()
{
   CLKBLK *pclkblk = KS_alloc_timer();

   KS_wait(DEMOSEM0);
   printl("Demo task 2: semaphore from Demo task 1\n",CONRES,CONOQ);

   KS_signal(DEMOSEM1);
  /* start cyclic timer */
   KS_start_timer(pclkblk,TMINT,TMINT,DEMOSEM2);

   for (;;)
   {
      KS_wait(DEMOSEM2); /* wait for sema from timer */
      printl("Demo task 2: timer\n, CONRES, CONOQ);
   }
}
```

Variations on a Theme

The tasks described above are, admittedly, trivial examples but they do serve to show how RTXC tasks look. Armed with the knowledge of how the demonstration application works, and with your confidence in the ability of your Make files to build an application, why not try a few modifications to **DEMO1** and **DEMO2** just for fun.

You can make changes directly into the *DEMO* directory or you can copy it to another directory. If you do copy *DEMO* to another directory, be sure that the paths used in your Make files are correct.

You will not have to rebuild the RTXC Kernel Library, **LIBRTXC**, as it should be unchanged for purposes of these exercises.

What do you do? A simple approach would be to implement some of the examples in Section 5 of this manual as parts of **DEMO1** and **DEMO2**. Doing so may not even require changes but to one task or the other. The choice is yours.

BUILDING YOUR OWN APPLICATION

Now that you have studied how RTXC works in the demonstration application and have written some elementary operations on your own, you are ready to launch into your own application.

How do you get started? The first thing to do is to create a new subdirectory under the *RTXC* directory and give it a name, say *MYAPPL*. Then copy the content of the distributed demonstration application to the new directory. Now you have a base from which to begin your own application development.

Device Drivers

Device drivers must be made or modified to meet your application's input and output requirements. Presuming you will use a periodic timer, you will want to have a clock driver. If you need a serial I/O driver, you have a place from which to begin. For other devices, you should follow the general outline of the clock or serial output drivers.

If any of the driver's processes interrupts, add the interrupt service prologue. RTXC distributions usually include an assembly language file that contains the ISR prologue for all devices. Any additional device drivers may require modification of that file. When calling the device servicing routine, there is one note of caution: **Be sure to observe your compiler's rules concerning passing data from an assembly language function to a C function.** You should see examples of argument

passing in the existing code, but you should be aware of why the code is written in that specific manner.

Any driver responding to interrupts will need a C language device servicing function. It will be called from the device's interrupt service prologue. Write it in C and make it a part of the file containing the driver's task level code.

Write any new driver as a task making note of its name and entry point as well as any other kernel objects used. You will need to define all of them during system generation.

Application Tasks

The application tasks are the real meat of your design. How do you determine what an application task should do? How many tasks are needed? How much of the system's function does an application task perform? The list could go on - there are no definite answers to these questions. The design and implementation of embedded real-time systems do not easily lend themselves to formalisms and cookbook recipes.

But instead of leaving it at that point, let's examine some concepts that have been used successfully. Perhaps, somewhere in there, is the piece of information you need.

Decomposition

This is not so much a concept as it is an organizational necessity. It refers to the breaking

down, the decomposition, of the application in its totality into component parts or functions. Those parts become the application tasks, the database, the input and output definitions. In effect, it becomes a high level functional specification of your design.

You should not be too concerned about the language used for implementation or whether or not a component such as a real-time kernel will be needed. If you do your application decomposition correctly, those requirements become self-evident. Hopefully, by the time you are reading this, you have already completed this job and have your system functionally defined.

Information Hiding

This is one of the most powerful concepts you can use for multitasking systems. You may know it by some other name, such as encapsulation, but the concept remains the same. In a multitasking system, each task has a responsibility to perform a certain repertoire of functions. But task A, in order to execute properly, does not need to know the methods used by task B when it executes. Task A needs to know only that task B produces a certain output or receives a certain input. Put simply, task B should be a "black box" to task A.

This is classic von Neumann architecture applied to software. Each task is a black box and produces certain output for certain input. Connect the outputs and inputs of the various black boxes, and you have a good diagram of system data flow. Data Flow Diagrams, a popular design tool today, are rooted in this concept.

The important thing to remember about information hiding is that it is the interface between the tasks and the process that determines how well your design ultimately works. The interface defines the information content and flow between the physical process and the tasks, as well as that between tasks. Usually, the information content and flow of an interface specification is a natural description. You don't need to get down to the bits and bytes of the interface until you start your actual design implementation.

What makes this concept so important is that it obscures the internal structure of each task from the rest of the system. And that is a good thing. A method within a task can change because you have determined a better way of doing something than you had originally programmed. You make the changes for the new method, and your system still runs because you did not change the interface for the task you modified. As you can see, this concept has tremendous ramifications in the areas of testing, maintenance, and product longevity.

Functionality

Once again, there are no rules in determining just how much a task should do. A good decomposition will show what the functions of each module need to be. However, there is no rule that says that you can only have one task per function module in the decomposed application. It is quite permissible, and often desirable, if not necessary, to use multiple tasks to accomplish a module's functions.

It is usually a reasonable design rule to limit the functions of a single task to a manageable set of operations. You will likely see from your decomposition that each task has a central, or primary, function. This might be a single but complex function or several simple, related, but mutually exclusive functions. You will have to be the judge as to what each task does. Just remember, you, or someone else, must maintain the beast after you get it working; so why complicate the job by making the tasks too complex.

Another trap to avoid is to make up a set of tasks which run in succession. For example, task A runs and then task B, followed by task C, and so forth. If none of the other two runs while the third is running, all three are part of the same function and could be justifiably combined into one task. If you create such a sequential operation, you defeat the power of multitasking and you can impact the responsiveness of the system.

Timeliness

Timeliness can be thought of as response time to an event. When taken collectively, the timeliness of all event handling determines the overall system performance. It is at best a difficult concept to describe. However, in your design, you will become aware that certain things must be done either at a certain time or within a certain time.

Just remember that multitasking allows you to have two or more tasks in some stage of operation concurrently. Each task runs during the time that other tasks are waiting for events to occur. This is

the foundation of multitasking, and you should use it to achieve the timeliness requirements of your application.

Priority

Assuming you are implementing your application on a single processor, you cannot escape a simple fact: all tasks have to share the use of the CPU. How do you share the CPU in such a way that it is possible to meet the system's timeliness requirements? Timeliness implies that some tasks receive control of the CPU more than others, or they complete operation within a specified period of time following an event, or they perform their function without preemption by other tasks. All of these are solved by using priorities for each task.

A task's priority is an indication of how you perceive its importance in the system relative to other tasks. You should look at each task and determine what its priority should be with respect to other tasks and to the system as a whole. If it must be executed within a very brief time following some event, you may want to give it a high priority. If it is a periodic task that runs at a low frequency, a low priority will probably suffice. You may even determine that there needs to be one or more tasks at the same priority.

Task Synchronization

RTXC make extensive use of semaphores for purposes of synchronizing tasks to both internal and external events. But there are other ways as well to synchronize without using events. You may pass information between tasks using queues and messages, and RTXC performs synchronization automatically with the flow of data. This is a

powerful feature because the system does it transparently, making it a part of the multitasking strategy.

Intertask Communication

Use FIFO Queues and Messages to move data between tasks. There are no particular rules as to when to use one and not the other. However, some guidelines may serve that purpose. FIFO Queues in RTXC are circular buffers, and as such, have defined lengths. Each entry into a queue requires that the data be moved from its source to the next available slot in the queue. If the number of bytes in the entry, its width, is large RTXC will spend a lot of time moving those bytes. That may not be desirable.

Queues are best employed to move entries having a width of a few bytes while maintaining chronological order. The maximum practical width of an entry is something that you have to determine with respect to your system's timing requirements. Keep in mind that RTXC will move the data into or out of a queue regardless of the width. However, the length of time it takes to perform the move may have an adverse effect on overall system performance.

If you have long buffers of data to move from one task to another, a better method is to employ mailboxes and messages. With messages, the data is not moved. Rather, only a pointer to it is sent to the mailbox. This results in a faster way to move large volumes of data around the system. The receiving task gets a pointer to the message and operates on the message via that pointer and some structural template of the message body.

RTXC Configuration Options

Being able to control the configuration of your real-time kernel is an important aspect of embedded systems development. Because memory is often at a premium, having present only those elements of the kernel that are needed by the application is one way to manage memory usage. It is possible to make RTXC fit practically any application's needs by the inclusion or exclusion of various kernel objects and services.

The RTXC distribution includes a file, **RTXCOPTS.H**, in the *KERNEL* directory for that purpose. This is a file that you must edit in order to make the necessary selections, However, we have tried to make it as easy as possible to use.

As you have seen, RTXC makes use of a set of kernel objects, and the kernel services are directly related to the presence of those objects. You must have Task and Semaphore objects in order for RTXC to operate at any level. Other than those two, however, you have great flexibility. For instance, if you are not going to use FIFO Queues, do not take up memory space with that code. You can eliminate all of the kernel services related to FIFO Queues by simply excluding the Queue kernel objects from the configuration. Not only does such an exclusion free the program space taken by the excluded kernel services, it also causes any related RAM to be freed as well.

It is just as likely that you may include a certain class of kernel object but not use certain related kernel services. You can exclude them as well by editing **RTXCOPTS.H**.

Editing RTXCOPTS.H

In order to edit **RTXCOPTS.H**, you first need to make some choices about what you want to include and exclude. Having made those selections, you can proceed with editing **RTXCOPTS.H** to produce your specific RTXC configuration.

There are several selections that you must make on which many subsequent choices depend. You must define the RTXC library configuration you want to use and that, in turn, defines the set of kernel services with which you have to work. For instance, let us assume your RTXC license is for an Extended Library. You may choose to use only the services from the Advanced Library for a given application. You may do this by defining RTXC_AL as the library configuration. The kernel services unique to the Extended Library would not be included in the RTXC configuration.

Likewise, if you have licensed the RTXC Advanced Library, your choices are limited to the set of services in Advanced Library and the Basic Library. Remember that the library configuration you choose is determined by the kernel services you need. For example, if you use just one kernel service from the Extended Library, you must define the RTXC model as RTXC_EL.

In **RTXCOPTS.H**, there are three possible definitions under the heading of RTXC Library Configuration. They are:

```
#define (or #undef) RTXC_EL  /* Extended Library */
#define (or #undef) RTXC_AL  /* Advanced Library */
#define (or #undef) RTXC_BL     /* Basic Library */
```

Only one of those three can be defined. The other two must be undefined.

If you have an Extended Library license and you define RTXC_AL for the Advanced Library, all kernel services in the Extended Library not in the Advanced Library will be excluded when RTXC is built.

There are several other options that you can control with **RTXCOPTS.H** by defining or undefining them. Some of them are specific to the processor or compiler and are beyond the scope of this document. But all of them are well described in the commentary associated with each option. You should read the text accompanying the file and determine your response to each choice.

Each class of kernel object, except for Tasks, Semaphores, and Timers may also be selected for inclusion or exclusion. Thus, it is possible to make global choices regarding MAILBOXES, PARTITIONS, QUEUES, and RESOURCES. Like the other options, you must define or undefine each of these kernel objects in order to include or exclude them respectively from the RTXC configuration. The choices appear as follows:


```
#define HAS_MAILBOXES
#define HAS_PARTITIONS
#define HAS_QUEUES
#define HAS_RESOURCES
```

To include a given class of kernel object, leave it selected with the *#define*. To exclude an object, change the *#define* to *#undef*.

You can further refine your RTXC configuration by eliminating unwanted kernel services associated with included kernel objects. Each kernel service that can be so excluded has been entered into **RTXCOPTS.H**. The options are organized according to kernel object inclusions. For example, if FIFO queues have been included, kernel services dealing with Queues may be selectively included or excluded. The file will appear something like the following excerpt:


```
/***********************************************/
/*    Define the Queue Services to be included   */
/***********************************************/

#ifdef HAS_QUEUES /* { */

#define HAS_DEQUEUE    /* REQUIRED */
#define HAS_ENQUEUE    /* REQUIRED */

#ifdef RTXC_AL /* { RTXC_AL */
#define HAS_DEQUEUEW   /* use #define or #undef */
#define HAS_ENQUEUEW   /* use #define or #undef */
#define HAS_PURGEQUEUE /* use #define or #undef */
#define HAS_DEFQSEMA   /* use #define or #undef */
#endif /* } RTXC_AL */

#ifdef RTXC_EL /* { */
#define HAS_DEQUEUET   /* use #define or #undef */
#define HAS_ENQUEUET   /* use #define or #undef */
#define HAS_INQQUEUE   /* use #define or #undef */
#define HAS_DEFQUEUE   /* use #define or #undef */
#endif /* } RTXC_EL */
#endif /* } HAS_QUEUES */
```

Finally, there is one warning associated with editing **RTXCOPTS.H**. At the end of the file are some sanity checks intended to catch gross errors you may have committed during your editing. It is probably not possible to catch every possibility that can cause you trouble but we have attempted to catch most problems. So, the warning is simple: **Be careful in your editing and don't change the sanity checks unless you absolutely have to.**

System Generation Using RTXCgen

After you have RTXC configured to suit your application, you will have to define your application's elements. Principally, the definitions are the kernel objects of each class that you will be using. RTXCgen is the system generation utility that you use to do this. RTXCgen is described in Section 6 of this manual and doesn't require much more explanation. It is an interactive program and is ready to run on a PC compatible host.

If you edited **RTXCOPTS.H** and changed the state of one or more of the switches, **FPU**, **DYNAMIC_TASKS**, **DYNAMIC_PARTITIONS**, or **SEMA_USE_TABLE**, you will need to compile and link the RTXCgen files to produce a new RTXCGEN.EXE.

Putting It All Together

By now you have seen the things you need to do in order to build your own application. What has gone unmentioned until now is the process of compiling and linking the various parts into an executable whole.

You compiled RTXC and built the library, **LIBRTXC**, by using the Make file provided in the RTXC distribution. You have used RTXCgen and have produced C source code for the various kernel objects you are using in your application. Lastly, you have written your application tasks and support routines. You will need to compile (and/or assemble) all of the source code modules in your application and the kernel object definitions. Hopefully, you have modified the distributed Make file in the *DEMO* directory to meet your application needs.

You will also have to specify the input information for the linker so that the various object code modules can be linked into the proper areas of memory. If you have done all of that, you should be in good shape to build your application executable file.

The figure on the following page gives a graphical presentation of what we have been discussing in this last subsection, proving once again that a picture is worth a thousand words.

Figure 4-1

SECTION 5
RTXC Kernel Services
Table of Contents

SECTION 5

RTXC KERNEL SERVICES

INTRODUCTION

Kernel Services (KS) are the functions that a real time kernel performs and serve to give it its flavor. This section will describe the complete set of the RTXC directives in two manners.

The first is an enumeration of each Kernel Service according to the class to which it belongs. Included in each description is a generalized C language prototype of the Kernel Service function's calling sequence.

The second description of Kernel Service will be in alphabetical order and will include a complete explanation of each Kernel Service function and an example of its usage.

CLASSES

The Kernel Services of RTXC are divided into the seven basic classes of:

Task Management Kernel Services deal with starting, stopping, and otherwise maintaining information about task states.

ISR Services perform a limited number of special operations while CPU control is in an interrupt service routine.

Intertask Communication and Synchronization functions provide the services by which Tasks pass data to other tasks via messages and queues. This class also is responsible for the primary synchronization services of RTXC.

Timer Management services deal with the RTXC Timer facility so that tasks may perform their operations with respect to time as an event.

Memory Partition Management Kernel Services deal with the maintenance of the RTXC memory partitions to ensure orderly usage of the system's RAM.

Resource Management services provide an orderly means to gain and release exclusive control of an RTXC resource.

Special Kernel Services provide for user defined extensions to RTXC which can be application dependent.

PROTOTYPES

Each RTXC port includes a file, **RTXCAPI.H**, which defines an ANSI C prototype for each Kernel Service. Because RTXC is designed with portability in mind, the API defined by **RTXCAPI.H** is essentially identical for all ports of RTXC. However, there are differences between some of the processors on which RTXC operates which lead to variations in sizes of certain parameters used by the Kernel Services. Similarly, there may be syntactical differences between C compilers of different manufacture.

For example, a C compiler may use the key words *near* and *far* to permit different memory models due to the processor's architecture. Another C compiler targeted to a different processor may not make use of a memory model requiring *near* and *far*.

Another example might be the size of an integer on a 8-bit microcontroller versus that on a 32-bit high performance processor.

You should refer to the RTXC header files, in particular **RTXCARG.H** and **TYPEDEF.H**, for actual sizes of the data elements if you are uncertain about a particular size.

GENERAL FORM OF KERNEL SERVICE REQUEST

The general form of an RTXC Kernel Service function call is:

```
KS_name([arg1][,arg2]...[,argn])
```

Where the character string "KS_" identifies *name* as an RTXC Kernel Service. This prefix should prevent *name* from being misidentified by a linker with some similarly named function in the runtime library of the compiler.

ARGUMENTS TO KERNEL SERVICES

The RTXC Kernel Service descriptions to follow will show the function prototypes with generalized RTXC arguments. Similarly, values returned from Kernel Service functions are shown symbolically. The list below is a brief description of those symbols:

SYMBOL	DESCRIPTION
char	character
CLKBLK	Address of a timer (clock) block
FRAME	Address of the stack frame of an interrupted process
ENTRY	Entry address of a task
QCOND	Queue condition code
int	Integer, single precision
KSRC	kernel service return code
MBOX	A mailbox identifier
PRIORITY	The priority of a task or a message

RESATTR	The priority inversion attribute code of a resource
RTXCMSG	Address of an RTXC message envelope
SEMA	A semaphore identifier
size_t	ANSI C compiler defined
TASK	A task identifier (not the task's priority)
TICKS	Units of time maintained by RTXC system time base
time_t	ANSI C compiler defined structure
void	No value returned or no argument required
KSRC	Kernel Service Return Code

TASK MANAGEMENT SERVICES

The task management services provided by RTXC allow for complete control of tasks and their respective interactions.

KS_alloc_task(void)

Allocate a TCB from the Pool of Free TCBs

KS_block(TASK, TASK)

Block a Range of Tasks from Running

KS_defpriority(TASK, PRIORITY)

Define Task Priority

KS_defslice(TASK, TICKS)

Define Task's Time-Slice Time Quantum

KS_deftask(TASK, PRIORITY,
char*, size_t, void (*)(void))

Define the Attributes of a Task

KS_deftask_arg(TASK, void *)

Define the Environment Arguments for a Task

KS_delay(TASK, TICKS)

Delay a Task for a Period of Time

KS_execute(TASK)

Execute a Task

KS_inqpriority(TASK)

Inquire on a Task's Priority

KS_inqslice(TASK)

Get the Task's Time-Slice Quantum

KS_inqtask(void)

Get Task Number of Current Task

KS_inqtask_arg(TASK)

Get the Current Task's Environment Arguments

KS_resume(TASK)

Resume a Task

KS_suspend(TASK)

Suspend a Task

KS_terminate(TASK)

Terminate a Task

KS_unblock(TASK, TASK)

Unblock a Range of Tasks

KS_yield(void)

Yield to Next Runnable Task

ISR SERVICES

ISR services provide a means of performing certain operations while in an interrupt service routine. These functions include allocating a block from a Memory Partition, signaling a semaphore to announce the occurrence of an event, processing a clock tick, and terminating an ISR.

KS_ISRalloc(MAP)

Allocate a Block of Memory from the Given Memory Partition.

KS_ISRexit(FRAME *, SEMA)

Exit Current Interrupt Service Routine and Optionally Signal Given Semaphore

KS_ISRsignal(SEMA)

Signal Given Semaphore from an Interrupt Service Routine

KS_ISRtick(void)

Perform System Required Processing for a Clock Tick Interrupt

INTERTASK COMMUNICATION AND SYNCHRONIZATION SERVICES

There are three subclasses of Kernel Services within this class. The subclasses consist of those functions which deal with RTXC Semaphores, RTXC Messages, and RTXC Queues respectively.

SEMAPHORE BASED SERVICES

A complete set of directives for managing semaphores is provided by RTXC. The C calling sequences for each directive will be described in the section that follows. The definition of a semaphore specification and prototyped functions are noted in C idiom below.

KS_defmboxsema(MBOX, SEMA)

Define Mailbox Semaphore

KS_defqsema(QUEUE, SEMA, QCOND)

Define Queue Semaphore

KS_inqsema(SEMA)

Return Current State of Semaphore

KS_pend(SEMA)

Force a Semaphore to a Pending State

KS_pendm(SEMA *)

Force Multiple Semaphores to Pending State

KS_signal(SEMA)

Signal a Semaphore

KS_signalm(SEMA *)

Signal Multiple Semaphores

KS_wait(SEMA)

Wait on Event

KS_waitm(SEMA *)

Wait on Multiple Events

KS_waitt(SEMA, TICKS)

Time Limited Wait on Event

MESSAGE BASED SERVICES

The message directives provide a means of transferring large amounts of data between tasks with minimal overhead since only pointers (addresses) are passed. Message receipt acknowledgment is also provided for task synchronization. The format of a RTXC message and function prototypes are noted.

KS_ack(RTXCMSG *)

Acknowledge Message

KS_receive(MBOX, TASK)

Receive a Message

KS_receivet(MBOX, TASK, TICKS, KSRC *)

Receive a Message, Limit Duration of Wait if MAilbox Empty

KS_receivew(MBOX, TASK)

Receive a Message, Wait if Mailbox Empty

KS_send(MBOX, RTXCMSG *, PRIORITY, SEMA)

Send a Message Asynchronously

KS_sendt(MBOX, RTXCMSG *, PRIORITY, SEMA, TICKS)

Send a Message Synchronously and Time Limit Duration of Wait

KS_sendw(MBOX, RTXCMSG *, PRIORITY, SEMA)

Send a Message Synchronously

QUEUE BASED SERVICES

Queue directives provide a means of passing multiple byte packets of information between tasks with automatic task synchronization of queue full and empty conditions.

KS_defqueue(QUEUE, size_t, int, void *, int)

Define Queue Attributes

KS_dequeue(QUEUE, void *)

Get Entry from a FIFO Queue

KS_dequeuet(QUEUE, void *, TICKS)

Get Entry from a FIFO Queue, Time Limited Wait if Queue Empty

KS_dequeuew(QUEUE, void *)

Get Entry from a FIFO Queue, Wait if Empty

KS_enqueue(QUEUE, void *)

Put Entry into FIFO Queue

KS_enqueuet(QUEUE, void *, TICKS)

Put Entry into FIFO Queue, Time Limited Wait if Queue Full

KS_enqueuew(QUEUE, void *)

Put Entry into FIFO Queue, Wait if Queue Full

KS_inqqueue(QUEUE)

Inquire on Number of Entries in Queue

KS_purgequeue(QUEUE)

Reset Queue to Empty State

RESOURCE MANAGEMENT SERVICES

Resource directives provide a means of managing and protecting logical resources. Typical resources might include a shared database, non-reentrant code modules, specialized hardware, or an expensive laser printer.

KS_defres(RESOURCE, RESATTR)

Define Priority Inversion Attribute for a Resource

KS_inqres(RESOURCE)

Inquire on the Owner of a Resource

KS_lock(RESOURCE)

Request Exclusive Use of a Resource

KS_lockt(RESOURCE, TICKS)

Request Exclusive Use of a Resource, Time Limited Wait if Busy

KS_lockw(RESOURCE)

Request Exclusive Use of a Resource, Wait if Busy

KS_unlock(RESOURCE)

Release Logical Resource

TIMER MANAGEMENT SERVICES

The time based directives provide for the synchronization of tasks with timed events. In addition, a generalized time based semaphore scheme for more advanced time based requirements is provided.

KS_alloc_timer(void)

Allocate a Timer

KS_elapse(TICKS *)

Compute Elapsed Time

KS_free_timer(CLKBLK *)

Free a Timer Block

KS_inqtimer(CLKBLK *)

Get Time Remaining on a Specified Timer

KS_restart_timer(CLKBLK *, TICKS, TICKS)

Restart an Active Timer

KS_start_timer(CLKBLK *, TICKS, TICKS, SEMA)

Start a Timer

KS_stop_timer(CLKBLK *)

Stop an Active Timer

MEMORY PARTITION MANAGEMENT SERVICES

The memory management directives provide a system-wide means of dynamically allocating and deallocating memory blocks to tasks on an as needed basis. Multiple tasks can thus share a common pool of memory. The basic unit of memory managed by these directives is noted below in C idiom.

KS_alloc(MAP)

Allocate a Block of Memory

KS_alloc_part(void)

Allocate a Memory Partition Header

KS_alloct(MAP, TICKS, KSRC *)

Allocate a Block of Memory with Time Limited Wait

KS_allocw(MAP)

Allocate a Block of Memory with Wait

KS_create_part(void *, size_t, size_t)

Create a Memory Partition with Given Attributes

KS_defpart(MAP, void *, size_t, size_t)

Define Attributes of a Memory Partition

KS_free(MAP, void *)

Free a Block of Memory

KS_free_part(MAP)

Free a Memory Partition Header

KS_inqmap(MAP)

Returns Size of Block in a Partition

SPECIAL SERVICES This is a class of directives which are included for special purposes.

KS_deftime(time_t)

Define Current Date/Time

KS_inqtime(void)

Get Current Date/Time

KS_nop(void)

No Operation

KS_user(int (*) (void *), void *)

User Defined Kernel Service

ALPHABETICAL LISTING OF KERNEL SERVICES

In the pages to follow, each KS will be shown in alphabetical order. Each KS will be described in a standard format:

Name

BRIEF FUNCTIONAL DESCRIPTION

CLASS	One of the 7 KS classes of which it is a member.
SYNOPSIS	The formal C declaration including argument prototyping.
DESCRIPTION	A description of what the KS does, data types used, etc.
RETURN VALUE	A description of the return values from the KS.
EXAMPLE	One or more typical KS uses. The examples assume the syntax of ANSI Standard C.
SEE ALSO	List of related Kernel Services that could be examined in conjunction with the current KS.
SPECIAL NOTES	Assorted notes and technical comments.

KS_ack

ACKNOWLEDGE MESSAGE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_ack(RTXCMSG *message)
```

DESCRIPTION When a task receives a message and finishes processing the message, it is good practice to let the task which sent the message know that it has been processed. The message acknowledge function is intended to perform that service. The receiving task has the address of the message envelope which was returned by a prior KS_receive, KS_receivet, or KS_receivew function call. The KS_ack function performs the signaling of the message semaphore specified by the sending task.

RETURN VALUE The function returns no value.

EXAMPLE Receive a message and save the pointer to the message envelope in pointer *p*. When finished processing the message body, inform the sending task of the event.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"              /* defines EMAIL */
#include "rtxstruc.h"         /* defines RTXCMSG */

struct{
   RTXCMSG msghdr;  /* Message header (required) */
   char data[10];       /* start of message body */
} MYMSG;

MYMSG *p;


/* get next message from mailbox EMAIL */
p = (MYMSG *)KS_receivew(EMAIL,(TASK)0);

... Perform message processing

KS_ack(p);     /* signal message processing done */
```

SEE ALSO KS_receive, KS_receivet, KS_receivew, KS_send, KS_sendt, KS_sendw

KS_alloc

ALLOCATE A BLOCK OF MEMORY

CLASS Memory Partition Management

SYNOPSIS

```
void  *KS_alloc(MAP map)
```

DESCRIPTION The KS_alloc Kernel Service function locates the next free block in the given RTXC Memory Partition specified by *map* and returns its address to the calling task as the value of the function. If no block is available in the specified partition, a value of NULL is returned.

RETURN VALUE The function returns a pointer to the memory block if successful. If there are no available blocks in the given partition, the map is said to be empty and a NULL pointer (void *(0)) is returned.

EXAMPLE In this example, a block of memory from one of the RTXC memory partitions, *MAP1*, is needed. If the allocation is successful, the pointer to the block is to be stored in a character pointer *p*. If there are no free blocks in the partition, the task is to output an appropriate message.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"         /* defines MAP1 */

char *p;


if ( (p = (char *)KS_alloc(MAP1)) == NULL)
{
   ... Deal with no memory available
}
else
{
   ... Allocation was successful
}
```

SEE ALSO KS_alloct, KS_allocw, KS_free, KS_inqmap

KS_alloc_part

ALLOCATE A MEMORY PARTITION HEADER

CLASS Memory Partition Management

SYNOPSIS

```
MAP  KS_alloc_part(void)
```

DESCRIPTION The KS_alloc_part Kernel Service function locates the next free Memory Partition header in the list of dynamic Memory Partitions and returns its *map* identifier to the calling task as the value of the function. No definition of the Map's attributes is done by this Kernel Service.

RETURN VALUE The function returns a Map identifier of a dynamic Memory Partition if successful. If no dynamic Memory Partition header is available, function value of zero (0) is returned.

EXAMPLE In this example, a task allocates a Memory Partition dynamically and then defines its attributes using some data values acquired during its operation. If the allocation is successful, the Map's identifier is to be stored in a variable, *map1*, of type MAP. If there are no free dynamic Memory Partitions available, the task is to output an appropriate message.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */

MAP map1;


if ( (map1 = KS_alloc_part()) == (MAP)0)
{
   ... Deal with no dynamic Maps available
}
else
{
   ... Allocation was successful
       Now define the Map's attributes
}
```

SEE ALSO KS_create_part, KS_defpart, KS_free_part

KS_alloc_task

ALLOCATE A TASK CONTROL BLOCK

CLASS Task Management

SYNOPSIS

```
TASK  KS_alloc_task(void)
```

DESCRIPTION The KS_alloc_task kernel service allocates the next available Task Control Block from the pool of free TCBs. The allocated TCB will be used in a dynamic task allocation and will be followed at some point by a request to define the allocated task's attributes prior to its execution.

RETURN VALUE The function returns the value of the identifier of the allocated Task Control Block if the allocation is successful.

If there are no available Task Control Blocks, the function returns a value of zero (0).

EXAMPLE In this example, the current task determines from the state of the system that it needs to spawn another task. It first allocates a TCB for the task to be spawned, then it defines the task's attributes. Optionally, it defines the new task's environment arguments, and finally, executes the task. If there are no available TCBs, the requesting task must handle the condition with special program logic.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */

extern void taskA(void);
struct newenvrg   /* taskA environment arguments */
{
   char arg1;
   int  arg2;
.   ..etc
}
TASK newtaskA;
PRIORITY newpri;
char *pstk;
int stksz;

if ((newtaskA = KS_alloc_task()) == (TASK)0)
{
   ... Deal with no TCBs available
}
else        /* TCB allocated. Okay to use it */
{
 - determine size of stack to allocate (stksz)
 - allocate space for task's stack (pstk)
 - assign a priority of the new task (newpri)
 - then define the task attributes as follows:
   KS_deftask(newtaskA, newpri, pstk, stksz, taskA);

 - optionally define any environment arguments for
   the task as follows:
   KS_deftask_arg(newtask,&newenvrg);

 - once that is all done, start the task executing:
   KS_execute(newtaskA);
}
```

SEE ALSO KS_deftask_arg, KS_execute, KS_terminate

KS_alloc_timer

ALLOCATE A TIMER

CLASS Timer Management

SYNOPSIS

```
CLKBLK  *KS_alloc_timer(void)
```

DESCRIPTION The KS_alloc_timer kernel service function allocates the next available timer from the pool of free timers and returns its address to the calling task. If no timer is available, a value of NULL (0) is returned. The address, or handle, of the timer will be used in subsequent RTXC kernel services when dealing with timer functions. A task may allocate more than one timer before one is deallocated.

RETURN VALUE The function returns a pointer to the timer block if successful.

If there are no available timers, a NULL pointer (void *(0)) is returned.

EXAMPLE In this example, a timer block is allocated and then a cyclic timer is started using the allocated timer. If the allocation is successful, the pointer to the timer block is returned and stored in a pointer *p*. If there are no free timer blocks, the task must handle the condition with special program logic.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cclock.h"        /* defines CLKTICK */
#include "csema.h"         /* defines AISEMA */

CLKBLK *p;


p = KS_alloc_timer();
if (p == (CLKBLK *)0)
{
   ... Deal with no timers available
}
else
   KS_start_timer(p, 250/CLKTICK, 1000/CLKTICK, AISEMA);
```

SEE ALSO KS_free_timer, KS_restart_timer, KS_start_timer, KS_stop_timer

KS_alloct

ALLOCATE A BLOCK OF MEMORY,
WAIT FOR LIMITED TIME
IF MEMORY UNAVAILABLE

CLASS Memory Partition Management

SYNOPSIS

```
void  *KS_alloct(MAP map,
                 TICKS ticks,
                 KSRC *ret_code)
```

DESCRIPTION

The memory allocation function allocates the next available block of memory from the specified partition and returns its address. If there is a block available in the specified memory partition, the function returns its address immediately to the requesting task. In addition, a value of **RC_GOOD** will be stored at the address indicated by the pointer to *ret_code*.

If there is no available block in the memory partition, the requesting task is blocked, removed from the READY List, and put into a WAIT state until memory in the requested partition becomes available. At the same time, a timeout timer is started to limit the duration of the task's wait to the period defined by *ticks* in the calling arguments to KS_alloct.

Either the timeout timer expiring or a block becoming available in the partition will cause the waiting task to be resumed. The latter cause returns the address of the allocated memory block. If, however, the timeout occurs and causes the task to be resumed, a NULL pointer (0) will be returned as the function value to indicate there was no block available within the specified timeout period. The function will store a value of **RC_TIMEOUT** at the *ret_code* parameter.

If there are multiple tasks waiting for memory from the same partition, the highest priority waiting task will get the first available block.

RETURN VALUE

The function returns a pointer to the memory block.

If the timeout occurs before there is memory to allocate, the function returns a NULL pointer (void *(0)) and **RC_TIMEOUT** via *ret_code*.

EXAMPLE

Allocate a block of memory from *MAP1* to be used for a character buffer. Store the address of the string in the character pointer *p*. If there is no memory available at the time of the request, wait for a period of 500 msec for a block to become available before proceeding. If there is no memory available and the timed wait expires, handle the situation with a special segment.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cclock.h"        /* defines CLKTICK */
#include "cpart.h"         /* defines MAP1 */


char *p;
KSRC code;


/* no return until requested memory is available */
/* or until timeout occurs. */
p = (char *)KS_alloct(MAP1, 500/CLKTICK, &code);
if (p == ((char *)0))
{
         ... Handle no memory availability here
}
else
{
   ... Memory allocated. Proceed.
}
```

SEE ALSO KS_alloc, KS_allocw, KS_free, KS_inqmap

This page is intentionally left blank.

KS_allocw

ALLOCATE A BLOCK OF MEMORY,
WAIT IF NONE AVAILABLE

CLASS Memory Partition Management

SYNOPSIS

```
void  *KS_allocw(MAP map)
```

DESCRIPTION The allocate memory with wait service function allocates the next available block of memory from the specified partition and returns its address. If there is no available memory, the requesting task is removed from the READY List, blocked, and put into a WAIT state until memory in the requested partition becomes available.

If there are multiple tasks waiting for memory from the same partition, the highest priority waiting task will get the first available block.

RETURN VALUE The function returns a pointer to the memory block.

EXAMPLE Allocate a block of memory from *MAP1* to be used for a character buffer. Store the address of the string in the character pointer *p*. If there is no memory available at the time of the request, wait for it to become available before proceeding.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"         /* defines MAP1 */

char *p;


p = (char *)KS_allocw(MAP1); /* no return until */
                             /* requested memory */
                             /* is available */
```

SEE ALSO KS_alloc, KS_alloct, KS_free, KS_inqmap

KS_block

BLOCK A RANGE OF TASKS

CLASS Task Management

SYNOPSIS

```
void  KS_block(TASK start,
               TASK end)
```

DESCRIPTION

The KS_block function provides a means of selectively blocking one or more tasks from running. This function implements another means to block a task in a manner similar to KS_suspend. The primary purpose of KS_block is to provide RTXCbug with a single call which blocks all other tasks. This function should be used with caution and critical tasks should never be blocked.

A runnable task to be blocked will be removed from the READY List. A task which is not currently runnable will be blocked again by this service. Once a task is blocked by this service, it will become runnable again only by invocation of the KS_unblock or KS_execute kernel services.

The range of specified tasks to be blocked may include the current task but RTXC guarantees the current task will not be blocked. A starting task number of 0 will block those tasks having a higher task number than the current task up to and in-

cluding the specified end task. An end task specification of 0 will block all tasks beginning with the start task up to, but not including, the current task. It is not legal to specify start task and end task as both having a value of 0.

RETURN VALUE The function returns no value.

EXAMPLE 1. Block tasks 5 through 8 inclusive.

```
#include "rtxcapi.h"      /* RTXC KS prototypes */

KS_block(5,8);          /* block 4 tasks, 5 -> 8 */
```

2. Block from task 5 up to but not including the current task.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#define SELFTASK (TASK(0))

KS_block(5,SELFTASK); /* block tasks 5 -> self-1 */
```

SEE ALSO KS_unblock

KS_create_part

CREATE A DYNAMIC MEMORY PARTITION

CLASS Memory Partition Management

SYNOPSIS

```
MAP  KS_create_part(void *body,
                    size_t blksize,
                    size_t n_blks)
```

DESCRIPTION

The *KS_create_part()* function provides a means of combining the two Basic Library Kernel Services, *KS_alloc_part()* and *KS_defpart()* into a single function. The function requires three arguments specifying the address of the RAM area to be used as the *body* of the Memory Partition (i.e. the blocks), the size of the blocks in the Map, *blksize*, and the number of blocks, *n_blks*.

If the Kernel Service finds an available dynamic Memory Partition header, it will use the function arguments to define the Map's attributes and then link all of the blocks in the Map.

The value of the block size argument, *blksize*, must be at least the size of a data pointer.

RETURN VALUE

If the function is successful, it will return the identifier of the allocated dynamic Memory Partition.

If the Kernel Service is unsuccessful, it returns a value of zero (0).

EXAMPLE

A task creates a dynamic Memory Partition having a block size of 18 bytes and 24 blocks. The body of the partition is a block of RAM allocated from another Memory Partition whose block size is 512 bytes. If successful, the Map's identifier will be stored in the variable of type MAP, *map1*.

```
#include "rtxcapi.h"      /* RTXC KS prototypes */
#include "cpart.h"            /* defines MAP512 */

MAP map1;
char *body;
size_t blksize, n_blks;

   ... blksize and n_blks defined by some means

if ( (body = (char *)KS_alloc(MAP512)) == NULL )
{
   ... Deal with no block available for dynamic
       Map's body. Maybe try another Map?
}

if ( (map1 = KS_create_part(body, blksize,n_blks) == (MAP)0 )
{
   /* the attempt to create a dynamic Map failed */
   /* free the unused RAM block */
   KS_free(MAP512, body);

   ... Then deal with the failure of the dynamic
       Memory Partition creation
}
else
{
   ... Creation was successful
}
```

SEE ALSO KS_alloc_part, KS_defpart, KS_free_part

KS_defmboxsema

DEFINE MAILBOX SEMAPHORE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_defmboxsema(MBOX mailbox,
                     SEMA sema)
```

DESCRIPTION The KS_defmboxsema permits the association of the Not_Empty condition of a mailbox with a semaphore. The association permits a task to use the KS_waitm Kernel Service to wait for the occurrence of that condition or other events with a single request.

RETURN VALUE The function returns no value.

EXAMPLE The current task is servicing two mailboxes, *HPMAIL* and *LPMAIL*. It needs to synchronize with the next message being sent to either mailbox, both of which are currently empty. It uses the KS_waitm Kernel Service to wait for mail to be sent to either mailbox. When the task continues upon detecting the presence of mail, it identifies the mailbox having the mail, receives it, and processes it. Upon completion of its processing, the task signals the message sender that processing is finished.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"  /* defines HPMAIL and LPMAIL */
#include "csema.h"    /* defines GOTHP and GOTLP */

struct{
   RTXCMSG msghdr;  /* Message header (required) */
   char data[10];       /* start of message body */
} MYMSG;

MYMSG *msg;
SEMA sema;
SEMA semalist[] =
{
    GOTHP, GOTLP, 0
};

KS_defmboxsema(HPMAIL,GOTHP);/* define semas for */
KS_defmboxsema(LPMAIL,GOTLP);  /* both mailboxes */
sema = KS_waitm(&semalist); /* wait for mail */
switch (sema)
{
   case GOTHP:
      /* receive message in HPMAIL from any task */
      msg = (MYMSG *)KS_receive(HPMAIL,(TASK)0);
      ... process received message
      break;

   case GOTLP:
      /* receive message in LPMAIL from any task */
      msg = (MYMSG *)KS_receive(LPMAIL,(TASK)0);
      ... process received message
      break;
}
/* acknowledge message receipt and processing */
KS_ack(msg);
```

SEE ALSO KS_receive, KS_receivet, KS_receivew, KS_send, KS_sendt, KS_sendw

KS_defpart

DEFINE MEMORY PARTITION ATTRIBUTES

CLASS Memory Partition Management

SYNOPSIS

```
void  KS_defpart(MAP map,
                 void *body,
                 size_t blksize,
                 size_t n_blks)
```

DESCRIPTION The *KS_defpart()* function provides the means to define the attributes of a new Memory Partition or to redefine those of an existing Map. The function requires four arguments including the Map identifier, *map*, the address of the RAM area to be used as the *body* of the Memory Partition (i.e. the blocks), the size of the blocks in the Map, *blksize*, and the number of blocks, *n_blks*.

Upon defining the Map's attributes, the function will link all of the blocks in the Map.

The value of the block size argument, *blksize*, must be at least the size of a data pointer.

RETURN VALUE The function returns no value.

EXAMPLE A task allocates a dynamic Memory Partition header and, if successful, stores the Map's identifier in the

variable of type MAP, *map1*. It then allocates the body of the partition from another Memory Partition whose block size is 512 bytes. Having all the necessary objects, the task uses *KS_defpart()* to define the Map's attributes.

```
#include "rtxcapi.h"      /* RTXC KS prototypes */
#include "cpart.h"           /* defines MAP512 */

MAP map1;
char *body;
size_t blksize, n_blks;

   ... blksize and n_blks defined by some means

if ( (map1 = KS_alloc_part()) == (MAP)0 )
{
   ... Deal with no dynamic Map available
}
else
{
   if ( (body = (char *)KS_alloc(MAP512)) == NULL )
   {
      ... Deal with no block available for dynamic
          Map's body. Maybe try another Map?
   }
   else
      KS_defpart(map1, body, blksize, n_blks);
}
```

SEE ALSO KS_alloc_part, KS_defpart, KS_free_part

KS_defpriority

DEFINE TASK PRIORITY

CLASS Task Management

SYNOPSIS

```
void  KS_defpriority(TASK task,
                     PRIORITY priority)
```

DESCRIPTION

This function permits a task to define (or change) the priority of itself or another task. The definition may be any legal priority be it higher or lower than the task's current priority.

For the current task, a change to a higher priority will not cause a context switch. If the change is to a lower priority, the current task may be preempted if another task in the READY List has a higher priority.

The current task may specify itself by the value of zero (0) in the task argument field in the calling sequence.

If the task whose priority is being changed is not the current task, a preemption will occur if the new priority of the object task becomes higher than the requesting task.

The priority of a task may be changed before it is referenced in a KS_execute request. This may be used to override the default priority setting which is set equal to the task number during system initialization and during the KS_terminate function.

RETURN VALUE The function returns no value.

EXAMPLE Change the priority of task *SERIALIN* from its current level to priority 3. Then change calling task to priority 6.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"         /* defines SERIALIN */


KS_defpriority(SERIALIN, 3); /* new priority = 3 */

KS_defpriority(SELFTASK, 6); /* new priority = 6 */
```

SEE ALSO KS_execute, KS_terminate

KS_defqsema

DEFINE QUEUE SEMAPHORE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_defqsema(QUEUE queue,
                  SEMA sema,
                  QCOND condition)
```

DESCRIPTION

The KS_defqsema service provides the ability to assign a semaphore to one of four conditions associated with a FIFO queue. The possible conditions are:

Queue_not_Empty,
Queue_not_Full,
Queue_Empty, and
Queue_Full.

These conditions have enumerated values of **QNE**, **QNF**, **QE**, and **QF** respectively. The specification in the calling arguments for the queue event, *condition*, should be given as one of these four values.

Defining a queue semaphore establishes a relationship with a queue condition. This association permits a task to wait on a condition of the queue to occur. This ability is most useful when a task needs to synchronize with a given condition. When several

queues are being used, a KS_waitm kernel service can be used to synchronize with any of the events associated with the specified queue conditions.

RETURN VALUE The function returns no value.

EXAMPLE A task needs to associate the *Queue_not_Empty* condition on queue *DATAQ* with semaphore *GOT1* so that it can synchronize with the event.

```
#include "rtxcapi.h"      /* RTXC KS prototypes */
#include "cqueue.h"       /* defines DATAQ */
#include "csema.h"        /* defines GOT1 */
struct entry
{
   int count;
   int values[8];
};


KS_defqsema(DATAQ,GOT1,QNE);


KS_wait(GOT1);
KS_dequeue(DATAQ, &entry)
```

SEE ALSO KS_defqueue, KS_dequeue, KS_dequeuet, KS_dequeuew, KS_enqueue, KS_enqueuet, KS_enqueuew, KS_inqqueue, KS_purgequeue

KS_defqueue

DEFINE QUEUE ATTRIBUTES

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_defqueue(QUEUE queue,
                  size_t width,
                  int depth,
                  void *body,
                  int currsize)
```

DESCRIPTION

The KS_defqueue service provides dynamic definition of a queue's attributes including *width* (entry size), *depth* (number of entries), address of queue *body* (array of entries), and the number of entries in the queue. This function does not create a new queue but rather modifies those queue attributes specified at system generation time.

The queue may be defined as containing the number of entries given by the value of *currsize* which may be zero, for an empty queue, or any number less than or equal to its defined *depth*. If *currsize* is equal to *depth*, the queue is full.

Once defined, the queue may be used in any RTXC queueing operation. KS_defqueue is intended to allow flexible queue sizing in environments where RAM memory is precious and buffering

requirements are dynamic and/or unknown until system operation is underway.

RETURN VALUE

The function returns two possible KSRC values.

If the function is performed successfully, a KSRC value of RC_GOOD is returned.

If the value of *currsize* exceeds the value of *depth*, the function will return RC_ILLEGAL_QUEUE_SIZE.

EXAMPLE

The current task must allocate a block of RAM from a Memory Partition containing a block size of at least 80 bytes and define new attributes for queue *DATAQ*. The queue will be defined as EMPTY.

The width of the entries is to be the size of the structure *entry* and the depth is to be the value previously defined as *NUM*. The body of the queue will be the allocated block of RAM whose address will be held in the pointer *pbody*.

```
#include "rtxcapi.h"      /* RTXC KS prototypes */
#include "cqueue.h"       /* defines DATAQ */
#include "cpart.h"        /* defines MAP128 */

#define NUM 10

void *pbody;
struct entry
{
   int count;
   int values[8];
};

/* allocate RAM for queue body */
pbody = KS_allocw(BUFFPART);

KS_defqueue(DATAQ,sizeof(struct entry),NUM,pbody,0);
```

SEE ALSO

KS_dequeue, KS_dequeuet, KS_dequeuew, KS_enqueue, KS_enqueuet, KS_enqueuew, KS_inqqueue, KS_purgequeue

SPECIAL NOTE

Any task(s) waiting on queue availability conditions (full, empty, not full, or not empty) at the time of the KS_defqueue may be left in an indeterminate state.

To minimize RAM usage, a queue that is to be redefined at runtime, should be defined as having a width of 1 byte and a depth of 1 entry during system generation. During the redefinition process, memory occupied by the original queue body is not reclaimed.

This page is intentionally left blank.

KS_defres

DEFINE PRIORITY INVERSION ATTRIBUTE FOR A RESOURCE

CLASS Resource Management

SYNOPSIS

```
KSRC  KS_defres(RESOURCE resource,
                RESATTR condition)
```

DESCRIPTION

The KS_defres kernel service defines the priority inversion attribute of the specified resource. This attribute determines if an attempt to lock a resource can result in a priority inversion and if RTXC is to handle the inversion. When enabled by ON as the condition, the attribute will cause RTXC to check for a priority inversion if an attempt to lock the resource fails. When the attribute is disabled, no such checking occurs. The default condition of the attribute is OFF.

The function requires a resource identifier and the condition of the priority inversion processing attribute. To enable the attribute, the condition is **PRIORITY_INVERSION_ON** while a value of **PRIORITY_INVERSION_OFF** disables it.

Defining the state of the resource's priority inversion attribute is only possible during the time when the resource is not busy. If the resource is busy, an

attempt to define the attribute will fail and the function will return a value of **RC_BUSY**.

RETURN VALUE

The function returns a value of **RC_GOOD** if successful.

A value of **RC_BUSY** is returned if the resource is busy when this kernel service is attempted.

EXAMPLE

The current task wants to enable the priority inversion processing for resource *ALARM_LIST*. Once the resource attribute is defined the task will lock the resource, use it, and then release the resource.

```
#include "rtxcapi.h" /* RTXC KS prototypes */
#include "cres.h    "/* defines ALARM_LIST */

/* enable priority inversion processing */
while(KS_defres(ALARM_LIST, PRIORITY_INVERSION_ON) == RC_BUSY)
{
   ... handle resource Busy condition
}
/* here when resource priority inversion ON */
KS_lockw(ALARM_LIST);  /* lock the resource */

... use the resource for something

KS_unlock(ALARM_LIST); /* release resource */
```

SEE ALSO KS_lock, KS_lockt, KS_lockw, KS_unlock

This page is intentionally left blank.

KS_defslice

DEFINE A TASK'S TIME-SLICE QUANTUM

CLASS Task Management

SYNOPSIS

```
void  KS_defslice(TASK task,
                  TICKS slice)
```

DESCRIPTION

The KS_defslice kernel service defines the amount of time the specified task is permitted to run before it is forced to yield in a time-sliced scheduling situation.

The function requires a task number and a time-slice time quantum as arguments. The time quantum period is specified as the number of RTXC clock ticks approximating the desired duration of the time-slice quantum. A task number of zero (0) has special significance as it indicates the calling task.

If time-slicing is not in operation for the specified task and the time quantum value is non-zero, the task will be readied for time-sliced operation. The task will begin time-sliced operation only when there is another task in the Ready List having the same priority and also ready for time-sliced operation.

If the task is either ready for time-slice operation or is in active time-slice operation, its time quantum

can be changed at any time. However, the new time quantum will not be put into force until the expiration of the time quantum currently in force.

A time quantum value of zero (0) causes time-sliced operation to cease on the specified task. The cessation will not go into effect until the expiration of the time quantum currently active.

RETURN VALUE The function returns no value.

EXAMPLE The current task is to begin time-sliced operation with a time quantum of 100 msec. After some period of time-sliced operation, the task will cease time-sliced operation.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cclock.h"            /* defines CLKTICK*/

/* define time slice quantum as 100 ms */
KS_defslice(SELFTASK,100/CLKTICK);

   ... Task now in time-sliced operation

/* turn off time-sliced operation */
KS_defslice(SELFTASK,(TICKS)0);
```

SEE ALSO KS_inqslice

KS_deftask

DEFINE A TASK'S ATTRIBUTES

CLASS Task Management

SYNOPSIS

```
KSRC  KS_deftask(TASK task,
                 PRIORITY priority,
                 char *stack,
                 size_t stacksize,
                 void (*entry)(void))
```

DESCRIPTION

The KS_deftask kernel service defines the attributes of an inactive task. While it can be used on both static and dynamically allocated tasks, it is generally found in association with the latter whose TCB has been allocated with the KS_alloc_task kernel service. The purpose of the service is to prepare the task for execution by establishing the attributes necessary for operation. The attributes include a task number, a priority, a stack, and a task entry address.

The definition of attributes may only occur under certain conditions. First of all, a definition may only take place on a task whose state is INACTIVE. Secondly, it is not permissible for a task to define its own attributes. Therefore, the use of a task number argument of zero (0) will be in error.

RETURN VALUE The function returns a value of **RC_GOOD** if the

definition is successful.

The function returns a value of **RC_ILLEGAL_TASK** if an attempt is made to specify the object task's identifier with a value of zero (0).

If the object task's state is not INACTIVE, the function returns a value of **RC_ACTIVE_TASK**.

EXAMPLE

The Current Task needs to spawn another task, newtaskA, whose TCB it must allocate. The task's entry address is taskA, and the task requires a stack size of 256 bytes which the Current Task allocates from memory partition MAP256. The task will run at priority 14. After defining the task's attributes, the Current Task starts newtaskA executing without defining any environment arguments for it.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"          /* Memory Partitions */

extern void taskA(void);

TASK newtaskA;
PRIORITY newpri = 14;
char *pstk;
int stksz = 256;

newtaskA = KS_alloc_task();

pstk = KS_allocw(MAP256); /* allocate space for
                          /* task's stack */
KS_deftask(newtaskA,newpri,pstk,stksz,taskA);

KS_execute(newtaskA);
```

SEE ALSO

KS_alloc_task, KS_deftask_arg, KS_execute

KS_deftask_arg

DEFINE A TASK'S ENVIRONMENT ARGUMENTS

CLASS Task Management

SYNOPSIS

```
void  KS_deftask_arg(TASK task,
                     void *arg)
```

DESCRIPTION The KS_deftask_arg establishes a pointer to a structure containing parameters which define the environment of the specified task. The content of the structure may be anything required by the application. Normal use of this kernel service would be preceded by a section of code which defines each member of the structure.

The function requires a task number and a pointer to the environment arguments structure of the specified task.

RETURN VALUE The function returns no value.

EXAMPLE The Current Task needs to spawn another task which is to operate on the port and channel specified by the content of two variables, *port* and *chnl*, which have been determined elsewhere. The task is an instance of taskA whose TCB must be allocated dynamically and whose identifier is in newtaskA. The task's entry address is taskA and the task

requires a stack size of 256 bytes which the Current Task allocates from memory partition MAP256. The task will run at priority 14. After defining the task's attributes, the Current Task defines two environment arguments, channel and port, in a structure and makes that structure known to taskA. Having done so, the Current Task then starts taskA executing.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"          /* Memory Partitions */

extern void taskA(void);

struct envargA /* environment argument structure */
{
   int port;
   int channel;
};

struct envarg envargA;
TASK newtaskA;
PRIORITY newpri = 14;
char *pstk;
int stksz = 256;
int port, chnl;

newtaskA = KS_alloc_task();
pstk = KS_allocw(MAP256); /* allocate space for
                          /* task's stack */
KS_deftask(newtaskA,newpri,pstk,stksz,taskA);

envargA.port = port
envargA.channel = chnl

KS_deftask_arg(newtaskA,&envargA);
KS_execute(newtaskA);
```

SEE ALSO KS_alloc_task, KS_deftask, KS_execute, KS_inqtask_arg

KS_deftime

DEFINE SYSTEM TIME-OF-DAY AND DATE

CLASS Special

SYNOPSIS

```
void  KS_deftime(time_t time)
```

DESCRIPTION The *KS_deftime()* service defines the Date and Time-of-Day for the system. The function requires a single argument which is a value of type *time_t* containing the date and time as the number of seconds since January 1, 1970. A function, *date2systime()* is provided in the RTXC distribution to convert from conventional calendar dates and clock times to a value of type *time_t*. Documentation on the uses of *date2systime()* is found in the Binding Manual.

RETURN VALUE The function returns no value.

EXAMPLE The current task needs to define the Time-of-Day which it gets from an ASCII buffer which was input from the system console.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cvtdate.h" /* defines time_tm & proto- */
                      /* type for date2systime() */

struct time_tm d;

sscanf(buffer,"%d/%d/%d %d:%d:%d",
               &d.tm_yr, &d.tm_mon, &d.tm_day,
               &d.tm_hr, &d.tm_min, &d.tm_sec);


KS_deftime(date2systime(&d));
```

SEE ALSO KS_inqtime, date2systime

KS_delay

DELAY A TASK FOR A PERIOD OF TIME

CLASS Timer Management

SYNOPSIS

```
void  KS_delay(TASK task,
               TICKS period)
```

DESCRIPTION

The KS_delay service blocks the specified task for a period of time. The delayed task may be the current task or another task and the object task may or may not be in the READY List. If the task is in the READY List when delayed, the function removes it from the READY List. If a task is not in the READY List, it will remain blocked at least until the delay period elapses. Once the task is blocked, a timeout timer is established for the specified delay period.

The function requires a task number and a delay period as arguments. The delay period is specified as the number of RTXC clock ticks approximating the desired time of the delay. A task number of zero (0) has special significance as it indicates the calling task. Thus, a task need not know its own task number to schedule a delay for itself.

If the current task uses delay time of zero (0) ticks, there will be no delay and the calling task will

immediately resume. A delay in progress on a task other than the current task can be terminated by calling *KS_delay()* using the delayed task's identifier and a delay time of zero (0) ticks.

Caution should be exercised when scheduling or canceling delays for other tasks.

RETURN VALUE The function returns no value.

EXAMPLE The current task is to delay itself for a period of 100 msec. After some processing, the task is to be again delayed for a period of 50 msec. Note the two methods of defining the time period of the delay.

```
#include "rtxcapi.h" /* RTXC KS prototypes */
#include "cclock.h"  /* defines CLKTICK */
#include "ctask.h"   /* defines SCANR */

/* delay SELF task for 100 ms */
/* clktick is defined as extern in cclock.c */
/* 100/clktick calc done at run time (Slower) */
KS_delay(SCANR,100/clktick);

... continue processing after delay

/* then do another delay */
/* CLKTICK is system wide #define in cclock.h */
/* 50/CLKTICK calculation done at compile */
/* time because it is Fast */
KS_delay(SELFTASK,50/CLKTICK);  /* delay SELF */
                                /* for 50 ms */
```


KS_dequeue

GET ENTRY FROM A FIFO QUEUE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_dequeue(QUEUE queue,
                 void *dest)
```

DESCRIPTION Dequeue is used to get the oldest entry from a FIFO queue. If the queue is not empty, the oldest entry in the queue is removed and stored at the destination address given in the calling sequence. When the dequeueing operation is successful, the function returns a value of **RC_GOOD**.

If the queue is empty, no entry can be dequeued. The function immediately returns a function value of **RC_QUEUE_EMPTY** indicating the function failed to dequeue an entry.

If the queue becomes empty as a result of the KS_dequeue request and if there is a semaphore previously associated with the given queue's Queue_Empty event (see KS_defqsema), and if there is a task waiting for that event, the associated semaphore will be signaled to notify the waiting task of the occurrence of the event.

RETURN VALUE The oldest entry in the queue is placed at the address

specified by the argument in the calling sequence.

The Kernel Service function returns a value of **RC_GOOD** if the dequeue is successful and a value of **RC_QUEUE_EMPTY** if it is not.

EXAMPLE

Dequeue an entry from *DATAQ* and store it in the structure called *entry*.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"        /* defines DATAQ */

struct            /* structure for receiving the */
{                 /* dequeued entry */
   int type;
   int value;
} entry;

/* get data from DATAQ until it is empty*/
while (KS_dequeue(DATAQ,&entry) == RC_GOOD)
{
   ... do something with the entry just dequeued
}
... DATAQ was empty, deal with it here ...
```

SEE ALSO

KS_defqsema, KS_dequeuet, KS_dequeuew, KS_enqueue, KS_enqueuet, KS_enqueuew

KS_dequeuet

GET ENTRY FROM A FIFO QUEUE,
WAIT FOR LIMITED TIME
IF QUEUE EMPTY

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_dequeuet(QUEUE queue,
                  void *dest,
                  TICKS timeout)
```

DESCRIPTION

KS_dequeuet is like KS_dequeuew except that any blockage of the requesting task due to a Queue_Empty condition is limited to the time period specified by the timeout argument in the calling sequence.

If the queue is not empty, the oldest entry in the queue is removed and stored at the destination address given in the calling sequence. The Kernel Service function returns a value of **RC_GOOD** when the dequeueing operation is successful.

An empty queue causes the current task to be blocked and removed from the READY List. After the task is removed from the READY List, a timeout timer is established with a duration as defined by the timeout argument of the function call.

The task will remain blocked until such time as either of two conditions occurs:

- Another task puts an entry into the queue via one of the kernel services which performs an enqueue function, or,
- the timeout period elapses.

If the queue becomes empty as a result of the KS_dequeuet request, and there is a task waiting on the Queue_Empty event, then the associated semaphore is signaled to notify the task of the occurrence of the event.

RETURN VALUE

The oldest entry in the queue is placed at the address specified by the argument in the calling sequence.

The Kernel Service function returns a value of **RC_GOOD** if the dequeue is successful.

If the timeout occurs, the function returns a value of **RC_TIMEOUT**.

EXAMPLE

Dequeue an entry from *DATAQ* and store it in the structure called *entry*. If *DATAQ* is empty, wait no longer than 250 msec for data to become available before proceeding.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"        /* defines DATAQ */
#include "cclock.h"        /* defines CLKTICK */
struct            /* structure for receiving the */
{                 /* dequeued entry */
   int type;
   int value;
} entry;

/* get data from DATAQ */
if (KS_dequeuet(DATAQ, &entry, 250/CLKTICK) == RC_GOOD)
{
   ... do something here with queue entry
}
else
{
   ... timeout occurred. Deal with it here.
}
```

Note that the units of *timeout* are milliseconds. The number of milliseconds in the *timeout* is divided by the number of milliseconds per RTXC timer tick. The quotient is the number of RTXC timer ticks required to approximate the defined *timeout*.

SEE ALSO KS_dequeue, KS_dequeuew, KS_enqueue, KS_enqueuet, KS_enqueuew

This page is intentionally left blank.

KS_dequeuew

GET ENTRY FROM A FIFO QUEUE,
WAIT IF EMPTY

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_dequeuew(QUEUE queue,
                  void *dest)
```

DESCRIPTION

KS_dequeuew, like KS_dequeue, is also used to get the oldest entry from a FIFO queue. If the queue is not empty, the oldest entry in the queue is removed and stored at the destination address given in the calling sequence. The Kernel Service function does not return a value when the dequeueing operation is successful.

Unlike KS_dequeue, however, an empty queue causes the requesting task to be blocked and removed from the READY List until such time when another task puts an entry into the queue via one of the kernel services which performs an enqueue function.

If the queue becomes empty as a result of the KS_dequeue request, and if there is a semaphore previously associated with the given queue's Queue_Empty condition, and if there is a task waiting for the Queue_Empty condition, that

semaphore is signaled to notify the task of the occurrence of the event.

RETURN VALUE

The function returns no value. The oldest entry in the queue is placed at the address specified by the argument in the calling sequence.

EXAMPLE

Dequeue an entry from *DATAQ* and store it in the structure called *entry*. If *DATAQ* is empty, wait for data to become available before proceeding.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"        /* defines DATAQ */

struct            /* structure for receiving the */
{                 /* dequeued entry */
   int type;
   int value;
} entry;

KS_dequeuew(DATAQ,&entry);/* get data from DATAQ */
```

SEE ALSO

KS_dequeue, KS_dequeuet, KS_enqueue, KS_enqueuet, KS_enqueuew

KS_elapse

COMPUTE ELAPSED TIME

CLASS Timer Management

SYNOPSIS

```
TICKS  KS_elapse(TICKS *etime)
```

DESCRIPTION

The KS_elapse function returns the elapsed time between two events. Correct calculation of an elapse time requires two calls to KS_elapse. The first sets the beginning time into the time marker, *etime*. The value returned by the first kernel service function is worthless and should be discarded. The second call is issued at the time of the event which marks the end of the period being measured. The value returned by the kernel service function after the second invocation will be the elapsed time of the period.

The elapsed time is computed as the number of RTXC clock ticks between the initial time marker as contained in *etime* and the current system time at the end of the period.

At the same time that the function is calculating the difference between the two times to get the elapsed time, the current system time is moved to the time marker, *etime*, so that serial events can be timed.

If the elapsed time of a set of serial events needs to be measured, the first period is measured as described. However, since *etime* is updated to the current system time at the end of the previous event, it is also the starting time of the next event. Consequently, the elapsed times of the second and successive events can each be obtained by a single call to KS_elapse.

Resolution of the elapsed time is limited only by the RTXC base clock frequency and is guaranteed to be less than 1 clock period (TICK).

RETURN VALUE

The function returns the elapsed time in system clock ticks.

EXAMPLE

Calculate the elapsed time of two changes of state on a switch, where the change-of-state event is associated with the semaphore, *SWITCH*.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cclock.h"        /* defines CLKTICK */
#include "csema.h"         /* defines SWITCH */

TICKS timestamp, diff;

KS_wait(SWITCH);           /* wait for the first */
                           /* change of state */

KS_elapse(&timestamp);     /* determine t(0) */

KS_wait(SWITCH); /* wait for switch change event */

diff = KS_elapse(&timestamp);/* get elapsed time */
                             /* since t(0) */

... use the elapsed time for something ...

KS_wait(SWITCH);  /* wait for next switch change */

diff = KS_elapse(&timestamp);/* get elapsed time */
                  /* since start of period known */
... Use the second period's elapsed time
```

This page is intentionally left blank.

KS_enqueue

PUT ENTRY INTO FIFO QUEUE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_enqueue(QUEUE queue,
                 void *entry)
```

DESCRIPTION

KS_enqueue inserts an entry into a FIFO queue. If there is room in the queue for at least one entry, the operation will succeed. If the queue is full, there is no room to insert the desired entry and the function cannot proceed normally. Consequently, it returns control to the requesting task with a value indicating the insertion did not happen.

If the entry inserted into the queue causes the queue to reach the Queue_Full condition, and if there is a semaphore associated with the Queue_Full condition on the given queue, and if there is a task waiting for the queue to become Full, the Queue_Full semaphore is signaled to notify the waiting task.

RETURN VALUE

The function returns a value of **RC_GOOD** if the enqueueing operation is successful.

A returned value of **RC_QUEUE_FULL** indicates the function failed to insert the data into the given queue.

EXAMPLE

Insert data found in the structure named *entry* into queue *DATAQ* making sure that the operation succeeded.

```
#include "rtxcapi.h"     /* RTXC KS prototypes */
#include "cqueue.h"      /* defines DATAQ */

struct
{
   int type;
   int value;
} entry;

/* enqueue packet of data into DATAQ */
if (KS_enqueue(DATAQ, &entry) == RC_GOOD)
{
   ... operation successful
}
else
{
   ... queue is FULL. Deal with it here.
}
```

SEE ALSO

KS_dequeue, KS_dequeuet, KS_dequeuew, KS_enqueuet, KS_enqueuew

KS_enqueuet

PUT ENTRY INTO FIFO QUEUE,
WAIT FOR LIMITED TIME
IF QUEUE FULL

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_enqueuet(QUEUE queue,
                  void *entry,
                  TICKS timeout)
```

DESCRIPTION

KS_enqueuet inserts an entry into a FIFO queue. If there is room in the queue for at least one entry, the operation will succeed and return to the requesting task. If the queue state is Full, the function cannot proceed normally and will cause RTXC to remove the current task from the READY List and block it.

The duration of the task's blocking, unlike KS_enqueuew, is limited by the period of time specified by the *timeout* argument in the calling sequence, or the Queue_Full condition being removed, whichever occurs first. When the Queue_Full condition is cleared by another task removing an entry from the queue via a dequeueing operation, the entry is inserted into the queue and the waiting task unblocked.

If the queue reaches the Queue_Full condition, and there is a semaphore associated with its Queue_Full event, and if there is a task waiting for the queue to become Full, the semaphore associated with Queue_Full is signaled to notify the waiting task.

RETURN VALUE

The function returns a value of **RC_GOOD** if it completes successfully.

If the Queue_Full condition persists longer than the *timeout* period, the function returns a value of **RC_TIMEOUT**.

EXAMPLE

Insert data found in the structure named *entry* into queue *DATAQ*. If the queue is Full, wait for 500 msec or until the enqueue operation is successful.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"
#include "cclock.h"

struct
{   int type;
    int value;
} entry;

/* enqueue packet of info into DATAQ */
if (KS_enqueuet(DATAQ,&entry,500/CLKTICK) == RC_GOOD)
{
   ... enqueue operation was successful
}
else
{
... Timeout. Queue was full longer than 500 ms.
}
```

SEE ALSO

KS_dequeue, KS_dequeuet, KS_dequeuew, KS_enqueue, KS_enqueuew

KS_enqueuew

PUT ENTRY INTO FIFO QUEUE,
WAIT IF QUEUE FULL

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_enqueuew(QUEUE queue,
                  void *entry)
```

DESCRIPTION

KS_enqueuew inserts an entry into a FIFO queue. If there is room in the queue for at least one entry, the operation will succeed and return to the requesting task. No function value is returned. If the queue is full, the function cannot proceed normally causing it to remove the current task from the READY List and block it until the Queue_Full condition is removed. When the Queue_Full condition is cleared by another task removing an entry from the queue via a dequeue operation, the entry is inserted into the queue and the requesting task unblocked.

If the entry inserted into the queue causes the queue to reach the Queue_Full condition, and if there is a semaphore associated with the Queue_Full condition on the given queue, and if there is a task waiting for the queue to become FULL, the Queue_Full semaphore is signaled to notify the waiting task.

RETURN VALUE The function returns no value.

EXAMPLE Insert data found in the structure named *entry* into queue *DATAQ*. If the queue is full, wait until the enqueue operation can succeed.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"        /* defines DATAQ */

struct
{
   int type;
   int value;
} entry;

KS_enqueuew(DATAQ,&entry);  /* enqueue packet of */
                            /* info into DATAQ */
```

SEE ALSO KS_dequeue, KS_dequeuet, KS_dequeuew, KS_enqueue, KS_enqueuet

KS_execute

EXECUTE A TASK

CLASS Task Management

SYNOPSIS

```
void  KS_execute(TASK task)
```

DESCRIPTION

The KS_execute function starts a task from its beginning address. The task may be idle or it may already be running. If the latter, it is removed from the READY List. The task is inserted into the READY List with its program counter (PC) and stack pointer (SP) initialized to their starting values. The task's starting address, priority, and stack pointer are specified during system generation or dynamically with the KS_deftask Kernel Service.

If the new task is of higher priority than the requesting (current) task, a context switch is performed and the new task runs. If the requesting task is of higher priority, control is returned to the caller.

RETURN VALUE The function returns no value.

EXAMPLE The current task starts task *SHUTDOWN* from its starting address.

```
#include "rtxcapi.h"    /* RTXC KS prototypes */
#include "ctask.h"      /* defines task SHUTDOWN */

KS_execute(SHUTDOWN);   /* execute SHUTDOWN task */
```

SEE ALSO KS_terminate, KS_deftask

KS_free

FREE A BLOCK OF MEMORY

CLASS Memory Partition Management

SYNOPSIS

```
void  KS_free(MAP map,
              void *p)
```

DESCRIPTION

The free memory kernel service returns a block of memory at a specified address to the free pool for the given memory partition.

WARNING: No checks are performed to determine that the specified memory block to be released "belongs" in the designated partition.

It is the programmer's responsibility to ensure adherence to the rule that a block is freed ONLY to the partition from which it was allocated. If this rule is violated, a partition's content can become corrupted with blocks of memory from other partitions.

However, this rule has at least one exception which can prove useful. It is possible during system generation to define more than one partition having the same size blocks. One large virtual partition can then be constructed dynamically by allocating the blocks from one partition and freeing them into another partition which will then contain the aggregate

number of blocks. This technique can overcome certain addressing limitations of segmented architecture computers that limit the size of a single RTXC memory partition.

Likewise, a partition may also be extended by allocating similarly sized blocks of memory from the heap or from another RAM area within the system's address space and freeing them to a given partition.

RETURN VALUE The function returns no value.

EXAMPLE Allocate a block of memory from the *BUFFMAP* partition, use it for a while as a character buffer and then return it to *BUFFMAP*.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"         /* defines BUFFMAP */

char *p;

p = (char *)KS_alloc(BUFFMAP);  /* get block for */
                                /* temporary use */
... /* use block for some operation */

KS_free(BUFFMAP,p); /* return block to BUFFMAP*/
```

SEE ALSO KS_alloc, KS_alloct, KS_allocw, KS_inqmap

KS_free_part

FREE A DYNAMIC MEMORY PARTITION HEADER

CLASS Memory Partition Management

SYNOPSIS

```
void  *KS_free_part(MAP map)
```

DESCRIPTION The free dynamic memory partition header kernel service returns a dynamic partition header to the free pool of dynamic partition headers.

RETURN VALUE The function returns a pointer to the block of memory that was passed to KS_alloc_part or KS_create_part when the dynamic memory partition was created.

EXAMPLE Allocate a block of memory from the *MAP512* partition, create a dynamic partition with it, use it for a while then free the dynamic partition header.


```
#include "rtxcapi.h"      /* RTXC KS prototypes */
#include "cpart.h"           /* defines MAP512 */

MAP map1;
char *body;
size_t blksize, n_blks;

   ... blksize and n_blks defined by some means

if ( (body = KS_alloc(MAP512)) == (char *)0 )
{
   ... Deal with no block available for dynamic
       Map's body. Maybe try another Map?
}

if ( (map1 = KS_create_part(body, blksize,n_blks) == (MAP)0 )
{
   /* the attempt to create a dynamic Map failed */
   /* free the unused RAM block */
   KS_free(MAP512, body);

   ... Then deal with the failure of the dynamic
       Memory Partition creation
}
else
{
   ... Creation was successful
   ... Use the partition a while
   ... Then free it
   body = KS_free_part(map1);
   ... Body may be used for another
   ...... dynamic partition
   ... Or released back to MAP512
}
```

SEE ALSO KS_alloc_part, KS_create_part, KS_defpart

KS_free_timer

FREE A TIMER BLOCK

CLASS Timer Management

SYNOPSIS

```
void  KS_free_timer(CLKBLK *timer)
```

DESCRIPTION KS_free_timer returns a given timer block to the pool of free timers. The calling argument *timer* is a pointer to the timer to be released. The function will stop an active timer prior to freeing it.

RETURN VALUE The function returns no value.

EXAMPLE Allocate a timer block and store its address in pointer *p*. Start a 250 msec timer using that timer block, and then free it when the timer expires.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cclock.h"        /* defines CLKTICK */
#include "csema.h"         /* defines AISEMA */

CLKBLK *p;

p = (CLKBLK *)KS_alloc_timer();
if (p == (CLKBLK *)0)
{
   ... Deal with no timers available here
}
else
{
   KS_start_timer(p, 250/CLKTICK, 1000/CLKTICK,
                  AISEMA);

   ... do some processing ...

   KS_free_timer(p);        /* release the timer */
}
```

SEE ALSO KS_alloc_timer, KS_start_timer

KS_inqmap

RETURNS SIZE OF BLOCK IN A PARTITION

CLASS Memory Partition Management

SYNOPSIS

```
size_t  KS_inqmap(MAP map)
```

DESCRIPTION KS_inqmap returns a value equal to the size of each block in the specified partition. This function is intended for applications using blocks from multiple partitions or those having no prior knowledge of the block sizes in the system.

RETURN VALUE The function returns a number equal to the size of a block in the specified memory partition.

EXAMPLE A task needs to compute the blocking factor for data to be packed into a block of memory from partition *MAPVCT*. It first inquires about the size of the block and then computes the blocking factor by dividing the block size by the size of the structure, *entry*, being used for one data packet.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"

size_t size;
int bfactor;         /* computed blocking factor */
struct entry
{
   int len;
   char data[10];
}

size = KS_inqmap(MAPVCT);   /* get size of block */
        /* calculate block factor for structure */
bfactor = size / sizeof(struct entry);
```

SEE ALSO KS_alloc, KS_alloct, KS_allocw, KS_free

KS_inqpriority

INQUIRE ON A TASK'S PRIORITY

CLASS Task Management

SYNOPSIS

```
PRIORITY  KS_inqpriority(TASK task)
```

DESCRIPTION The KS_inqpriority function allows the calling task to make a direct inquiry about the priority of a task. A *task* value of zero (0) specifies the current task.

RETURN VALUE The function returns the priority of the specified task.

EXAMPLE Look at the priority of the current task and reduce its priority by 2.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"            /* defines HISTASK */
#define SELF (TASK(0))  /* Used for Current Task */

PRIORITY mypri, hispri;

mypri = KS_inqpriority(SELF);

/* raise Current Task's priority by 2 */
KS_defpriority(SELF, mypri-2);

hispri = KS_inqpriority(HISTASK);

... Do something with the other task's priority
```


SEE ALSO KS_defpriority

KS_inqqueue

INQUIRE ABOUT NUMBER OF ENTRIES IN QUEUE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
int  KS_inqqueue(QUEUE queue)
```

DESCRIPTION The KS_inqqueue function allows the calling task to make a direct inquiry about a queue's current size. The current size is expressed in terms of entries in the queue rather than the number of bytes.

RETURN VALUE The function returns the number of entries currently in queue.

EXAMPLE Look at the current size of *CHARQ* and signal the *XOFF* semaphore if the queue contains more than 20 entries. Signal the *XON* semaphore if the current size of the queue is less than 4 entries.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"             /* defines CHARQ */
#include "csema.h"       /* defines XOFF and XON */

int depth;

depth = KS_inqqueue(CHARQ);/* get depth of CHARQ */
if (depth > 20)
  KS_signal(XOFF);
if (depth < 4)
  KS_signal(XON);
```

SEE ALSO KS_dequeue, KS_dequeuet, KS_dequeuew, KS_enqueue, KS_enqueuet, KS_enqueuew

SPECIAL NOTE The current queue size may change between the time the task calls the KS_inqqueue service and its next request for an enqueue or dequeue service.

KS_inqres

INQUIRE ON THE OWNER OF A RESOURCE

CLASS Resource Management

SYNOPSIS

```
TASK  KS_inqres(RESOURCE resource)
```

DESCRIPTION The KS_inqres function allows the calling task to determine the owner, if any, of a specified resource.

RETURN VALUE The function returns the identifier of the task that currently owns the given resource or a value of zero (0) if the resource is unlocked.

EXAMPLE Determine the owner of the PRINTER resource and see if is owned by the Alarm Output task, ALRMTASK.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cres.h"          /* defines PRINTER */
#include "ctask.h"         /* defines ALRMTASK */


if (KS_inqres(PRINTER) == ALRMTASK)
{
   ... do something
}
else
{
   ... do something else if resource is unlocked
}
```


SEE ALSO KS_lock, KS_lockt, KS_lockw

KS_inqsema

RETURN CURRENT STATE OF SEMAPHORE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
SSTATE  KS_inqsema(SEMA semaphore)
```

DESCRIPTION KS_inqsema returns a value indicating the state of the given semaphore.

NOTE: The state of the semaphore may actually change between the time the request is issued and the time the semaphore state is returned, due to an exception which interrupts the kernel service and alters the state of the semaphore.

RETURN VALUE The function returns a number equivalent to the state of the semaphore as follows:

- SEMA_DONE
- SEMA_PENDING
- Task ID number of waiting task

EXAMPLE The current task wants to determine if semaphore AIDONE is in a DONE state. If so, it is to perform some processing.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"         /* defines AIDONE */

if (KS_inqsema(AIDONE) == SEMA_DONE)
{
   ... semaphore is DONE, process something
};
```

SEE ALSO KS_pend, KS_signal, KS_wait, KS_waitm, KS_waitt

KS_inqslice

GET TIME-SLICE TIME QUANTUM

CLASS Task Management

SYNOPSIS

```
TICKS  KS_inqslice(TASK task)
```

DESCRIPTION The KS_inqslice function allows the calling task to obtain the value of the time-slice time quantum for the object task. If there has been no time-slice time quantum defined for the specified task, the function returns a value of zero (0).

RETURN VALUE The function returns the value of the specified task's time-slice time quantum in units of system clock ticks.

EXAMPLE Get the time-slice time quantum for the task *SCANR* and see if it has been defined. If not, define the task's time quantum at 100 msec.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"              /* defines SCANR */
#include "cclock.h"           /* defines CLKTICK */


if( (KS_inqslice(SCANR) == (TICKS)0)
   KS_defslice(SCANR,100/CLKTICK);
```

SEE ALSO KS_defslice

KS_inqtask

GET NUMBER OF CURRENT TASK

CLASS Task Management

SYNOPSIS

```
TASK  KS_inqtask(void)
```

DESCRIPTION The KS_inqtask function allows the calling task to obtain its task identifier.

RETURN VALUE The function returns the task number of the Current Task.

EXAMPLE Get the task number of the Current Task and use it as an argument in changing the priority of the task to 10.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */

TASK mytaskid;

mytaskid = KS_inqtask();

KS_defpriority(mytaskid, 10);
```

SEE ALSO KS_defpriority

This page is intentionally left blank.

KS_inqtask_arg

GET ADDRESS OF TASK'S ENVIRONMENT ARGUMENTS

CLASS Task Management

SYNOPSIS

```
void  *KS_inqtask_arg(TASK task)
```

DESCRIPTION

The KS_inqtask_arg function allows the calling task to obtain a pointer to the structure containing the environment arguments for the specified task. The *task* argument may be zero (0) to indicate that the request is made for the calling task's environment arguments.

This call may be used by any task whose environment arguments have been previously defined to RTXC by the KS_deftask_arg function. Normally, the function will be used by tasks which have been dynamically defined by the KS_deftask function. Those tasks would likely have an associated environment argument structure in order to determine the parameters they need to operate.

RETURN VALUE

The function returns a pointer to the specified task's environment argument structure. If no such definition has been made, the function returns a NULL pointer.

EXAMPLE

The Current Task is a communications channel driver and is an instance of a task which may have clones also in operation. In order to run, it needs to determine the operational parameters, the communications port and channel, on which it will operate. It will do that by getting the data from its environment argument structure which contains the port and channel identifiers. The environment argument structure has been previously defined.

While the example below is simple, it demonstrates some of the basic concepts in organizing a task which is dynamically allocated, defined, and executed. In contrast to a static task, the dynamic task is normally used in situations where each instance of the task serves one particular purpose. In the example to follow, the purpose is to handle a single communications port and the channel on that port.

Other instances of the same task may already be in operation on other port/channels. Thus it is necessary for the task to determine which it is and the critical data it needs in order to operate. That data should be found in the task's environment argument structure, the content of which was probably filled by the task that spawned the Current Task.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#define SELF (TASK(0))    /* define Current Task */

void commchnl(void)
{
   struct myargs
   {
      short port;  /* port number */
      short chnl;  /* channel number */
   };
   struct myargs *envargs;
   int chnlstat;          /* port/channel status */

   /* first find out which we are by getting the */
   /* environment arguments */
   envargs = KS_inqtask_args(SELF);

   while((chnlstat = get_chnl_stat(envargs->port,
                     envargs->chnl)) != 0)
   {
      ... while the status is non zero, do some
          work with the port and channel to
          process the data stream
   }
   KS_terminate(SELF); /* terminate when the */
                              /* status is 0 */
}
```

SEE ALSO KS_deftask, KS_deftask_arg

This page is intentionally left blank.

KS_inqtime

GET CURRENT TIME-OF-DAY AND DATE

CLASS Special

SYNOPSIS

```
time_t  KS_inqtime(void)
```

DESCRIPTION A task needing to determine the current time-of-day and/or date can use the KS_inqtime Kernel Service. The function returns the System Calendar as a value of type *time_t*. If the task needs to present the System Time as normal calendar and clock data, the value returned by the function should be passed to the *systime2date()* function. Documentation on *systime2date()* is found in the Binding Manual.

RETURN VALUE The function returns System Time as a single value of type *time_t*. If there has been no definition of an actual date, the returned value represents the number of seconds that have elapsed since the system was initialized. If there has been a date defined, the returned value represents the number of seconds that have elapsed from January 1, 1970 to the present time.

EXAMPLE The Current Task wants to output the current date and time-of-day to the console via the Console Output Queue.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cqueue.h"             /* defines CONOQ */
#include "cvtdate.h" /* defines time_tm & proto- */
                      /* type for systime2date() */


struct time_tm timenow;
char buffer[40];



systime2date(KS_inqtime(), &timenow);
                /*get the date*/* & time-of-day */
/* now prepare the output string */
sprintf(&buffer,"DATE: %d/%d/%d TIME: %d:%d:%d\n",\
        timenow.tm_mon, timenow.tm_day,
        timenow.tm_yr, timenow.tm_hr,
        timenow.tm_min, timenow.tm_sec)

/* send string to console */
printl(&buffer,0,CONOQ);
```

SEE ALSO KS_deftime, systime2date

KS_inqtimer

GET TIME REMAINING ON A TIMER

CLASS Timer Management

SYNOPSIS

```
TICKS  KS_inqtimer(CLKBLK *timer)
```

DESCRIPTION The KS_inqtimer function allows the calling task to obtain the time remaining on a specified timer, the pointer to which is passed as an argument to the function. If the specified timer is in an ACTIVE state, the remaining time will be returned in units of RTXC clock ticks. If the timer status is not ACTIVE, the function will return a value of zero (0).

RETURN VALUE The function returns the number of ticks remaining on the given timer if the timer is ACTIVE. Otherwise, it returns a value of zero (0).

EXAMPLE The Current Task starts a 500 msec timer and then waits on *TMRSEMA*, the timer expiration, or another event using semaphore *INTSEMA*. When either event occurs, the task determines which event happened and sets up a variable, *remainder*, that contains the time remaining on the active timer. If the event associated with *INTSEMA* occurred, the remaining time is obtained and the timer is stopped. Otherwise, the value of *remainder* will be zero (0).


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"  /* defines INTSEMA & TMRSEMA */
#include "cclock.h"           /* defines CLKTICK */

TICKS remainder;
CLKBLK *pclkblk;
SEMA sema;
SEMA *semalist[] =
{
   INTSEMA, TMRSEMA, 0
};
/* allocate a timer and start it */
pclkblk = KS_alloc_timer();
KS_start_timer(pclkblk,500/CLKTICK,0,TMRSEMA);

/* now wait for either the event or the timer */
sema = KS_waitm(semalist);
switch (sema)
{
   case INTSEMA:               /* event occurred */
      remainder = KS_inqtimer(pclkblk);
      KS_stop_timer(pclkblk);
      break

   case TMRSEMA:               /* timer occurred */
      remainder = 0;
      ... timer elapsed before event occurred
          at this point both semaphores are back
          in a PENDING state and the timer is in
          an INACTIVE state.
      break:
}
... now do something with the remainder
```

SEE ALSO KS_start_timer, KS_stop_timer

KS_ISRalloc

ALLOCATE A BLOCK OF MEMORY FROM AN ISR

CLASS ISR Services

SYNOPSIS

```
void  *KS_ISRalloc(MAP map)
```

DESCRIPTION The KS_ISRalloc Kernel Service function allows an interrupt service routine to allocate a block of memory. The function locates the next free block in the given RTXC Memory Partition specified by *map* and returns its address to the calling interrupt service routine as the value of the function. If no block is available in the specified partition, a value of NULL is returned. Interrupts are disabled while the function is executing and a context switch during the kernel call is not possible.

RETURN VALUE The function returns a pointer to the memory block if successful. If there are no available blocks in the given partition, the map is said to be empty and a NULL pointer (void *(0)) is returned.

EXAMPLE In this example, a block of memory from one of the RTXC memory partitions, *MAP1*, is needed. If the allocation is successful, the pointer to the block is to be stored in a character pointer *p*. If there are no free blocks in the partition, the interrupt service routine must take the appropriate action.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cpart.h"         /* defines MAP1 */

char *p;


if ( (p = (char *)KS_ISRalloc(MAP1)) == NULL)
{
   ... Deal with no memory available
}
else
{
   ... Allocation was successful
}
```

SEE ALSO KS_ISRexit, KS_ISRsignal, KS_ISRtick

KS_ISRexit

EXIT AN INTERRUPT SERVICE ROUTINE

CLASS ISR Services

SYNOPSIS

```
FRAME  *KS_ISRexit(FRAME *frame,
                   SEMA sema)
```

DESCRIPTION The KS_ISRexit service provides a generalized means of terminating an interrupt service routine and informing RTXC of the event. The function requires that the pointer to the interrupted context be passed to RTXC. Optionally, a semaphore may also be signaled as part of this function. If no semaphore is to be signaled, the semaphore identifier should be passed as a value of zero (0).

RETURN VALUE The service returns a pointer to the stack frame of the highest priority task in the Ready List. The stack frame pointer is used by the ISR epilogue to restore the context of the highest priority task.

EXAMPLE When a Push Button is pressed it causes an interrupt. Upon acknowledgment of the request, the Current Task is interrupted and CPU control is granted to the associated Interrupt Service Routine (ISR). After clearing the source of the interrupt, the device servicing routine needs to inform RTXC of the Push Button event by exiting the ISR and

signaling semaphore PBISEMA. The value returned by the function points to the context of the highest priority task in the Ready List.

```
/* Interrupt service example - Push Button input
*/
/* C level Push Button device service function */
FRAME *pbic(FRAME *frame)
{
... clear the interrupt source

return(KS_ISRexit(frame, PBISEMA));
}
```

SEE ALSO KS_ISRsignal, KS_ISRtick, KS_ISRalloc

KS_ISRsignal

SIGNAL SEMAPHORE FROM AN INTERRUPT SERVICE ROUTINE

CLASS ISR Services

SYNOPSIS

```
void  KS_ISRsignal(SEMA sema)
```

DESCRIPTION The KS_ISRsignal provides a means by which an interrupt service routine may signal a semaphore. This function supplements the semaphore signaling capability of *KS_ISRexit()* and is intended for use when the ISR needs to signal more than one semaphore.

RETURN VALUE The service returns no value.

EXAMPLE In an interrupt service routine for a full duplex serial I/O driver, it is possible that two events can be detected during the course of servicing the UART device. If such a situation occurs, signal both.


```
FRAME *uartc(FRAME *frame)
{
   /* test source of interrupt */
   if (USART_STATUS == TX_BUFF_EMPTY)
   {
      ... Output: clear output interrupt here
      /* now see if input also happened */
      if (USART_STATUS == RX_READY)
      {
         /* input is also READY */
         ... read character and clear interrupt
         /* signal serial input semaphore */
         KS_ISRsignal(SERINSEMA);
      }
      /* exit and signal serial output semaphore */
      return(KS_ISRexit(frame, SEROUTSEMA));
   }
   else  /* if here it is USART input */
   {
      ... Input: read character and clear interrupt
      /* now see if output happened
      if (USART_STATUS == TX_BUFF_EMPTY)
      {
         /* output DONE */
         ... clear interrupt source
         KS_ISRsignal(SEROUTSEMA); /*signal event*/
      }
      /* signal input semaphore and end ISR */
      return(KS_ISRexit(frame,SERINSEMA));
   }
}
```

SEE ALSO KS_ISRexit, KS_ISRtick, KS_ISRalloc

KS_ISRtick

PROCESS A CLOCK TICK INTERRUPT

CLASS ISR Services

SYNOPSIS

```
int  KS_ISRtick(void)
```

DESCRIPTION The KS_ISRtick service provides a means of performing all of the RTXC dependent functions necessary when a clock TICK interrupt occurs.

RETURN VALUE The kernel service returns an integer value of **1** if the function determines that a timer has expired and needs to be signaled.

It returns a value of **0** if no timer expired as a result of the clock TICK.

EXAMPLE A model for the device servicing function of a clock driver is the only place where *KS_ISRtick()* should appear.


```
FRAME *clkc(FRAME *frame)
{
... clear device specific interrupt

... do any application specific processing

KS_ISRtick(); /* process the clock tick */
/* return from interrupt */
return(KS_ISRexit(frame, (SEMA)0));
}
```

SEE ALSO KS_ISRexit, KS_ISRalloc

KS_lock

REQUEST EXCLUSIVE USE OF A RESOURCE

CLASS Resource Management

SYNOPSIS

```
KSRC  KS_lock(RESOURCE resource)
```

DESCRIPTION

The KS_lock service provides a generalized means of requesting or managing a logical resource during a period of exclusive use. A logical resource can be anything, such as a shared database, non-reentrant code (i.e., BIOS/DOS), math coprocessor or emulator library, etc. Nested lock requests by the current owner are supported. However, unlock requests by non-owners are ignored.

If the specified resource is idle, it is marked BUSY to prevent other tasks from using it, and a function value of **RC_GOOD** is returned. If the resource is owned at the time of request, the calling task resumes with a function value of **RC_BUSY** being returned from KS_lock.

RETURN VALUE

The kernel service returns a value of **RC_GOOD** if the lock attempt succeeds for the initial lock. A value of **RC_NESTED** is returned if the resource is already owned by the caller.

It returns a value of **RC_BUSY** if the specified resource is owned by another task.

EXAMPLE

The current task wants to output a system status report to the system printer without interspersed messages from other system monitors. When the report is finished, exclusive use of the printer is to be released.

If the printer is unavailable, perform a code segment to handle the situation.

In this example it is known that the current task does not own the resource prior to the call to KS_lock.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cres.h"          /* defines PRINTER */

if (KS_lock(PRINTER) == RC_GOOD)
{
   ... PRINTER is now locked for exclusive use
       during printing of status report

   KS_unlock(PRINTER);   /* release PRINTER lock */
}
else
{
...PRINTER is locked by another task. Deal with it.
}
```

SEE ALSO KS_lockt, KS_lockw, KS_unlock

KS_lockt

REQUEST EXCLUSIVE USE OF A RESOURCE,
WAIT FOR LIMITED TIME IF BUSY

CLASS Resource Management

SYNOPSIS

```
KSRC  KS_lockt(RESOURCE resource,
               TICKS timeout)
```

DESCRIPTION

KS_lockt operates like the KS_lockw kernel service except that it limits the duration of the waiting period should the object resource be busy. It provides a generalized means of requesting or managing a logical resource to be used for exclusive use. A logical resource can be anything, such as a shared database, non-reentrant code (i.e., BIOS/DOS), math coprocessor or emulator library, etc. Nested lock requests by the current owner are supported. However, unlock requests by non-owners are ignored.

If the specified resource is inactive, it is marked BUSY to prevent other tasks from using it. If the resource is BUSY at the time of request, the calling task is blocked and removed from the READY List until the task currently using the resource unlocks it. A timeout timer is started with a duration as specified by the timeout argument in the calling sequence.

RETURN VALUE

If the calling task already owns the resource, a timeout timer is not started and a value of **RC_NESTED** is returned immediately.

If the ownership of the resource is gained before the timeout expires, the function returns a value of **RC_GOOD**.

If the timeout occurs, the function returns a value of **RC_TIMEOUT**.

EXAMPLE

The current task wants to output a system status report to the system printer without interspersed messages from other system monitors. When the report is finished, exclusive use of the printer is to be released.

If the printer is unavailable for a period of 5 seconds, perform a code segment to handle the situation and then try it again.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cres.h"          /* defines PRINTER */
#include "cclock.h"

/* lock the Printer resource. */
/* Limit WAIT to 5 sec. */
while ((KS_lockt(PRINTER, 5000/CLKTICK)) == RC_TIMEOUT)
{
   ... Resource unavailable. Timeout occurred.
}
...PRINTER resource is now locked and no other
   task may gain access to it. Print report.

KS_unlock(PRINTER);     /* release PRINTER lock */
```

SEE ALSO

KS_lock, KS_lockw, KS_unlock

This page is intentionally left blank.

KS_lockw

REQUEST EXCLUSIVE USE OF A RESOURCE, WAIT IF BUSY

CLASS Resource Management

SYNOPSIS

```
KSRC  KS_lockw(RESOURCE resource)
```

DESCRIPTION

The KS_lockw service provides a generalized means of requesting or managing exclusive use of a logical resource. A logical resource can be anything, such as a shared database, non-reentrant code (i.e., BIOS/DOS), math coprocessor or emulator library, etc. Nested lock requests by the current owner are supported. However, unlock requests by non-owners are ignored.

If the specified resource is idle, it is marked BUSY to prevent other tasks from using it. If the resource is BUSY at the time of the request and is not owned by the calling task, the calling task is blocked and removed from the READY List until the task currently using the resource unlocks it.

RETURN VALUE

The function returns a value of **RC_GOOD** for the initial KS_lock call by a task.

A value of **RC_NESTED** is returned if the calling task already owns the resource.

EXAMPLE

The current task wants to output a system status report to the system printer without interspersed messages from other system monitors. When the report is finished, exclusive use of the printer is to be released.

If the printer is busy, do not proceed. Wait for it to become available.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cres.h"          /* defines PRINTER */

KS_lockw(PRINTER);

...PRINTER resource is now locked and no other task
   may gain access to it

KS_unlock(PRINTER); /* release PRINTER lock */
```

SEE ALSO

KS_lock, KS_lockt, KS_unlock

KS_nop

NO OPERATION

CLASS Special

SYNOPSIS

```
void  KS_nop(void);
```

DESCRIPTION The KS_nop function is included in the set of kernel services for completeness. It can serve as a means of benchmarking performance for entry into and exit from the kernel.

RETURN VALUE The function returns no value.

EXAMPLE Perform 10,000 iterations of the KS_nop kernel service and compute the elapsed time of those calls in units of system clock ticks.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */

int i;
TICKS timestamp, et;

KS_elapse(&timestamp);        /* dummy for setup */
for (i = 0; i < 10000; i++)
   KS_nop();

et = KS_elapse(&timestamp); /* read elapsed time */
                            /* after 10000 loops */
printf("10000 KS_nops in %d ticks\n",et);
```

This page is intentionally left blank.

KS_pend

FORCE A DONE SEMAPHORE
TO A PENDING STATE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_pend(SEMA sema)
```

DESCRIPTION KS_pend forces the state of a semaphore to PENDING if the state is currently DONE. If it is WAITING, no change is made. Normally, the state of a semaphore is automatically maintained by RTXC. However, there may be a requirement to wait on some event unconditionally, regardless of whether it has previously occurred. In other words, disregard prior occurrences of the event and wait for the next instance of the event. Forcing the semaphore associated with the event to a PENDING state, followed closely by a call to an event wait function, will achieve that result.

RETURN VALUE The function returns no value.

EXAMPLE Force semaphore SWITCH to the PENDING state before waiting on the event associated with it.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"   /* defines SWITCH semaphore */

KS_pend(SWITCH);      /* force semaphore PENDING */
KS_wait(SWITCH);      /* wait on change-of-state */
```

SEE ALSO KS_wait, KS_waitm, KS_waitt, KS_pendm

KS_pendm

FORCE MULTIPLE DONE SEMAPHORES TO PENDING STATE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_pendm(SEMA *semalist)
```

DESCRIPTION The KS_pendm function performs the same operation as done by the KS_pend function except that it operates on a list containing one or more semaphores. All semaphores in the list which are in a DONE state will be set to a PENDING state. This directive reduces the number of kernel calls when multiple semaphores must be set to PENDING.

A semaphore list is a null terminated array of semaphore identifiers.

RETURN VALUE The function returns no value.

EXAMPLE The current task needs to set the semaphores associated with the change-of-state events on two pushbuttons. The semaphores are named *SWITCH1* and *SWITCH2*. After forcing the PENDING state, the task is to wait for either event to occur.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"  /* defines SWITCH1 & SWITCH2 */

SEMA semalist[] =
{
   SWITCH1,
   SWITCH2,
   0                /* list terminator */
};

SEMA event;

KS_pendm(semalist);   /* forget switch histories */
event = KS_waitm(semalist);   /* wait for either */
                              /* switch to change */
```

SEE ALSO KS_pend, KS_wait, KS_waitm, KS_waitt

KS_purgequeue

RESET QUEUE TO EMPTY STATE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_purgequeue(QUEUE queue)
```

DESCRIPTION The KS_purgequeue service forces a queue to a known virgin condition (empty, no waiting tasks for any full/empty conditions). Note, any tasks waiting to enqueue (due to Queue_Full condition) or dequeue (due to Queue_Empty condition) will be at risk.

RETURN VALUE The function returns no value.

EXAMPLE Two tasks, PUTTER and GETTER, need to begin execution knowing that queue DATAQ is empty. Before starting the tasks, DATAQ is cleared.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"    /* defines PUTTER & GETTER */
#include "cqueue.h"             /* defines DATAQ */

KS_purgequeue(DATAQ);    /* reset DATAQ to empty */

KS_execute(PUTTER);       /* start producer task */
KS_execute(GETTER);       /* start consumer task */
```


SEE ALSO KS_dequeue, KS_dequeuet, KS_dequeuew,
KS_enqueue, KS_enqueuet, KS_enqueuew

KS_receive

RECEIVE A MESSAGE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
RTXCMSG  *KS_receive(MBOX mailbox,
                     TASK task)
```

DESCRIPTION

The KS_receive function fetches messages from a specified mailbox and returns the pointer to the message. If there are no messages in the mailbox, the function returns a NULL pointer to indicate the empty condition.

If the TASK argument contains a value of zero, the first message in the mailbox, from any sender, is returned. Because the messages are placed in the mailbox in priority order as specified by the sender, they are processed in the same sequence.

It is possible, however, to override the strict priority processing. If the receiving task specifies a non-zero task number in the calling sequence, the first message in the mailbox from that task will be returned.

RETURN VALUE The function returns a pointer to message if a message was received.

If no message was available, the function returns a NULL pointer.

EXAMPLE

A task wants to receive the next message in its mailbox, *MYMAIL*, from any sender. If a message is received, it will be processed and at the conclusion of processing, the sending task will be notified. If no message is in the mailbox, the task will execute special code to deal with the situation.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"            /* defines MYMAIL  */
#include "rtxstruc.h"         /* defines RTXCMSG */
struct{
   RTXCMSG msghdr;  /* Message header (required) */
   char data[10];       /* start of message body */
} MYMSG;
MYMSG *msg;

/* receive next message from any task */
msg = (MYMSG *)KS_receive(MYMAIL,(TASK)0);
if (msg != (MYMSG *)0 )
{
   ... message received, process it ...
   KS_ack(msg); /* acknowledge message processed */
}
else {
       ... Deal with no message available
     }
```

SEE ALSO

KS_ack, KS_receivet, KS_receivew, KS_send, KS_sendt, KS_sendw

KS_receivet

RECEIVE A MESSAGE,
WAIT FOR LIMITED TIME
IF MAILBOX EMPTY

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
RTXCMSG  *KS_receivet(MBOX mailbox,
                      TASK task,
                      TICKS timeout,
                      KSRC *ret_code)
```

DESCRIPTION

The KS_receivet function fetches messages from a specified mailbox and returns the pointer to the message. If there are no messages in the mailbox, the requesting task is blocked and removed from the READY List. The task will remain blocked until another task sends a message to the specified mailbox or until the expiration of a period of time defined by the timeout argument in the calling sequence.

When either the next message is sent to the mailbox, or the timeout occurs, the waiting receiver task will be unblocked and inserted into the READY List. The function also returns a value indicative of how it processed the request. This value is stored in the address pointed to by the *ret_code* parameter in the calling arguments. It is useful in determining which event caused the resumption of the requesting task.

If the *task* argument contains a value of zero, the first message in the mailbox, from any sender, is returned. Because the messages are placed in the mailbox in priority order as specified by the sender, they are processed in the same sequence. It is possible, however, to override the strict priority processing. If the receiving task specifies a non-zero task number in the calling sequence, the first message in the mailbox from that task will be returned.

If a message was received, the task is resumed with a pointer to the message returned as the value of the function. If the timeout occurred, the function returns a NULL pointer as the value of the function and stores the value **RC_TIMEOUT** via *ret_code*.

RETURN VALUE

The function returns a pointer to the received message if one was found in the mailbox. The value **RC_GOOD** is also stored via *ret_code*.

If the timeout timer expires before there is any mail sent to the mailbox, the function returns a NULL pointer and stores the value **RC_TIMEOUT** via *ret_code*.

EXAMPLE

The current task is to receive the next message from its mailbox, *MYMAIL*. If there is no mail in the mailbox, the task is to wait for a period of up to 500 msec for something to arrive. If the 500 msec period elapses without receipt of mail, the task is to resume and perform a special code segment to handle the timeout situation.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"            /* defines MYMAIL  */
#include "cclock.h"           /* defines CLKTICK */
#include "rtxstruc.h"         /* defines RTXCMSG */

struct{
   RTXCMSG msghdr;  /* Message header (required) */
   char data[10];       /* start of message body */
} MYMSG;
MYMSG *msg;
TICKS timeout = 500/CLKTICK;
KSRC ccode;

/* receive next message from any task */
if ( (msg = (MYMSG *)KS_receivet(MYMAIL, (TASK)0, timeout, &ccode)
                                                    == (RTXCMSG *)0 )
{
   ... timeout occurred or there were no timer
       blocks available. Deal with it here.
}
else
{
   ... message received, process it.

   KS_ack(msg);               /* signal sender */
}
```

SEE ALSO KS_ack, KS_receive, KS_receivew, KS_send, KS_sendt, KS_sendw

This page is intentionally left blank.

KS_receivew

RECEIVE A MESSAGE,
WAIT IF MAILBOX EMPTY

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
RTXCMSG  *KS_receivew(MBOX mailbox,
                      TASK task)
```

DESCRIPTION

The KS_receivew function fetches messages from a specified mailbox and returns the pointer to the message. If there are no messages in the mailbox, the requesting task is blocked and removed from the READY List. The task will remain blocked until another task sends a message to the specified mailbox. When the next message is sent to the mailbox, the waiting receiver task will be unblocked and inserted into the READY List. The function will return a pointer to the received message.

With a zero task number in the calling sequence, the first message in the mailbox from any sender is returned. Because the messages are placed in the mailbox in priority order as specified by the sender, they are processed in the same sequence. It is possible, however, to override the strict priority processing. If the receiving task specifies a non-zero task number in the calling sequence, the first

message in the mailbox from that task will be returned.

RETURN VALUE The function returns a pointer to the received message.

EXAMPLE The task is to receive the next available message from its mailbox *MYMAIL*. If there is no mail available, the task is to wait until a message is sent to the mailbox.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cmbox.h"            /* defines MYMAIL  */
#include "rtxstruc.h"        /* defines RTXCMSG */

struct{
   RTXCMSG msghdr;  /* Message header (required) */
   char data[10];       /* start of message body */
} MYMSG;
MYMSG *msg;

/* receive next message from any task */
msg = (MYMSG *)KS_receivew(MYMAIL, (TASK)0);
```

SEE ALSO KS_ack, KS_receive, KS_receivet, KS_send, KS_sendt, KS_sendw

KS_restart_timer

RESTART AN ACTIVE TIMER

CLASS Timer Management

SYNOPSIS

```
KSRC KS_restart_timer(CLKBLK *timer,
                      TICKS period,
                      TICKS cycle_period)
```

DESCRIPTION The purpose of KS_restart_timer is to change the initial or recycle period of an active timer. The function is equivalent to a KS_stop_timer function call followed by a KS_start_timer function. KS_restart_timer combines both operations into a single kernel service. It does not affect the status of a PENDING semaphore associated with the timed event. If the associated semaphore is in a DONE state, however, it is set PENDING.

RETURN VALUE The function returns a value of **RC_GOOD** if the timer was restarted without a problem.

A value of **RC_TIMER_ILLEGAL** is returned if the timer does not have a valid clock block.

EXAMPLE Having previously allocated a timer block, a task starts a one-shot timer with a duration of 250 msec and associates the expiration of the time with semaphore *SWITCH*. During the timer's initial

period, it restarts it as a 1 second cyclic timer with a new initial period of 1500 msec.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "cclock.h"           /* defines CLKTICK */
#include "csema.h"             /* defines SWITCH */

CLKBLK *timer;

/* allocate timer block for task */
timer = KS_alloc_timer();


/* start a one-shot timer of 250 msec */
KS_start_timer(timer,250/CLKTICK,(TICKS)0,SWITCH);

... do some processing

/* then restart the timer as a 1-sec cyclic */
/* timer following a 1.5 second delay */
KS_restart_timer(timer, 1500/CLKTICK,
                        1000/CLKTICK);
```

SEE ALSO KS_alloc_timer, KS_free_timer, KS_start_timer, KS_stop_timer

KS_resume

RESUME A TASK

CLASS Task Management

SYNOPSIS

```
void  KS_resume(TASK task)
```

DESCRIPTION KS_resume clears the suspended state of a task caused by a prior KS_suspend operation. If the resumed task becomes runnable it is inserted into the READY List at a position dependent upon its priority. If the resumed task is of higher priority than the requesting task, a context switch is performed. Otherwise, control is returned to the requesting task.

RETURN VALUE The function returns no value.

EXAMPLE A task suspends the analog input task, AIREADER, performs some operations, and then resumes the analog input task.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"           /* defines AIREADER */


KS_suspend(AIREADER);   /* suspend task AIREADER */

... perform some operations

KS_resume(AIREADER);     /* resume task AIREADER */
```


SEE ALSO KS_suspend

KS_send

SEND A MESSAGE ASYNCHRONOUSLY

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_send(MBOX mailbox,
              RTXCMSG *msghdr,
              PRIORITY priority,
              SEMA sema)
```

DESCRIPTION

KS_send sends a message asynchronously to the specified mailbox. There may or may not be a task waiting to receive the message from the specified mailbox. If there is no waiting receiver task, the message is inserted into the mailbox at a position with respect to the priority given in the kernel service function call.

If there is a receiving task waiting to receive a message, the message is passed directly to the receiver task. The receiver task is then unblocked and, if found to be runnable, is placed into the READY List at a position dependent on its priority.

If the receiving task is of higher priority than the sending task, a task switch is performed.

If the receiving task is of lower priority than the sending task, control is returned to the sending task.

RETURN VALUE The function returns no value.

EXAMPLE Send a message asynchronously at priority 4 to mailbox *MAILBOX3*. The message is in a structure named *mymessage*. Associate the semaphore *GRAFSEMA* with the completion of message processing. After sending the message, perform some other operations and then wait for the completion of processing of the message.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"           /* defines GRAFSEMA */
#include "cmbox.h"           /* defines MAILBOX3 */
#include "rtxstruc.h"        /* defines RTXCMSG  */

struct {
   RTXCMSG msghdr;  /* Message header (required) */
   int command;         /* start of message body */
   char data[10];
} mymessage;

/* send msg to MAILBOX3 at a priority of 4 and */
/* associate semaphore GRAFSEMA with the message */
KS_send(MAILBOX3, &mymessage.msghdr, (PRIORITY)4, GRAFSEMA);
... do some more processing and then wait for
    the event associated with completion of
    message processing

KS_wait(GRAFSEMA);
```

SEE ALSO KS_receive, KS_receivet, KS_receivew, KS_sendt, KS_sendw

KS_sendt

SEND A MESSAGE SYNCHRONOUSLY
WITH TIME LIMITED
WAIT FOR ACKNOWLEDGMENT

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_sendt(MBOX mailbox,
               RTXCMSG *msghdr,
               PRIORITY priority,
               SEMA sema,
               TICKS timeout)
```

DESCRIPTION

KS_sendt sends a message synchronously to the specified mailbox. There may or may not be a task waiting to receive the message from the specified mailbox. If there is no waiting receiver task, the message is inserted into the mailbox at a position with respect to the message priority given in the kernel service function call.

The sending task is removed from the READY List and blocked by a wait on the message semaphore specified in the function call. Simultaneously, a timeout timer is established to limit the duration of the wait to that amount of time specified by the *timeout* argument in the function call. A duration of zero (0) will not cause a timer to be started and is thus equivalent to the kernel service KS_sendw.

If there is a receiving task waiting to receive a message, the message is passed to the receiver. The receiver task is then unblocked and, if found to be runnable, is placed into the READY List at a position dependent on its priority.

The sending task will resume operation when it receives either the acknowledgment that the receiver task has completed processing the message or the expiration of the timeout period occurs. The function returns a value indicative of the form of completion.

RETURN VALUE

The function returns a value of **RC_GOOD** when the message is successfully sent and processed within the specified timeout duration.

If the timeout occurs, the function returns a value of **RC_TIMEOUT**.

EXAMPLE

The task synchronously sends a message located in the structure *mymessage* to the mailbox *MAILBOX3*. The priority of the message is to be 4 and the completion event is associated with semaphore *GRAFSEMA*. A timeout period of 250 msec is to be used for the duration of the waiting period. If the wait for acknowledgment of processing exceeds 250 msec, handle the situation with a special code segment.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"         /* defines GRAFSEMA */
#include "cmbox.h"         /* defines MAILBOX3 */
#include "cclock.h"        /* defines CLKTICK  */
#include "rtxstruc.h"      /* defines RTXCMSG  */

TICKS timeout = 250/CLKTICK;

struct {
   RTXCMSG msghdr;  /* Message header (required) */
   int command;     /* start of message body */
   char data[10];
} mymessage;

/* send message msg synchronously to MAILBOX3 at */
/* priority 4. Associate semaphore GRAFSEMA with */
/* the message. Wait up to 250 ms for message to */
/* be processed */
if (KS_sendt(MAILBOX3, &mymessage.msghdr, (PRIORITY)4, GRAFSEMA,
             timeout) == RC_GOOD)
{
   ... message sent and processed successfully
}
else
{
   ... message not completed within timeout period
}
```

SEE ALSO KS_ack, KS_receive, KS_receivet, KS_receivew, KS_send, KS_sendw

This page is intentionally left blank.

KS_sendw

SEND A MESSAGE SYNCHRONOUSLY,
WAIT FOR ACKNOWLEDGMENT

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_sendw(MBOX mailbox,
               RTXCMSG *msghdr,
               PRIORITY priority,
               SEMA sema)
```

DESCRIPTION

KS_sendw sends a message synchronously to the specified mailbox. There may or may not be a task waiting to receive the message from the specified mailbox. If there is no waiting receiver task, the message is inserted into the mailbox at a position with respect to the priority given in the kernel service function call.

The sending task is removed from the READY List and blocked by a wait on the message semaphore specified in the function call. If there is a receiving task waiting to receive a message, the message is passed to the receiver task. The receiver task is then unblocked and, if found to be runnable, is placed into the READY List at a position dependent on its priority.

The sending task will resume operation when it receives the signal that the receiver task has completed processing the message.

RETURN VALUE The function returns no value.

EXAMPLE The task synchronously sends a message located in the structure *mymessage* to the mailbox *MAILBOX3*. The priority of the message is to be 4, and the completion event is associated with semaphore *GRAFSEMA*. After sending the message, the task waits for the signal that the message has been processed.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"         /* defines GRAFSEMA */
#include "cmbox.h"         /* defines MAILBOX3 */
#include "rtxstruc.h"      /* defines RTXCMSG  */
struct {
   RTXCMSG msghdr;  /* Message header (required) */
   int command;     /* start of message body */
   char data[10];
} mymessage;

/* send message msg synchronously to MAILBOX3 at */
/* priority 4. Associate semaphore GRAFSEMA with */
/* message. Wait for the message to be processed */
KS_sendw(MAILBOX3, &mymessage.msghdr, (PRIORITY)4, GRAFSEMA);
```

SEE ALSO KS_receive, KS_receivet, KS_receivew, KS_send, KS_sendt

KS_signal

SIGNAL A SEMAPHORE

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_signal(SEMA sema)
```

DESCRIPTION

KS_signal sets the state of a specified semaphore to DONE. If the semaphore is currently in a WAIT state, the Event Wait state of the waiting task is removed, and the semaphore is set PENDING. If the waiting task becomes runnable, it is inserted into the READY List at a position dependent on its current priority. A context switch will occur if the task which was waiting on the signaled semaphore is of higher priority than the signaling task.

If the state of the semaphore was either PENDING or DONE, the semaphore is placed in the DONE state, and the current task is resumed following the KS_signal function call.

RETURN VALUE

The function returns a value of **RC_MISSED_EVENT** if the semaphore is already in the DONE state.

EXAMPLE

The task signals semaphore *SWITCH* to indicate that the associated event has occurred.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"       /* defines SWITCH */


KS_signal(SWITCH);       /* signal sema SWITCH */
```

SEE ALSO KS_pend, KS_pendm, KS_signalm, KS_wait, KS_waitm, KS_waitt

KS_signalm

SIGNAL MULTIPLE SEMAPHORES

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
void  KS_signalm(SEMA *semalist)
```

DESCRIPTION The KS_signal functions performs like the KS_signal kernel service except that it signals all semaphores found in a list of semaphores provided as an argument to the function. The list must be null terminated. Its intent is to minimize RTXC kernel calls and context switching when multiple semaphores need to be signaled as one logical operation.

Unlike KS_signal, KS_signalm does not return a value when signaling a semaphore which is already in a DONE state.

RETURN VALUE The function returns no value.

EXAMPLE The task signals semaphores *ISWITCH* and *RESTART* that a particular event has occurred.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"   /* defines ISWITCH, RESTART */

SEMA semalist[] =
{
   ISWITCH,
   RESTART,
   0                    /* null terminated list */
};

KS_signalm(semalist);
```

SEE ALSO KS_pend, KS_pendm, KS_signalm, KS_wait, KS_waitm, KS_waitt

KS_start_timer

START A TIMER

CLASS Timer Management

SYNOPSIS

```
CLKBLK  *KS_start_timer(
                    CLKBLK *timer,
                    TICKS initial_period,
                    TICKS cycle_time,
                    SEMA sema)
```

DESCRIPTION

The KS_start_timer function starts a timer whose handle is given in the argument list to the function. The timer can be cyclic or one-shot. A one-shot timer has an initial_period argument greater than zero (>0) and a cycle_time argument value of zero (0). A cyclic timer will have both the initial_period and cycle_time argument values greater than zero (>0). The duration of the timer's initial_period and the cycle_time period are specified in terms of the system clock ticks.

The timer expiration event is associated with a semaphore as defined in the arguments of the function call. At the time of the function call, the semaphore is forced to a PENDING state so that the task may subsequently call a blocking function such as KS_wait to await the event. After the timer is

inserted into the Active Timer List, the current task is resumed.

A NULL pointer can be passed in place of the CLKBLK pointer and the function will automatically assign the timer block and return a pointer to the timer. If no timer blocks are available, the function returns a NULL pointer and the task will have to deal with that situation with special code.

Two special features of the KS_start_timer function are as follows. If the function is called with an initial_period of zero (0) and a cycle_time greater than zero (>0), the associated semaphore will be signaled and a cyclic timer will be started. When KS_start_timer is called with both the initial_period and cycle_time equal to zero (0), the only action taken is that the associated semaphore will be signaled.

RETURN VALUE

The function returns the pointer to the timer block used for the timer.

The function returns a NULL pointer if an attempt was made to do an automatic allocation of a timer block and there were none available.

EXAMPLE

A task wants to start a timer using a previously allocated timer block *timer1*. The timer is to have an initial period of 150 msec and a cyclic period of 100 msec. The time expiration event is associated with semaphore *SEMA6*. After starting the timer, the task waits for the timer to expire.

After the first timer's initial period expires, a second timer having only an initial period of 150 msec is also started but the timer block is to be automatically allocated. The second timer is associated with semaphore *SEMA7*. After the second timer is started, the task is to wait for either timer to expire.

```
#include "rtxcapi.h"     /* RTXC KS prototypes */
#include "cclock.h"      /* defines CLKTICK */
#include "csema.h"       /* defines SEMA6, SEMA7 */

SEMA semalist[] =
{
   SEMA6, SEMA7,
   0                     /* null terminated list */
};
SEMA sema;
CLKBLK *timer1, *timer2;

timer1 = KS_alloc_timer();

/* start timer with initial period of 150 ms and */
/* cyclic period of 100 ms. */
KS_start_timer(timer1, 150/CLKTICK, 100/CLKTICK, SEMA6);

KS_wait(SEMA6);     /* wait for timer to expire */

... Do some more processing, then

/* start one shot timer with duration of 150 ms */
/* have system automatically allocate timer block*/
timer2 = KS_start_timer((CLKBLK *)0, 150/CLKTICK, (TICKS)0, SEMA7);
if (timer2 == (CLKBLK *(0)))
{
   ... No timer blocks available. Deal with it here
}
else
{
   sema = KS_waitm(semalist); /* wait for either */
                              /* timer to expire */
}
```


SEE ALSO KS_alloc_timer, KS_restart_timer, KS_stop_timer

KS_stop_timer

STOP AN ACTIVE TIMER

CLASS Timer Management

SYNOPSIS

```
KSRC  KS_stop_timer(CLKBLK *timer)
```

DESCRIPTION The KS_stop_timer service function stops the specified timer, the pointer to which is provided as the function argument, and removes it from the list of active timers.

NOTE: A task may stop only those timers which it has initiated via a prior KS_start_timer() or KS_restart_timer().

RETURN VALUE If the timer was active when stopped, the function returns a value of **RC_GOOD**.

If timer was inactive, the function returns a value of **RC_TIMER_INACTIVE**.

If the task attempts to stop a timer it does not own, the function returns a value of **RC_TIMER_ILLEGAL**.

EXAMPLE A task allocates a timer block and stores the handle to it in pointer *p*. If there is a timer block available, the task needs to wait no longer than 250 msec for

the occurrence of either of two switch closure events associated with semaphores *SWITCH1* and *SWITCH2*. After starting a 250 msec one-shot timer, the task waits for the occurrence of either event or the expiration of the timer. The timer expiration is associated with semaphore *WATCHDOG*. If the task continues as a result of a switch closure, the task is to stop the one-shot timer and free it.

If there are no timer blocks available when attempting to assign pointer *p*, the task must execute special code to deal with the situation.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"
#include "cclock.h"
#include "csema.h"

CLKBLK *p;
TICKS period = 0;          /* one-shot timer */
TICKS initial = 250/CLKTICK;

SEMA semalist[] = {
   WATCHDOG,
   SWITCH1,
   SWITCH2,
   0                      /* list terminator */
}

if ((p = KS_alloc_timer()) != (CLKBLK)0)
{
   KS_start_timer(p, initial, period, WATCHDOG);

   sema = KS_waitm(semalist); /* wait for switch */
   if (sema != WATCHDOG)
      KS_stop_timer(p);       /* stop timer */

   ... continue processing
}
else
{
   ... no timer available. Deal with it here
}
```

SEE ALSO KS_alloc_timer, KS_restart_timer, KS_start_timer

This page is intentionally left blank.

KS_suspend

SUSPEND A TASK

CLASS Task Management

SYNOPSIS

```
void  KS_suspend(TASK task)
```

DESCRIPTION The KS_suspend directive causes the specified task to be placed into a suspended state and removed from the READY List. The suspended state will remain in force until it is removed by a KS_resume or KS_execute kernel service function invoked by another task. A task may suspend itself. An argument value of 0 indicates the SELF task.

RETURN VALUE The function returns no value.

EXAMPLE The current task suspends another task, *LKDETECT*, and then suspend itself.

```
#include "rtxcapi.h"      /* RTXC KS prototypes */
#include "ctask.h"       /* defines LKDETECT */

KS_suspend(LKDETECT);    /* suspend LKDETECT task */

KS_suspend(TASK(0));     /* suspend self */
```

SEE ALSO KS_resume

This page is intentionally left blank.

KS_terminate

TERMINATE A TASK

CLASS Task Management

SYNOPSIS

```
void  KS_terminate(TASK task)
```

DESCRIPTION

KS_terminate stops a task's operation by removing the task from the READY List if it is runnable and by setting its status to INACTIVE. A task number of zero (0) indicates self-termination. This is the normal mode of use for this kernel service. While it is possible to terminate another task, such usage should only be done under circumstances where the terminator knows that the act will not jeopardize system integrity.

In all cases following self-termination, the next highest priority task in a runnable state will execute next. If a task has an active timeout timer, it is stopped and removed from the list of active timers. If the task is a waiter on some kernel object, it will be removed from that object's list of waiters. If the task's Task Control Block was dynamically allocated, the TCB is returned to the Free TCB Pool.

WARNING: Other than the items mentioned above, tasks that are currently using, or have allocated,

kernel objects are not "cleaned up" by the termination process. It is the programmer's responsibility to ensure that all such system elements are released to the system prior to the act of termination.

RETURN VALUE The function returns no value.

EXAMPLE The current task terminates another task, *TASKBX*, and then terminates itself.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "ctask.h"            /* defines TASKBX */
#defines SELFTASK (TASK(0)) /* defines this task */


KS_terminate(TASKBX);   /* terminate task TASKBX */


KS_terminate(SELFTASK);  /* now terminate self */
```

SEE ALSO KS_execute

KS_unblock

UNBLOCK A RANGE OF TASKS

CLASS Task Management

SYNOPSIS

```
void  KS_unblock(TASK start,
                 TASK end)
```

DESCRIPTION The KS_unblock directive is the opposite of the KS_block kernel service function. It may be used to enable the operation of one or more tasks, the range of which is specified by the starting and ending task numbers in the function arguments. The range of tasks to be unblocked begins with the specified start task and includes the specified end task. If the specified end task of the range is the current task (end task = 0), the unblocking action will range from the start task up to, but not including the current task.

RETURN VALUE The function returns no value.

EXAMPLE The current task unblocks tasks 5 through 10 inclusively.

```
#include "rtxcapi.h"       /* RTXC KS prototypes */

KS_unblock(5,10);/* remove blocks for tasks 5-10 */
```


SEE ALSO KS_block

KS_unlock

RELEASE LOGICAL RESOURCE

CLASS Resource Management

SYNOPSIS

```
KSRC  KS_unlock(RESOURCE resource)
```

DESCRIPTION

The KS_unlock kernel service is the opposite of the KS_lock function. KS_unlock releases a logical resource previously locked by the requesting task. Only the task which locked the resource, i.e., the resource "owner", may unlock that resource. Unlocking a resource which is not currently owned causes no change in the state of the resource and a value of **RC_BUSY** will be returned to the caller.

Normally, locks and unlocks of a resource will occur in pairs. That is, for each KS_lock of a specific resource, there will be a corresponding KS_unlock of that same resource by the locking task. However, RTXC supports nested locks of a resource by the same task. Nesting occurs when a resource owner locks the resource again, be it deliberately or inadvertently. When unnesting, the owner task must issue the same number of unlocks as there were locks in the nest. A return value of **RC_NESTED** will be returned until the resource is no longer nested. Then, **RC_GOOD** will be returned for the final unlock.

RETURN VALUE

The function returns **RC_GOOD** when the resource is unlocked and not nested.

A value of **RC_NESTED** is returned if the calling task has not issued as many unlocks as locks.

If the resource is owned by another task, a value of **RC_BUSY** is returned to the calling task.

EXAMPLE

The current task needs to update a resident database, and it must be done without other tasks preempting the operation. Thus, exclusive access to the database is necessary during the update operation. The database is associated with resource *DATABASE*. After performing the update, the task will permit other tasks to access the database.

```
#include "rtxcapi.h"     /* RTXC KS prototypes */
#include "cres.h"        /* defines DATABASE */


KS_lockw(DATABASE);     /* grab resource */
update_db();           /* update shared database */
while(KS_unlock(DATABASE) == RC_NESTED)
       /* release resource */
```

SEE ALSO

KS_lock, KS_lockt, KS_lockw

KS_user

USER DEFINED KERNEL SERVICE

CLASS Special

SYNOPSIS

```
int  KS_user(int (*func) (void *),
             void *arg)
```

DESCRIPTION

The user may execute the specified function, func, as if it were an RTXC kernel service function. This basically defines the function to be indivisible with respect to preemption. Interrupts are permitted and serviced during execution of the function.

The KS_user calling sequence requires a pointer to the function, func, and a pointer to an arbitrary structure, arg, which will be passed to function, func, when invoked. The return value from the specified function, func, will be returned to the caller as the value of the kernel service, KS_user.

RETURN VALUE

The KS_user function returns the return value of the specified function.

EXAMPLE

The task wants to call a function, *foobar*, so that it can execute as though it were a kernel service. Arguments to the function are found in the structure, *args*, the pointer to which is passed in the calling arguments to the function.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */

int status;
struct fooarg
{
   int opcode;
   int val;
} args;


extern int foobar(struct fooarg *);


/* execute function foobar as a KS */
status = KS_user(foobar, &args);
```

KS_wait

WAIT ON EVENT

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_wait(SEMA sema)
```

DESCRIPTION

The KS_wait function is a fundamental function in RTXC. It is used to block a task for a specified event to occur. The event must be associated with the given semaphore. If the semaphore is found to be in a PENDING state, the task is placed into an Event Wait state and removed from the READY List. The semaphore state is changed to WAITING.

If the semaphore is in a DONE state, no wait occurs nor is the task blocked. Instead, the task resumes immediately returning a code indicative of the success of the wait.

The state of the given semaphore should be either in a PENDING or DONE state when KS_wait is called. If it is already in a WAITING state, the function returns immediately with a value indicating the conflicting semaphore usage. In this conflict situation the function does not change the semaphore state. It will be the responsibility of the user to resolve the conflict.

RETURN VALUE

The function returns a value of **RC_GOOD** if the wait was successful.

If the semaphore was already in a WAITING state, the function returns a value of **RC_WAIT_CONFLICT**.

EXAMPLE

The current task needs to synchronize its operation with the occurrence of a keypad character being pressed. The event is associated with semaphore *KEYPAD*.

```
#include "rtxcapi.h"   /* RTXC KS prototypes */
#include "csema.h"   /* defines KEYPAD */


KS_wait(KEYPAD);     /* wait for KEYPAD hit sema */
```

SEE ALSO

KS_pend, KS_pendm, KS_signal, KS_signalm, KS_waitm, KS_waitt

KS_waitm

WAIT ON MULTIPLE EVENTS

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
SEMA  KS_waitm(SEMA *semalist)
```

DESCRIPTION The KS_waitm service performs the same function as the KS_wait directive, except that it uses a list of semaphores associated with the various events. The states of the listed semaphores must follow the same rules as for KS_wait. The KS_waitm function operates as a logical OR, in that the occurrence of an event associated with any one of the semaphores in the list will cause resumption of the waiting task.

RETURN VALUE The function returns the identifier of the semaphore associated with the event which occurred.

NOTE: In the situation where multiple events simultaneously occur, the function returns the semaphore number of the first event serviced. The semaphores associated with the other events which occurred will be in a DONE state. Each subsequent call to the KS_waitm service will immediately return the identity of the next semaphore in the list which is in a DONE state. In this manner all events will be correctly processed.

EXAMPLE

The current task needs to know when any of three events occurs. Two of the events, *SWITCH1* and *SWITCH2*, are associated with switch closures while the third is associated with a timer, *TIMERA*. When any one happens, the task performs a code segment specific to that event.

```
#include "rtxcapi.h" /* RTXC KS prototypes */
#include "csema.h"   /* defines SWITCH1, SWITCH2,
                                      TIMERA */
SEMA cause;
SEMA semalist[] =
{
   SWITCH1,
   SWITCH2,
   TIMERA,
   0             /* null terminated list */
};
for (;;)
{
   /* wait for any of 3 events */
   cause = KS_waitm(semalist);
   switch(cause)
   {
      case SWITCH1:
         ... process SWITCH1 event...
         break;

      case SWITCH2:
         ... process SWITCH2 event...
         break;

      case TIMERA:
         ... process TIMERA event...
         break;

   }  /* end of switch */
}  /* end of forever */
```

SEE ALSO

KS_wait, KS_signal

KS_waitt

TIME LIMITED WAIT ON EVENT

CLASS Intertask Communication and Synchronization

SYNOPSIS

```
KSRC  KS_waitt(SEMA sema,
               TICKS timeout)
```

DESCRIPTION

The KS_waitt is used to block a task for a limited period of time while waiting for a specified event to occur. The event must be associated with the given semaphore. The state of the given semaphore should be either in a PENDING or DONE state when KS_waitt is called.

If the semaphore is found to be in a PENDING state, the task is placed into an Event Wait state and removed from the READY List. The semaphore state is changed to WAITING. At the same time, a timeout timer is started with a period defined by the calling argument, *timeout*.

If the semaphore is in a DONE state at the time of the function call, no wait occurs nor is the task blocked. Instead, the task resumes immediately.

Either the occurrence of the timeout or the event will cause the requesting task to resume. The

function returns a value indicative of the cause of the task's resumption.

The state of the given semaphore should be either in a PENDING or DONE state when KS_wait is called. If it is already in a WAITING state, the function returns immediately with a value indicating the conflicting semaphore usage. In this conflict situation the function does not change the semaphore state. It will be the responsibility of the user to resolve the conflict.

RETURN VALUE

The function returns a value of **RC_GOOD** if the expected event occurs within the time of the timeout duration.

If a timeout occurs, the function returns a value of **RC_TIMEOUT**.

If the semaphore is already in a WAITING state at the time of the function call, the function returns a value of **RC_WAIT_CONFLICT**.

EXAMPLE

The current task needs to wait for a keypad character to be pressed, but it can't wait for more than 100 msec as it has other jobs to do. It uses the KS_waitt kernel service to perform a time limited wait on the event, *KEYPAD*.


```
#include "rtxcapi.h"       /* RTXC KS prototypes */
#include "csema.h"         /* defines KEYPAD */
#include "cclock.h"        /* defines CLKTICK */


/* wait 100 msec for KEYPAD to be hit */
if(KS_waitt(KEYPAD, 100/CLKTICK) == RC_GOOD)
{
   ... keypad was hit, process the event
}
else
{
   ... keypad not hit and timeout happened
       or no timers were available, or
       a Wait Conflict exists
}
```

SEE ALSO KS_pend, KS_signal, KS_wait

This page is intentionally left blank.

KS_yield

YIELD CPU CONTROL

CLASS Task Management

SYNOPSIS

```
KSRC  KS_yield(void)
```

DESCRIPTION

The KS_yield function permits a voluntary release of control by a task without violating the policy of the highest priority runnable task being the current task. This service is of use only when there are two or more tasks operating at the same priority. When KS_yield is invoked and there is at least one more task in the Ready List at the same priority, the calling task is removed from the READY List and reinserted into the READY List immediately following the last runnable task having the same priority. The task remains unblocked.

Yielding when there is no other task at the same priority causes no change in the READY List, and the calling task is immediately resumed.

RETURN VALUE

If there is another task at the same priority, the function yields CPU control to it and returns a value of **RC_GOOD**. If no yield can occur, the function returns a value of **RC_NO_YIELD**.

EXAMPLE

The current task has reached a point in its

processing where it will yield to another task if that task is running at the same priority as the current task. If not, this kernel service operates without changing the READY List.

```
#include "rtxcapi.h"     /* RTXC KS prototypes */


/* yield to next READY task at same priority */
if (KS_yield() == RC_NO_YIELD)
{
   ... no READY task exists at same priority level
       take whatever action is required
}
/* otherwise, the yield was successful */
```

SEE ALSO KS_defpriority

SECTION 7
DEVICE DRIVERS
Table of Contents

SECTION 7

DEVICE DRIVERS

INTRODUCTION

Device drivers are special programs which provide an organized software interface between a physical device and the application programs which use it. The intent of a device driver is to mask the specifics of the device's hardware peculiarities from the application software. By doing so, the application code need only conform to the protocol which the device driver expects in order to perform a function. Each device driver may have a unique protocol related to its function. For instance, the interface between a task and a disk driver would be different than between a task and an analog-to-digital converter.

In order to accommodate the wide range of devices which are found in real time systems, device drivers in RTXC are structured as tasks. As a task, a device driver has the most flexible environment with

complete access to system resources via the executive service requests. In some cases, a device driver may even require more than one task to perform its required functions.

INTERRUPT HANDLING BASICS

A device attached to a computer often makes use of an interrupt to indicate completion of a function or some event associated with its operation. Interrupts are efficient in a multitasking design in that they permit the CPU to continue doing useful work without wasting time waiting for a device to perform some function.

With the use of interrupts, the device performs its designated function and indicates that fact by asserting an interrupt request. When dealing with devices which generate interrupts, the system must have a way of acknowledging an interrupt request, identifying the requesting device, breaking the program flow, and then servicing the device to remove the interrupt.

Much of what transpires in interrupt handling is actually done in hardware. More is done in some cases, less in others. It depends on the system. It is quite common, for example, for processor hardware to be able to recognize an interrupt request, acknowledge it, identify the device, and to vector CPU control directly to a service routine to perform the functions necessary to remove the interrupt.

After handling the interrupt, the system is allowed to continue as though nothing had happened. Indeed, all interrupt handling must be totally transparent to normal system operations.

Interrupts and Preemption

Interrupts are what their name implies, a breaking of the normal flow of system operation. Interrupts require that they be serviced before the system can return to its normal operations. Thus, they are given a processing priority above that of normal operations. This priority leads to the necessity of creating a method for dealing with interrupts and their consequences to the flow of system operations.

Interrupt handling can occur at any time and usually requires some part of the processor's context in order to service the interrupt. One fundamental requirement in interrupt handling, therefore, is that the processor state must be saved to the degree that it can be restored in the future without loss of context.

A device causing an interrupt needs servicing in order to remove the source of the interrupt. But there may be some detail of the device's operation which requires that it be performed as quickly as possible at the task level. A direct consequence of the interrupt is the potential to suspend current operations while a high priority task performs the device's needed operations. In other words, an interrupt may require the system to switch from the task being performed at the time of the interrupt to another task of higher priority. This is the fundamental idea behind the policy of preemption.

Stacks

RTXC supports two types of stacks, task stacks and a system stack. While they are used in identical

manners, it is important to understand why the two exist and when they are employed.

Task Stack

Every task has a stack which contains its context. Function arguments, automatic variables, and preempted processor states all share the task's stack space. When the task is granted control by RTXC, it is the task's stack which is the system's active stack. The active stack has a pointer to its top somewhere in the processor context. This stack pointer is normally found in the CPU register set. If not in hardware, it is in a software construct known to and managed by the C compiler and serves the same purpose. Either can be referred to as the stack pointer.

When the processor's stack pointer points to the top of the task's stack, an interrupt will cause the processor state to be stored on the task's stack. As it was the task which had CPU control when the interrupt occurred, the stacked processor state is actually the task's state. Thus, the task's state is preserved.

System Stack

If an ISR runs with interrupts enabled, permitting higher priority interrupts to be serviced, it is not desirable to store a second processor state on the task's stack. Instead, RTXC switches to a System Stack whenever processing is active for an interrupt which occurred at the task level. Any new interrupts will have their contexts saved on the System Stack instead of the task's stack. This kind of stack usage minimizes the amount of space required for each task's stack, a desirable feature when working with

systems which have a tight RAM budget.

INTERRUPT SERVICE ROUTINES

In RTXC, a device which employs interrupts for event notification is composed of two parts: the driver task, and an (ISR). The purpose of the task is to initiate device operations and to deal with data flow requests from other tasks. The purpose of the ISR is to service the device when it causes an interrupt. Usually the task and its related ISR are very closely coupled.

The general philosophy of RTXC drivers is to minimize the time spent in the interrupt service routine and let the task portion handle the real work of the driver. This design concept places few functional restrictions on the design of both the task and the ISR.

Basic ISR Flow

RTXC interrupt servicing follows a definite procedure. Section 3 described how ISR processing begins in an assembly language routine, the prologue, and then branches to a C routine for the specific device servicing. The C routine returns to a second assembly language routine, the epilogue, from which normal operations are resumed.

Prologue

The first level assembly language routine should always do the following steps:

1. Save the processor state.

2. Handle any requirement for switching stacks.

3. Call a user written C routine to service the interrupt.

Saving the processor state in Step 1 may be performed by hardware on some processors. If not, it must be done by software in the early stages of interrupt processing. The processor state is always stored on the currently active stack.

In Step 2, the ISR prologue must determine if it is necessary to switch stacks. When operating normally, the Current Task's stack is the active stack. An interrupt always forces a stack change to the System Stack if the current stack is a task stack. The prologue must establish the stack pointer for whichever stack is appropriate for the ISR's operation. For example, a call by a task for a kernel service will force a change to the System Stack. An interruption of an ISR will not cause a stack switch because the first interrupt caused a switch to the System Stack.

After saving the complete register set on the stack, and, if necessary, changing the stack pointer to the System Stack, the pointer to the top-of-stack of the interrupted process is pushed on the System Stack. This copy of the stack pointer, called a frame pointer, is used as the argument for the user-written device servicing routine.

Device Servicing

The device servicing routine portion of the ISR in Step 3 is normally written in C. Ideally, the majority of it should be performed with interrupts enabled to allow other exceptions to be serviced. Likewise, the

clearing of the hardware interrupt registers and devices should be performed as early as possible in the logic to allow for other interrupts at the same level to function. The routine then processes the interrupt as needed.

Most ISRs are associated with a single event and consequently need only a single semaphore associated with the event. Usually the signaling of the event is combined atomically into the exit logic of *KS_ISRexit()* (see below). However, some ISRs are associated with multiple events and multiple semaphores and it may be necessary to signal more than one during the course of the device servicing routine. RTXC ISRs are permitted to do so by using the special service *KS_ISRsignal()*. One semaphore can be signaled for each call to *KS_ISRsignal()*, the identifier of the semaphore being the only argument to the function.

When the device servicing routine of the ISR reaches a point of completion, it needs to inform RTXC of that fact. It does so by calling the Interrupt Service Routine Exit function, *KS_ISRexit()*, passing it the frame pointer of the interrupted process. One additional argument, an identifier of the semaphore associated with the interrupt, may be employed to cause that semaphore to be signaled.

One goal in the RTXC design is to be as hardware independent as possible. The primary areas that are hardware dependent are those concerned with interrupt service and the register context switch.

Epilogue

The last element in the basic flow of an ISR is the epilogue. It is written in assembly language routine and receives the frame pointer of the highest priority task in the Ready List as returned from *KS_ISRexit()*. The function does the following operations:

1. Switches to the appropriate stack of the process to be resumed.

2. Restores the context of the highest priority task in the READY List.

3. Resumes normal system operations.

COMMON ISR EXIT FUNCTION

The RTXC Interrupt Service Routine Exit Function, *KS_ISRexit()*, exists to perform those processor independent functions which all interrupt handling must do, including signaling one or more semaphores, determining interrupt nesting level, and performing a context switch if necessary.

The ISR device servicing routine calling *KS_ISRexit()* may pass an argument indicating that the semaphore associated with the interrupt is to be signaled. If so, the semaphore handle is appended to the Signal List. However, actual signaling of the semaphores in the Signal List does not occur until *KS_ISRexit()* determines that there is no other interrupt processing pending. This must be done since it is possible for interrupts to be nested (see below). Returning to the point of the original exception cannot occur until that interrupt has been completely serviced. As each ISR completes and calls *KS_ISRexit()*, the exit logic checks to see if the current interrupt occurred in an ISR or a task. If an ISR was interrupted, *KS_ISRexit()* ends and causes resumption of the interrupted ISR.

When it is determined that the ISR calling *KS_ISRexit()* is the one which interrupted a task, the exit logic causes all of the semaphores in the Signal List to be signaled. As each listed semaphore is signaled, any task waiting on the associated event has its WAIT state removed. If the task is found to be completely unblocked, it is made runnable and inserted into the Ready List. When all semaphores in

the list have been similarly processed, the highest priority runnable task becomes the current task and is resumed at the point indicated by its stored context. Thus, nested interrupts and re-entrant ISRs are handled cleanly, quickly, and automatically by RTXC.

NESTED INTERRUPTS

One common problem area concerning interrupts is nesting of interrupts (different levels) and reentrancy on the same hardware interrupt level. These are handled by RTXC in different manners according to the nature of the particular processor. It may be by simply maintaining an internal interrupt counter, or, by a more complex method related to processor/interrupt priority levels. Regardless of the technique, it is in the initial interrupt service for each interrupt that it is implemented. The method for implementing this feature is supplied as part of the RTXC distribution and is peculiar to the processor and C compiler being employed.

INTERRUPT HANDLING CAVEATS

An important factor in interrupt processing throughput is the length of time interrupts are disabled during interrupt handlers. On all interrupts, interrupts must be disabled long enough to execute the code necessary to set up the interrupt nesting control mechanism. After performing that logic, interrupts may be enabled. This logic is very processor dependent and is supplied as part of the standard RTXC Distribution.

RTXC imposes rules on the construction of interrupt service routines that the user must observe. Perhaps the most important of these rules is that **an ISR should make no calls to RTXC kernel services other than to those in the special ISR class.** (See *KS_ISRalloc()*, *KS_ISRexit()*, *KS_ISRsignal()*, and *KS_ISRtick()*) Such misuse of a kernel service by an ISR can cause corruption of the System Stack with indeterminate consequences. If it is necessary to call a kernel service, get out of the ISR as quickly as possible and into the task level part of the driver where complete access to RTXC executive services is available.

COPROCESSOR CONTEXT

The coprocessor (FPU) context typically involves several floating point registers with a few bytes of status and control information. For instance, on the iAPX 8087/80287 coprocessors, the context is 94 bytes. From the size of the context, it is impractical to swap the FPU context along with CPU context on every task switch. Real-time systems, typically have only a small number of the tasks in the entire suite of tasks which require floating point support. To minimize context switch time, RTXC tasks are categorized at RTXCgen time as having a FPU requirement or not. Context switch time is then optimized by performing the FPU context swap only on demand. Demand swapping means that the context will be swapped only when granting control to a task which uses the FPU that is different than the last task to use the FPU. These combined techniques provide efficient sharing of a math coprocessor between multiple tasks.

SECTION 8

RTXCbug DEBUGGING TOOL

Table of Contents

SECTION 8

RTXCbug DEBUGGING TOOL

INTRODUCTION TO RTXCbug

is the system level debugging tool for RTXC. Its purpose is to provide snapshots of RTXC internal data structures as well as perform some limited task control. RTXCbug operates as a task and is usually set up as the highest priority task. Whenever RTXCbug runs, it freezes the rest of the system, thereby permitting coherent snapshots of RTXC system data components. RTXCbug is not intended as a replacement for other debugging tools but is meant to assist the user in tuning the performance of or checking out problems within the RTXC environment.

RTXCbug uses the input and output ports of a user-defined console device. The console device is usually a keyboard and a CRT display. Commands are given to RTXCbug via the console input port, and output from RTXCbug is displayed on the console output device. These devices may be reassigned during a system generation procedure.

Because RTXCbug usually operates as the highest priority task in the system, all other tasks are blocked except for the console input and output drivers. All interrupts are serviced as usual while RTXCbug is active, but lower priority tasks are not dispatched. Active timers are not adjusted while RTXCbug is active, as that could cause the timer to behave improperly.

ENTRY INTO RTXCbug

RTXCbug is designed to be entered through two different mechanisms.

1. The user entering an exclamation point (**!**) on the console input device.

2. By a task calling a special function within RTXCbug.

Once RTXCbug is entered, the version of RTXC and the main menu are displayed as:

```
** RTXCbug - RTXC x.xx   mm/dd/yy

  K - RTXC
  G - Go to Multitasking Mode
  X - Exit RTXCbug
```

where x.xx is the version number and mm/dd/yy is the date of that version. The menu is followed by the RTXCbug command prompt:

```
RTXCbug>
```

Selecting "K" causes the following prompt to be displayed.

```
RTXCbug - RTXC Objects>
```

From the RTXC command prompt, you may enter any of the primary RTXCbug commands. All

commands must be terminated by an **Enter** (<cr>) key.

RTXCbug COMMANDS

Whenever you wish to review the RTXCbug command options, you may display the RTXCbug Command Menu by entering an "**H**" (or "**h**") followed by an **Enter** (<cr>) key in response to the RTXCbug prompt. The Command Menu appears as:

```
T - Tasks
M - Mailboxes
P - Partitions
Q - Queues
R - Resources
S - Semaphores
C - Clock/Timers
K - Stack Limits
Z - Zero Partition/Queue/Resource Statistics
$ - Enter Task Manager Mode
# - Task Registers
G - Go to Multitasking Mode
H - Help
U - Return to Main Menu
X - Exit RTXCbug
```

TASKS

Selection of this option produces a snapshot of the state of all the tasks in the system as shown below. The snapshot contains four columns of information:

- Task Number
- Task Name
- Task Priority
- Task State

Task Number

The task number is the numerical equivalent of the task's name.

Task Name

The task name shows the symbol associated with the task number as defined during the configuration process.

Task Priority

The priority column shows the task's current priority.

Task State

The task state column shows the current state of the task and some related information. For instance, if a task is blocked, the state column shows the cause of the blockage. The possible state conditions are:

- **INACTIVE** - The task has not been executed.
- **READY** - The task is active and is in the READY List. A minus sign in front (-**READY**) indicates that the task is Ready but is being blocked by RTXCbug.

- **DELAY** - The task is delayed for a period of time. The amount of time remaining in the delay period is shown adjacent to the task state.

- **SUSPENDED** - The task is suspended.

- **Semaphore** - The task is waiting on one or more events using semaphore whose name(s) appear(s) adjacently. If the event is associated with a timeout, the amount of time remaining is shown adjacent to the semaphore name.

- **QueueEmpty** - The task is waiting because a queue is Empty. The name of the queue is shown adjacent to the task state. If the task is in a time limited wait, the amount of time remaining in the timeout period is shown adjacent to the queue name.

- **QueueFull** - The task is waiting because a queue is Full. The name of the queue is shown adjacent to the task state. If the task is in a time limited wait, the amount of time remaining in the timeout period is shown adjacent to the queue name.

- **Mailbox** - The task is waiting because a mailbox is empty. The name of the mailbox is shown adjacent to the task state. If the task is in a time limited wait, the amount of time remaining in the timeout period is shown adjacent to the mailbox name.

- **Resource** - The task is waiting because a resource is Locked. The name of the resource is shown adjacent to the task state. If the task is in a time limited wait, the amount of time remaining in the timeout period is shown adjacent to the resource name.

- **Partition** - The task is waiting because a Partition is empty. The name of the partition is shown adjacent to the task state. If the task is in a time limited wait, the amount of time remaining in the timeout period is shown adjacent to the partition name.

A sample Task snapshot is shown below.

```
** Task Snapshot **
  #   Name    Priority  State
  1  RTXCBUG     1      READY
  2  PRTSC       9      -READY
  3  CONODRV     6      READY
  4  CONIDRV     5      READY
  5  HISTASK    12      INACTIVE
  6  COMODRV    10      QueueEmpty COMOQ
  7  CAL         8      Semaphore ONESEC <500 ms>
  8  DINP        8      Semaphore DINTSEMA SDINSEMA
```

In the example, tasks 1, 3, and 4 are active and in the READY List reflecting RTXCbug's use of the console input driver (CONIDRV) and the console output driver (CONODRV). Task 2 is not used by RTXCbug and, while ready to run, is blocked by RTXCbug. The minus sign prefix on READY indicates the task is blocked by RTXCbug.

Task 5 has not been started and is idle. Tasks 6, 7, and 8 are waiting for certain events to occur. Task 6 waits for something to be put into the COM Output Queue, COMOQ. Task 7 is waiting for a timer to expire which has another 500 milliseconds to run. The timed event is associated with semaphore ONESEC. Task 8 is waiting for either one of two events to occur. One is associated with the semaphore DINTSEMA and the other with semaphore SDINSEMA.

QUEUES

This command produces a snapshot of the queues in the system as shown below. Seven columns are used in the snapshot. The first two, queue number and name, are self-explanatory.

Current Size
Maximum Depth

The columns for Current/Depth show the current sizes of the queues and their maximum depths.

Worst Case Usage

The column entitled "Worst" shows the worst case usage, i.e., largest current size, of the queue.

Total Usage

The "Count" column shows the total number of entries that have been put (enqueued) into the queue.

Waiters

The "Waiters" column shows the name of the tasks, if any, which are waiting on the queue.

The Queue snapshot appears as:

```
** Queue Snapshot **
  #   Name  Current/Depth   Worst   Count   Waiters
  1 CONIQ         0/   16       1      19
  2 CONOQ       108/ 1024     546    3413
  3 COMOQ         0/  128       0       0   COMODRV
```

If there are condition semaphores defined for a given queue, they are shown adjacent to the column for Waiter tasks. The code for the queue condition associated with the semaphore is also displayed next

to the semaphore name. The queue condition codes are as follows:

- **<E>** Empty
- **<F>** Full
- **<NE>** Not Empty
- **<NF>** Not Full

SEMAPHORES — The four-column snapshot of the RTXC semaphores is shown below. The first two columns give the semaphore number and its symbolic name.

State — Column three, labeled "State" shows one of the three possible states in which a semaphore can exist:

- **PEND** - Pending (Not yet happened and no waiter)
- **WAIT** - Waiting (Not yet happened and a task is waiting for it)
- **DONE** - Done (Event has happened)

Waiters — The last column shows the name of the tasks waiting for the semaphores.

An example of the snapshot appears as:

```
** Semaphore Snapshot **
  #    Name       State    Waiter
  1  PRNSEMA      PEND
  2  PRTSCSEM     DONE
  3  COMISEMA     WAIT     COMIDRV
```

RESOURCES

RTXCbug produces a Resource Snapshot such as that shown below when the R command is entered. Six columns of status information are displayed. The first two, resource number and name, need no explanation.

Lock/Unlock Cycles

The third column shows the total number of times the given resource has been locked and unlocked.

Lock Conflicts

The "Conflicts" column shows the number of times there has been an attempt to lock the resource when it was already locked by another user.

Owner

The name of the task which is currently locked on the resource is shown in the column entitled "Owner".

Waiters

The names of any tasks which are waiting to use the resource are shown in the last column, "Waiters".

The Resource snapshot appears as:

```
** Resource Snapshot **
  #   Name    Count   Conflicts    Owner    Waiters
  1  PRNRES   36742       0
  2  DOSRES       1       1       PRTSCRN   FILMGR
```

MEMORY **P**ARTITIONS

The Memory Partition Snapshot produced by the P command is shown below. Eight columns of information make up the snapshot. The memory partition number and name are the first two columns.

Current Available
Total Available

The next two columns are headed "Avail/Total" and show the available number of blocks and the total number of blocks in the map.

Worst Case Usage

The column titled "Worst" show the worst case usage of blocks in terms of the maximum number of blocks allocated at any one time.

Total Usage

The "Count" column shows the total number of block allocations processed by RTXC.

Block Size

The size of each block in the partition is shown in the column entitled "Bytes".

Waiters

The last column shows the name of any task waiting on the availability of memory.

An example of a Partition snapshot appears below.

```
** Partition Snapshot **
 #  Name   Avail/Total  Worst Count  Bytes  Waiters
 1 PRTSCMAP    3/    4      1     0   2010
 2 AIMAP       0/   20     20   482     64  AINP
```

MAILBOXES

The Mailbox Snapshot produced by the M command is shown below. Five columns of information make up the snapshot. The mailbox number and name are the first two columns.

Current Content

The next column, headed "Current", shows the number of messages currently in the mailbox.

Total Usage

The column labeled "Count" displays the number of messages sent to the mailbox.

Waiters

The last column, "Waiters", shows the name of any task waiting for messages to arrive at the mailbox.

A sample Mailbox Snapshot is shown below.

```
** Mailbox Snapshot **
  #    Name     Current    Count    Waiters
  1  FSRVMBOX         0    31472    FILESRVR
  2  PRNMSG           0     3720    PRNDRV
```

CLOCK/TIMERS

This command produces a display of the Clock Snapshot such as that shown in the example below. The five columns of the snapshot show all the information about each active timer.

Time Remaining

The first column, titled "Time Remaining", shows the amount of time remaining on each active timer in units of milliseconds.

Cyclic Value

The "Cyclic Value" column contains a value if the timer is cyclic in nature. The value shown defines the cyclic period of the timer in milliseconds. A period of 0 msec indicates a one-shot timer.

Task

The "Task" column shows the name of the task waiting for the timer to expire.

Timer Type and Object

The fourth and fifth columns, "Timer Type" and "Object" are associated. Timer Type shows the type of timer being used while the Object column shows the name of the associated object. The permissible types are:

- **Timer** - A general purpose timer. A blank field following the word "Timer" indicates no semaphore is associated with the timer.
- **Delay** - A task delay
- **Partition** - A timeout on allocation from an empty memory partition. The partition name appears adjacently.

- **Semaphore** - A timed wait for an event. The name of the semaphore appears adjacently.

- **QueueEmpty** - A timed wait for data to be put into an empty queue. The name of the queue is shown also.

- **QueueFull** - A timed wait for data to be removed from a full queue. The name of the queue is shown also.

- **Resource** - A timed wait before gaining ownership of a currently locked resource. The name of the resource is also shown.

- **Mailbox** - A timed wait for mail to arrive at an empty mailbox. The name of the mailbox is shown adjacently.

In addition to the columnar information about the timers, there is some general information about the clock. Specifically, its rate in Hertz, its time granularity expressed as a tick interval in msec., and the maximum number of timers in the system are shown also in the snapshot. Due to space limitations, this general clock information is not shown in their exact columnar locations.


```
** Clock Snapshot **

Clock rate is xxxx Hz, Tick interval is xxx ms,
Maximum of xx timers. Tick timer is    37046,
ET is     126 ticks, RTC time is    486

   Time       Cyclic     Task       Timer    Object
Remaining     Value      Name       Type      Name
   500        1000       CAL       Timer    CALSEMA
```

The Tick timer shown in the example above represents the number of Ticks since the system was started. The term ET is the number of Ticks which have elapsed since the last entry into RTXCbug.

STAC**K** LIMITS — This function is intended to assist the user in tuning the use of RAM needed for stack space by tasks as well as by RTXC. The snapshot, as shown in the example below, consists of five columns.

Task Number
Task Name — The first two columns are used to identify the task number and name.

Stack Size — The third column shows the "Size" of the stack, in bytes, as allocated during the system generation procedure.

Used — The fourth column shows how much of that allocated stack has been used during the course of operation.

Spare — The fifth column shows how much of that allocated stack has been unused during the course of operation. (Size = Used + Spare)

The stack snapshot appears as:

```
** Stack Snapshot **

  #    Task     Size    Used    Spare
  1  RTXCBUG     768     610      158
  2  PRTSC       512     124      388
  3  CONODRV     512     250      262

RTXC Kernel      256      68      188
Worst case interrupt nesting = 3
Worst case Signal List Size = 2
```

The snapshot also shows the usage of the RTXC system stack(when supported) under the same columns (Size, Used, and Spare) as for the task stacks. The worst case levels of interrupt nesting and ISR semaphore signalling are also shown.

ZERO QUEUE/ PARTITION/ RESOURCE STATISTICS

This command will cause all of the usage statistics for queues, memory partitions, mailboxes, and resources to be reset. The worst case levels for interrupt nesting depth and ISR semaphore signalling are also reset. No other user input is required.

$ - TASK MANAGER MODE

Task Manager Mode allows the user to do some types of task management operations via the debug console. Selection of this command causes a special prompt to indicate that RTXCbug is in Task Manager Mode. The prompt appears as:

```
$RTXCbug>
```

The Task Manager Mode menu may be displayed by responding to the prompt with an "H" (or "h") followed by an **Enter** (<cr>) key. The Task Manager Mode menu is shown below.

```
S - Suspend
R - Resume
T - Terminate
E - Execute
C - Change task priority
B - Block (-1=All)
U - Unblock (-1=All)
/ - Time slice
H - Help
X - Exit Task Manager Mode
```

Except for the Exit (X) command, all of the commands in the Task Manager Mode require that a task number or name be entered. The task identifier prompt appears as:

```
Task>
```

The user's response to the prompt is a decimal task number or the task's symbolic identifier as defined during the system generation procedure. The entry is terminated by an Enter (<cr>) key.

Suspend Execution of this command causes the specified task to be suspended. The task cannot be restarted until it is resumed by another task or by operator command via RTXCbug.

Resume This command removes the state of suspension on the specified task. If no other blocking condition exists, the task is made ready to run.

Terminate This command causes the specified task to cease operation. All active timers associated with the task are purged.

Execute A task may be started by the selection of this command. The specified task is started at the entry point specified during the system generation procedure.

Change task priority The priority of the specified task is changed by the selection of this command with immediate effect.

Block A task may be blocked by the selection of this command. If the special task identifier of -1 is given, it causes all tasks to be blocked with the exception of RTXCbug and its supporting input and output tasks.

Unblock This command is used to remove the blocking condition set by the RTXCbug Block command on a specific task. The task identifier is entered in the same manner as on the Block command. The special task identifier of -1 also applies to the unblock command.

Time slice This command is used to define the time slice time quantum for a specified task. A time quantum value greater than zero enables time slicing and a value of 0 disables it.

Help This command causes the RTXCbug Command Menu to be displayed.

Exit Task Manager Mode This command causes the Task Manager Mode to terminate and to return to RTXCbug snapshot mode. The standard RTXCbug command prompt is reissued.

- TASK REGISTERS

This command displays the processor register context for a given task. You must enter the desired task number in response to a query from RTXCbug.

```
Task>
```

Enter the task number and terminate the entry by pressing the **Enter** (<cr>) key. RTXCbug will immediately display the register context for the indicated task. The display format of the registers is processor dependent.

GO TO MULTITASKING MODE

When you have finished your session with RTXCbug and you wish to resume normal system operations, select this menu option.

HELP

To display the RTXCbug Command Menu, select this option. The Command Menu appears as:

```
T - Tasks
M - Mailboxes
P - Partitions
Q - Queues
R - Resources
S - Semaphores
C - Clock/Timers
K - Stack Limits
Z - Zero Partition/Queue/Resource Statistics
$ - Enter Task Manager Mode
# - Task Registers
G - Go to Multitasking Mode
H - Help
U - Return to Main Menu
X - Exit RTXCbug
```

RETURN TO MAIN MEN**U**

If you wish to return to the RTXCbug main menu, select this option.

E**X**IT RTXCbug

If you wish to terminate RTXC operations, select this option from the Command Menu. If your RTXCbug terminal is a workstation with an operating system, selecting this option will cause the return to that environment. For a system without an operating system, this option is the same as the "G" option.

SECTION 9
APPLICATION NOTES
Table of Contents

SECTION 9

APPLICATION NOTES

INTERTASK COMMUNICATION USING MESSAGES

Messages allow a very efficient means of passing information between two tasks either synchronously or asynchronously. Rather than attempting to describe some hypothetical use of messages, an example of usage may be more useful and informative. You will find below an example of two tasks which use messages as a means of intertask synchronization and communication. The code is obviously edited to leave only the essentials of message use.

As a brief synopsis of the tasks, *smcsrv()* is a Stepper Motor Controller SeRVer handling multiple steppers. It receives command messages from other tasks to make a given stepper motor move to a given position. Thus, *smcsrv()* is the receiver task.

The task, *motor1()*, is a sender task which sends a command message to *smcsrv()*.

In the example, the breakdown by line number range is as follows:

001-024	Commentary and file includes.
025-065	Message structures. These are for example and their content does not reflect a mandatory method for the organization of a structure to be used for an RTXC message.
066-072	More declarations.
073-110	Task *smcsrv()*.
113-164	Task *motor1()*.

```
1   /*************************************************************************/
2   /*  SMCSRV.C                                                             */
3   /*                                                                       */
4   /*  This is an excerpt from a stepper motor driver system. There are a few   */
5   /*  undefined symbols such as IDLE etc. but they are unimportant to the      */
6   /*  example. This is a task which receives a stepper motor command and then  */
7   /*  processes that command. How it does it is not important to the example.  */
8   /*  This is primarily intended as an example of how messages are used.       */
9   /*************************************************************************/
10
11  #include "rtxcapi.h"
12  #include "csema.h"
13  #include "ctask.h"
14  #include "cclock.h"
15  #include "cmbox.h"
16
17      /************************/
```

```
18       /*       STRUCTURES       */
19       /*************************/
20
21   /* Here is a message structure for a stepper motor command sent by some task */
22   /* to the stepper motor server task, smcsrv(). What the various parts of the */
23   /* structure are and how they are used is beyond the scope of this example.  */
24
25   typedef struct smcmsg              /*  Stepper motor control message         */
26        {
27           RTXCMSG  hdr;                        /* header message               */
28           char    motor_num;                   /* stepper motor number         */
29           char    command;                     /* motor command                */
30           char    pos_num;                     /* position number              */
31           char    comp_code;                   /* completion code*/
32           char    n_params;                    /* number of parameters to send*/
33           char    filler;
34           struct
35                {
36                    char  param_code;           /* stepper motor command code   */
37                    char  c_count;
38                    int   par_val;              /* number of steps to move      */
39                } pval[7];
40        } SMCMSG;
41
42   /* The following structure is a block of information used by each stepper    */
43   /* motor while it is in operation. The stepper motor server task and the     */
44   /* stepper motor driver task (not part of this example) use these tables to  */
45   /* maintain control over the motors as each can be in a different state than */
46   /* the others. Multiple commands can be sent to a single motor and linked via*/
47   /* the two link pointers, cptr and lptr. */
48
49   typedef struct                     /* Stepper motor control block           */
50        {
```

```
51          char      status;                   /* status                       */
52          SMCMSG    *cptr;                    /* current pointer              */
53          SMCMSG    *lptr;                    /* last pointer                 */
54          POS_ARAY  pos_aray;                 /* position array               */
55          char      n_posn;                   /* number of positions          */
56          char      cur_posn;                 /* current position             */
57          char      card_drv;                 /* card driver                  */
58          char      flag_dno;                 /* flag device number           */
59          char      rate;                     /* motor rate (fast rate)       */
60          char      slope;                    /* motor slope                  */
61          char      slp_dvz;                  /* motor slope divisor (accel.) */
62          char      jumprate;                 /* motor jumprate (slow rate)   */
63          char      direct;                   /* motor direction towards home */
64       } SMCBLK;
65
66  #define NUMMOTOR 4
67
68  extern  SMCBLK mcb[NUMMOTOR];      /* these are motor control blocks used to */
69                                     /* hold information about each motor as it*/
70                                     /* is being used. All motors are able to  */
71                                     /* operate independently.                 */
72
73  /*************************************************************************/
74  void smcsrv(void)
75  {
76      int     i;
77      SMCMSG  *mptr;                 /* see headers.h for definition of SMCMSG */
78
79      /* Do any initialization prior to the start of the "forever" loop   */
80      for(;;)
81      {
82          /* first function is to receive the message, or rather a pointer     */
83          /* to the message. The next available message is received since the  */
```

```
 84          /* task argument to KS_receive() is 0.  */
 85
 86          mptr = (SMCMSG *)KS_receivew(SMCMBOX,(TASK)0); /* get next command   */
 87          i = mptr->motor_num;                        /* get motor # as index   */
 88          if ((mcb[i].status & SMASK) == IDLE)
 89          {                                           /* if IDLE, then ...      */
 90              mcb[i].cptr = mptr;                     /* set up current pointer */
 91              mcb[i].lptr = mptr;                     /* set last current ptr   */
 92              mcb[i].status = (mcb[i].status &(~SMASK)) + READY;
 93              ...
 94              /* more processing */
 95              ...
 96          }
 97          else
 98          {
 99              mcb[i].lptr->hdr.link = (RTXCMSG *)mptr;
100              mcb[i].lptr = mptr;
101          }
102          mcb[i].lptr->hdr.link = 0;                  /* Last Pointer = Null    */
103
104         /* now that the message has been processed, signal completion */
105
106         KS_ack((RTXCMSG *)mptr);
107
108     }
109 }
110
111
112
113 /*************************************************************************/
114 /* MOTOR1.C  --  Example of a task which operates one stepper motor.     */
115 /*************************************************************************/
116
```

```
117  #define POSone 1
118  #define POStwo 2
119
120  /*************************************************************************/
121  void motor1(void)
122  {
123      /* declare some different messages for stepper motor operations */
124      SMCMSG  pos1msg, pos2msg;
125
126      /* then fill in a few values */
127      pos1msg.pos_num = POSone;
128      pos1msg.motor_num = SM1;
129      pos1msg.command = 'M';
130
131      pos2msg.pos_num = POStwo;
132      pos2msg.motor_num = SM1;
133      pos2msg.command = 'M';
134      ...
135      /* processing */
136      ...
137      for(;;)
138      {
139      /* Send a message to the stepper motor server task, SMCDRV, but do not  */
140      /* wait for the completion of its processing. Continue with the task.   */
141
142          /* move SM1  to park  */
143          KS_send(SMCMBOX,(RTXCMSG *)&pos1msg,
144                  (PRIORITY)3, SM1_DONE);
145          KS_delay((TASK)0, 50/CLKTICK);         /* delay for 50 milliseconds */
146          ...
147          /* processing */
148          ...
149          /* wait on the completion of the above message here.               */
```



```
150          KS_wait(SM1_DONE);
151          ...
152          /* more processing */
153          ...
154       /* then send a stepper motor command to SMCSRV and wait on the completion*/
155       /* of the message processing before proceeding. The semaphore SM1_DONE is*/
156       /* used as the completion event flag. Notice the use of coercing the     */
157       /* address of the message to be of type RTXCMSG.*/
158
159          KS_sendw(SMCMBOX, (RTXCMSG *)&pos2msg, (PRIORITY)3, SM1_DONE);
160          ...
161          /* and still more processing */
162          ...
163       }
164    }
```

SIGNALLING MULTIPLE EVENTS ON THE SAME INTERRUPT

Hardware devices are often used which utilize one interrupt for several events associated with the device. This may be done to preserve interrupt lines on an interrupt controller chip, or to simplify the hardware and associated costs, or just because it makes good design sense. Usually, there is some sort of interrupt request status register which can be read to determine the actual source of the interrupt. In many of these implementations, it is possible for one or more events to occur simultaneously. It is this case which bears special attention.

In the situation where a single interrupt is associated with two or more different events, some special processing is required when writing the device servicing function of the interrupt service routine. Recall from Sections 3 and 7 that the ISR completes its processing by calling the Common Interrupt Service Routine Exit function, *KS_ISRexit()*. Recall also that one of the arguments passed to *KS_ISRexit()* may be the identifier of a semaphore associated with the interrupt. The semaphore, if specified, is passed to *KS_ISRexit()* so that it may be signalled. This scenario is the normal case and is well described in Sections 3 and 7. But when more than one semaphore needs to be signalled when the ISR is complete, some special techniques are needed.

When signalling a semaphore, *KS_ISRexit()* places its identifier into a list of semaphores, the Signal

List. The content of the Signal List is processed subsequent to some status checks. It is possible and permissible to make insertions into the Signal List from outside of *KS_ISRexit()*. For instance, an ISR may need to signal other semaphores besides the one it passes to *KS_ISRexit()* as an argument. RTXC provides for this requirement with the special function *KS_ISRsignal()*. The function requires the identifier of the semaphore that is to be signalled.

RTXC provides a global pointer to the semaphore Signal List, *siglist*, to permit such usage. *KS_ISRsignal()* consists of three steps. First, it disables interrupts. Secondly, it appends one semaphore identifier to *siglist*. Lastly, it enables interrupts. The inline code could be substituted for *KS_ISRsignal()* with equivalent results.

When there are two or more semaphores to signal, you have two options to accomplish the feat. First, call *KS_ISRsignal()* to signal one while still in the ISR and then call *KS_ISRexit()* with the second. Alternatively, you could signal both by calling *KS_ISRsignal()* twice, once for each semaphore, and end the ISR by calling *KS_ISRexit()* without passing a semaphore identifier.

Take, for example, the ISR for a UART driver. In it, one interrupt is used for both the INPUT_READY and OUTPUT_DONE events. A status register can be interrogated to determine the source of the UART interrupt. Since the input and output ports on the UART are asynchronous, it is possible for both

interrupts to occur simultaneously. When such an event occurs, the two semaphores, INPUT_SEMA and OUTPUT_SEMA, need to be signalled. A code example from a UART driver illustrates the first method.

```
/*
 * Interrupt service for input and output ports on a single UART
 *
 * Note, in some UARTS more than one interrupt may be pending at the same
 * time (i.e. TX done, RX ready).  KS_ISRexit() only allows for signaling a
 * single semaphore per interrupt.  In order to signal multiple semaphores,
 * KS_ISRsignal() must be called to signal any other
 * semaphore(s) associated with the event.
*/

/* C level CONsole interrupt handler */
FRAME *uartc(FRAME *frame)
{
   char inchar;
   extern SEMA *semaput;

   if (USART_STATUS == RX_DATA_READY)       /* test source of the interrupt */
   {                                        /* INPUT READY */
      inchar = read_char();                 /* read char and save */
      if (USART_STATUS == TX_BUFF_EMPTY)    /* test output port's status */
      {                                     /* OUTPUT DONE */
         KS_ISRsignal(OUTPUT_DONE_SEMA);    /* add semaphore to list */
      }

      /* put clear interrupt logic here if necessary */

      /* exit and signal char input semaphore */
      return(KS_ISRexit(frame, CONISEMA));
```



```
29   }
30
31   if (USART_STATUS == TX_BUFF_EMPTY)       /* test source of interrupt */
32   {                                        /* OUTPUT DONE */
33      if (USART_STATUS == RX_DATA_READY)    /* test input port's statuS */
34      {                                     /* INPUT READY */
35         inchar = read_char();              /* read char and save */
36         KS_ISRsignal(INPUT_DONE_SEMA);     /* signal char input semaphore */
37      }
38
39      /* put clear interrupt logic here if necessary */
40
41      /* exit and signal char output semaphore */
42      return(KS_ISRexit(frame, CONOSEMA));
43   }
44   return(KS_ISRexit(frame,0))              /* neither device interrupted */
45}
```

RTXC INDEX

—E—

—F—

—G—

—H—

—I—

—K—

—M—

—R—

—S—

—T—

—U—

—W—